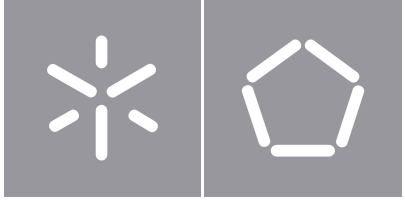
University of Minho
School of Engineering

Pedro Miguel Borges Rodrigues

Analysis of I/O patterns for data management systems

august 2023



University of Minho
School of Engineering

Pedro Miguel Borges Rodrigues

Analysis of I/O patterns for data management systems

Masters Dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
João Tiago Medeiros Paulo
Ricardo Manuel Pereira Vilaça
Tânia da Conceição Araújo Esteves

august 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I want to express my sincere gratitude to all those who contributed to the completion of this master's thesis. This thesis marks the culmination of a five-year journey at the University of Minho, during which I have grown both personally and professionally.

I am deeply grateful to all the collaborators and supervisors of this thesis. I thank Professor João Paulo for his invaluable guidance, continuous support, and encouragement throughout the writing of this dissertation. I extend my appreciation to Tânia Esteves for sharing her knowledge and experience, guiding me on the best possible path throughout this journey. I also want to express my thanks to Ricardo Vilaça for the numerous suggestions that enriched this work.

I am also grateful to the University of Minho and the HASLab laboratory for providing essential resources for the completion of this master's degree.

My thanks go to Dipcode and all its collaborators for their motivation and the knowledge they shared, which undoubtedly made me more capable of accomplishing this work.

Lastly, I want to thank my family and friends, especially my parents, my brother Afonso, and my girlfriend Vânia, for their unwavering support and motivation. To my friends and course colleagues André, José Pedro, and Luís, I also extend my gratitude for their support, motivation, and assistance throughout the academic journey.

In conclusion, this thesis would not have been possible without the collective efforts and support of all those involved. Thank you all for your invaluable contributions.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, august 2023

Pedro Miguel Borges Rodrigues

Abstract

The exponential growth of digital information that has been witnessed in recent years requires a continuous evolution and optimization of data management systems, such as databases and storage solutions.

In order to provide efficient processing and storage capabilities for large amounts of data, data management systems must adopt different optimizations (e.g., caching, replication, data reduction) that increase their complexity. As a result, developing, configuring and maintaining a data management system becomes increasingly difficult and costly.

Tracing and analyzing the interactions and exchanges between components of these systems is fundamental to uncover performance, correctness and dependability issues almost unavoidable in any complex solution. On the other hand, this presents several challenges, such as minimizing the impact on applications' performance and storage space, improving tracing accuracy and achieving real-time analysis, that must be explored.

With this thesis, we present a tracing and analysis pipeline capable of capturing and analyzing the I/O patterns of these data-centric systems in order to better understand their behavior, using LTTng as tracing tool.

In particular, the proposed solution includes a tracing component that efficiently collects disk and network I/O metrics originated by the target application. This component is the major focus of this thesis and allows for the capture of system calls that the application executes, as well as their arguments, in a non-intrusive and almost real-time way. The rest of the pipeline facilitates the analysis and visualization of captured events through search queries and diagrams, allowing the user to find potential performance and optimization problems.

In the end, we demonstrate that the proposed solution allows for the identification of inefficient and redundant I/O patterns in production applications without causing significant impacts on the runtime performance of the application and allowing for near real-time analysis.

Keywords analysis, data management, I/O patterns, LTTng, performance, tracing

Resumo

O crescimento exponencial de dados digitais a que se tem assistido nos últimos anos exige uma evolução e otimização dos sistemas de gestão de dados, como bases de dados e sistemas de armazenamento.

De modo a fornecer processamento e armazenamento eficaz de grandes quantidades de dados, os sistemas de gestão de dados devem adotar diferentes otimizações (e.g., caching, replicação, redução de dados) que aumenta a sua complexidade. Desta forma, desenvolver, configurar e manter um sistema de gestão de dados tornam-se cada vez mais tarefas difíceis e dispendiosas.

Analisar as interações entre componentes destes sistemas é fundamental para descobrir problemas de desempenho, correção e confiabilidade quase inevitáveis em qualquer solução complexa. Por outro lado, isto apresenta vários desafios, como minimizar o impacto no desempenho das aplicações e no espaço de armazenamento, melhorar a precisão do tracing e alcançar análise em tempo real, que devem ser explorados.

Com esta tese, apresentamos uma *pipeline* de tracing e análise capaz de capturar e analisar padrões I/O destes sistemas com o objetivo de melhor compreender o seu comportamento, utilizando o LTTng como ferramenta de tracing.

Em particular, a solução proposta contempla uma componente que coleciona de forma eficaz os pedidos de disco e métricas de E/S de rede originados pela aplicação alvo. Esta componente é o foco principal da tese e permite a captura de todas as chamadas ao sistema operativo que a aplicação execute, bem como os seus argumentos, de forma não intrusiva e quase em tempo real. O resto da pipeline facilita a análise e visualização dos eventos capturados através de interrogações de pesquisa e de diagramas, permitindo ao utilizar encontrar potenciais problemas de desempenho e otimização.

No final, demonstramos que a solução proposta permite identificar padrões E/S ineficientes e redundantes em aplicações de produção sem causar impactos significativos na execução normal da aplicação e possibilitando análise quase em tempo real.

Palavras-chave análise, captura, desempenho, gestão de dados, LTTng, padrões E/S

Contents

List of Figures

List of Tables

Acronyms

CaT Content-aware Tracing.

CBT Ceph Benchmarking Tool.

CPU Central Processing Unit.

CTF Common Trace Format.

eBPF Extended Berkeley Packet Filter.

ELK Elasticsearch, Logstash, and Kibana.

GB Gigabyte.

GCC GNU Compiler Collection.

GDB GNU Debugger.

GiB Gibibyte.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

IDC International Data Corporation.

JSON JavaScript Object Notation.

KB Kilobyte.

LTTng Linux Trace Toolkit: next generation.

MCAS Maneuvering Characteristics Augmentation System.

OS Operating System.

PID Process identifier.

PPID Parent process identifier.

RADOS Reliable, Autonomous, Distributed Object Store.

RAM Random Access Memory.

TID Thread identifier.

UUID Universally Unique Identifier.

YCSB Yahoo! Cloud Serving Benchmark.

Part I

Introductory material

Chapter 1

Introduction

The amount of digital data and the demand for its durability and dependability have grown exponentially in recent years due to technological advances. People rely on digital information more than ever before and this reliance will only increase in the future as consumers progressively embrace the technological evolution. The consequence of this increasing reliance on data will be a never-ending expansion in the amount of data that is created, captured and replicated around the world. International Data Corporation (IDC) predicts that this value will grow from 84 zettabytes (1 zettabyte is 1 000 000 000 terabytes) in 2021 to 221 zettabytes by 2026 with an annual growth of 21.2% [39].

Data management systems, such as databases and storage solutions, must implement several optimizations (*e.g.*, caching, replication, deduplication, compression) in order to provide effective processing and storage capabilities for large amounts of data, which increases their complexity. As a result, developing, configuring and maintaining a data management system becomes increasingly complex and expensive [28].

Users' lives can be affected and their trust eroded by system flaws. Consider the example of the Boeing 737 Max accidents in 2018 and 2019. These tragic incidents were a result of a software glitch in the Maneuvering Characteristics Augmentation System (MCAS) system. This flaw caused two separate plane crashes, claiming the lives of 346 individuals. The MCAS software malfunction triggered the planes to pitch downwards, ultimately leading to the catastrophic accidents [66]. This emphasizes the critical importance of addressing software vulnerabilities to ensure safety in complex systems.

As a result, detecting and troubleshooting anomalous behavior in a timely and effective manner is crucial to ensure that services are trustworthy [59]. However, modern applications interact with the rest of the system in complex and volatile ways, making it difficult to understand and analyze their behavior [2].

The oldest and most commonly used tools to understand and analyze application's behavior are *debuggers* (*e.g.*, GNU Debugger (GDB)). They can assist the developer in finding and fixing bugs, but they are unable to help on discovering performance issues or understanding the application's multiple interactions

with the Operating System (OS) and the outside world. On the other hand, *profilers* are tools that can be used to discover performance issues. These tools are important to understand where the system's performance is affected. However, they do not provide any indication of the cause of the performance loss at any particular time. To do so, it would be essential to look into the execution history as well as the overall context and status of the applications. This is where *tracing* is useful. Tracers, which work similarly to a tape recorder collecting executions' events, help developers to understand what is happening in a running application [53].

Therefore, the tasks of understanding and analyzing application's behavior can be helped by tracing and analysis frameworks, which allow for the monitoring of applications requests as they propagate over a distributed system, providing useful insights into how the system's state develops over time [28], making them almost unavoidable for developers and maintainers to work with. Tracing and analysis are fundamental to find performance bottlenecks, security vulnerabilities, malicious behavior, etc. [2].

1.1 Problem statement

Tracing tools are widely available in today's Linux distributions (*strace* [45], *ltrace* [11], *ebpf* [23], *ftrace* [62], *gperftools* [34], *dyninst* [8] etc.). However, current tools often come with various problems:

Intrusion. Many tracing tools require changing the application's source code or adding binary instrumentation. However, application code is often not available, making it impossible to use these tracers. This intrusive approach increases the complexity of the tracer setup for developers and maintainers, making it as difficult as the application's complexity [28].

Performance impact. It is impossible to build a tracer that does not affect the system's performance. However, this can be problematic if the user perceives the impact or if the overhead changes the behavior of the application, making the tracer irrealistic and inappropriate for production environments [2]. This effect is proportional to the number of events registered by the tracer, as well as the amount of data collected by it. If there are several collection points and/or large quantities of data being gathered, it might result in considerable overheads in terms of performance and resource usage (*e.g.*, RAM, CPU, storage) [28]. As a result, a balance must be struck between the system capacity, the number of collection points and the amount of information collected.

Scope. The scope at which data is collected is relevant. Collecting information at the userspace level provides more precise data about the application, but increases the number of events gathered and implies more intrusive solutions [9]. Capturing data at the kernel level, on the other hand, is less invasive,

but less accurate (*i.e.*, contains less information). Therefore, it is important to find a balance that allows for collecting relevant data on the execution of the target application without being intrusive (*e.g.*, capturing and analysing system calls).

File system impact. Most of the current tracers save the trace results in disk. This means that all captured events are stored in a file, resulting in files that are too large and introducing overhead in the file system. Therefore, the ability to filter events or the exploration of new types of storage (*e.g.*, efficient databases) is necessary.

Complexity of results. The use of a tracing tool, by itself, does not allow the user to obtain a complete view and easy understanding of the execution of a certain system. This is due to the complexity and the enormous amount of data resulting from the events captured by the tracing tool. In order to address this, these tools should be combined with an analysis pipeline to automate and facilitate the reading and understanding of the results obtained.

Analysis timing. The ideal scenario would allow the analysis and visualization of applications' behavior during their execution, that is, without waiting for the end of the execution. However, this approach requires events to be sent sent to a specific analysis pipeline during the execution of the application, which impacts the performance of the application being traced.

1.2 Objectives and contributions

This thesis aims to assist in the development, configuration and maintenance of data management systems by presenting a new tool for capturing and analyzing I/O patterns of complex systems to better understand their behavior.

Taking into account the issues raised in 1.1, the solution must be non-intrusive, meaning that no changes to the target application's source code are required; have minimal performance impact and use as few storage resources as possible, making it suitable to run in production environments; and provide a way to analyze and visualize the tracing results in near real-time.

To achieve this goal, this thesis explores the LTTng tool to provide a non-intrusive and low-overhead solution [53]. LTTng is used to capture the I/O requests of the target application at the system call level, providing a view of the requests executed by the application. Additionally, our solution integrates the LTTng tracer with an analysis pipeline, using the ELK stack, which includes Elasticsearch and Kibana, to automate the process of storing, correlating, analyzing, and visualizing the obtained data [25].

Hence, a new plugin for Babeltrace, which is a trace processing framework used for analyzing and

converting trace data from various sources and formats [6]), was developed to send the data collected by the tracer to Elasticsearch in near real-time, taking advantage of LTTng's [53] live mode.

To validate the proposed solution, performance tests were conducted on the various introduced components, and two real-world applications were traced and analyzed. This allowed us to assess the performance of the solution and validate its usefulness in real-world case studies.

Through the conducted tests, we have demonstrated that the presented solution does not cause significant impacts on the normal execution of applications and allows for near real-time analysis. Additionally, we have shown that the solution enables not only understanding the execution of a real-world application but also identifying potential performance issues through undesired I/O patterns. We have also demonstrated that our solution can be used to assist in identifying the root cause of reported performance problems in real-world applications.

1.3 Document structure

Chapter 2 presents a comprehensive study on tracing tools and tracing and analysis pipelines. Related works are also addressed, along with examples of alternative solutions to LTTng and tracing and analysis pipelines. Additionally, at the end of the chapter, a discussion is presented highlighting the advantages and disadvantages of the alternatives compared to LTTng, as well as the distinguishing elements between the proposed solution and all the related work.

Next is Chapter 3, which presents the principles followed in the design of the proposed solution, as well as the architecture and important aspects of its implementation.

In Chapter 4, we present evidence of the usefulness of our solution through the execution of test scenarios. This chapter provides an overview of the test setup, the methodology employed and a description of each executed test case. Following the functional tests, a brief discussion is provided, gathering the facts that demonstrate the utility of the proposed solution in this thesis.

In the final chapter, Chapter 5, a conclusion of the work carried out in this thesis is presented. Further, future work is proposed to enhance the presented solution.

Chapter 2

State of the Art

The use of tracing and analysis pipelines has become increasingly prevalent in the analysis and optimization of complex systems. As a result, a large number of research has been dedicated to the development of efficient and effective methods for tracing, analysis, and visualization.

In this chapter, we provide a comprehensive review of the current state of the art in this field. We begin by outlining the key challenges and open questions that remain in trace-based analysis, including issues related to scalability, accuracy, and interpretability. Next, we explore multiple methods and approaches that have been proposed for tracing, analysis, and visualization, discussing their strengths and limitations.

2.1 Background

Monitoring a system's performance and security is critical throughout its lifetime. It assists developers in testing and understanding efficiency problems early in the implementation process, allowing them to improve the implemented solution. During the deployment phase, it assists administrators in identifying and predicting potential bottlenecks in the system, allowing them to make judgments on possible changes to the configuration to improve its performance and security. After deployment, in the support phase, when costumers report an issue, it assists maintainers in observing and understanding the cause of the problem, minimizing wasted time and increasing efficiency in the problem identification process [68].

Software tracing is an effective method of recording information about the execution of a program. It is used by programmers, system administrators and technical support employees to diagnose problems, improve application performance, and get a better knowledge of system behavior. Tracing is important for analysing/debugging applications that include multiple processes interacting with the operating system and with one another (*e.g.*, web servers, databases, telecommunications systems, storage systems, etc.). Tracers often capture events that occur frequently. Thus, tracers must be tuned to handle large volumes of data while having the least influence on system performance [69].

Merely capturing information using tracing tools is insufficient to achieve a thorough understanding of complex systems. As aforementioned, tracing tools must be complemented with data analysis and visualization tools to facilitate the interpretation and comprehension of the large volume of collected data. Consequently, it is necessary to integrate the tracing tool into an analysis pipeline that simplifies and automates the processing and analysis of data obtained through tracing.

Analysis pipelines, which are responsible for analyzing and presenting the results in a visual and summarized way, make it easier for consumers to employ tracing tools. These pipelines automate the tracing process by passing the tracer's captured results to the analysis component, which then passes the information to the visualization component, where the final results are displayed, thus serving as a supplement to the tracers and improving the usability and effectiveness of the tracing process.

2.1.1 Tracing tools

Tracing tools are a reliable and efficient method of debugging and reverse engineering complex systems. Many tracers have arisen during the last decade, covering all layers of the software stack and even reaching the hardware level (*e.g.*, Intel VTune Profiler [38]).

In general, tracing is a method for intercepting and recording events generated by applications, such as functions invoked by the application. Typically, function handlers (known as *probes*) are attached to points of interest (*e.g.*, system call tracepoints) to be invoked whenever an event reach these (*e.g.*, when a system call is called). These probes are responsible for collecting the desired data about the event that occurred, such as timestamps, event type and/or arguments [33].

Tracepoints are points of interest located in strategic points of the code (*e.g.*, throughout the kernel) that provide a hook for invoking a probe. These tracepoints can be turned on or off (*i.e.*, connected to the probe or not). When a tracepoint is off, it has no performance impact on the traced system other than to add a tiny time and space penalty for checking conditions. On the other hand, when the tracepoint is on, the probe is called whenever it is reached [18]. Probes connected with tracepoints should be low cost to avoid large overheads and have no influence on applications' performance.

Dynamic and static instrumentation

Tracepoints can be set either at compile time (statically) or during runtime (dynamically). In static instrumentation, the instrumentation code is injected into the binary at build time, and the location of the tracepoints is known beforehand. On the other hand, in dynamic instrumentation, this instrumentation code is injected during runtime by following user-defined points. Dynamic tracing is frequently misun-

derstood with dynamic instrumentation, however dynamic tracing means that tracing can be enabled or disabled at runtime, and it can support both static and dynamic instrumentation [33].

A static instrumentation example is having each function call of an application preceded by a call to a common tracing probe. In other words, the binary contains explicit calls to a specific probe upon each function entry (or exit). This may be accomplished by using the `-pg` flag on GCC, which generates a binary with the `mcount` routine call as a preamble for each function [32]. Hence, whenever the application invoke a function, the `mcount` probe will be called and executed.

Trap-based instrumentation is an example of a dynamic instrumentation mechanism. It makes use of operating system's traps feature to insert and execute custom probes at almost any position in the kernel. Linux kernel implements this method via Kprobe infrastructure [33]. When a Kprobe is registered, it replaces the instruction at the probepoint with a breakpoint instruction [55]. When the breakpoint is later executed, the kernel's breakpoint handler is called to store the application's state (*e.g.*, registers, stack, etc.) and give control to the Kprobe infrastructure, which then calls the tracing probe. Once this process completes, the original instruction (*i.e.*, replaced by the breakpoint) is executed and the application's execution resumes with the instruction following the probepoint [43].

Kernel and user space tracing

Tracing can be done in either kernel or user space and while many tools provide both options, the majority of them cannot perform both at the same time.

The impact of capturing events at the kernel level differs from that of capturing events at the user space level. User space tracing provides more detailed information about the event that occurred (*e.g.*, can obtain information about the application's functions that were executed and their arguments), but these events occur more frequently, and the creation of tracepoints in the application is necessary, making user space tracing quite intrusive.

On the other hand, at the kernel level, it is not necessary to introduce tracepoints into the application, making the tracing process less intrusive. This makes it possible to capture events without changing the source code of the target application. Consequently, the level of detail of the information collected at the kernel level is lower than at the user space level (*e.g.*, provide information about low level events such as system calls instead of application's functions).

As one of the main goals is to minimize intrusiveness, this thesis focuses on kernel-level tracing, which allows capturing system calls and their associated arguments.

2.1.2 Kernel-based tracing tools

strace

Strace is a command-line utility for tracing system calls in Unix systems [46, 41]. It is essential for system administrators and troubleshooters to find and fix problems in applications when the source code is not available since tracing them with strace does not require recompilation. Additionally, it can be useful for students to better understand how simple programs and system calls work. Furthermore, developers can use strace to closely examine the interactions between kernel and user space, assisting them in understanding what the application is doing [45].

It makes use of ptrace system call that allows one process (tracer) to monitor and control the execution of another process (targeted process). It provides tracing of system calls, signals and read and write operations to memory and registers. In detail, the targeted process is stopped once it enters or exits a system call, allowing for the tracer to collect the program values (*i.e.*, system call name, arguments and return value) [28, 37, 45].

As a disadvantage, stopping the target process execution on each system call may impose significant performance overhead on the targeted application, which is not suitable for production environments [33, 28].

eBPF

The Extended Berkeley Packet Filter (eBPF) is a universal in-kernel virtual machine that allows user space applications to inject code into kernel at runtime, without changing kernel source code [56]. Today, eBPF is widely used in a variety of scenarios, including high-performance networking and load-balancing, extracting observability data at low overhead, application tracing and much more [23].

It allows attaching small programs to specific points in kernel and user space (*i.e.*, tracepoint, kprobe, uprobe), in order to be executed anytime an event (*e.g.*, system call or kernel function) arises. It also includes maps (key-value data storage) for sharing data between eBPF programs, kernel and user space. Moreover, it uses a ring buffer, which is a circular buffer used to store tracing data, with a fixed size that overwrites old data when it is full.

Furthermore, eBPF is acknowledged as a secure method of instrumenting the Linux kernel since its Verifier component runs a sanity-check on the eBPF program before attaching it. This method ensures that it does not compromise the kernel's stability and security. Meanwhile, this mechanism has implications such as the restriction of the total amount of instructions and stack space [23, 28].

ftrace

Ftrace is a tracer included in the Linux kernel that traces all kernel function entries to reveal insights into its internal operations. It includes multiple configurations like the buffer's size and the clock source to timestamp the events. Ftrace can be used for a variety of purposes, including function tracing, tracepoints, system call tracing and dynamic instrumentation [32, 64].

Function tracing reports the entry and exit of kernel level functions. Ftrace leverages the `TRACE_EVENT` for static instrumentation or the kprobe for dynamically hook into different points of the kernel. Additionally, ftrace's probe uses a ring buffer to store the events captured. The ring buffer can be configured to either overwrite the oldest events or drop incoming events when it is full [33].

By default, ftrace timestamps the recorded events using the local clock (*i.e.*, a CPU-local clock source for quicker reads). However, this clock does not guarantee monotonicity or synchronization with other clocks. Alternatively, ftrace can be configured to use other clock sources, such as a global clock (*i.e.*, system-wide clock) or a logical counter [33].

Ftrace segments its ring buffers into pages and manages each one separately, thus limiting the size of an event, including its payload [63]. Although this implementation choice has advantages such as avoiding lazy memory allocations, it also results in two major drawbacks: i) the size of an event, including its payload, is limited to the size of a page, and ii) a page can only be read when it is full [33].

LTTng

The Linux Trace Toolkit: next generation (LTTng) is an efficient and full-system tracer that supports both kernel and user space tracing. It was designed to have minimal performance impact [69, 48]. Similar to ftrace, LTTng leverages per-CPU buffers to avoid concurrent writing and synchronization. LTTng's trace files are produced in the CTF format, which is a file format optimized for the analysis of multi-gigabyte data [20, 33].

LTTng can create multiple recording sessions, enable and disable recording event rules, filter events efficiently, start and stop tracing, etc. Generated trace files can either be saved on disk or sent over the network. Moreover, LTTng can be configured to execute user-defined actions when it emits an event (*trigger*). Although creating an event might be processing costly because there is a need to evaluate the arguments of the instrumentation point, LTTng implements multiple optimizations to avoid it when the tracer knows, thanks to independent properties, that the instrumentation point would never emit such an event. This knowledge come from properties like the instrumentation point type, name and log level.

Additionally, LTTng provides multiple instrumentation point types depending on the tracing domain

used. For kernel tracing, it provides *LTTng tracepoint* (*i.e.*, a statically defined point in the source code of the kernel image or module), *Linux kernel system call* (*i.e.*, entry, exit or both of a Linux kernel system call), *kprobe* or *kretprobe* (*i.e.*, a single probe dynamically placed in the compiled kernel code) and *Linux user space probe* (*i.e.*, a single probe dynamically placed at the entry of a compiled user space application function through the kernel). As for the user space tracing it provides *LTTng tracepoint* (*i.e.*, a statically defined point in the source code of a C/C++ application).

It also has four recording session modes: i) local mode, in which the trace data is written into the local file system, ii) network streaming mode, where the trace data is sent over the network after the trace is finished, iii) snapshot mode, in which the trace data is only written when LTTng takes a snapshot, and iv) live mode, where the trace data is sent over the network for live reading.

In order to manage ring buffers, LTTng uses channels. A channel is an object that is in charge of a group of ring buffers, with at least one ring buffer per CPU. Each ring buffer is divided into multiple sub-buffers. When a recording event rule matches an event, LTTng can record it to one or more sub-buffers of one or more channels. When a sub-buffer runs out of space, the tracer marks it as consumable and forwards subsequent events to another sub-buffer that is available. A marked sub-buffer returns to the available state after being consumed by a consumer daemon [19].

At some point, all sub-buffers can be full, leaving no space to record the following events. By default, LTTng loses event records to prevent substantial delays in the execution of the application, privileging performance over integrity. However, the user space tracer support a blocking mode. Moreover, LTTng can be configured to either take a discard approach, where it discards the newest event records until a sub-buffer becomes available or an overwrite approach, where the sub-buffer holding the oldest event records is emptied so that the newest event records can be written [53].

2.1.3 Tracing and analysis pipelines

Using a complete tracing and analysis pipeline offers several advantages over using standalone tracing. It allows the collection of a more complete and detailed picture of a system's behavior, which may lead to a deeper understanding of system performance. A tracing and analysis pipeline can also save time and effort for developers and system administrators by automating the processing and analysis of large amounts of data.

Additionally, the visualization tools provided by a tracing and analysis pipeline can make it easier to read and understand the data collected by the tracing process, leading to faster issue detection and resolution [42].

Generally, tracing and analysis pipelines have three main components, as depicted in Figure 1:

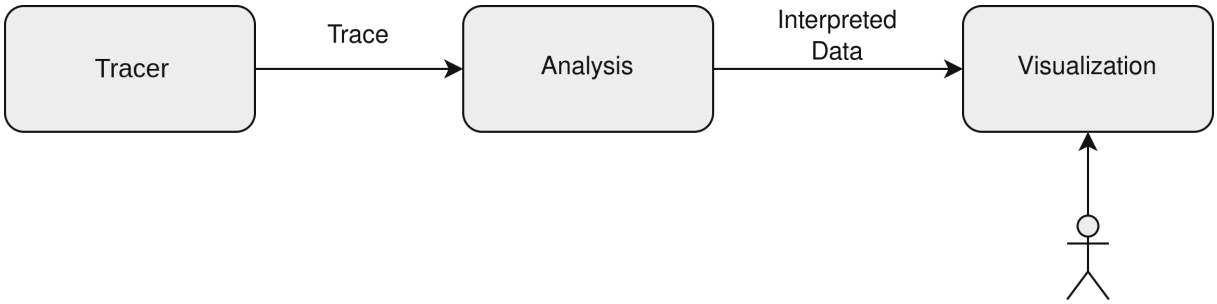


Figure 1: Base architecture for a tracing and analysis pipeline.

Tracer. This component is responsible for capturing the events. It relies on tracing tools such as strace, eBPF and LTTng [45, 23, 53]. These tools are used by the tracer component to intercept the context (*e.g.*, type of event, timestamp, PID) and the content (*e.g.*, buffers) of network (*e.g.*, connect, disconnect, send, receive) and storage (*e.g.*, open, close, read, write) requests, preferably in a non-intrusive way [28]. As in a distributed system many events are collected, the data (*e.g.*, files) resulting from this component is extremely large, leading to increased storage overhead. Hence, pipelines must implement a solution to face these problems.

Analysis. The analysis component is responsible for receiving events related to the tracing of applications' executions and store, process and correlate them in order to return detailed and summarized information about the applications' behavior. In this regard, the main challenges of this module are how and where to store the data, how to correlate it, and how to provide it to the user.

Visualization. This component is responsible for presenting the results generated by the analysis component to the user. Generally the results' presentation is graphical through tables, time series and/or dashboards. Typically, these charts are paired with filters and various customizations, such as colors and labels, that eases the interpretation of more complex systems.

A good example of analysis and visualization components is the ELK stack [25], where Elasticsearch (analysis component) stores the data for fast and powerful search that scales with ease, allowing the execution of complex queries on stored data [24], and Kibana (visualization component) displays the Elasticsearch data, providing multiple graphics, filters and customizations to understand and show the way application flows [26].

2.2 Related work

In this section, we present multiple examples of tracing and analysis pipelines. There are several examples of incomplete pipelines (without all components) and complete pipelines, some of which use LTTng in the tracer component and others use other tracing tools mentioned earlier (*e.g.*, strace, eBPF).

Given that, this section is divided into three different subsections: Section 2.2.1 presents complete pipeline solutions that do not use LTTng in the tracing component, Section 2.2.2 details pipelines that use LTTng to capture data but do not have a complete pipeline, and Section 2.2.3 presents complete pipeline solutions with LTTng as the tracing tool.

2.2.1 Complete pipelines without LTTng

Falcon [58], CaT [28], and Horus [59] are good examples of complete pipelines without LTTng as the tracing tool.

Falcon focus on the causality of events. It can receive data from different sources (*e.g.*, log4j, strace, tshark [44, 36, 13]) and causally relates their events to present a space-time diagram with their ordered global execution [58, 52, 17]. The other two examples are based on Falcon. CaT is a monitoring pipeline that innovates in the tracing and analysis phase, as it also captures the content where the events operate (network and storage data buffers), and calculates the similarity these to detect data mutation, corruption or leakage in identical I/O messages [28]. Finally, Horus innovates in the visualization component, where the results are saved in Neo4j, which is a graph database that includes a powerful query language for filtering, refining and visualizing graphs [51, 59].

Another good example of a complete pipeline is DIO. This tool focuses on tracing system calls and uses eBPF as its tracing mechanism, which provides greater flexibility in data collection without the need to modify the application's source code [23]. Additionally, DIO provides a near real-time pipeline as it can asynchronously and in real-time send events to an Elasticsearch, enabling their subsequent visualization in Kibana [29].

The solution presented in this thesis is inspired by and based on the pipeline introduced in DIO, with the main objective of using LTTng as the tracing tool to feed a similar pipeline.

2.2.2 Incomplete pipelines with LTTng

Re-animator

Re-animator [2] is a sophisticated system-call tracing tool that concentrates on storage-related system calls (*e.g*, read, write, open, close) and collects as much information as possible, including full data buffers and arguments. The traces are written in a compact, efficient and self-explanatory format named DataSeries [3]. Re-animator also includes a working prototype replayer that concentrates on calls related to file-system state.

The goal of this tool is to improve the accuracy of capturing and replaying a complete set of collected events while minimizing overhead and being scalable and verifiable. It also supports summary hashes for the content of data contained in system calls, such as write and reads, avoiding storage overhead from increasing, as recording all available information might result in enormous log files.

In addition to the system call arguments, Re-animator records the exact time the call began and ended, the PID, TID, parent and process group PID, and errno, as well as the data buffers for reads and writes. Re-animator's main tracing tool, is based on LTTng [53], however, there is also a library to use strace [45]. The LTTng-based version imposes low performance overhead on the traced program, but requires kernel patches installation. On the other hand, the strace-based version has higher overhead but does not require kernel changes or special privileges.

The strace solution allows them to record event information without changing the source code [45]. However, in order to capture the same type of content using LTTng, they had to make adjustments to the LTTng source code, modifying the probes pre-defined by it [2].

Because it lacks an analysis pipeline, this approach comes up short of becoming a complete monitoring pipeline.

Performance Anomaly Detection through Sequence Alignment

Performance issues within a program or task can be difficult to identify, especially when the program or task has a complex structure or involves a significant number of steps. One method of uncovering performance issues is through the analysis of the execution's critical path. The critical path represents the sequence of steps that must be completed in a specific order and without delay in order for the program or task to be completed successfully. By identifying the critical path, it is possible to prioritize the most important steps and focus on any bottlenecks or other issues that may be impacting the performance of the program or task [65].

Janecek et al. present a tool for identifying performance issues through the examination of the execution critical path. After the data is collected, a preliminary analysis is conducted to identify any anomalies. In the event that anomalies are detected, machine learning algorithms and sequence alignment techniques are utilized to identify and locate the source.

The tool in question receives two distinct sets of traced data. The first set represents a normal execution of the program, while the second set represents an execution that is to be analyzed for possible anomalies. These data sets are collected using the LTTng tool and are separated into individual executions. The critical path is then calculated for each execution, and two types of critical path vectors are generated. This method is similar to the approach described in DepGraph [73], where the various possible states of a thread are used to identify the internal and external dependencies of a given execution [40].

After the data is collected and the various vectors are calculated, a preliminary analysis is conducted to identify potential performance issues within a particular execution. If no anomalies are detected, the execution is deemed normal and the analysis is terminated. In the event that anomalies are identified, the anomalous executions are subjected to further scrutiny in the following phases. Machine learning algorithms are used to extract the different types of execution from the calculated vectors, and similarity algorithms are utilized to identify executions with abnormal behavior. These executions are then subjected to the final phase of the pipeline, which involves the localization of the anomaly through the use of sequence alignment techniques to identify the specific differences between normal and anomalous executions [4, 70].

The data analysis is primarily automated, and not all necessary data is provided to enable the user to understand the traced execution. Additionally, no analysis or visualization tools are provided to facilitate reading and filtering of the collected data. Therefore, it cannot be considered a complete tracing and analysis pipeline.

2.2.3 Complete pipelines with LTTng

Automatic Integrated Anomaly Detection Framework

Performance anomaly analysis is essentially finding patterns in the execution flow that do not conform to expected behavior. These anomalies can have different sources such as application bugs, hardware failures, etc. Performance anomalies are different from high resource utilization. An application may be using a high number of resources, as long as it is considered as expected, it is not considered an anomaly [54, 31].

One of the main ways of detecting these anomalies is through tracing, as it continuously provides us with detailed information about the system. However, most existing solutions require human administrators to analyze the huge amount of data, decreasing efficiency and even accuracy in detecting anomalies. That said, Kohyarnejadfard et al. propose a precise anomaly detection framework with minimal human intervention.

As shown in Figure 2, this framework is divided into different modules. System call data is collected through LTTng during program execution. The tracing data is then sent to a Data Extraction module which processes it. Data extraction and transformation is done using Trace Compass, that is a Java tool for analysing and visualizing any type of logs or traces. This module is responsible for processing the data and preparing it for the detection module. This data contains vectors extracted from the tracing data collected by LTTng that are used both to train the model and to detect anomalies, based on the execution time and frequency of the system calls [48].

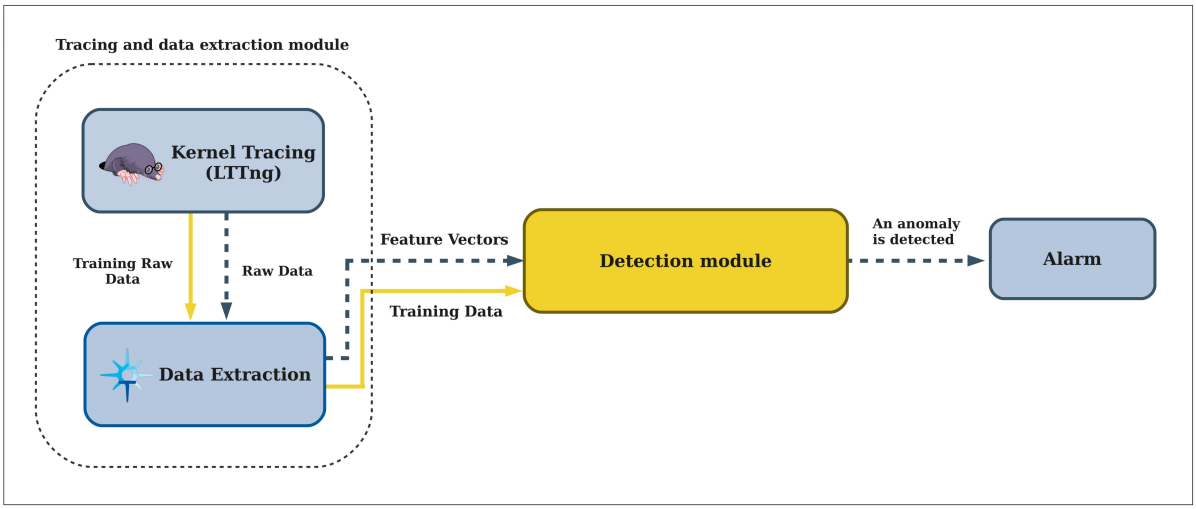


Figure 2: Base architecture for the anomaly detection framework. Extracted from [49]

DepGraph

DepGraph is a tool whose main objective is to detect bottlenecks caused by blocking dependencies. This tool is based on tracing to identify blocking dependencies between threads and presents a graph showing the relationships between different threads, as well as the resources used during the execution of a given task [73, 47].

Despite being innovative, it is not the only tool capable of generating a graph of blocking dependencies. wperf is another example, however, it only focuses on identifying events with a large impact on all threads [74]. DepGraph innovates in identifying the reason for these unexpected latencies, focusing on

the performance of a single thread interacting with other threads and all system resources.

DepGraph's main job is to extract all the dependencies needed to complete a given run. For this, certain system calls are collected through tracing at the kernel level. In order to reduce the size of the tracing data, they transform the trace content into states. A thread can have 5 different states: running, waiting for disk, waiting for another thread, waiting for cpu or contented on a lock. After extracting the state transitions of the threads, a Waiting Dependency Graph is constructed. This graph allows one to analyze and compare dependencies between different threads and between threads and resources. In the end, it is possible to identify the main reasons for latency problems, since it is possible to see in which situations a thread is waiting for another [73].

As mentioned above, the tracing component is one of the main components of this tool and is composed by LTTng. In this case, DepGraph does not need to capture all system calls in the system. It focuses on collecting the most relevant events that can help identify performance issues, such as *socket_accept* and *socket_shutdown*, as they indicate the start and end of a specific request. To allow the identification of blocking dependencies between different threads, the tracing component starts by identifying state changes in the threads. For this purpose, it captures specific system calls that mark these state transitions: *sched_switch* indicates that a thread is going to be replaced by another; *irq_handler*, *softirq*, and *hrtimer_expire* indicate that a particular thread is currently blocked; *sched_wakeup* indicates that a thread has woken up and is no longer blocked.

Analysis of distributed storage clusters

Distributed storage systems are increasingly popular due to their flexibility, scalability, and fault tolerance [71]. However, as mentioned earlier, the complexity of these systems increases the difficulties in analyzing and detecting problems on behalf of administrators. To address this, there are different tools that automate the process of analysis in order to assist system administrators.

Benchmarking tools, such as CBT and RADOS Bench, allow for the evaluation of a cluster's performance using different workloads and analyzing various collected metrics (e.g., execution time, resource utilization), however, these tools work with synthetic workloads and cannot be used in real-world scenarios [60, 35]. On the other hand, monitoring tools provide another type of information about the state of the cluster. In this case, metrics are collected in real-time and allow for graphical visualization of them. However, the information collected from the various metrics used is not sufficient to identify the cause of the problems found.

Daoud and Dagenais propose a performance analysis framework that provides important data about

the system, collecting information at both the kernel and userspace levels.

The tracer component uses LTTng to perform kernel and userspace tracing on the different nodes of the system. However, in order to use tracing in production environments, they have created a different approach. This approach is based on two different tracing sessions: exhaustive tracing and lightweight tracing. With this approach, it is possible to keep tracing active indefinitely without impacting the rest of the system's behavior. In the case of lightweight tracing, it is always active and collects a smaller number of events that can be immediately analyzed to detect potential problems. If no problems are detected, the tracing results can be discarded, avoiding taking up disk space. When a problem is detected, exhaustive tracing is activated, which collects all the necessary tracepoints for a more detailed analysis of the problem. In this case, the tracing results are saved and sent to Trace Compass for later analysis and visualization [16].

Recovering disk storage metrics

Performance metrics are numerical values that are used to assess the performance of a storage system. These metrics may include measures of throughput, latency, efficiency, and reliability [12]. Tracing, on the other hand, is a technique that involves the systematic collection of data related to the operation of a system over a period of time. By analyzing traces of a storage system, it is possible to gain a more detailed understanding of the system's behavior and identify patterns and correlations that may not be apparent through the analysis of performance metrics alone. Tracing can therefore be a valuable tool for supplementing performance metrics and identifying the root causes of performance issues.

In [15], a complete pipeline solution for analyzing storage subsystems is presented . This pipeline consists of the various different components of a tracing and analysis pipeline pipeline: tracer, analysis, and visualization.

The tracer component of this solution uses LTTng. System call event tracing provides a wealth of information, but it is not sufficient for a thorough understanding of the inner workings of the storage subsystem. In order to gain insight into storage activity beyond system calls, tracepoints must also be used. While the Linux kernel includes several storage-related tracepoints, they do not provide sufficient detail for in-depth analysis. To overcome this limitation, a module based on Kprobes was added to LTTng to provide the necessary additional tracepoints for more detailed tracing data analysis [50, 15].

This tool enables the user to view and filter metrics that are derived from the tracing data. However, due to the large volume of data, it is not practical to compute and calculate these metrics on demand every time the user requires them. As a result, after data collection, the trace result is fully processed and

the various metrics are calculated and stored in efficient data structures to enhance the efficiency of user filtering and visualization [57]. Data visualization is accomplished using Trace Compass, which required the development of a plugin to provide various views and charts of the storage subsystems [1].

2.3 Discussion

Of the four tracing tools presented in 2.1.2, only strace does not provide user space tracing or static instrumentation. On the other hand, eBPF, ftrace and LTTng support user space and kernel space tracing, as well as static instrumentation (*e.g.*, tracepoins) and dynamic instrumentation (*e.g.*, kprobe, uprobe).

However, strace guarantees that no events are lost along the trace, unlike other solutions that cannot guarantee this. In the remaining solutions, if the speed of creating new events is greater than the speed of consuming them, the ring buffers will be full and events will have to be discarded. LTTng and ftrace can be configured to have the same behavior as eBPF or to simply drop new events until there is free space again. Additionally, LTTng's user space interface can be configured to adopt a blocking approach like strace, thus ensuring that events will not be missed, however, this decision can increase the impact on application performance.

In terms of features and configuration, LTTng is the most complete tracing tool, allowing the propagation of trace results over the network, as well as a live recording mode where events can be consumed and read in real time. This approach makes it unnecessary to create a trace file, which can grow uncontrollably, thus reducing the storage overhead caused by the tracer. Further, the trace file generated by LTTng is in a compressed format, thus taking up less space than the files generated by the other tools.

Gebai and Dagenais conducted an assessment of the application performance impact of different tracers and concluded that strace had the greatest impact on the application. The tracer with the least impact on the application was LTTng, followed by ftrace and finally, but before strace, eBPF. They also concluded that, when comparing the fastest tracers, that is, LTTng and ftrace, the main advantage of LTTng was in data serialization, where it managed to do it in almost half the time compared to ftrace.

As a result, it is a goal to this thesis to explore and integrate LTTng into a tracing and analysis pipeline. It will be necessary to understand how the communication between LTTng and an analysis platform can be done. LTTng's live session mode is a possible solution to accomplish that. In this mode, LTTng spreads the collected data over the network instead of storing it on disk. Thus, the captured data can be sent to the analysis component while having less storage overhead. However, this integration brings implementation challenges, such as, the incompatibility of the format generated by LTTng with the input format of the

analysis component. To solve this, it is necessary to introduce an extra component that takes the data from LTTng and translates it into a format that the analysis component supports.

Hence, this master thesis aims to overcome the aforementioned challenges, resulting in a complete pipeline with a tracer, analysis and visualization components. The tracer component includes LTTng that is responsible to collect and send data over the network to the rest of the pipeline while being non-intrusive and having less overhead as possible.

The solutions presented in 2.2.3 also use LTTng as the main tracing component and provide a complete pipeline. However, all of these tools focus only on detecting problems, and if no anomalies are detected, they do not provide any information about the application's execution. In contrast, our solution allows the user to understand the behavior of the application without the need for anomalies to occur and allows the creation of dashboards, charts, and searches that allow the user to quickly detect problems in a particular execution. The tracing and analysis pipeline presented by our solution provides total freedom to the user to define the filters, searches, and/or rules that determine whether an execution contains any anomaly, unlike the previously presented solutions that rely on an algorithm that can fail and mask potential problems that are not included in it.

Therefore, the solution presented in this thesis provides a complete tracing and analysis pipeline that allows the user to understand and visualize the entire execution of the application, as well as create complex dashboards, charts, and searches on the captured data to quickly detect possible problems.

Part II

Core of the Dissertation

Chapter 3

Design and Architecture

The previous chapter explores tracing and analysis pipeline concepts, as well as examples of existing tracing tools and related pipelines. Based on this research, this chapter details the design and architecture of our solution.

Our solution involves a complete tracing and analysis pipeline capable of addressing the problems mentioned in Section 1.1 using LTTng as a tracing tool. Therefore, our solution follows the principles indicated in Section 3.1 and the architecture presented in Section 3.2.

3.1 Design principles

Our solution contemplates the following design principles to address the problems and achieve the objectives mentioned earlier.

One of the first and main problems mentioned was the intrusion of current tracing tools. As one of the main objectives was to address this issue, we presented a solution that explores kernel-level tracing tools, such as LTTng. Although LTTng allows for tracing at the userspace level, this solution focuses on kernel-level tracing, allowing it to be non-intrusive to the application and not requiring any changes to its source code.

Another objective was to minimize the overhead introduced by the tracer in the application's execution. Once again, the use of LTTng allows us to achieve this objective, as Gebai and Dagenais showed, it is one of the tracing tools with the least impact on the application's performance.

Another challenge previously identified is the high number of generated events and the difficulty in correlating and analyzing these events as the application itself becomes more complex. Therefore, the presented solution provides analysis and visualization components that use the ELK stack, allowing users to store, analyze, perform complex filtering, and view charts to ease the understanding of the data resulting from the tracing phase.

Finally, reducing the storage impact on the file system and obtaining near-real-time analysis are also objectives of this solution, which can be achieved through the live mode tracing feature provided by LTTng. This feature allows the collected data to be sent over the network as LTTng captures events, avoiding the need to wait for the end of execution to collect the captured data.

3.2 System architecture

As depicted in Figure 3, our solution is composed by a tracer, a parser, an analyzer and a visualizer. The tracer component intercepts the system calls selected by the user, and the parser receives the collected system calls, parses them, and sends them in real-time to the analyzer component. The analyzer component persists the received system calls, and allows for complex queries and summaries/statistics about them. After the data is persisted, it can be viewed through the visualizer component, which queries the analyzer component and allows for the construction of dashboards and charts that facilitate the user's reading of the results.

Only the tracer needs to be deployed on the same machine as the target application. The other components, including the parser, can be distributed across different machines, reducing the performance overhead imposed on the machine that runs the application.

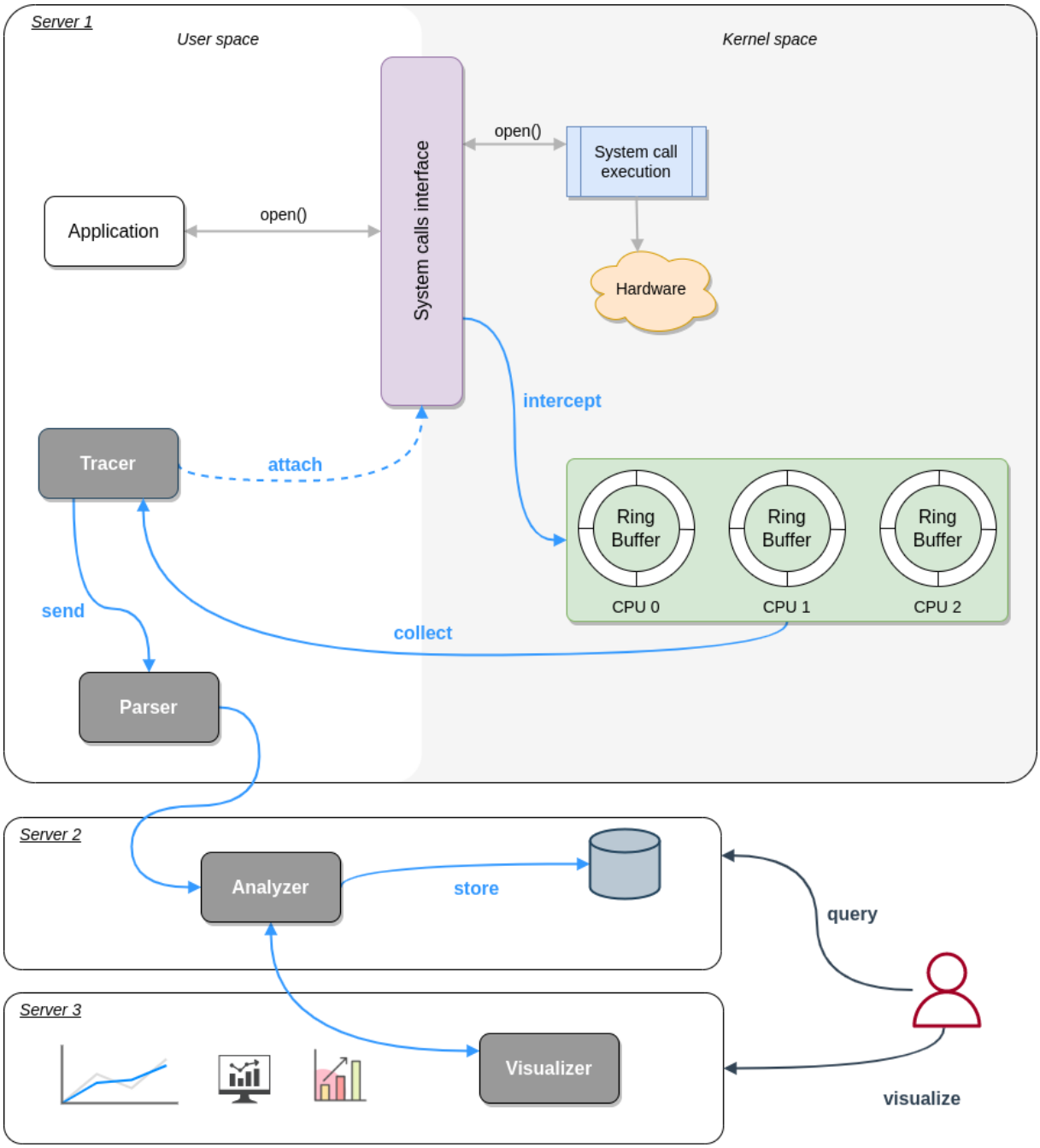


Figure 3: Architecture.

3.2.1 Tracer and parser

The tracer must run alongside the target application and collect data from the system calls generated by it. In this way, LTTng [53] is used as the tracing tool to trace the application in a non-intrusive manner. To do this, LTTng uses probes that generate events at each entry and exit of a system call, capturing all of their arguments as well as the return value. These events are stored in ring buffers to be consumed later

at the user space level.

The tracer consumes these events present in the ring buffer and can save them to disk or send them over the network to a listening reader. These events are always stored on disk or sent over the network in the CTF format [21].

Our solution is capable of intercepting all system calls supported by LTTng, which currently are 340 different system calls[1]. Additionally, although LTTng does not directly allow tracing on a specific executable (*i.e.*, like strace, which traces the executable process itself and any possible childrens), a solution was implemented that allows LTTng to run on a specific executable command, which filters the events captured by the process being executed and by its child processes.

By default, an event generated by LTTng captures all arguments of the system call, as well as the result returned by it. However, it is possible to add extra context to each of the events generated by the tracer in order to ease their analysis in a more advanced stage of the pipeline. Currently, LTTng provides 144 more extra context variables that can be collected together with the system call arguments. Among these are some essential variables for processing this data, such as PID, PPID, TID, hostname, and process name. Many of these context variables are also performance counters that allow monitoring the impact of the system call on CPU and memory usage [5].

The parser can receive data directly from the tracer over the network or read it from disk. It is composed by babeltrace2 [6], which allows translating the CTF format to a human-readable format. This parser is responsible not only for translating the received events but also for organizing them in a way that obtains a structure with all the information about the system call. Since two events are generated per system call (an entry event and an exit event), it is at this stage that these two events are merged, generating a single event that includes the execution of the system call with all available information, such as the arguments, return value, entry and exit timestamps, and extra process information (*e.g.*, PID, PPID, TID, hostname, process name).

In the end, a JSON object is generated with this event representing the execution of a system call, in order to facilitate its analysis in the next stages of the pipeline, and sent to the analyzer.

3.2.2 Analyzer

This component is composed by Elasticsearch [24]. Elasticsearch allows persisting and analyzing the collected data. This data is persisted by indexing it in a document-based fashion, which increases the performance of executing complex search queries. In addition to executing search queries, Elasticsearch also

[1] Retrieved from the execution of LTTng's list command

allows updating existing documents and transforming data before indexing it through an ingest pipeline.

3.2.3 Visualizer

The visualizer provides a visual interface for exploring the previously indexed events. It allows executing queries and filtering the captured events, and provides a graphical visualization through dashboards and diagrams of the analysis results on the collected data. This component uses Kibana [26], which allows creating multiple dashboards and different types of charts (*e.g.*, histograms, time series charts) that represent the data indexed by Elasticsearch.

3.3 Optimizations, pipeline integration and implementation

As aforementioned, it was necessary to implement a command to use LTTng to start a tracing session on a specific executable and a Babeltrace2 plugin that parses and sends the results obtained to Elasticsearch.

3.3.1 Process filtering

The default behavior of LTTng is to start a tracing session to capture events generated throughout the system. It also provides functionality to enable tracing specific PIDs. However, while it is possible to use the concept of this thesis in any tracing mode, it would be beneficial to enhance its usability by enabling the execution of LTTng with an executable specified as an argument. This would ensure that the tracing data collected is exclusively related to the execution of that specific program, similar to how strace operates [45]. In addition to simplifying result analysis, this approach also helps minimize performance impact and disk space usage, which are key objectives of this thesis.

Although LTTng supports tracing for a specific PID, it lacks the capability to trace a particular executable. To address this limitation, we have implemented a solution that enables LTTng to be run on an executable, thereby enabling tracing the execution of the targeted program (*i.e.*, main process) and all its related processes (*i.e.*, any child process).

In order to achieve the aforementioned behavior, two components were implemented: *lttng-trace*, which starts a tracing session on a specific executable, and *lttng-noty* which is responsible for adding the PIDs of child processes as they are created. Both of these components were implemented using the C library, *liblttng-ctl*, provided by LTTng itself.

lttng-trace

The behavior of this component is similar to the one provided by the *lttng-cli* command. The goal is to accept an executable as an input parameter and activate tracing for the specified executable and any child processes that may be spawned during its execution.

Thus, this component is responsible for creating and starting the tracing session, and subsequently activating the desired tracepoints. The goal was to simplify the process and abstract as much as possible the commands required to run LTTng. In this sense, the activation of tracepoints and the addition of context to events is done in this component.

The program can be used in two different ways: run with an executable as an argument or without arguments. If the command is run without arguments, then the created session is simply started and captures events generated by the entire system.

However, if an executable is passed as an argument, our program *forks* and activates the tracing session for that PID only, leaving the child process on a sleep state while creating and activating triggers so that, when a *sched_process_fork* event is captured, a notification is sent to the application with the PID of the created process. However, it is necessary an application that listens to these notifications and adds the PIDs to the whitelist of processes that are targets for the session's tracing. This is where the other component, *lttng-noty*, comes in. Thus, after the triggers are activated, a child process is created that runs *lttng-noty*.

As triggers are active and the *lttng-noty* is running in a child process, the tracing session is started and a signal is sent so that the child process that will run the received executable can continue with its execution.

When the execution is finished, the child process will terminate and the tracing session will be destroyed, along with the deactivation of the previously activated triggers.

lttng-noty

The purpose of this program is to subscribe to notifications emitted by LTTng when a *sched_process_fork* event is captured and to add the new PID to the whitelist.

Similar to *lttng-trace*, the *liblttng-ctl* library provided by LTTng was used to subscribe to notifications generated by the capture of a *sched_process_fork* event. After receiving these notifications, the program extracts the data captured in the generated event and adds the PID of the new child process to the tracing session so that this process is added to the list of processes targeted by tracing.

3.3.2 Pipeline integration

In order to send the captured data to Elasticsearch, it was necessary to implement a plugin for Babeltrace2. As previously mentioned, there are already plugins that implement components that allow data collection from LTTng, both from file and real-time reading from a relay daemon. However, there are no plugins that provide the offloading of events directly to Elasticsearch.

To address this, the *elastic* plugin was created, which incorporates a sink component specifically designed to transmit data to an instance of Elasticsearch.

Despite the ease of use of the python bindings provided by Babeltrace2 in developing a plugin, they have a huge impact on performance. Given the expectation of dealing with a large number of events per second, using them to implement the plugin was not feasible. Therefore, the C library, *libbabeltrace2*, provided by Babeltrace2 was used instead.

In order to implement a sink component, the C library provided by Babeltrace2 offers an API that iterates over the messages generated by the filter component. As we have seen, LTTng creates two events for each call to a given system call, an entry event and an exit event. Our goal with this plugin is to correlate these two events and transform them into a single one, sending the result to an instance of Elasticsearch.

Firstly, it was necessary to establish a strategy for associating the entry and exit events of a system call, as they may not be sequential. To solve this problem, a data structure was created that is responsible for storing input events that are pending for the output event.

Figure 4 depicts the flow followed by an event from the entry point of the developed sink component until its exit. As we can see, an event is classified as an entry or exit event. When an entry event is processed, the data is extracted to a data structure and it is stored in an HashTable of pending events where the key is the concatenation of the session name with the TID that generated the event. Then, when the corresponding exit event is processed, the key is searched in the HashTable, the information extracted from the exit event is added to the data structure, and a JSON object with the data of the respective system call is generated and sent to Elasticsearch.

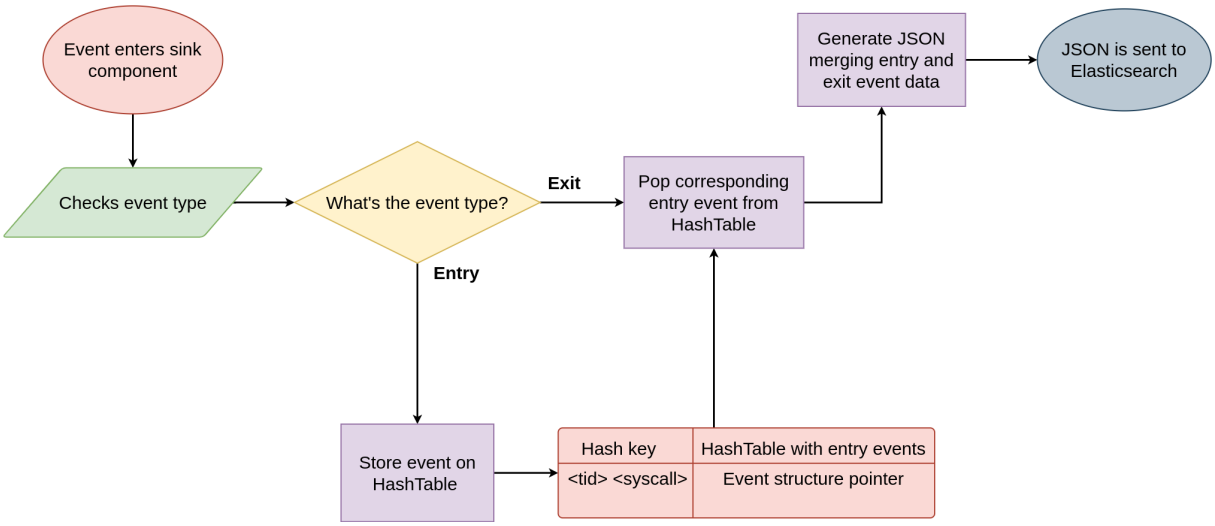


Figure 4: Event flow on developed sink component.

In the process of generating the JSON object to send to Elasticsearch, several relevant extra fields are added. Fields such as *category*, *event_type*, and *type* are created to categorize the system calls based on the data they work with, for example, to differentiate a system call that works at the storage or network level. These fields are useful for later understanding the amount of system calls related to storage or network. Additionally, a field representing the execution time of the system call (*execution_time*), resulting from the subtraction of the timestamp of the exit event from the entry event, is also included.

After sending the data to Elasticsearch, the data structure representing the system call is no longer useful and is removed from memory.

Given the high number of events, it is not feasible to send the data to Elasticsearch through generic HTTP libraries such as *curl*, as the overhead would be excessive. Therefore, to facilitate communication with Elasticsearch and take advantage of its bulk indexing capabilities, the *go-elasticsearch*[2] library provided by Elasticsearch was used. This library allows adding messages to a *BulkIndexer* to be sent asynchronously in bulk. To integrate this with our C-written plugin, it was necessary to create a Go module that encapsulated communication with Elasticsearch.

3.4 Configuration and usage

The installation of our solution should be performed in two distinct phases. Initially, an infrastructure with part of the analysis pipeline should be set up, including Elasticsearch and Kibana. As demonstrated earlier, these components can be installed and initialized on separate servers to reduce the overhead on

[2] https://pkg.go.dev/github.com/elastic/go-elasticsearch/v7@v7.13.1

the target application. After deploying the analysis pipeline, the user can proceed with the installation and configuration of the tracer and parser.

To proceed with the installation of the tracer, it is necessary to install the LTTng kernel modules, which can be achieved by installing the package *lttng-modules*. Once the LTTng modules are installed, no further installation is required. Simply use the Docker images built as part of this thesis to run the tracer and parser.

The parser should be started before the tracer to be immediately listening for events. Therefore, the parser can be initialized as follows:

```
$ docker run -it --name parser --net=host -e ES_ADDRESSES=<elasticsearch address> -e
↪ ES_USER=<elasticsearch user> -e ES_PWD=<elasticsearch password> -e
↪ ES_INDEX_NAME=<elasticsearch index> -e LTTNG_ADDRESS=<tracer address> -e
↪ pedrordgs/babeltrace:latest
```

Previous command will create a container with a Docker image where the parser is installed and running according to the configurations provided as environment variables. Next, we can start the tracer as follows:

```
$ docker run -it --name lttng_ls --pid=host --privileged --cap-add=ALL --net=host -v
↪ /lib/modules:/lib/modules -v /usr/src:/usr/src -v
↪ /sys/kernel/debug/:/sys/kernel/debug/ -e LTTNG_SESSION_NAME=<session name> -e
↪ LTTNG_SYSCALLS='open;close;creat;read;write' -e
↪ LTTNG_EXTRA_CONTEXT='pid;tid;hostname;procname;ppid' pedrordgs/lttng:latest
↪ <command>
```

This command creates a container with the tracer running on the specified command and sending events over the network. The system calls and additional information to capture are defined through environment variables, as shown above.

With this setup, the tracer will capture the system calls generated during the execution of the received command and send them over the network. Meanwhile, there will be an instance of the parser running and listening for events at a specific address. As it collects the events, it correlates them and sends them to an Elasticsearch instance, allowing for their visualization through Kibana.

Chapter 4

Evaluation

In this chapter, our solution was subjected to several tests in order to evaluate its performance impact, as well as to confirm its usefulness against real-world application use-cases. In this regard, with this chapter, we intend to answer two fundamental questions:

- What is the performance overhead introduced by the different components of the solution?
- How can the solution be used to understand and discover issues in real-world applications?

Therefore, two different types of tests were performed. To answer the first question, several stress tests were created, where the application under analysis was subjected to a high amount of load to understand the impact that our solution imposes when analyzing large volumes of data. The goal of this type of test is to validate the impact caused by each component of our solution.

On the other hand, to answer the second question, two real-world applications, Elasticsearch and RocksDB [24, 30], were analyzed using our solution in order to obtain data that facilitate the understanding of them, as well as the detection of possible unexpected I/O patterns and issues.

In the next sections, we detail the experimental setup used in the tests, as well as all the test cases executed, including their workloads and the results obtained.

4.1 Experimental setup

All the indicated tests were run with the same setup: one server with the application, tracer, and parser, and two separate servers for the remaining analysis and visualization components.

All servers ran on *Linux Ubuntu 20.04.5 LTS* with kernel version *5.4.0-125-generic*. The application was running on a server with a 4-core *Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz*, 16 GiB of RAM, consisting of two *8GiB DIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0.4 ns)* devices, a *Samsung SSD 970 EVO Plus 250GB*, and a *250GB Samsung SSD 860*.

Additionally, both nodes running Kibana and Elasticsearch are equipped with a 6-core *Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz*, 16 GiB of RAM, with a *16GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)* device, a *Samsung SSD 970 EVO Plus 250GB* and a *HDD 500GB WDC WD5000AZLX-7*.

4.2 Performance tests

The first set of experiments addresses the question of the impact of our solution on the execution of an application issuing a large volume of storage requests. Thus, we intend to determine the impact of the different components of our solution with different configurations, allowing us to answer this question in more detail. In the following subsections, we detail the workload used, as well as the methodology and results obtained from the tests.

4.2.1 Workload

In order to evaluate the performance of our solution, we conducted tests using the Filebench benchmarking tool [67]. Filebench is a flexible and powerful tool for testing and benchmarking file systems and storage solutions.

We used a workload that simulates a file server, where clients are reading and writing files of different sizes in parallel. The workload was designed to test the performance of the system under heavy load, with a large number of files and threads being processed simultaneously.

The workload was designed to create 10,000 files of 128KB each in a single directory. The workload then performed a number of operations on these files, including reading, writing, appending, and deleting files.

To simulate a real-world file server scenario, we used 4 threads to read and write files in parallel. Each thread performed a series of operations on a randomly chosen file, such as creating, writing, reading, and deleting the file. The workload was designed to be I/O intensive, with large I/O sizes and frequent I/O operations.

To ensure consistent and accurate results, we ran the workload for a fixed duration of 20 minutes. We also preallocated the files and cleared the file system cache before each test run to eliminate any caching effects.

4.2.2 Test methodology

For each test run, metrics were collected regarding analysis statistics (*e.g.*,number of processed/discarded events, execution time), Filebench throughput and latency, and system information such as CPU, RAM, and disk usage obtained through dstat, which is a versatile tool that allows users to gather system resource usage data and monitor performance metrics in real-time [72].

To ensure more reliable results, each test case was run three times, and the average and standard deviation of the three runs were calculated.

4.2.3 Configuration

As mentioned before, seven different test cases were created. Although LTTng supports 343 different system calls and 144 extra context variables, it doesn't make sense to activate all of them. Taking this into account, and since we intend to conduct performance testing, we have configured our solution to collect 55 different system calls: *read*, *write*, *open*, *close*, *stat*, *fstat*, *lstat*, *lseek*, *pread64*, *pwrite64*, *readv*, *writev*, *socket*, *connect*, *accept*, *sendto*, *recvfrom*, *sendmsg*, *recvmsg*, *bind*, *listen*, *socketpair*, *setsockopt*, *getsockopt*, *fsync*, *fdatasync*, *truncate*, *ftruncate*, *rename*, *creat*, *unlink*, *readlink*, *mknod*, *fstatfs*, *readahead*, *setxattr*, *lsetxattr*, *fsetxattr*, *getxattr*, *lgetxattr*, *fgetxattr*, *listxattr*, *llistxattr*, *flistxattr*, *removexattr*, *lremovexattr*, *fremovexattr*, *openat*, *mknodat*, *newfstatat*, *unlinkat*, *renameat*, *readlinkat*, *accept4* e *renameat2*; and we add only 4 context variables necessary for event correlation: *pid*, *tid*, *hostname*, *procname* e *ppid*.

Vanilla

This test case only runs Filebench without any type of active tracing. The main goal is to obtain a reference point to calculate the impact of the remaining components in relation to the normal execution of the application being analyzed.

Impact of LTTng

For this test case, only the tracer with LTTng running in normal mode, which writes the collected events to a file, was activated. After the Filebench execution, the obtained results were passed to the parser to extract metrics such as the number of captured and discarded events. Metrics of latency and system information were collected separately for the tracing and parsing phases.

Impact of LTTng live mode

Similar to the previous test case, in this test case, we have the tracing phase separated from the parsing phase. The main difference is the use of LTTng with live mode, allowing us to understand the impact of this feature on the tracing process. Although LTTng operates in live mode, the results are also written to disk in the CTF format. This strategy enables separating the parsing phase from the tracing phase. Even if there are no readers actively receiving the events sent over the network, they are stored on disk, making later parsing possible.

Impact of babeltrace2

Unlike the previous setups, all the following test cases no longer have the tracing phase separated from the parser phase, therefore, the tracer and the parser run concurrently and in live mode. In this case, the parser runs babeltrace2 with its default plugin, where it was configured to discard all received events.

Impact of writing results to stdout

This test case is very similar to the previous one. The only difference is that the default plugin of babeltrace2 was configured to write the results to stdout.

Impact of the newly developed plugin

Similar to the previous case, the results obtained are written to stdout, however, the new developed plugin (*i.e.*, elastic plugin) is used, which generates the data in JSON format ready to be sent to Elasticsearch.

Impact of sending data to Elasticsearch

In this last test case, unlike the previous one, the collected data is sent to Elasticsearch through the plugin developed on this thesis instead of being written to stdout.

4.2.4 Results

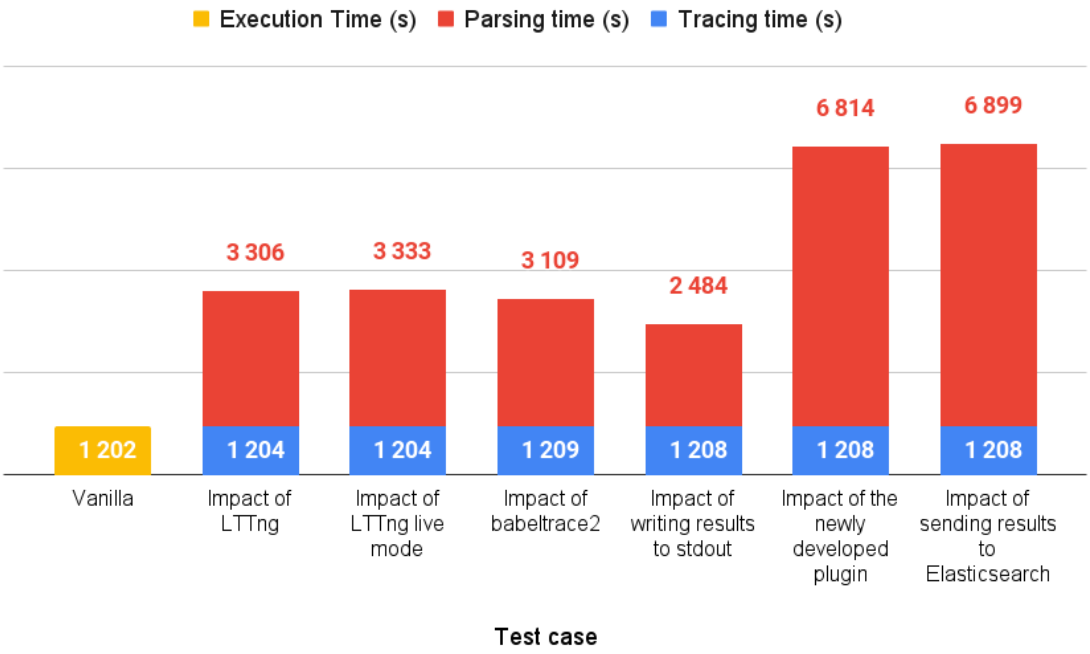


Figure 5: Tracing and parsing time in seconds.

Figure 5 shows the tracing and parsing times for each test case, except for the vanilla execution, which represents the execution time of filebench vanilla. For the LTTng impact and live mode test cases, since the parsing phase is run separately after tracing, the total time corresponds to the sum of these two times. In the remaining test cases, where the parsing runs at the same time as the tracing, the total time corresponds to the longer of the two times, which is always the parsing time.

Therefore, we observed that in all test cases, in terms of tracing time, the tracing ends almost simultaneously with the application's execution. This indicates that LTTng is capable of generating events and organizing all the information during the application's execution, allowing us to obtain results immediately after its execution.

Regarding the parsing time, we noticed that ignoring the events or writing them to stdout does not have a significant impact. In this case, writing the results to stdout even took less time than simply ignoring the events, which can be explained by the lower number of events generated when they are written to stdout, as observed in Table 1.

The most significant impact on the parsing time occurs in the test cases where the plugin developed as part of this thesis is used, resulting in the parsing time being doubled. While the default plugin of

babeltrace2 receives the entry and exit events of the system call and simply prints them to stdout, the new plugin contains additional logic to relate them, generate a single event resulting from the combination of both, and convert the event into a JSON object ready to be sent to Elasticsearch. This extra processing causes the parsing time to be longer with the developed plugin.

Comparing the test cases that use the new plugin, we can conclude that sending the events to Elasticsearch takes almost the same time as simply writing them to stdout. Although fewer events were generated when sent to Elasticsearch, the parsing times are not significantly different. This is because the events are sent to Elasticsearch asynchronously and in bulk, taking advantage of Elasticsearch's bulk indexing feature.

Test case	**Metrics**			
	Operations	**Handled events**	**Lost events**	**% lost events**
Vanilla	195 180 700	-	-	-
Impact of LTTng	173 487 606	473 163 902	307	0.000065
Impact of LTTng live mode	174 493 605	475 832 700	65 968	0.0139
Impact of babeltrace2	137 917 662	376 031 409	114 787	0.0305
Impact of writing results to stdout	95 223 469	259 690 345	3 011	0.00116
Impact of the newly developed plugin	142 317 007	388 028 682	105 579	0.0272
Impact of sending results to Elasticsearch	121 574 601	331 550 168	14 316	0.00432

Table 1: Metrics average values for each test case

In Table 1, the averages of the most significant metrics extracted in each test case are presented. By analyzing this table, we can observe that activating tracing did not cause a significant impact on the execution of Filebench. The biggest impact on the execution of Filebench was when we activated the parser simultaneously with tracing, which caused the number of operations generated by Filebench to be lower. This phenomenon can be explained by the fact that the parser consumes system resources, preventing Filebench from executing as many operations as it would if it were running "alone". Additionally, we can observe that whenever more resource-consuming logic was added, such as writing to stdout or sending to Elasticsearch, the number of operations generated by Filebench was lower.

Regarding the other metrics, we can conclude that the live mode feature of LTTng causes a higher percentage of lost events, however, this percentage remains quite low, always below 0.05%, which makes this impact negligible.

Test case		**Metrics**	
		CPU (% usage)	**RAM (GBytes)**
Vanilla		96.715	2.31 GB
Impact of LTTng	*Tracing*	94.524	1.97 GB
	Parsing	26.528	14.30 GB
Impact of LTTng live mode	*Tracing*	94.086	13.85 GB
	Parsing	26.558	15.84 GB
Impact of babeltrace2		53.999	14.94 GB
Impact of writing results to stdout		69.853	14.87 GB
Impact of the newly developed plugin		38.103	15.58 GB
Impact of sending results to Elasticsearch		43.323	15.54 GB

Table 2: CPU and memory usage for each test case

Observing Table 2, which shows the CPU percentage and the number of memory bytes used in the execution of each test case, we can observe that there was not a significant impact on resource usage when activating the application tracing without live mode. We can also verify that the parser does not require many CPU resources, however, on the contrary, we can observe that there was a large memory usage in the execution of this component. Additionally, we observed that when activating the live mode of LTTng, there was a significant increase in the amount of memory used during the execution of Filebench.

4.3 Use cases

In this section, we present two examples of real-world scenarios where the use of the solution proposed in this thesis allows the detection of performance issues and unexpected I/O patterns. As mentioned earlier, two real applications, Elasticsearch and RocksDB, were analyzed.

In the first case, we aim to analyze I/O patterns and identify unexpected patterns in the execution of a benchmark on Elasticsearch. On the other hand, in the case of RocksDB, we aim to identify the possible cause for some performance issues.

In both tests, LTTng was configured to run in live mode, and babeltrace2 was configured to use the newly developed plugin, sending the data to Elasticsearch.

4.3.1 Elasticsearch

The aim of this test is to demonstrate that our solution can be used to obtain information about the I/O operations of the application, as well as identify inefficient file access patterns.

For this purpose, Elasticsearch v8.3.0 [24] was chosen as the application to be analyzed. This use case test involves loading the Elasticsearch search engine with data from the *geonames* dataset using the Rally benchmark [10], which includes information on geographical locations such as cities, countries, and postal codes. The aim is to simulate a real-world scenario where large amounts of data need to be indexed and searched efficiently.

The benchmark was configured to run the *geonames* dataset, which, by default, indexes 11 396 503 documents in an Elasticsearch instance, with 8 concurrent clients. Additionally, our solution, similar to the performance tests, was configured to collect nearly all available system calls as the objective of the test was to understand the application's execution. This results in a higher number of events, but it provides more information about the application's behavior, thus facilitating its comprehension.

Through the captured data and subsequent analysis in Kibana, we were able to understand that, with the workload used, Elasticsearch executed over one million system calls, all related to storage operations. Additionally, we observed that approximately 90% of the system calls were data-related operations.

In Figure 6, taken from a Kibana dashboard, we can see the most frequently executed system calls by Elasticsearch during the test execution. From this, we can observe that more than 60% of the system calls were writes, and almost 25% were reads, indicating that approximately 85% of the system calls were either reads or writes (*i.e.*, *read* and *pread64* system calls). This highlights the significant I/O load introduced by the benchmark used.

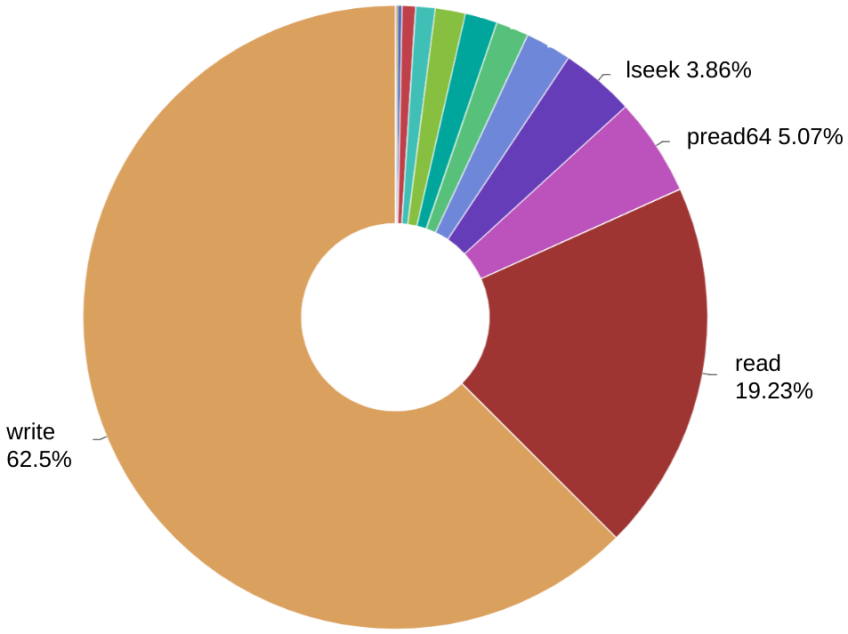


Figure 6: Top system calls by number of executions.

Additionally, we were able to gather information about the most frequently accessed relative paths of Elasticsearch (*/usr/share/elasticsearch-8.3.0/data/**) using the openat system call during the test. This insight was obtained through the visualization presented in Figure 7.

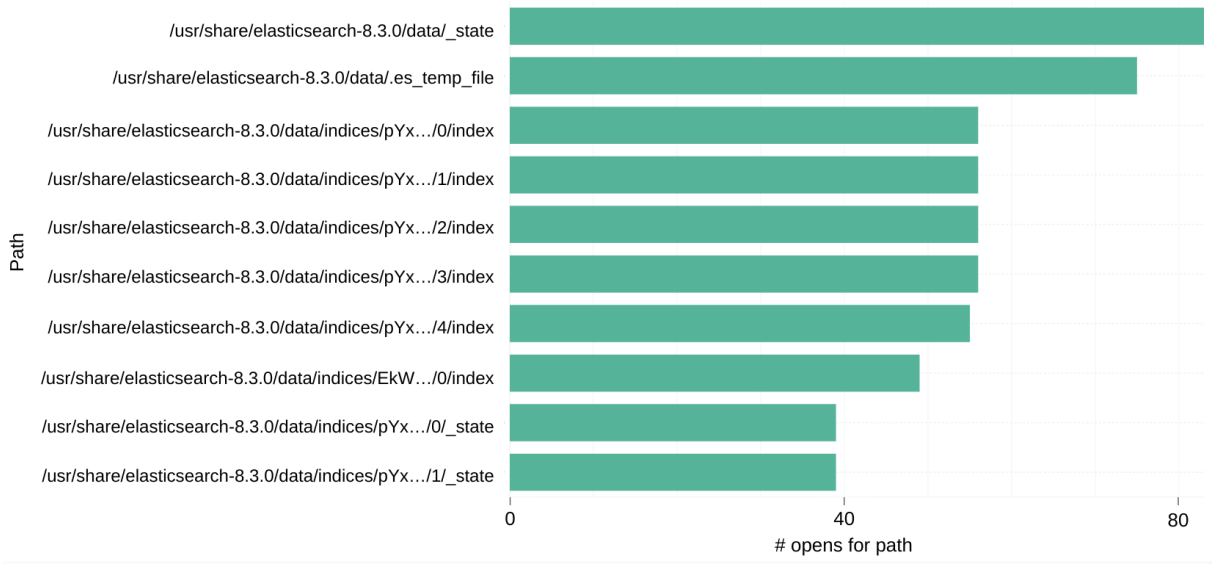


Figure 7: Top open paths for files from elasticsearch data directory.

We can also analyze how Elasticsearch accesses these files. To do so, we studied the *.es_temp_file* file to understand how Elasticsearch accesses and manipulates it. With this in mind, using Kibana, we filtered

the various system call arguments that can receive the file path (*i.e.*, *path*, *pathname*, and *filename*):

```
args.filename.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file"
or args.path.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file" or
args.pathname.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file"
```

Later on, when analyzing the data returned by the previous filter, we selected a thread that accesses these files and filtered the system calls by the selected TID in order to reduce the number of events and obtain a more detailed study:

```
(args.filename.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file"
or args.path.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file" or
args.pathname.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file") and tid:
1556857
```

Since the system calls that manipulate files (*i.e.*, *write*, *read*, *close*) receive the file descriptor as an argument rather than the file path, we are not including them in this filter. To overcome this, we selected two file descriptors returned by the previous results and added them to the filter:

```
(args.fd: 376 or args.fd: 377 or args.filename.keyword : "/usr/share/elasticsearch-
8.3.0/data/.es_temp_file" or args.pathname.keyword :"/usr/share/elasticsearch-
8.3.0/data/.es_temp_file" or args.path.keyword : "/usr/share/elasticsearch-
8.3.0/data/.es_temp_file") and tid : 1556857
```

By applying the necessary filter, we obtain the visual representation presented in Figure 8. Analyzing these visual representations, we can observe that for each access of this file, Elasticsearch opens the file, performs a write operation, opens the file again to perform an fsync, closes the two file descriptors, and subsequently deletes the file. From this analysis, we can conclude that Elasticsearch opens this file twice, once for writing and once for performing fsync. During this time period, this thread keeps two file descriptors open pointing to the same file.

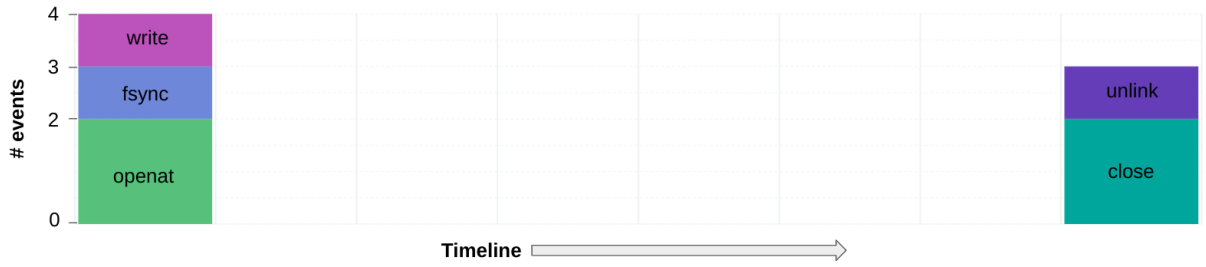


Figure 8: Sequence of system calls on */usr/share/elasticsearch-8.3.0/data/.es_temp_file* file access.

Additionally, we can analyze the number of times Elasticsearch opens this file by applying the following filter:

> args.filename.keyword : "/usr/share/elasticsearch-8.3.0/data/.es_temp_file" and system_call_name.keyword : openat

As we can see in Figure 9, Elasticsearch opens the file in question twice every two minutes. These two openings correspond to the problem mentioned above. Thus, we demonstrate that the observed access pattern occurs in a periodic manner, with a frequency of every 2 minutes.

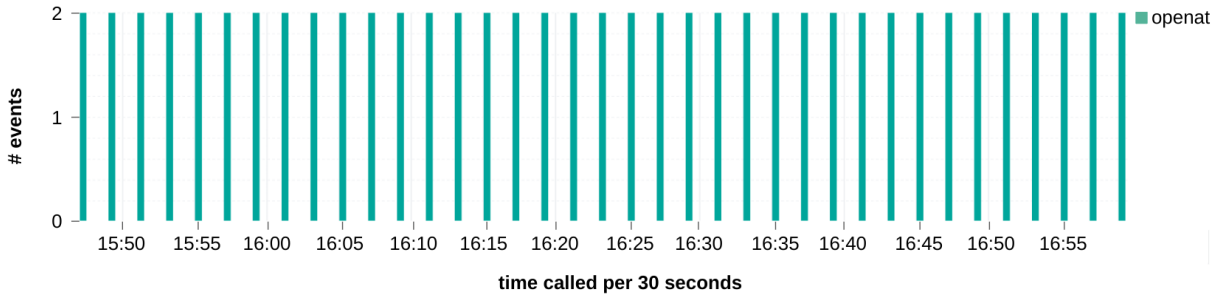


Figure 9: Timeline for */usr/share/elasticsearch-8.3.0/data/.es_temp_file* opens.

After observing how Elasticsearch accesses and manipulates the *.es_temp_file*, we can compare our findings with the code in their repository [27]. By searching for file accesses to *.es_temp_file*, we find the file *elasticsearch/server/src/main/java/org/elasticsearch/monitor/fs/FsHealthService.java*, where in the *monitorFSHealth* method of the *FsHealthMonitor* class, we can identify the previously observed pattern. Upon analyzing the code, we observe the following sequence:

```
168 try (OutputStream os = Files.newOutputStream(tempDataPath,
    ↪  StandardOpenOption.CREATE_NEW)) {
169   os.write(bytesToWrite);
170   IOUtils.fsync(tempDataPath, false);
171 }
172 Files.delete(tempDataPath);
```

Upon analyzing the documentation of the *fsync* method in the *IOUtils* library, we concluded that this method opens a channel to a specific file and closes it after the synchronization is completed. This means that the method opens and closes the file again, resulting in two unnecessary system calls (*open* and *close*), as the file is already open and the thread holds a file descriptor pointing to it. This problem can cause performance degradation in cases of I/O-intensive workloads.

Based on this analysis, we were able to map the information obtained from the system calls using our solution with the code in the Elasticsearch repository. The first *open* corresponds to the creation of the OutputStream to write to *.es_temp_file*. Next, the *write* operation corresponds to *os.write* method, that writes a random UUID to the previously opened stream. The following three system calls (*open*, *fsync*, and *close*) are issued by *fsync* method from *IOUtils* library. After exiting the *try* block, the *OutputStream* is closed, resulting in the subsequent *close* operation. Finally, the deletion of the temporary file issued the *unlink* operation.

Additionally, we can confirm that this pattern is executed every two minutes, as demonstrated in our analysis. Upon reviewing the code, we can conclude that this temporary file is used for performing health checks. The interval between these health checks is configurable, but it has a default value of two minutes, defined in line 66 of the *FsHealthService.java* class file.

Therefore, we can conclude that the solution proposed in this thesis not only allows us to analyze general metrics and aid in understanding the execution of an application but also identify undesirable I/O patterns that can impact the application's performance.

4.3.2 RocksDB

With this test case, we aim to demonstrate that our solution assists developers in identifying potential causes of performance issues. To accomplish this, we used our solution to pinpoint the reason behind high tail latency observed in client requests sent to RocksDB. RocksDB is a high-performance, persistent key-value storage engine developed by Facebook. It is specifically designed to enable efficient storage and

retrieval of data for modern, high-scale applications [30].

This issue was initially discovered in the Silk study by Balmau et al.. The research highlighted the occurrence of latency spikes in key-value storage systems such as RocksDB. While these storage systems are designed to offer high write throughput, they can experience latency spikes during read operations caused by the compaction process. Compaction involves merging smaller data files into larger ones, which can introduce long pauses, ultimately affecting read latency and throughput.

In order to investigate this problem, we performed a test using the RocksDB's *db_bench* benchmark with 8 client threads executing a combination of read and write requests in a closed-loop manner, following the YCSB workload [14]. As for RocksDB, it was configured with 8 background threads, where 1 thread was dedicated to handling flushes and 7 threads were responsible for managing compactions.

Since this problem focuses on data-related operations, our solution was configured to collect only the following system calls: *open*, *openat*, *creat*, *read*, *pread*, *readv*, *write*, *pwrite*, and *close*. This selective collection ensures that fewer events are generated, reducing the analysis time and preventing unwanted events from overwhelming the Elasticsearch database.

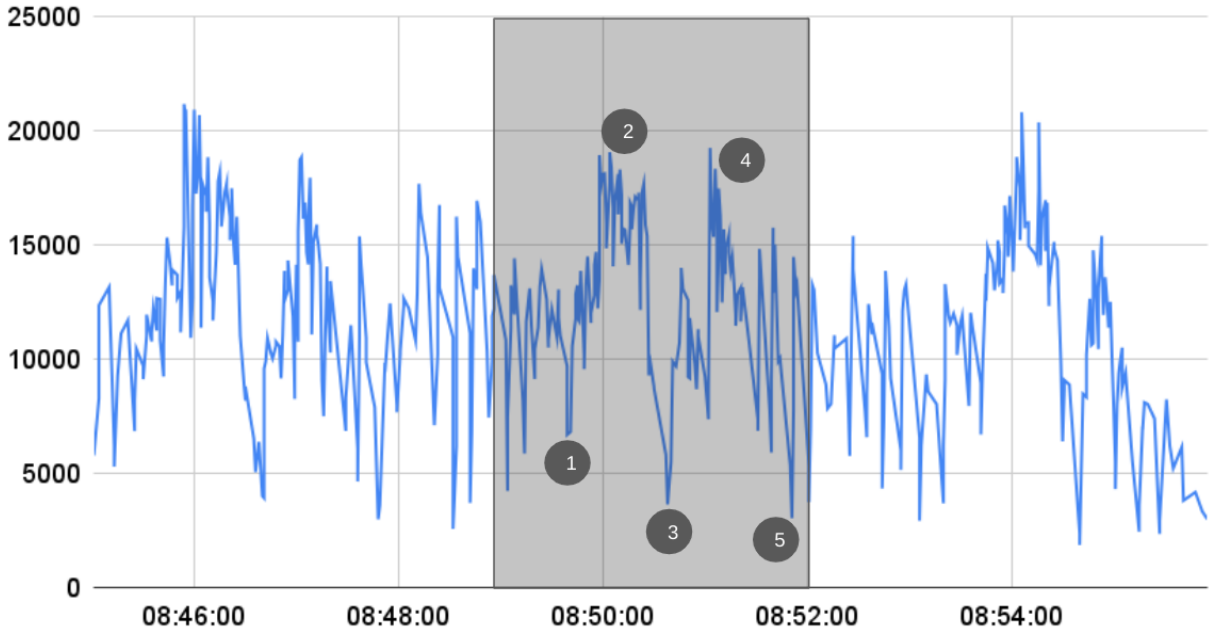


Figure 10: Average of latency on clients requests over 10 minutes of the execution.

Figure 10 represents the latency of client requests made to RocksDB. This visualization was extracted from benchmark data and not from Kibana visualizations. By observing it, we can observe a pattern where multiple latency spikes are identified in the operations performed by RocksDB clients. It can be concluded that the latency of these operations ranged between 5 000 and 25 000 milliseconds.

After analyzing the issue in our visualization component, we observed that over 40 million events were generated. In order to facilitate the analysis of the reported problem, we selected the time interval between 08:49 and 08:52, where we could easily identify latency spikes from the previous visual representation (Figure 10). Additionally, a visualization was created in Kibana to identify the number of events executed over time, grouped by thread group.

In this case, there are three different types of thread groups: *db_bench*, *rocksdb:high*, and *rocksdb:low*. The *db_bench* thread group represents the operations performed by the clients. The *rocksdb:high* thread group handles high-priority tasks such as writes and critical operations, while the *rocksdb:low* thread group manages lower-priority background tasks. This division ensures a balanced and efficient execution of different operations within the RocksDB database engine.

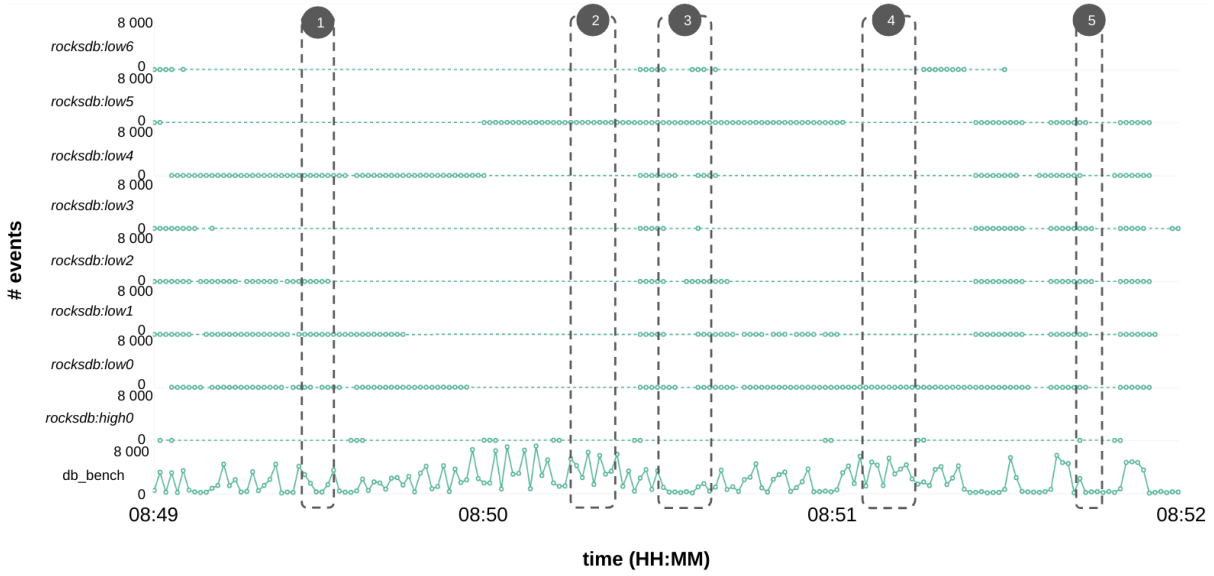


Figure 11: Number of system calls executed over time grouped by thread groups.

By observing Figure 11, we can identify the specific time periods in which the latency spikes occurred, as mentioned and shown earlier. Additionally, we can identify a pattern that explains these spikes. The moments with higher latency correspond to instances when the lower-priority threads are under heavier load and performing more I/O operations (moments 1, 3 and 5). This results in a decrease in the number of system calls executed by the *db_bench* threads, which are responsible for client requests, leading to increased latency for those requests. On the other hand, when the number of events from the *rocksdb:low* threads is lower, the number of events from the *db_bench* threads increases, resulting in faster processing of requests and reduced latency observed by the clients (moments 2 and 4).

The observed phenomenon can be easily understood in the context of RocksDB's operation principles.

As previously mentioned, RocksDB employs three distinct thread types: *db_bench* for processing client requests, *rocksdb:high* for high-priority tasks like flushes and *rocksdb:low* for low-priority tasks such as compactions.

Flushes play a vital role in ensuring the durability of data stored in memory-based structures. Initially, when data is written to RocksDB, it is stored in memory for faster access. However, for long-term storage and reliability, the data needs to be flushed to disk. During a flush, the data is sequentially written to disk, freeing up memory for new write operations and maintaining data consistency. This process creates a balance between performance and durability in data storage [22].

Compactions, on the other hand, are essential background operations that optimize data storage and enhance performance. As data accumulates in RocksDB, it can become fragmented and inefficient to access. Compactions address this issue by merging and reorganizing data files, reducing storage space requirements, and improving read and write efficiency. Different types of compactions, such as level-based and universal compactions, are designed to handle specific scenarios. During a compaction, overlapping key-value pairs are identified, obsolete data is discarded, and data files are compacted into a more efficient format. This process ensures optimal data organization, reduces disk space usage, and mitigates write amplification. By regularly performing compactions, RocksDB maintains data organization, enhancing overall performance and storage efficiency [61].

However, when a compaction is triggered, it requires significant disk I/O and CPU resources to merge and rearrange data files. This can lead to increased contention for shared resources, causing delays in processing client requests. As compactions consume disk bandwidth and CPU, the availability of these resources for handling client operations decreases. Consequently, client requests may experience higher latency as they wait for their turn to be processed, particularly during periods of heavy compaction activity. Furthermore, compactions can introduce I/O amplification, where the read and write amplification caused by compaction workloads further intensifies latency issues. Therefore, the interaction between compaction threads and client operations, especially in scenarios with high compaction workloads or limited system resources, can result in latency spikes in client requests.

4.3.3 Discussion

After executing the presented use cases, we can conclude that through our solution, we gained a better understanding of the execution of a real-world program like Elasticsearch, enabling us to identify various important metrics for its comprehension. Additionally, we were able to study in more detail the access patterns of a specific file. Upon analysis, we discovered that unnecessary operations were being performed,

which could lead to performance issues under high request loads.

Using our solution, we were also able to demonstrate that it is possible to identify the cause of a previously reported performance issue by studying the problem of latency spikes in RocksDB. In this way, we were able to observe the problem and identify its root cause without the need for code analysis. Identifying this problem through code analysis would involve comprehending and analyzing over 440K lines of RocksDB code. Moreover, relying solely on a tracing tool would require manually examining more than 40 million events. Therefore, our solution greatly facilitated the identification and analysis of the root causes behind the reported latency issues.

Therefore, we have shown that our solution not only helps in understanding a specific execution but also enables the identification of potential future issues in cases of higher intensity. Additionally, it allows us to pinpoint the root cause of these problems.

Chapter 5

Conclusion

The technology is constantly evolving, and the complexity and amount of data have grown exponentially. Consequently, the applications and systems we rely on in our daily lives are becoming larger and more complex. At the same time, they are becoming increasingly critical, and it is imperative to ensure their reliability. In this sense, it is necessary to understand and monitor the operation of these applications in order to detect and promptly address potential performance and reliability issues.

Thus, in this thesis, we present a comprehensive study of the state of the art and existing technologies that facilitate the understanding and analysis of these applications. After this study, we concluded that it was necessary not only to present specific metrics and data about the execution of a particular application but also to provide automated analysis tools to facilitate the reading and comprehension of the collected data from a specific execution.

We concluded, therefore, that in order to facilitate the analysis of an application, it was necessary to provide a tool that could collect, interpret and present informative data about the execution of the application in question. At the same time, it was important to meet criteria such as not requiring visibility into the application's code and not impacting the normal operation of the application. It was also a goal of this thesis to explore LTTng as the data collection tool (*i.e.*, tracing tool).

Therefore, in this thesis, we present a new tracing and analysis pipeline that uses LTTng as a tracing tool. It is capable of collecting, interpreting, correlating traced data and sending it to Elasticsearch, enabling analysis and the creation of visualization dashboards in Kibana, automating the analysis process.

In order to validate that the solution presented in this thesis fulfills the mentioned objectives, such as being non-intrusive, have minimal impact and allow near real-time analysis, performance tests and utility tests were conducted. In the performance tests, we demonstrated that enabling tracing with LTTng does not cause significant impact on the application's execution. On the other hand, when adding the parsing phase, some impact on the application's performance was observed. However, the parser and the rest of the pipeline can be executed on a separate server due to the live mode of LTTng, which ensures that

our solution does not cause significant impacts on the application's execution. Additionally, through the utility tests on real applications, we demonstrated that our solution not only facilitates the understanding of the application but also enables the prediction of potential issues and the identification of their possible causes. All of this without the need for code instrumentation at of the application.

5.1 Future work

In this section, we highlight potential enhancements that can be made to the solution presented in this thesis, aiming to enrich its contributions. Throughout the development of this thesis, various ideas and curiosities emerged, leading us to explore other areas to further enhance our work. Therefore, the following ideas for improving the developed tool are presented, which would undoubtedly make it more valuable and better equipped to achieve its goal of assisting programmers and system administrators in understanding applications, anticipating potential issues, and identifying their root causes:

Add file information to events. In general, system calls only receive the file path during the file opening. For subsequent file accesses, whether for reading or writing, the file descriptor is used as the argument to identify the file. This means that it is not directly possible to associate a read or write event with a specific file. Additional manual analysis work is required, as done in the Elasticsearch utility test, to identify the file name corresponding to the file descriptor received by the system call at that moment. Therefore, it would be important to establish this relationship in the parsing phase, namely in the plugin developed for Babeltrace2, where file path information could be added to the read/write event.

Capturing content from references. LTTng captures all the arguments used in a system call invocation. However, many system calls receive memory addresses referring to structures/buffers as arguments. Although LTTng captures some of these references, such as the path of the file being opened, it does not capture most of them. To provide more information to users, it is possible to capture additional references passed to certain system calls. For example, it would enable observing the content written/read to a file by reading the buffer or collecting network information passed as a reference to a structure in some system calls. Currently, this functionality needs to be implemented manually, requiring the creation of a custom probe for these system calls.

Add artificial intelligence to detect problematic patterns. In order to further automate the analysis process, it would be interesting to add artificial intelligence capable of automatically detecting undesired I/O patterns in the application's execution, without the need for human detection.

Bibliography

[1] Trace compass. https://www.eclipse.org/tracecompass/. [Online; accessed 7-January-2023].

[2] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. Re-animator: Versatile high-fidelity storage-system tracing and replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR '20, pages 61–74, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375887. doi: 10.1145/3383669.3398276. URL https://doi.org/10.1145/3383669.3398276.

[3] Eric Anderson, Martin Arlitt, Charles B. Morrey, and Alistair Veitch. Dataseries: An efficient, flexible data format for structured serial data. *SIGOPS Oper. Syst. Rev.*, 43(1):70–75, jan 2009. ISSN 0163-5980. doi: 10.1145/1496909.1496923. URL https://doi.org/10.1145/1496909.1496923.

[4] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[5] Tim Arnold. Performance counters for linux. In *Linux Symposium*, 2006. URL https://www.kernel.org/doc/ols/2006/ols2006v1-pages-225-230.pdf.

[6] Babeltrace. babeltrace2(1) [v2.0]. https://babeltrace.org/docs/v2.0/man1/babeltrace2.1/. Accessed: January, 2022.

[7] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. *ACM Trans. Comput. Syst.*, 36(4), may 2020. ISSN 0734-2071. doi: 10.1145/3380905. URL https://doi.org/10.1145/3380905.

[8] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11,

page 9–16, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308496. doi: 10.1145/2024569.2024572. URL https://doi.org/10.1145/2024569.2024572.

[9] Jan Blunck, Mathieu Desnoyers, and Pierre-Marc Fournier. Userspace application tracing with markers and tracepoints. In *Proceedings of the 16nd Linux Kongress*, Linux Kongress '09, 2009.

[10] Elasticsearch B.V. Rally: Macrobenchmarking framework for elasticsearch. https://esrally.readthedocs.io/en/stable/install.html, 2023.

[11] Juan Cespedes and Petr Machata. ltrace(1) — linux manual page. https://man7.org/linux/man-pages/man1/ltrace.1.html.

[12] Peter M Chen and David A Patterson. Storage performance-metrics and benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, 1993.

[13] Jim S Tiller CISSP and Bryan D Fish CISSP. Packet sniffers and network monitors. *Information systems security*, 2006.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[15] Houssem Daoud and Michel R. Dagenais. Recovering disk storage metrics from low-level trace events. *Software: Practice and Experience*, 48(5):1019–1041, 2018. doi: https://doi.org/10.1002/spe.2566. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2566.

[16] Houssem Daoud and Michel R. Dagenais. Performance analysis of distributed storage clusters based on kernel and userspace traces. *Software: Practice and Experience*, 51(1):5–24, 2021. doi: https://doi.org/10.1002/spe.2889. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2889.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[18] Mathieu Desnoyers. Using the linux kernel tracepoints. https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[19] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

[20] Mathieu Desnoyers and Michel Dagenais. Low disturbance embedded system tracing with linux trace toolkit next generation. In *ELC (Embedded Linux Conference)*, volume 2006. Citeseer, 2006.

[21] DIAMON Research Group. Ctf (common trace format). https://diamon.org/ctf/. Accessed on 16th May 2023.

[22] Siying Dong, Mark D. Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *Conference on Innovative Data Systems Research*, 2017.

[23] eBPF. ebpf - introduction, tutorials & community resources. https://ebpf.io/. Accessed: Januray, 2022.

[24] Elastic. Elasticsearch: The heart of the free and open elastic stack. https://www.elastic.co/elasticsearch/, . Accessed: October, 2021.

[25] Elastic. Elk stack - elasticsearch and kibana. https://www.elastic.co, . Accessed: December, 2021.

[26] Elastic. Kibana: Your window into the elastic stack. https://www.elastic.co/kibana/, . Accessed: October, 2021.

[27] Elasticsearch contributors. Elasticsearch v8.3.0. URL https://github.com/elastic/elasticsearch/tree/v8.3.0.

[28] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. Cat: Content-aware tracing and analysis for distributed systems. In *Proceedings of the 22nd International Middleware Conference*, (Middleware '21), 2021.

[29] Tânia Esteves, Ricardo Macedo, Rui Oliveira, and João Paulo. Diagnosing applications' i/o behavior through system call observability. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023.

[30] Facebook. Rocksdb: A persistent key-value store for fast storage environments. https://rocksdb.org, 2023.

[31] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128, 1996. doi: 10.1109/SECPRI.1996.502675.

[32] Mike Frysinger. Function tracer design. https://www.kernel.org/doc/html/latest/trace/ftrace-design.html.

[33] Mohamad Gebai and Michel R. Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2), mar 2018. ISSN 0360-0300. doi: 10.1145/3158644. URL https://doi.org/10.1145/3158644.

[34] Google. gperftools. https://github.com/gperftools/gperftools, Accessed 2023.

[35] Diana Gudu, M. Hardt, and Achim Streit. Evaluating the performance and scalability of the ceph distributed storage system. *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, pages 177–182, 01 2015. doi: 10.1109/BigData.2014.7004229.

[36] Ceki Gülcü. *The complete log4j manual*. QOS. ch, 2003.

[37] Red Hat. Chapter 8. strace. Red Hat Customer Portal.

[38] *Intel VTune Profiler User Guide*. Intel Corporation, 2021 edition, 12 2021.

[39] International Data Corporation. Worldwide global datasphere and global storagesphere forecast, 2022-2026. 2022.

[40] Madeline Janecek, Naser Ezzati-Jivan, and Abdelwahab Hamou-Lhadj. Performance anomaly detection through sequence alignment of system-level traces. 2022.

[41] Michael K. Johnson and Erik W. Troan. *Linux Application Development*. Addison-Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201308215.

[42] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, mar 1987. ISSN 0734-2071. doi: 10.1145/13677.22723. URL https://doi.org/10.1145/13677.22723.

[43] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel probes (kprobes). https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[44] Michael Kerrisk. ptrace(2) — linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html, .

[45] Michael Kerrisk. strace(1) — linux manual page. https://man7.org/linux/man-pages/man1/strace.1.html, .

[46] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, USA, 1st edition, 2010. ISBN 1593272200.

[47] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. Introperf: transparent context-sensitive multi-layer performance inference using system stack traces. *ACM SIGMETRICS Performance Evaluation Review*, 42:235–247, 06 2014. doi: 10.1145/2637364.2592008.

[48] Iman Kohyarnejadfard, Mahsa Shakeri, and Daniel Aloise. System performance anomaly detection using tracing data analysis. In *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, ICCTA 2019, page 169–173, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371810. doi: 10.1145/3323933.3324085. URL https://doi.org/10.1145/3323933.3324085.

[49] Iman Kohyarnejadfard, Daniel Aloise, Michel R. Dagenais, and Mahsa Shakeri. A framework for detecting system performance anomalies using tracing data analysis. *Entropy*, 23(8):1011, Aug 2021. ISSN 1099-4300. doi: 10.3390/e23081011. URL http://dx.doi.org/10.3390/e23081011.

[50] R. Krishnakumar. Kernel korner: Kprobes-a kernel debugger. *Linux J.*, 2005(133):11, may 2005. ISSN 1075-3583.

[51] Mahesh Lal. *Neo4j Graph Data Modeling*. Packt Publishing, 2015. ISBN 1784393444.

[52] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563.

[53] LTTng. Lttng: an open source tracing framework for linux. https://lttng.org/. Accessed: October, 2021.

[54] Surekha mariam varghese and K. Jacob. Anomaly detection using system call sequence sets. *JSW*, 2:14–21, 01 2007. doi: 10.4304/jsw.6.1.14-21.

[55] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. Probing the guts of kprobes. In *Linux Symposium*, volume 6, page 5, 2006.

[56] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[57] Alexandre Montplaisir, Naser Ezzati-Jivan, Florian Wininger, and Michel Dagenais. Efficient model to query and visualize the system states extracted from trace data. pages 219–234, 09 2013. ISBN 978-3-642-40786-4. doi: 10.1007/978-3-642-40787-1_13.

[58] Francisco Neves, Nuno Machado, and José Pereira. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 534–541, 2018. doi: 10.1109/DSN.2018.00061.

[59] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. Horus: Non-intrusive causal analysis of distributed systems logs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 212–223, 2021. doi: 10.1109/DSN48987.2021.00035.

[60] Henrik Nordahl and Adrian Wells. Testing the metacognitive model against the benchmark cbt model of social anxiety disorder: Is it time to move beyond cognition? *PL o S One*, 12(5), May 2017. ISSN 1932-6203. doi: 10.1371/journal.pone.0177109.

[61] Keren Ouaknine, Oran Agra, and Zvika Guz. Optimization of rocksdb for redis on flash. In *Proceedings of the International Conference on Compute and Data Analysis*, ICCDA '17, page 155–161, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352413. doi: 10.1145/3093241.3093278. URL https://doi.org/10.1145/3093241.3093278.

[62] Steven Rostedt. ftrace: Linux kernel tracing. https://www.kernel.org/doc/Documentation/trace/ftrace.txt, .

[63] Steven Rostedt. Lockless ring buffer design. https://www.kernel.org/doc/html/latest/trace/ring-buffer-design.html, .

[64] Steven Rostedt. Finding origins of latencies using ftrace. 01 2009.

[65] Martin Schulz. Extracting critical path graphs from mpi applications. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005. doi: 10.1109/CLUSTR.2005.347035.

[66] Vijay Shrivastava. A study on the crash of boeing 737 max. *International Journal of Science and Research (IJSR)*, 9:411, 08 2020. doi: 10.21275/SR20805210709.

[67] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, Jan 2016.

[68] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 3–14, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933190. doi: 10.1145/1140277.1140280. URL https://doi.org/10.1145/1140277.1140280.

[69] Dominique Toupin. Using tracing to diagnose or monitor systems. *IEEE Software*, 28(1):87–91, 2011. doi: 10.1109/MS.2011.20.

[70] Matthias Weber, Ronny Brendel, and Holger Brunst. Trace file comparison with a hierarchical sequence alignment algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 247–254. IEEE, 2012.

[71] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association. ISBN 1931971471.

[72] Dag Wieers. Dstat: Versatile resource statistics tool. Website, Accessed 2023. URL http://dag.wieers.com/home-made/dstat/.

[73] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. Depgraph: A dependency-driven accelerator for efficient iterative graph processing. In *2021 IEEE International*

Symposium on High-Performance Computer Architecture (HPCA), pages 371–384, 2021. doi: 10.1109/HPCA51647.2021.00039.

[74] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. Wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 527–543, USA, 2018. USENIX Association. ISBN 9781931971478.