Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 8478076, 17 pages
https://doi.org/10.1155/2021/8478076

Research Article

Debugging of Performance Degradation in Distributed Requests Handling Using Multilevel Trace Analysis

Naser Ezzati-Jivan,[1] **Houssem Daoud**,[2] **and Michel R. Dagenais**[2]

[1]*Brock University, St. Catharines, Ontario, Canada L2S 3A1*
[2]*Polytechnique Montreal, Montreal, Quebec, Canada H3T 1J4*

Correspondence should be addressed to Naser Ezzati-Jivan; nezzati@brocku.ca

Received 6 May 2021; Revised 29 June 2021; Accepted 16 October 2021; Published 16 November 2021

Academic Editor: Yong Zhang

Root cause identification of performance degradation within distributed systems is often a difficult and time-consuming task, yet it is crucial for maintaining high performance. In this paper, we present an execution trace-driven solution that reduces the efforts required to investigate, debug, and solve performance problems found in multinode distributed systems. The proposed approach employs a unified analysis method to represent trace data collected from the user-space level to the hardware level of involved nodes, allowing for efficient and effective root cause analysis. This solution works by extracting performance metrics and state information from trace data collected at user-space, kernel, and network levels. The multisource trace data is then synchronized and structured in a multidimensional data store, which is designed specifically for this kind of data. A posteriori analysis using a top-down approach is then used to investigate performance problems and detect their root causes. In this paper, we apply this generic framework to analyze trace data collected from the execution of the web server, database server, and application servers in a distributed LAMP (Linux, Apache, MySQL, and PHP) Stack. Using industrial level use cases, we show that the proposed approach is capable of investigating the root cause of performance issues, addressing unusual latency, and improving base latency by 70%. This is achieved with minimal tracing overhead that does not significantly impact performance, as well as $O(\log n)$ query response times for efficient analysis.

1. Introduction

When performance degradation occurs within a distributed system, it can have multiple causes. For instance, it may be caused by insufficient system resources, a problem in the network layer, a bug in the software code within a connecting node, incorrect input data, or the misconfiguration of one of the active modules or nodes. The situation gets more serious when we notice that it is not easy to locate the problem in the system, as it is running continuously in parallel with other software and machines that cannot be stopped for debugging.

Therefore, most issues within distributed systems are hard to recognize and investigate. Active monitoring of distributed system execution using runtime information can be helpful in this matter [1–3]. The runtime execution data, which is usually collected by logging and tracing tools, can help monitor the actual executions of systems, detect possible runtime problems, and hopefully pinpoint their root causes. Tracing is a method that consists of collecting execution logs from a system at runtime [4]. Unlike profiling, which usually provides statistics about a time range, tracing can display the state of the system at various levels, including the active processes, running system calls, function call stack, network usages, and active elements of disk queues at different time points, e.g., when a latency problem is detected in the system [5].

There are already trace-based solutions to debug performance problems of distributed systems [5–7], however, extracting the root cause of a performance problem requires much more depth than these solutions can provide since they mainly rely on a single level of data, either kernel,

user-space, or network level [8]. Conversely, we present a trace-oriented solution that monitors the system in several layers (from the application layer to the driver layer), to improve visibility. The proposed solution uses trace data gathered from Linux Trace Toolkit Next Generation (LTTng), a low-overhead kernel and user-space tracer [4]. The traces collected from the different components of the distributed system are synchronized together based on an event matching algorithm so that all events are related to a unified clock. Our approach is generic and can be used to debug different distributed system issues, from problems in web servers to network file systems. However, our focus in this paper is mostly on web servers (distributed or centralized) and more specifically the Linux, Apache, MySQL, and PHP (LAMP) Stack. We instrument the different LAMP stack modules and provide tracing probes to collect runtime information from the web server, application server(s), database server(s), and from the client(s).

The collected trace is analyzed using a stateful top-down approach using a historical data-store that is built while the trace is first read. This data-store keeps track of the state of the system throughout execution and includes metrics from the various system layers which it measures, aggregates, and organizes. Its purpose is to give users an overview of the system and to help pinpoint any bottlenecks or performance misbehaviors. This data-store uses a tree structure with search complexity of $O(\log\ n)$ which is efficient for recovering the desired analysis data, without having to reread the substantial trace events every time the user selects a new area or system resource (e.g., a CPU, a process, or a virtual machine) to analyze. The database can be used to compare metric values from different points of execution to determine if there is misbehavior (e.g., an unexpectedly high response time or resource overload) at any time during execution. Upon discovery of problematic symptoms in the first overview step, the next analysis phase is to dig deeper into the problem using the information gathered and analyzed from the other layers, to investigate and detect the root cause of the problem.

The main contributions of the paper are

(i) A generic, top-down, and multi-level analysis solution to detect and locate performance bottlenecks of distributed systems

(ii) Using an optimized data structure to store the state of the system for the trace duration, which makes the analysis much more interactive afterward

(iii) A unified data collection approach to trace the different components of the LAMP stack (Linux, Apache (web server), MySQL (database server), and PHP (application server)) to efficiently collect the required run time execution data for a whole-chain web request analysis

The remainder of the paper is organized as follows: we discuss the related work of distributed system and web server performance analysis in Section 2, followed by the architecture of our performance analysis method in Section 3. In Section 4, we highlight the usefulness and functionality of our work through some real-world use-cases. We evaluate the overhead of our proposed method in Section 5 and conclude with a look at some interesting future work.

2. Related Work

A distributed system is a system whose processes and threads are running on multiple computers with different configurations. The possible incompatibility in the runtime environments makes the debugging of performance problems more difficult. Runtime execution data, collected by tracing tools, can help monitor the system execution, detecting performance problems, and uncovering their root causes.

Tracing is a method that consists of collecting execution logs from a system at runtime [9, 10]. The payload of a trace event usually contains the event name, the ID of the processor on which the event is executed, the timestamp, and the arguments. A trace event can be a system call, a function call, a signal, an IRQ, etc. Unlike debugging, where the program is executed step by step to get its current state, tracing collects data during the execution, and the trace file is often analyzed offline. The overhead of tracing should be minimal to preserve the normal behavior of the system.

Linux Trace Toolkit Next Generation (LTTng) [4] is an open-source Linux tracing tool initially developed by DORSAL (Distributed Open Reliable Systems Analysis Lab (DORSAL) http://www.dorsal.polymtl.ca) to provide very low-overhead tracing capabilities. It is packaged as an out-of-tree kernel module, and it is available in all major Linux distributions. LTTng supports kernel and user-space tracing, which is useful to correlate high-level application events with low-level kernel events to support a multilevel trace-based analysis. Execution traces can be used to study the runtime behavior of software applications. Daoud and Dagenais [11] proposed a trace-based framework to provide detailed workload characterizations of Java virtual machines. Surveys of trace-based dynamic analyses methods including different applications of multilevel execution traces in software analyses are presented in [10, 12].

Some previous works have applied kernel tracing to aid in distributed system performance monitoring. For instance, in [13], vectors representing system call sequences are created and used along with machine learning classification algorithms to identify anomalies. This work demonstrates how valuable kernel tracing can be in the identification of CPU or memory-related problems. Furthermore, in [14], Ates et al. utilize tracing methods to identify regions of high performance variability for the purposes of automatically enabling additional instrumentation. By including user-space and network level trace data, one can go beyond performance degradation identification and conduct precise root cause analysis within a distributed system.

Google Dapper [15], a tracing framework for distributed systems, works by instrumenting the RPC (remote procedure call) libraries to trace distributed requests. In this system, a unique identifier is assigned to each request at the entry point and is used to follow the request through the whole distributed system. This unique ID is used to

temporarily join and relate the components interacting to serve the request. At the visualization time, a request is drawn recursively in a trace tree timeline, including spans and arrows, where each span includes the basic unit of work and arrows display the communication and interactions between spans. While Dapper is an excellent tool to break down a request into all its components, it does not offer much information to analyze the problem's root causes.

Pinpoint [6] monitors call request paths and clusters them into success and failure groups, to find latency-anomalous components. However, the fault may come from the operating system and not the software components, in which case this tool will not offer interesting insight.

Spectroscope [7] is aimed at diagnosing performance changes by comparing request flows. It applies root cause analysis by comparing the way the system services requests from different normal and problematic periods. Spectroscope extracts critical paths of requests from the normal and problematic parts of the system execution and groups the similar paths into a cluster using k-mean algorithms. The clusters correspond to different types of requests and the critical paths within a cluster show different ways of handling the same type of requests. They use data from only a single layer, whereas our approach uses several.

TraceCompare [5] uses a similar idea but leaves the selection of normal and slow requests to users, where they can see all requests in side-by-side distributions and can choose any two areas to compare. TraceCompare [5] extracts critical paths of requests and converts them to an enhanced calling context tree (ECCT). It then extracts a differential flame graph from the various requests to compare the differences. By comparing two different requests, a user can get the detailed differences of their executions and possibly figure out the reasons for the latency of the slow request. Although it uses traces from several levels, it depends on comparing various executions, which is different from our proposal.

In this paper, we address the runtime issues which may occur in the different modules, nodes, or layers of distributed systems (i.e., LAMP stack including a web server, a database server, and an application server). We collect trace data from multiple levels (application, system call, disk block layer, and network layer). The data from each layer has its specificity and should be processed based on the characteristics of that layer. A comparison of using user-space and kernel-space trace data in software anomaly detection is presented in [16]. The separate analysis of different layers of kernel data is studied in [10], while the processing of system call traces is reviewed in [17]. Another important layer is the physical storage (disk) layer. The performance of disk operations may directly affect the performance of applications. Different applications, like web and database servers, usually provide a user-space caching mechanism to reduce disk accesses, but it is still impossible to hold all the required data in memory, and accesses to disk are necessary at some point. The storage subsystem itself is composed of different layers: the virtual file system (VFS), the file system (ext4, ext2, btrfs, etc.), the block layer, and the disk driver. A user-space application requests I/O operations through system calls, and these requests go down through the layers until they reach the disk drive where they get processed. Daoud and Dagenais [11] proposed a comprehensive Linux tool to recover high-level storage metrics from low-level trace events collected at different layers of the storage subsystem. Similar tools like Oracle ZFS Storage Software [18] and IBM XIV [19] are available in other operating systems.

Many storage debugging tools can be used to debug storage issues. The traditional tools like *iostat* and *iotop* provide information about disk activity, and the processes causing this activity, by parsing the kernel statistics available in the *proc* file system. This information is useful to monitor disk activity, but it cannot be used to debug difficult problems. Block-level tracing provides a more precise solution to debug disk performance. Tracers like *Blktrace* [20] provide low-level information about I/O requests and the behavior of the disk scheduler, but this information is usually very detailed and difficult to analyze manually. Visualization tools like *Seekwatcher* [21], IOprof [22], and BTT [23] can be used to read the trace files generated by *blktrace* and to generate storage metrics from them. However, the visibility of *blktrace* is limited to the block layer, and it does not cover the other layers of the storage subsystems.

All the above studies review different aspects of trace-based analysis. However, they lack a unified way to analyze multilevel multinode trace data, which we address. We propose a solution to collect execution data from different layers and different sources, process them, extract the required information, and place them in a common data store to use in a posteriori analysis phase. This analysis synchronizes the data and performs a global analysis based on the common features (i.e., clock time and process names) between the data at different layers and from different sources.

3. Architecture

A distributed system is composed of different computers interconnected through the network. To ensure a comprehensive analysis, it is important to collect traces at different levels: user-space level, kernel level, and network level. The collected information is then sent to an automated tool for unified analysis. The general architecture of the proposed solution is presented in Figure 1. The next section will provide a detailed description of each of those components.

3.1. Data Collection

3.1.1. User-Space Tracing. User-space tracing is a technique to collect program runtime data by probing the different points of application code. It can be used for anomaly and fault localization in software code. Developers can embed different probes (tracepoints) in the important parts of the software code to get execution logs at runtime when the execution reaches them. Then, by analyzing these logs, users can understand what is actually happening in the different parts of the software.

However, it is also possible to trace a user-space application without changing its source code. To do so, the core library is replaced with a wrapper library and the program

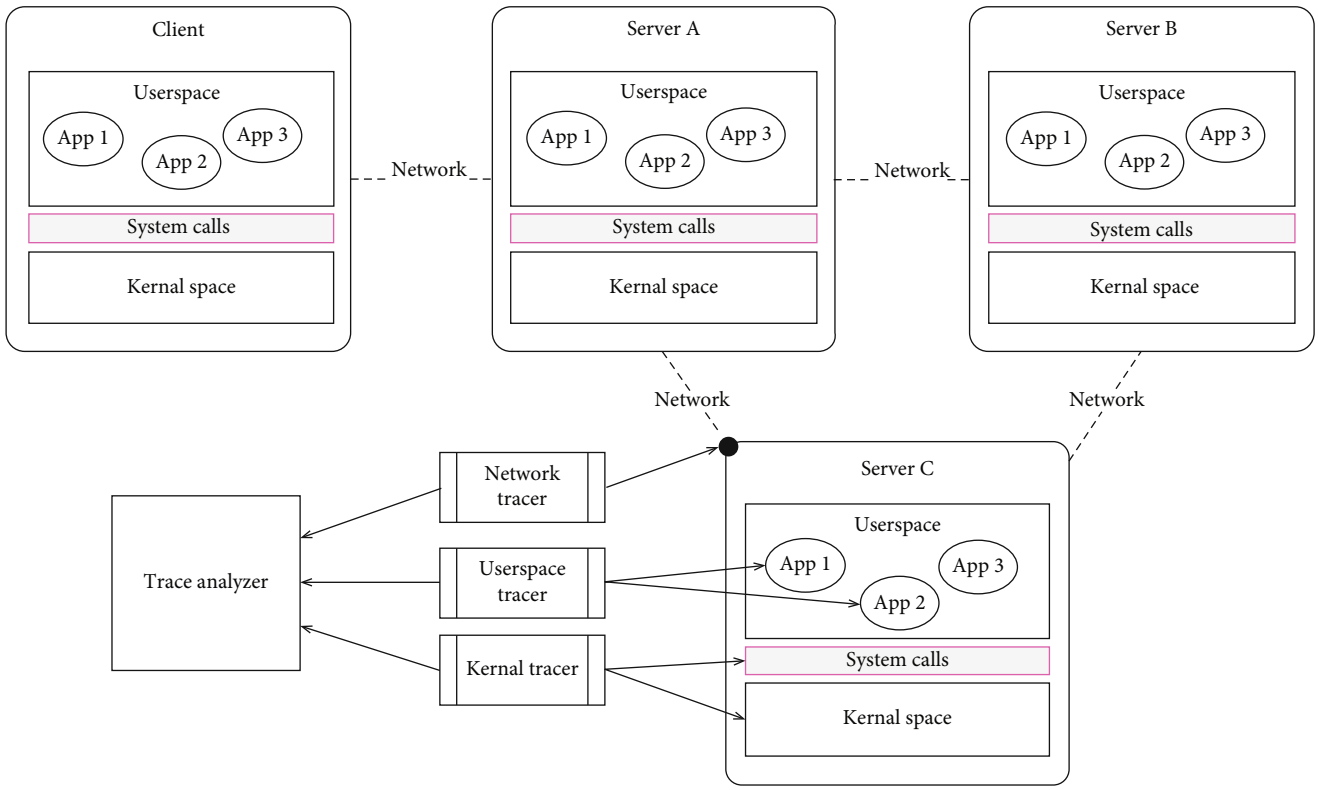

Figure 1: General architecture of the proposed multilevel analysis.

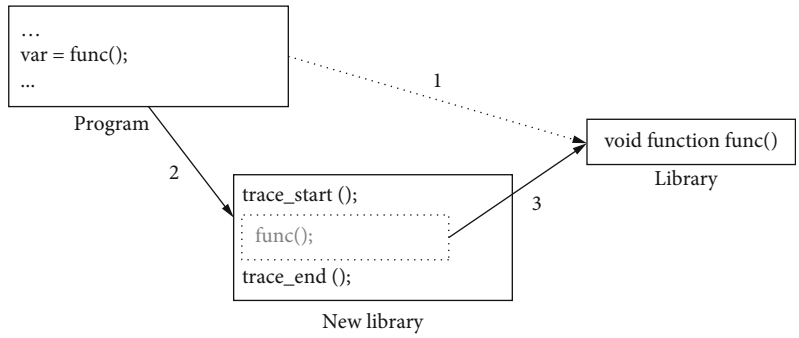

Figure 2: User-space tracing using the wrapping method.

calls the functions from the new wrapper library using LD_PRELOAD or other interception techniques. The wrapper library in turn calls the functions from the original library but includes some tracing before and after. Figure 2 shows this technique. Arrow number 1 shows the original path between the program and the library. Arrows 2 and 3 show the new calling path going through the wrapper library.

In the first phase of the technique proposed in this paper, the core of the LAMP stack (Linux, Apache, MySQL/MariaDB, and PHP) was changed to support tracing. A few tracepoints have been added to the libraries of the different modules to ensure that all important aspects of the LAMP stack are covered. Using this modified core of the LAMP stack, users can trace their web applications without needing to change their program source code.

For Apache, we have written a module to trace Apache requests. For each request, it records the start and end times, in addition to all the details about each request (e.g., client IP, file accessed, method, and user-agent).

In PHP, the core functions are changed to enable tracing at the request level, function call level, and code execution level. For each request, it records the start and end times and details such as client IP, files accessed, methods, and ports. For each function call, it records the filename, function name, class name, and status of the function execution. For each execution, it records the line number, file and function names, and execution status. These LAMP extensions (both the Apache module and the PHP extension) are open-source (https://github.com/naser/) and available for public use.

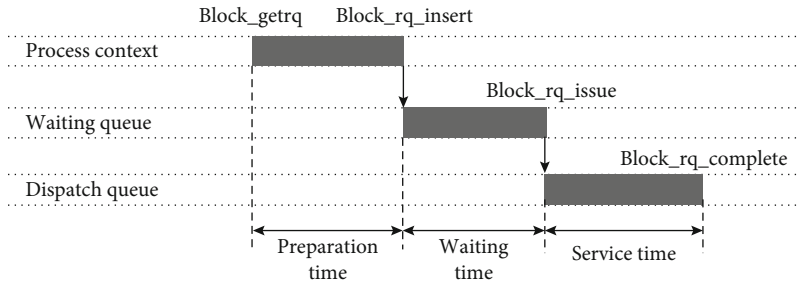

Figure 3: Request latency.

MariaDB (a fork of MySQL) is also instrumented to trace database connections, commands, and queries. Using these tracepoints, it is possible to gather precise runtime information about query execution, query response time, latency, or time elapsed in different parts of the query execution steps (e.g., query parsing, query caching, and query real execution).

What is important to note is that when the tracepoints have not been chosen for tracing or are disabled, they do not execute any tracing code, so they impose no additional overhead. When they are enabled, they may incur some very low overhead. We measure this additional overhead cost in Section 5. Using the mentioned tracepoints, we can debug the whole LAMP stack. For each request, we can give the exact time elapsed in each part of the stack and what fraction of the request time was spent in Apache, PHP, or the database. This means we can locate the potential bottlenecks of any request.

3.1.2. Kernel Tracing. System calls form a boundary between the user-space and the kernel-space. They are used by the applications to interact with the operating system and hardware resources. Tracing the system call layer provides useful information, which can be used to analyze the behavior of the system and to evaluate its performance. Detecting problematic calls is done by tracing the entry and exit of each system call and calculating its duration. In the context of our analysis, we need to monitor all system calls related to I/O operations (opening and creating files, write/read operations, reposition file offset, duplicate file descriptor, etc.).

System call events provide limited visibility about what is happening inside the storage subsystem. For example, a *READ* system call may return quickly because the data was found in the page cache, not because the disk offers good performance. To get a wider visibility, it is important to trace lower-level events that give more precise information about storage activity. The three important components to be considered in a deep storage performance analysis are the file system, the page cache, and the block layer.

The file system is the key component of the operating system responsible for storing and retrieving data from storage. It presents the data to the user-space applications in terms of files and internally keeps the match between each file and its physical location on disk. Typical file system operations are creating, reading, writing, and deleting a file. System calls are the entry points to the file system. For example, a read system call is directly linked to the *read* handler defined by the file system to *struct file_operations*. In addition to the basic operations, the file system uses many advanced mechanisms to ensure the coherence and security of the data. Some interesting file system events that can be used for a detailed analysis include inode creation and deletion, starting and processing writing operations, and when the file system journal starts to commit an operation.

The page cache is used to store data in the main memory, to minimize disk I/O operations. The file system always tries to recover the required data from the page cache before issuing a read request to the disk. The same happens for writing operations: the data is written to the page cache before being transferred to the disk asynchronously by the write-back mechanism. In Linux, the page cache is represented as a Radix Tree, since it provides a good search complexity. It is possible to evaluate the page cache lookup efficiency by tracing the entry and exit of *find_get_page*. However, in our analysis, we decided to generate a trace event only when a cache miss happens. The reasoning behind this decision was twofold. First, tracing all cache lookup operations adds too much overhead to the system. Second, the lookup time is insignificant compared to the cache miss handling time.

The block layer is the operating system component responsible for I/O request management. By instrumenting the block layer, we can get very precise information about I/O requests that are being processed, and how they are managed by the disk scheduler. We follow the I/O request and collect trace events from creation, through the waiting and dispatch queues, to completion by the disk drive.

The request latency can be broken into three main parts, as shown in Figure 3. The preparation time is the time needed to create the request data structure. The waiting time is the time during which the request resides in the waiting queue. The service time is the time taken by the disk drive to handle the request.

3.1.3. Kernel Modules Tracing. The Linux kernel offers many helper functions to facilitate driver development. For example, disk drivers typically use the exported functions of the block layer to manage requests and waiting queues. However, information may be needed from inside the driver and, for that, additional instrumentation is required. In the case of storage devices, it is important to know exactly when the interaction with the hardware happens. Four tracepoints are used for this:

(i) *scsi_dispatch_cmd_start* is executed when the request is sent to the disk controller

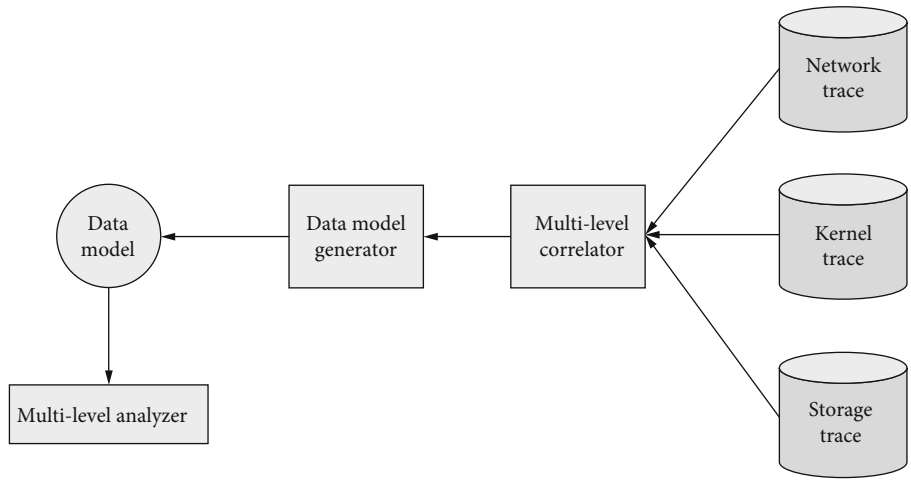

Figure 4: Data analyzer architecture.

(ii) *scsi_dispatch_cmd_error* is executed if the request delivery has failed

(iii) *scsi_dispatch_cmd_done* is executed if the command was handled correctly by the disk drive

(iv) *scsi_dispatch_cmd_timeout* is executed if the disk drive does not respond in time

3.1.4. Network Tracing. Network communication is an important part of a distributed system. Latency in the transmission of network packets has a direct impact on the performance of the system. In such cases, the application must be blocked until the required information is transmitted through the network. To get precise information about network latency, we based our analysis on many network-related system calls such as *connect()*, *accept()*, and *shutdown()*.

Network tracing is also used to synchronize traces between different machines. LTTng recovers event timestamps using the monotonic clock provided by the kernel of the traced machine. The absence of a global clock analysis in a distributed system is challenging and may even cause messages to seemingly be received before being issued (message inversion). The convex hull algorithm was proposed by Jabbarifar et al. [24] to solve this problem. The idea is to use event matching between different hosts to unify the clock sources. The couple {inet_sock_local_out, inet_sock_local_in} can be used for synchronization in the case of a distributed system because one event triggers the other and they share a common ID in their payload.

3.2. Data Analysis. Trace events usually contain very detailed information about the runtime behavior of the system, which is essential in the detection of hard problems. However, trace files are often very large, and pinpointing a problem inside them is not an easy task. An automated analysis system should be used to read the trace and generate higher-level, easier to understand information.

Data-driven abstraction is an abstraction method that consists of generating compound events by grouping low-level trace events. Ezzati-Jivan and Dagenais [10] proposed an approach that aims to summarize trace files based on a pattern library. The idea is to replace semantically-related trace events with a compound event like SEND_NETWORK_PACKET or WAIT_ON_MUTEX. This method improves the legibility of the trace but the oversimplification may hide important details, making the detection of some problems impossible. In the case of distributed systems, the problems are usually at a very low level; a deeper analysis is therefore required.

Metric-based abstraction is another approach that helps to identify problems without reducing the precision of the analysis. A metric is computed from one or many trace events, and a hierarchy of metrics can be defined so that a high-level metric is computed from lower-level metrics. Performance metrics are very useful for an efficient top-down approach. This technique is useful in our case: the troubleshooter can pinpoint the origin of the problem by looking at different metrics provided from the different layers. It starts by monitoring high-level metrics, and, when something unexpected happens, it goes down to see what happens at a deeper level of detail.

3.2.1. Proposed Architecture. In this section, we propose a generic architecture to process and analyze trace events provided from different layers. This architecture is illustrated in Figure 4.

3.2.2. Multilevel Correlator. It is the component responsible for getting the trace data from the different layers and synchronizing it. If the trace events are gathered from the same machine, the synchronization is based on the event timestamps. LTTng follows the monotonic time-base of the Linux kernel to reliably recover the exact time of each event [4].

The Linux kernel has two internal clocks: CLOCK_REALTIME and CLOCK_MONOTONIC. CLOCK_REALTIME (kernel function do_gettimeofday) tracks the wall clock and gets updated and corrected by the Network Time Protocol (NTP) daemon and is not monotonic. CLOCK_MONOTONIC represents the absolute elapsed wall-clock time since the machine boot time (which is an optional fixed prior time point). CLOCK_MONOTONIC, therefore, monitors more closely the hardware timers and does not jump in time. CLOCK_MONOTONIC is generally used for tracing, since it

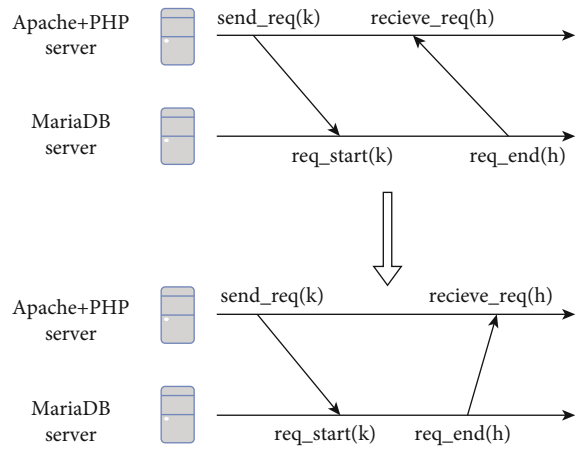

Figure 5: Trace synchronization in a distributed system.

is precise, has a fine granularity, and is most consistent across small and large time intervals. Moreover, the fact that it is monotonic avoids the ambiguity present for timestamps with CLOCK_REALTIME, when the clock is adjusted backwards.

LTTng follows the second approach, like CLOCK_MONOTONIC. However, to be safe from integer arithmetic imprecision, it directly reads and outputs the CPU timestamp counters which include the most accurate CPU calibration frequency and leaves the other integer arithmetic operations to the trace postprocessing analysis tools like Trace Compass or Babeltrace.

This time-base guarantees a partial ordering of events, even if the events are generated by different CPUs. The synchronization is more complex if the traces are collected from different machines. In this case, the causality between events is the only way to get a partial ordering, which is enough for the analysis. The fully incremental convex hull synchronization algorithm [24] is used to solve the problem of trace synchronization in distributed systems. The algorithm is based on matching an event a from one trace and an event b from the other. The couple {a,b} must satisfy the following requirement:

(i) a triggers b

(ii) a and b must have a common value in their payload

(iii) Every event b must have a corresponding event a in the other trace

For example, if PHP and MariaDB are installed on different machines, the traces can be synchronized using the matching events {send_req,req_start} and {req_end, receive_req}, as shown in Figure 5.

3.2.3. Data Model Generator. Detecting a performance bottleneck is a complex operation. To analyze the trace, users have to navigate horizontally by selecting different time regions and vertically by zooming in on regions of interest. Computing the metrics every time the user selects a new time range is a very slow operation because it requires rereading the same events again and again to generate the statistics.

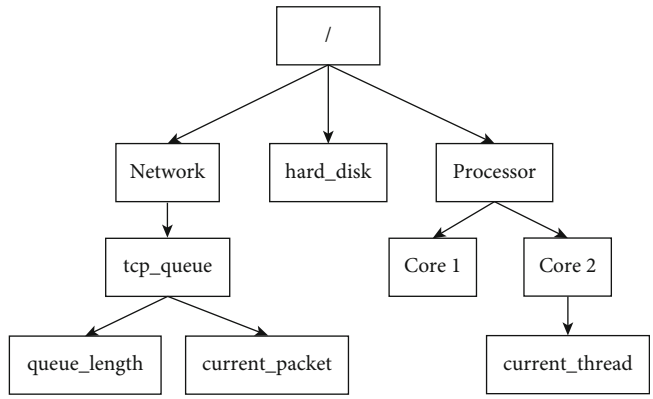

Figure 6: Attribute tree.

To avoid this extra computation, we decided to use a stateful approach in which the state of the system is kept in an incremental database when the trace is read for the first time. Every time a new time range is selected, the metrics can be quickly retrieved from the database, without reading the trace file.

Using a memory-based data-structure like Segment-tree or R-tree is not a viable solution, since trace files may be huge and the model generated cannot be kept in memory. On the other hand, disk-based solutions like relational databases are not tailored to tracing and provide a very slow response time in that context, as shown in Section 5.3.

Here, we use the modeled state system a custom-build database to keep track of the execution states of the different components of the system during its execution. The main components of the modeled state system are the attribute tree and the state history tree. The attribute tree represents the resource hierarchy of the system, and each node can be accessed using an absolute or relative path. A sample of the attribute tree used in our analysis is shown in Figure 6. For example, we can easily access the queue length of the TCP stack using the path/network/tcp_queue/queue_length.

The state history tree saves the state of the system on the disk in terms of nodes. A node is defined by a key, a time range, and a state, where the key is the path of the system component in the attribute tree. States are added continuously into a node until it is full and a new sibling is created, as shown in Figure 7. The state history tree offers a fast query time ($O(\log\ n)$ where n is the number of nodes), making it very convenient for an interactive analysis when a big trace is involved.

Formally, every event EV is a tuple like $(r1, \cdots, rn)$ which represents an interaction between a set of system resources, e.g., processors, files, disks, processes, and network sockets at a specific timestamp ti, and with one or more output values like $v_i \in N$.

$$\begin{cases} \mathrm{EV} = \{(t_i, r_1, \cdots, r_n, v_1, \cdots, v_m) | t_i \in T, r_j \in \mathrm{SR}, v_k \in N\}, \\ T\{t | t \in N\}, \\ \mathrm{SR} = \mathrm{SystemResources}. \end{cases} \quad (1)$$

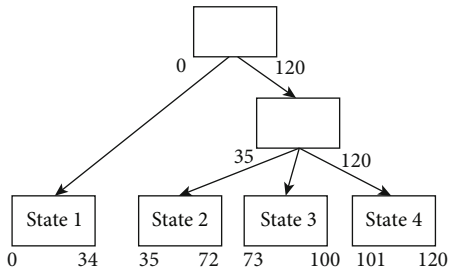

Figure 7: State history tree.

For example, an event like (t1, p1, read, fd1, cpu0, 100) shows that at time t1, process p1 reads 100 bytes from a file associated with the file descriptor fd1, while running on cpu0.

A trace TR may be seen as a set of ordered events. Events in a trace are ordered by their timestamp, and no two distinct events have the same time value:

$$\mathrm{TR} = \left\{ e_i \mid e_i \in \mathrm{EV}, i < j \longrightarrow t_{e_i} < t_{e_j} \right\}. \tag{2}$$

An abstract event, which we also call "State Change," is a high-level event that is built by grouping some raw level events. Each state has a duration, a key, and a value associated with that attribute for the given time duration. Abstract events (state changes) can be used to denote high-level concepts like an active network connection, a process blocked, a process running, and a CPU preempted. In summary, each state change is associated with time duration and contains a key and a value. For instance, the state change of "an active network connection" includes a key (i.e., the network connection socket id or the source and destination IP addresses), a value (i.e., active), and time duration (e.g., t1 to t2) during which the state value is valid for the key.

$$\begin{cases} \text{Time Duration TD} = \left\{ \left[t_i, t_j \right] \mid t_i, t_j \in N, t_i < t_j \right\}, \\ \mathrm{SV} = \left\{ (td_i, at_i, v_i) \mid td_i \in \mathrm{TD}, at_i \in \text{Attributes}, v_i \in N \right\}, \\ \text{State Database } SD = \left\{ sv_i \mid sv_i \in \mathrm{SV} \right\}. \end{cases} \tag{3}$$

As shown in the above equations, each state value sv_i includes a time range, an attribute, and a value. The attribute is a way to describe an aspect of a system resource. For example, an attribute fd shows a file descriptor (i.e., an aspect) of a file (i.e., a system resource). Other examples are the id of a process or a thread, the CPU core number, the address of a socket, the parent PID of a process, and so on.

Analysis AN is a function from the trace events TR to the state values database SD.

$$\text{Analysis AN} : \mathrm{TR} \longrightarrow \mathrm{SV}. \tag{4}$$

For instance, the analysis to extract the CPU utilization of each process $(\mathrm{CPU}A(p_i))$ may contain data like the following:

$$\begin{cases} \mathrm{TD} = \{ [ti, tj] \mid ti, tj \in N, ti < tj \}, \\ \mathrm{PR} = \{ p_i \mid p_i \in \text{System Processes} \}, \\ \mathrm{CU}_s = \{ (td_i, p_i, v_i) \mid tdi \in \mathrm{TD}, p_i \in \mathrm{PR}, v_i \in N \}, \\ \mathrm{CPU}A(p_i) \colon \mathrm{TR} \longrightarrow \mathrm{CU}_s. \end{cases} \tag{5}$$

Each analysis (like the analysis CPUA in the previous example) has a set of different mapping rules to convert events to state values. Depending on the analysis, the type of input events, and their effect on the state of the system attributes, the rules can take the form of a simple "if-then-else" or a complex transition pattern. The rules can be defined in the analysis tool source code, in the form of JAVA code, or can be specified dynamically using XML patterns. The possibility to define the analysis rules in XML form allows defining complex transitions from the trace events to states, to fulfill the requirements of specific use cases and problems.

$$\mathrm{RL} = \{ (e_i \longrightarrow sv_i) \mid e_i \subset \mathrm{TR}, sv_i \subset \mathrm{SD} \}, \tag{6}$$

$$\mathrm{AN} = \{ r \mid r \in \mathrm{RL} \}. \tag{7}$$

In the same way, we wrap all the analysis belonging to the same host into a container called a model. A model is a set of analyses, for a specific system, which can be used to reason about the underlying system from different points of view. A Model Cloud, in turn, is the superset of all existing models: a container for all analyses of all tracing systems (Figure 8).

$$\text{Model Cloud MDC} = \{ an_i \mid an_i \in \mathrm{AN} \}, \tag{8}$$

$$\begin{cases} \mathrm{AS} : \dfrac{\text{Attributes of a specific system}}{\text{host}}, \\ \mathrm{MDC} = \{ an_i \mid an_i \in \mathrm{AN}, A_a n_i \in \mathrm{AS} \}, \\ \text{Model MD}_s \subset \mathrm{MDC}. \end{cases} \tag{9}$$

Using the proposed model, users may aim to trace a distributed system, spanning different nodes (virtual machines or physical hosts). After tracing all machines, analyses for each are performed and a model for each is built. All such models together constitute a Model Cloud. The generated Model Cloud is then used in the analysis phase to render user-friendly views.

3.2.4. Multilevel Analyzer. As mentioned earlier, analyses for a model can be defined using JAVA code, recompiled into the tool before opening the trace, or can be dynamically defined using XML language patterns. The resulting model can be huge if many distributed nodes are involved, and opening all of them may slow down the trace analysis. Therefore, the analyses are partitioned into two main categories: base and subsidiary. The base analyses are those that are required to render the basic views (like control flow, histogram, etc.) and are executed when the trace is first opened. The subsidiary analyses, however, are not opened with the trace and are executed only upon a specific user request.

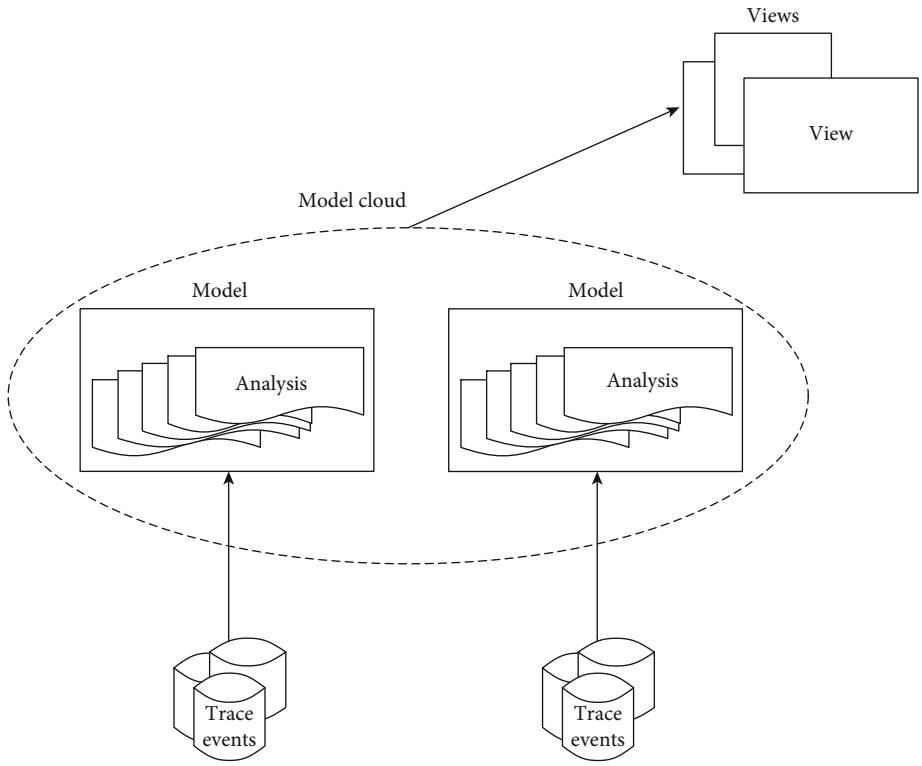

Figure 8: Trace events to analysis to model to Model Cloud.

This prevents the system from being slow when the trace is opened for the first time. Users can nonetheless see a list of the available analyses and can execute them whenever they want.

3.2.5. Generic Detection Algorithm. As explained above, the data collected from different sources is synchronized using timing and causality relationships with the multilevel correlator, and then modeled and saved in the State History database by the Data Model Generator. The multilevel analyzer recovers the data from the generated data model and uses an advanced algorithm to help the user detect performance bottlenecks. Bottlenecks can reside in any system layer. For example, in the case of a web server, the latency can reside in the web engine, the database, the network, the storage subsystem, etc. A good strategy is to start by analyzing the high-level layers and then go deeper if a problem is detected. The generic detection algorithm used by our tool is described in Algorithm 1 and shown graphically in Figure 9 to find latency problems in the context of a web server.

This algorithm is based on a top-down approach. It starts by computing high-level metrics from the user-space tracepoints. If an anomaly is detected, it is important to know if the problem resides in the application itself or if it is caused by the environment. An anomaly is detected if a web request takes longer than the normal distribution. We collect statistics about the number of requests, the latency, and the type of requests, and we use that as a baseline with which we compare. The moment we detect a request that took a longer time than normal, we consider it as abnormal and we use the proposed analysis for further investigation. To this end, the tool looks at the user-kernel boundary and checks if there is a specific system call that caused the latency. In such a case, we can use the kernel system calls to uncover the deeper reason for the latency.

In Figure 9, we describe in more detail how a problem is detected in the context of a web server. If a web request is slow (slower than a threshold), it can be because of the user-space logic (e.g., a long loop in the application code); in this case, no system calls are involved. However, the slowness may also be caused by a long system call, for example, a read syscall. In this case, the two main reasons are possible. The file is big and needs a lot of time to be read or the file is small, but there is disk contention, which makes it slow to read. To uncover the root cause, kernel events are needed, in addition to system calls. Figure 9 describes the decision process for this detection.

The procedure shown in Algorithm 1 covers system calls and other types of events at several levels of the execution. The term multilevel here means the different LAMP stack layers (Apache/PHP/MySQL), and also user-space/system calls/kernel. An operation can be slow because many computations have to be done on the CPU, or because it is reading a big file, or because it is waiting for a network packet, etc. The only way to know the reason for the latency is to have full visibility about what is happening in the system and to use the provided algorithm to narrow down the reason for the slowness.

The proposed method is implemented in Trace Compass [25] (an open-source trace viewing and analysis tool), under the name of Trace Compass Incubator and is available to the public [25].

Input:
User-space Tracepoints
System calls
Kernel Tracepoints
Events relationships map
Output: Problem
Compute high level metrics from user-space tracepoints
If user-space latency detected
  Find the system calls that caused the latency
  If system calls latency detected **then**
    Get the kernel events related to the problematic system calls
    If kernel anomaly detected
      Problem ⟵ *Kernel*
    Else
      Problem ⟵ *System calls*
    End
  Else
    Problem ⟵ *User − space*
  End
Else
  Problem ⟵ *NONE*
End

Algorithm 1: Generic detection algorithm.

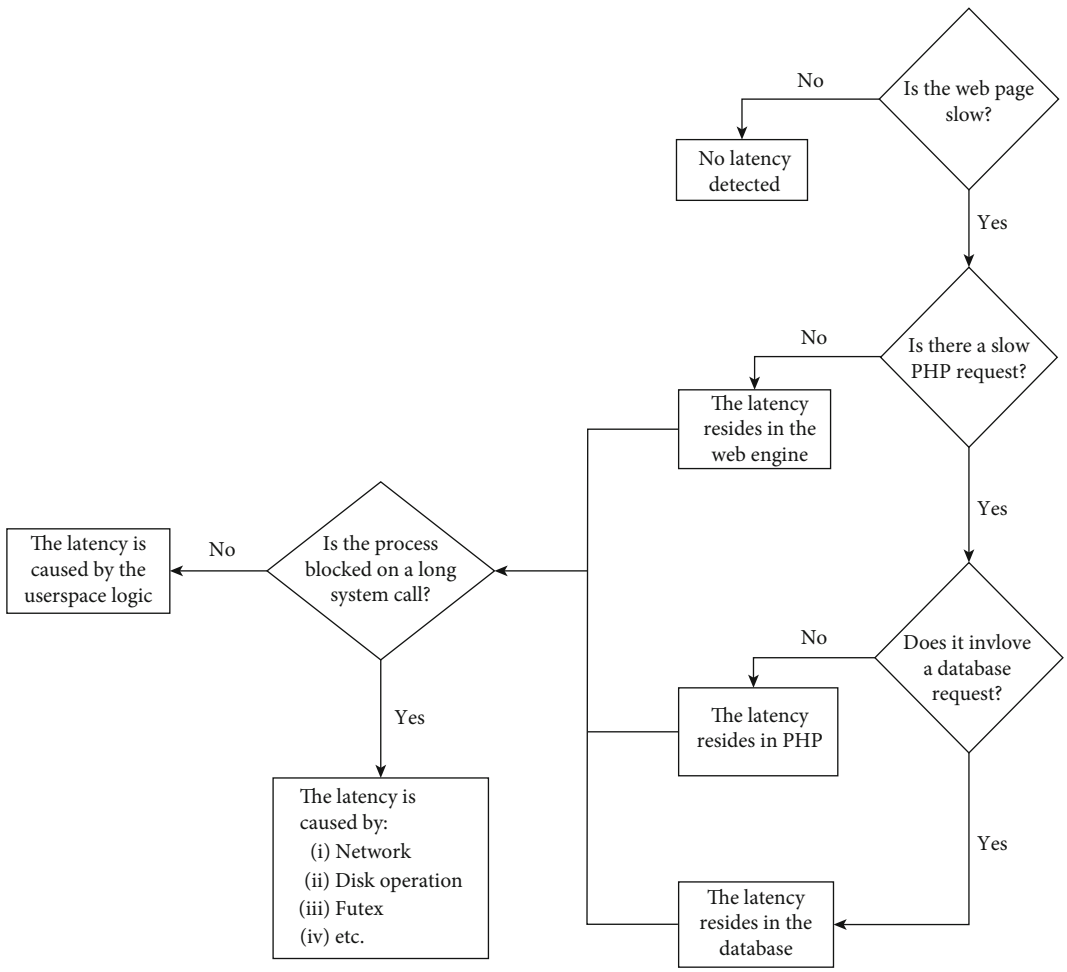

Figure 9: Decision diagram.

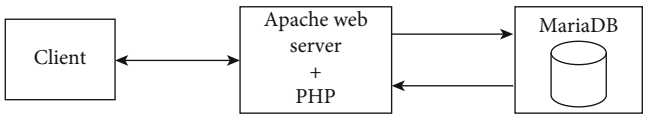

Figure 10: Web server architecture.

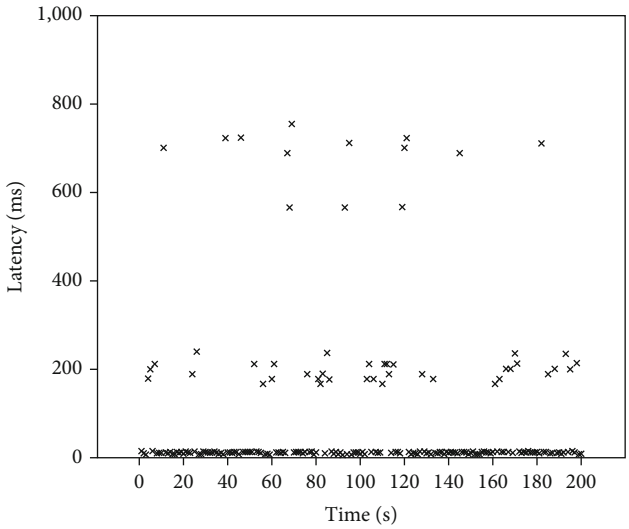

Figure 11: Web server latency.

4. Use Cases

The architecture we are proposing is generic and can detect performance problems in distributed systems, regardless of their precise architecture. It can be a distributed database, a computing cluster, a distributed file system, a distributed web server, etc. In this section, we present a detailed use case where the proposed solution was used to find some performance issues, difficult to detect using other monitoring tools.

The users of a web server complained about some long response times. The architecture of the server was as follows: a first machine runs Apache 2.4.23, and PHP 5.4, and is connected to a remote database server running MariaDB 10.2 (shown in Figure 10).

Our objective is to answer two main questions:

(i) *Base Latency*. Why is the web site always slow?

A web page is supposed to be processed in 20 ms (20 ms being the average response time of other similar web sites on the same hardware), but in our case, the processing takes at least 50 ms.

(ii) *Unusual Latency*. Why is the response time much slower sometimes, but fine at other times?

The processing time is sometimes 10x more than usual.

4.1. Base Latency. A website may be slow for different reasons. It may be a misconfiguration or a problem on the web server, application server, database layer, network layer, or in the host machine operating system. Therefore, finding the root cause without having precise runtime information from each module would be impossible or very difficult. In this use case, we investigated different areas with our proposed tool and identified the root cause of the problem.

We had problems with a website running the MediaWiki (https://www.mediawiki.org) engine, taking 10 to 15 seconds of loading time. Apache logs were only showing that the response times are high, without helping us to pinpoint the problem. MariaDB slow query logs were also showing nothing.

Using our proposed method, we can see the different layers, having enough details at each level to investigate problems. We were able to see the response time for each request, including the portion clasped in each module separately. From this view, we found that the application layer (i.e., PHP) seems to be the bottleneck (or at least the first place to investigate) because it consumed the largest portion of the response time. This leads us to look at the PHP level to see if there is a problem.

Looking at the top function calls for the problematic server showed that all requests go to functions called wp_∗ which are functions from an installed WordPress. It was surprising to see WordPress here because the Mediawiki engine is a completely separate application and does not need any other application like WordPress. More investigations in the analysis data show that a WordPress module is installed to unify the authentication of two related sites, one running Mediawiki and the other WordPress, both being based on PHP sessions.

In other words, each MediaWiki request keeps checking the PHP session to see if there is an authenticated user in WordPress, to let them sign into the Mediawiki engine, which was taking around 70% percent of each request. By changing the module source code and commenting out the part that was continuously checking the session, we could confirm that the problem was because of this module.

Moreover, our tool helped us to uncover problems in other layers as well, like the database layer. We saw that some requests spent around 50% of their time in the database layer. We used our MySQL layer data to dig deeper into these requests.

We found a database connection that had some update queries that take time and look costly. Expanding on this view, we can easily see which queries take a longer time than others. Looking at the queries, we noticed that the update queries are for gathering statistics about the WordPress component of the site. By looking at the installed WordPress plugins, and disabling the statistics plugin, we could solve the problem of these requests as well.

The above use cases, which are only a few samples out of many possible analyses with our tool, show that, with the proposed methods and tools, we can quickly see the modules, requests, pages, and database tables involved in the problem. It then becomes relatively easy to understand the situation and to solve any unexpected behavior by disabling the problematic nonrequired default modules, or by modifying the configuration that had seemed interesting at first but added much latency.

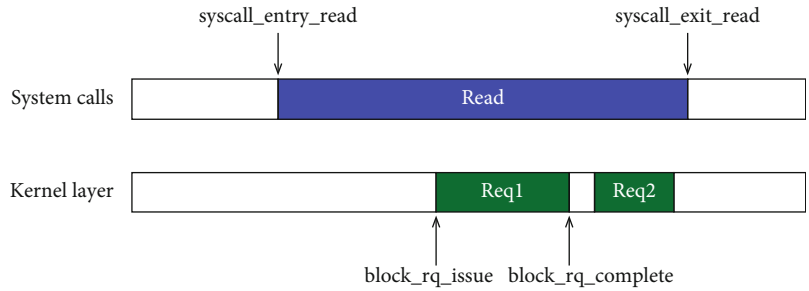

Figure 12: Cache miss ratio.

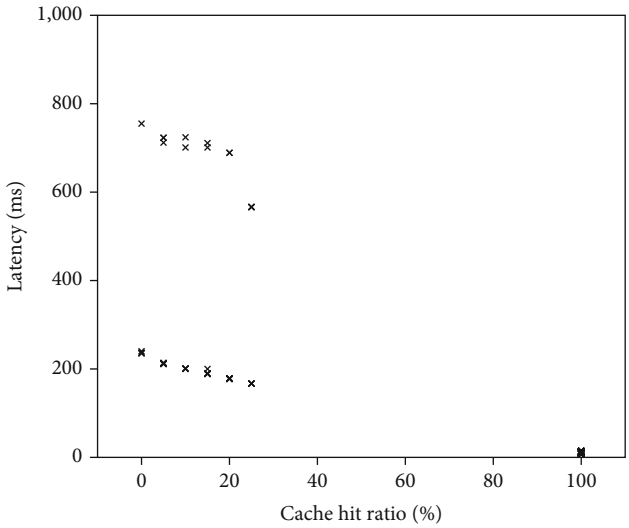

Figure 13: Correlation between response time latency and cache hit ratio.

4.2. Unusual Latency. In the previous use case, after detecting and solving the latency problem, we were able to get a much better response time. To ensure that everything was working as expected, we used ApacheBench to stress the website. A scatter chart of response times is presented in Figure 11.

The response time of the server is unstable. Most web requests are processed in 10 ms. However, in some cases, the processing takes much more time. Figure 11 shows three main categories of requests:

(C1) A latency of 10 ms. Most requests belong to this category.

(C2) A latency of (180 ms, 220 ms).

(C3) A latency of (600 ms, 800 ms).

The user-space trace provided by the LAMP tracing module is not enough to understand this problem. More precise information from the operating system is needed for a deeper analysis.

One of the major factors that can affect the speed of the website is the storage device access, whether to process database queries or to load static web pages. Disk caching plays a major role in reducing the frequency of disk accesses, by keeping the required data in memory. Many web servers and database engines implement a user-space cache, and others rely on the operating system cache. User-space caching is generally more efficient since the caching mechanism is tailored by the database itself.

Our tool can generate precise information about cache hits/misses that happen in the context of the webserver, based on system calls and block events. The computation is performed as follows: a read operation starts with the event syscall_entry_read and ends with syscall_exit_read. If all the required data is found in the cache, no disk operation is observed. Otherwise, block device requests will be created to recover the missing information from the disk. The miss ratio is the amount of data read from the disk divided by the amount returned by the *read* system call. For example, the miss ratio of Figure 12 can be computed as follows:

$$\text{MissRatio} = \frac{\text{Size(Req1)} + \text{Size(Req2)}}{\text{Size(READ)}}. \quad (10)$$

And therefore,

$$\begin{aligned}\text{HitRatio} &= 1 - \text{MissRatio} \\ &= \frac{\text{Size(READ)} - \text{Size(Req1)} + \text{Size(Req2))}}{\text{Size(READ)}}. \end{aligned} \quad (11)$$

Figure 13 shows that there is a clear correlation between the response time and the cache hit ratio. Two cases, with respect to Figure 14, are to be considered:

(1) *Web Requests That Are Fully Served from the Cache.* These requests are the fastest ones and they correspond to category (C1). All the data required is present in the cache and no disk access is required

(2) *Web Requests That Are Partially or Totally Served from the Disk (Cache Hit Ratio between 0% and 30%).* These requests correspond mainly to categories (C2) and (C3).

The cache hit analysis was useful to understand the origin of the slowness of some requests. However, the difference between categories (C2) and (C3) is still unknown. Why do (C3) requests take more time to get the required information from the disk? To answer this question, a more detailed analysis at the level of the storage subsystem should be performed. It is important to understand why an I/O request is sometimes slower than usual. Our tool gives very detailed information about the I/O scheduler, such as the length of the waiting queue and the identity of the processes

Figure 14: Difference between fast and slow requests.

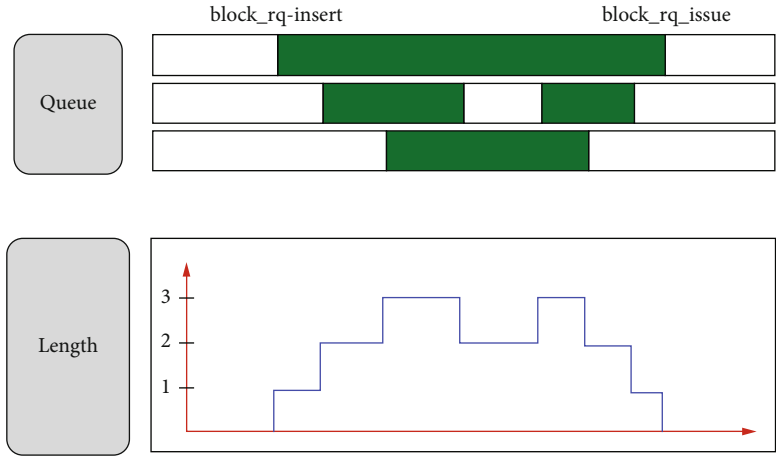

Figure 15: Queue length computation.

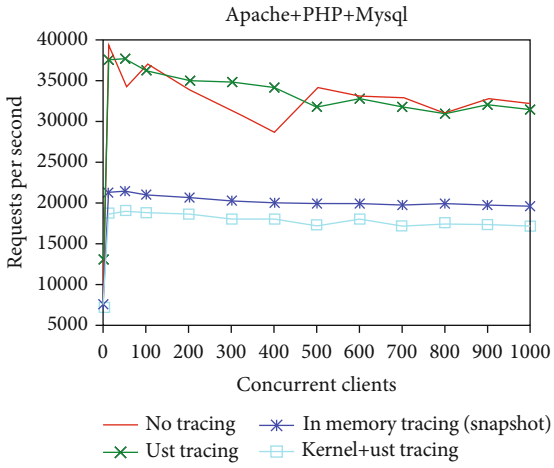

Figure 16: Comparison of kernel and user-space tracing costs.

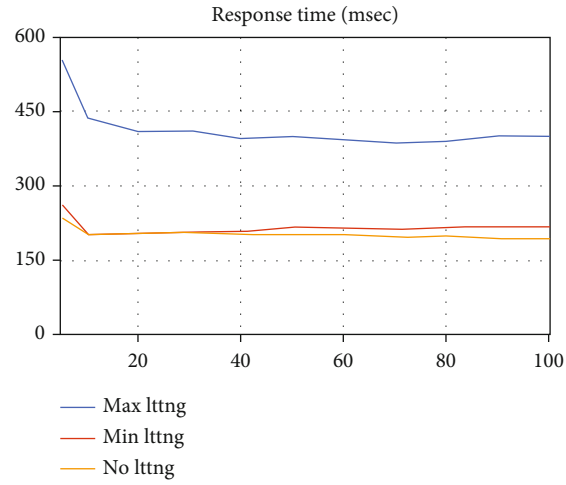

Figure 17: Performance of user-space tracing with 65 million lines of code.

generating I/O requests during a selected time range. Computing the waiting queue is done based on the events *block_rq_insert* and block_rq_complete, as shown in Figure 15.

Our tool was able to show that the difference between (C2) and (C3) is that (C3) requests are processed when the disk drive is busy processing other requests using a process called *backup.sh*. This process is copying server files to a backup medium for recovery purposes in case of failure.

5. Evaluation

It is crucial to minimize the overhead of tracing to avoid having biased results. A high overhead can change the normal behavior of the system, which makes the solution unusable in production systems.

The analysis cost is another important factor that must be taken into account. Building the state system database must be performed in a reasonable time to improve the user experience. The trace files are often very large and inspecting them must be done efficiently.

This section shows the different tests realized to ensure that our tool has low overhead under different circumstances.

5.1. Environment. The tests are executed in a machine with the following configuration.

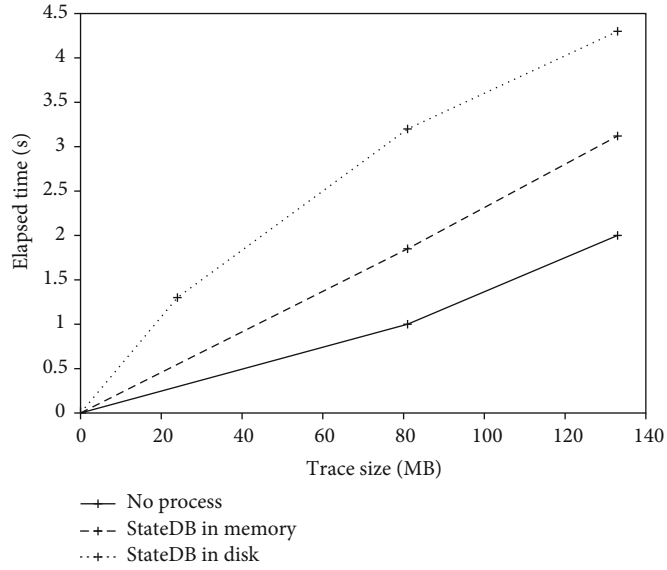

Figure 18: Comparison of state database construction cases.

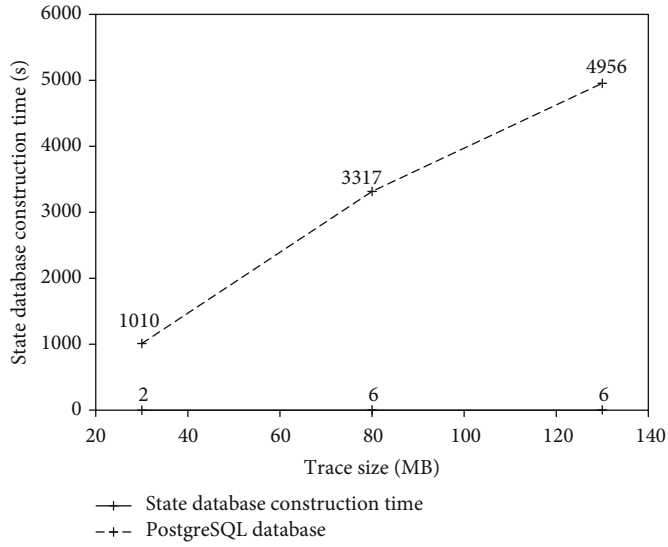

Figure 19: Comparison of construction time against PostgreSQL.

(a) Hardware configuration:

(i) Intel i7-4790 CPU @ 3.60GHz

(ii) 32 GB RAM

(b) Software configuration:

(i) Linux kernel version 4.4

(ii) LTTng 2.7

(iii) Apache 2.4.23, PHP 5.4, and MariaDB 10.2

5.2. Tracing Cost. The performance analysis is achieved using a real configuration composed of an Apache Web server and a MariaDB database. The workloads applied are generated using *ab* (ApacheBench), and the traces are collected using LTTng 2.7. ApacheBench is used to simulate the behavior of many clients navigating throughout the site. The experiment is performed using different numbers of clients (between 1 and 1000) with the following configurations:

(i) *No Tracing*. The tracing is disabled

(ii) *Required Events*. Only events required for the analyses are activated

(iii) *All Events in Memory*. All kernel and user-space events are activated and the trace is kept in memory

(iv) *All Events*. All kernel and user-space events are activated and the trace is written to the disk

The results of the experiment are presented in Figure 16.

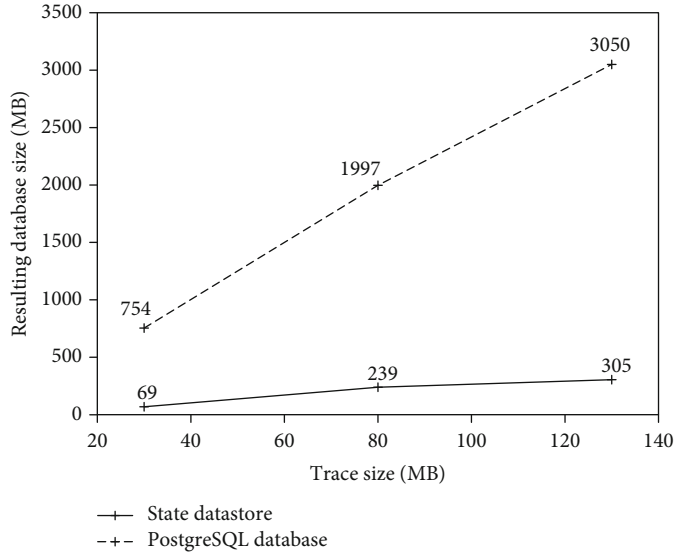

Figure 20: Comparison of disk usage against PostgreSQL.

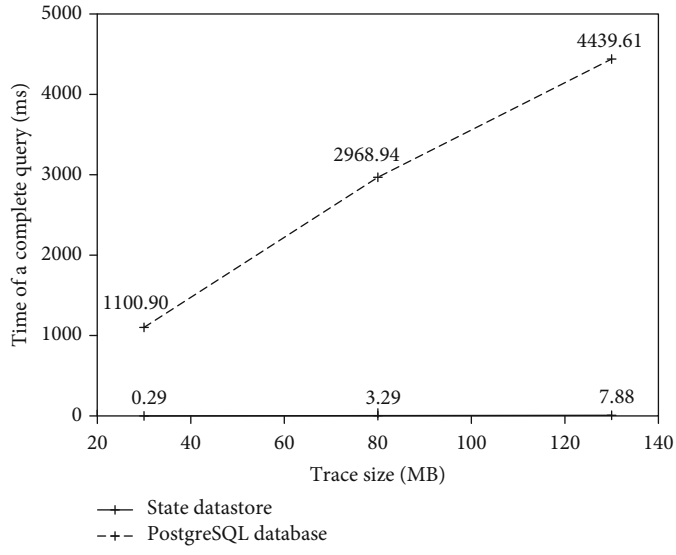

Figure 21: Comparison of full query time against PostgreSQL.

The graph shows that enabling the required events does not have a significant impact on the website's performance. The server can process about 30000 requests per second in both cases. The impact of tracing becomes significant if all tracepoints (including kernel tracepoints) are activated. Kernel tracing appends a lot of system details to the trace and increases the overhead. In the case where the kernel tracing is enabled, the processing speed goes down to 21000 if the trace is written in memory, and 19000 if it is written to the disk. Writing data to the disk does not add much overhead, because it is done asynchronously using a separate process. Please note that in Figure 16, for some points, the ust-tracing acts slightly better than the no-tracing case. This should not have happened if they were running under identical conditions. However, there is always some variability in operating system execution (interrupts, scheduling, memory page faults...). In that case, since the overhead of tracing a relatively small number of ust events is almost negligible, this overhead is difficult to measure, being smaller than the operating system variability.

The experiment is conducted on a Drupal website where serving a web request requires executing 30 k lines of code on average. We repeated the test on a benchmark of 65 million lines of code, to test if executing more lines of code will make a big difference. Figure 17 shows the response time in three different cases: (1) when all UST trace events are enabled, (2) when all trace events are enabled except the PHP line-execution events (ust_PHP:execution_start, ust_PHP:execution_end), and (3) tracing is disabled.

As shown in Figure 17, when there are 65 million lines of code (which is very rare for a web application), the response time is twice as long with tracing, compared to when there is no tracing (or minimum tracing). This is predictable because there are two PHP events activated and triggered for the

execution of each PHP line of code, which provides very detailed information but increases the response time. However, when the line execution events are disabled (i.e., the min-LTTng case), the response time becomes almost the same as when tracing is disabled, showing that UST tracing does not impose a significant performance overhead.

5.3. Analysis Cost. As mentioned earlier, the proposed technique uses a tree-based data structure to store the system state, extracted from the raw trace events, at different time points, to be used later in the analysis phase. Since the state database offers a fast query response time ($O(\log\ n)$), locating the required data within the (usually considerable) trace data becomes faster, leading to efficient analysis and faster graphical rendering. In this section, we evaluate the costs of using this database and compare it to a spatial database like PostgreSQL, which can store multidimensional data on the disk, enabling it to store state intervals for large trace files.

First, we look at the time needed to extract the required state from the trace events and construct the state database. Figure 18 shows the time required to construct the attribute tree and state database. It compares the different cases. Not surprisingly, the fastest time is for the case where the system only reads the trace events without parsing and analyzing them. The second fastest case is when the system reads the traces, analyses them, extracts states but stores them only in memory and not on disk, therefore, no disk I/O is involved. The worst case is when the state database is built and written to disk.

As Figure 18 shows, the I/O required to write the database to disk is the most time-consuming part.

We then compare the construction time of the state database using our solution, with the one using the PostgreSQL database. Figure 19 shows that PostgreSQL takes a much longer time than the state database, especially with larger trace files.

We also compared the disk usage and the full query time between the two databases (Figures 20 and 21). PostgreSQL requires more disk space to save the data and takes more time to serve full queries.

The results above show that even if PostgreSQL provides highly optimized mechanisms to manage generic data sets, it cannot beat the performance of a special-purpose database such as the state database. The state database is tailored to trace files, and it stores the data in ways that make it quickly accessible while using minimal disk space.

6. Conclusion

We have presented a unified analysis method for studying trace data gathered from different layers and sources. The concept is to analyze the collected data, based on the relations between the layers (timestamp, process id, etc.). Our solution extracts the required information from the raw input data and stores it in a multidimensional data store where it can then be analyzed.

This solution has been evaluated by real-world web applications experiencing performance degradation (i.e., MediaWiki, Drupal, and WordPress). For instance, when a website is running slow or when there is an unexpected latency. The solution uses trace data from multiple layers and helps uncovering the root cause(s) of the problem.

Since the proposed solution works by gathering runtime data from different layers; which contains valuable information about the different aspects of the running system; it can be utilized to investigate a wider variety of problems, including network attacks and host-based anomalies. This work would also benefit from a multilevel visualization display of the analyses data. Both will be investigated as future work.

Data Availability

All the data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, and EfficiOS for funding this project.

References

[1] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: precise discovery of request processing paths from black-box observations of thread and network activities," in *In Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX'09*, Berkeley, CA, USA, 2009.

[2] E. Thereska, B. Salmon, J. Strunk et al., "Stardust: tracking activity in a distributed storage system," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 3–14, 2006.

[3] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: event-based tracing to measure mobile application and platform performance," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, Montreal, QC, Canada, 2013.

[4] M. Desnoyers and M. R. Dagenais, "The lttng tracer: a low impact performance and behavior monitor for gnu/linux," *In OLS (Ottawa Linux Symposium)*, vol. 2006, pp. 209–224, 2006.

[5] F. Doray and M. Dagenais, "Diagnosing performance variations by comparing multi-level execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2016.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 595–604, Washington, DC, USA, June 2002.

[7] R. R. Sambasivan, A. X. Zheng, M. De Rosa et al., "Diagnosing performance changes by comparing request flows," in *In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, pp. 43–56, Berkeley, CA, USA, 2011.

[8] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "Perfcompass: online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1742–1755, 2016.

[9] M. Bligh, M. Desnoyers, and R. Schultz, "Linux kernel debugging on google sized clusters," in *Proceedings of the Linux Symposium*, pp. 29–40, Ottawa, Ontario, Canada, 2007.

[10] N. Ezzati-Jivan and M. R. Dagenais, "Multi-scale navigation of large trace data: a survey," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, article e4068, 2017.

[11] H. Daoud and M. Dagenais, "Multilevel analysis of the java virtual machine based on kernel and userspace traces," *Journal of Systems and Software*, vol. 167, article 110589, 2020.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[13] I. Kohyarnejadfard, D. Aloise, M. R. Dagenais, and M. Shakeri, "A framework for detecting system performance anomalies using tracing data analysis," *Entropy*, vol. 23, no. 8, p. 1011, 2021.

[14] E. Ates, L. Sturmann, M. Toslali et al., "Association for Computing Machinery," in *An automated, crosslayer instrumentation framework for diagnosing performance problems in distributed applications. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '19*, pp. 165–170, New York, NY, USA, 2019.

[15] B. H. Sigelman, L. Andr'e Barroso, M. Burrows et al., *Dapper, a large-scale distributed systems tracing infrastructure*, Technical report, Google, Inc., 2010, https://research.google.com/archive/papers/dapper-2010-1.pdf.

[16] S. S. Murtaza, A. Sultana, A. Hamou-Lhadj, and M. Couture, "On the comparison of user space and kernel space traces in identification of software anomalies," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 127–136, Szeged, Hungary, March 2012.

[17] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Graph summarization methods and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–34, 2018.

[18] B. Gregg, "Visualizing system latency," *Communications of the ACM*, vol. 53, no. 7, pp. 48–54, 2010.

[19] O. Rodeh, H. Helman, and D. Chambliss, "Visualizing block io workloads," *ACM Transactions on Storage*, vol. 11, no. 2, 2015.

[20] A. D. Brunelle, *Block i/o layer tracing:blktrace*http://duch.mimuw.edu.pl/~lichota/09-10/Optymalizacja-open-source/Materialy/10%20-%20Dysk/gelatoICE06aprblktracebrunellehp.pdf.

[21] C. Mason, *Seekwatcher*http://oss.oracle.com/~mason/seekwatcher.

[22] B. Donie, *Ioprof*https://github.com/01org/ioprof.

[23] A. Brunellehttps://usermanual.wiki/Document/bttmanual.1495776143/view.

[24] M. Jabbarifar, M. Dagenais, and A. Shameli-Sendi, "Online incremental clock synchronization," *Journal of Network and Systems Management*, vol. 23, no. 4, pp. 1034–1066, 2015.

[25] *Eclipse trace compass*https://www.tracecompass.org.