Transparent Trace Annotation for Performance Debugging in Microservice-oriented Systems (Work In Progress Paper)

Adel Belkhiri
École Polytechnique de Montréal
Montreal, Canada
adel.belkhiri@polymtl.ca

Ahmad Shahnejat Bushehri
École Polytechnique de Montréal
Montreal, Canada
ahmad.shahnejat-bushehri@polymtl.ca

Felipe Gohring de Magalhaes
École Polytechnique de Montréal
Montreal, Canada
felipe.gohring-de-magalhaes@polymtl.ca

Gabriela Nicolescu
École Polytechnique de Montréal
Montreal, Canada
gabriela.nicolescu@polymtl.ca

ABSTRACT

Microservices is a cloud-native architecture in which a single application is implemented as a collection of small, independent, and loosely-coupled services. This architecture is gaining popularity in the industry as it promises to make applications more scalable and easier to develop and deploy. Nonetheless, adopting this architecture in practice has raised many concerns, particularly regarding the difficulty of diagnosing performance bugs and explaining abnormal software behaviour. Fortunately, many tools based on distributed tracing were proposed to achieve observability in microservice-oriented systems and address these concerns (e.g., Jaeger). Distributed tracing is a method for tracking user requests as they flow between services. While these tools can identify slow services and detect latency-related problems, they mostly fail to pinpoint the root causes of these issues.

This paper presents a new approach for enacting cross-layer tracing of microservice-based applications. It also proposes a framework for annotating traces generated by most distributed tracing tools with relevant tracing data and metrics collected from the kernel. The information added to the traces aims at helping the practitioner get a clear insight into the operations of the application executing user requests. The framework we present is notably efficient in diagnosing the causes of long tail latencies. Unlike other solutions, our approach for annotating traces is completely transparent as it does not require the modification of the application, the tracer, or the operating system. Furthermore, our evaluation shows that this approach incurs low overhead costs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Performance analysis, Microservices, Software tracing, Distributed systems

ACM Reference Format:
Adel Belkhiri, Ahmad Shahnejat Bushehri, Felipe Gohring de Magalhaes, and Gabriela Nicolescu. 2023. Transparent Trace Annotation for Performance Debugging in Microservice-oriented Systems (Work In Progress Paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3585030

1 INTRODUCTION

Microservice architecture is becoming increasingly popular in the cloud thanks to the flexibility and scalability it enables within the application [1]. Based on this architecture, a complex application can straightforwardly be distributed over multiple hosts as it is structured into several small, autonomous, and single-purpose services. These services communicate with each other through well-defined interfaces using lightweight APIs (e.g., RESTful APIs). This approach to application development offers numerous advantages over the traditional method of developing monolithic applications. For instance, since services are run independently, they can easily be scaled up and down in order to meet changing demands. In addition, because services are implemented as isolated units, developers can use different programming languages and frameworks depending on their skills and the required functionalities. They can also update a particular service without changing the code of other services, which would shorten the development and deployment time.

Despite the well-known advantages of microservices, the adoption of this architecture brings many challenges, especially when it comes to debugging performance issues [5]. In brief, diagnosing performance bugs in microservices-based applications is notoriously hard for the following reasons. First, processing user requests usually requires the collaboration of a set of services through the request/response synchronous communication (e.g., gRPC) or asynchronous event-driven messaging systems (e.g., Kafka). Therefore, a slow or bottlenecked service can cause a chain reaction that will eventually decrease the performance of other services. As latencies propagate through the components of the distributed application, diagnosing the root causes of the problem becomes extremely

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3585030

difficult. Second, typical microservice-based software comprises hundreds or even thousands of interconnected services. Therefore, the complexity of identifying performance issues and diagnosing their causes increases with the number of services and interdependencies. Third, the services are likely implemented in different programming languages, developed based on various libraries and frameworks, and communicate through heterogeneous networking systems. Hence, the complexity of this type of software impedes the ability of the practitioner to understand its operations and, thus, makes the troubleshooting process for performance anomalies long and tedious.

The most effective approach to detecting latency-related problems in microservices lies in tracking user requests as they travel across multiple services and hosts [17]. This approach, known as distributed tracing, provides end-to-end visibility of the request execution and enables uncovering the dependencies between services. Spans are the fundamental building blocks of distributed tracing. A span represents a tagged time interval that denotes the execution of a particular operation (e.g., RPC or function calls) as part of a service workflow. It is worth noting that spans are recorded with information that describes their operational contexts, such as the operation name, the start/end timestamps, and the causal relationship between each other. On the other hand, a trace is a series of related spans generated as a result of executing a given user request or transaction.

The literature reports many methods for monitoring microservice-based applications and diagnosing their performance bugs. For instance, X-Trace [11] and Dapper [21] have established the principles of distributed tracing, notably achieving causality between requests by propagating metadata between the services composing the distributed system. From ancestor solutions to modern open-source tools (e.g., Jaeger [15] and Zipkin [23]), distributed tracing has been used extensively for pinpointing abnormal latencies. Nevertheless, as these tools rely on gathering high-level information, they are unable to diagnose intricate performance issues or conduct in-depth analysis of latency causes. Several proposals based on the use of general-purpose tracers (e.g., LTTng [9] and eBPF [13]) have been proposed to circumvent this limitation. For instance, there have been a number of attempts to leverage host-based tracing to troubleshoot performance problems in distributed systems [2, 4, 16, 18, 19]. However, the difficulty of synchronizing fine-grained traces recorded on different hosts makes these proposals fail to diagnose performance bugs occurring from the interaction of many remote services. Conversely, there exist diagnostic tools that use cross-layer tracing to improve their visibility into microservice-oriented systems [3, 12, 20]. The major limitations of these tools are the substantial overhead they create and the necessity to recompile either the OS kernel, the tracer, or the microservice application.

In this article, we propose an efficient approach to address the shortcomings above by annotating distributed traces with relevant information and metrics derived from kernel events. The annotation added to the traces provides valuable insights into the operations of the application and the operating system during the execution of key operations. This information can help diagnose the causes of unexpected latencies and identify potential performance optimization opportunities. For instance, by examining the system calls executed by a microservice, practitioners can identify which ones are contributing to latencies or consuming the most resources. The analysis of system calls related to block I/O, such as *read()* and *write()*, enables identifying parts of the application that are experiencing excessive latency due to I/O operations, and showing how much time is being spent on each I/O operation. In addition, the analysis of system calls related to networking, such as *connect()*, *send()*, and *recv()*, can provide insights into the time spent on each communication. Hence, by adding annotations derived from kernel events to distributed traces, practitioners can identify areas of the application that require optimization, which can ultimately improve the overall performance.

Our approach is distinctly non-intrusive compared to others as it does not necessitate any alterations to the application, tracer, or kernel. Besides, the framework that implements our approach does not rely on any host-based tracer, and its overhead is minimal. To sum up, the contribution of this paper is two-fold: **Firstly**, we propose an efficient approach for annotating traces with useful kernel information. **Secondly**, we develop a framework based on this approach that can work with many distributed tracers.

This paper organizes the remainder as follows. Section 2 looks at the main techniques and tools for discovering performance bugs in microservice-based applications. Section 3 presents our approach and describes the design details of our framework. Section 4 reports the experimental results and evaluates the overhead of the proposed framework. Finally, Section 6 concludes the article and presents our future work.

2 RELATED WORK

Previous works aimed to achieve observability in microservices environments by focusing on two research directions. First, devise efficient tracing techniques adapted to distributed applications. Second, detect and diagnose software anomalies through trace analysis. Since this paper focuses on the first direction, we will describe it in the following paragraphs.

There is a large body of research focusing on the topic of tracing microservice-based systems. To date, the most mature approach to achieving this goal is tracing the end-to-end execution path of user requests. This approach relies on inserting unique identifiers into requests and establishing causality between them by propagating metadata between processes and distributed system components. X-Trace [11] is one of the earliest implementations of this approach. It is a tracing framework that enacts application tracing by propagating metadata down the request datapath and along all the sub-requests through the modification of networking protocols. Dapper [21], Google's distributed tracing system, shares many of its core ideas with X-trace but makes end-to-end performance tracing more convenient for production. For instance, it leverages sampling techniques to lower the tracing overhead. It also restricts instrumentation to a small corpus of threading, control flow, and RPC library code to increase application-level transparency. Canopy [17] makes tracing more comprehensive by decoupling instrumentation, collection of traces, and transformation of tracing data. Inspired by the ideas and tracing models proposed by these pioneering tracing systems, several open-source and proprietary distributed tracing tools have emerged (e.g., Jaeger [15], and Zipkin [23]). All these

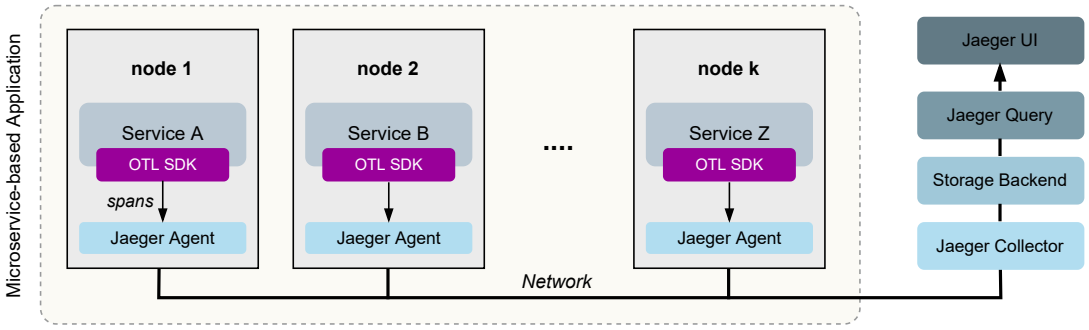

Figure 1: Reference architecture for a microservice-based application

tools can pinpoint several latency-related issues in monitored applications. However, as their operations are based on collecting high-level tracing data, they mostly fail to diagnose the root causes of detected issues.

Unlike the traditional approach adopted by most distributed tracing systems, a few proposals provide transparent tracing of microservices without requiring changes to the application's source code [6, 14, 22]. Frameworks based on this approach use service mesh [1] for intercepting communication between services. Hence, containers in a microservice system are deployed with sidecars (e.g., Envoy) that extract metadata from requests, generate tracing data, and report it to a distributed tracer. Furthermore, the authors in [14] proposed to monitor specific system calls (e.g., *read* and *sendto*) to establish causality between the requests. The high overhead and the necessity to modify the deployment of microservices represent the main limitations of these frameworks.

Distributed tracing frameworks produce high-level information providing a comprehensive overview of the end-to-end request processing. Practitioners can use this information to depict and analyze the operations of the traced system at the workflow level. Unfortunately, this information is usually insufficient to characterize the execution state of the system and, therefore, enact latency root cause analysis. As a result, numerous performance diagnostic frameworks use host-based tracing to collect detailed performance data from microservice systems [2, 16, 18, 19]. There is a multitude of host-based tracers for almost every OS. For instance, tracers like LTTng [9] and eBPF [13] are very popular among Linux users.

On the other hand, a few diagnostic tools leverage cross-layer tracing to improve their visibility into distributed systems. For instance, authors in [12] used a patched Jaeger client to capture and synchronize kernel and distributed request events. Moreover, authors in [3] inject application-level events into kernel traces by executing a stylized sequence of innocuous system calls at each high-level event of interest (e.g., the start and end of an RPC call). These system calls are used as synchronization anchors in the trace to interleave both high- and low-level events. The work closest to ours is presented in [20]. The authors used a custom kernel module to inject the contexts of distributed requests into kernel space. They used a patched version of the Linux kernel to add some information to the *task_struct* structure of the current process. They also modified LTTng to incorporate system call information into the userspace trace. The downside of the proposed framework is its high overhead as it is incapable of only annotating spans of interest. Besides, the need to recompile the modified kernel and tracer is another serious limitation.

This paper proposes a framework that leverages kernel events for annotating traces generated by distributed tracers. Our framework aims to provide practitioners with useful information to help them diagnose the causes of long tail latencies. Compared to other works, our approach presents less overhead and ensures monitoring transparency and ease of deployment since it does not require the modification of the application, the distributed tracer, or the operating system.

3 PROPOSED SOLUTION

This section introduces our approach to annotating distributed traces transparently. For the sake of simplicity, we present in Fig. 1 a reference architecture for a microservice-based application. A typical microservice application is composed of a collection of services deployed on several nodes interconnected by a network. Services are instrumented using OpenTelemetry (OTL). The latter is an open-source project that provides a unified framework for generating, capturing, and transmitting telemetry data (i.e., logs, traces, and metrics). It also offers a set of APIs and libraries enabling the instrumentation of applications across many programming languages in a vendor-agnostic way. It is worth noting that the setup in our experiments relies on Jaeger to propagate and process the traces generated by OTL. Nevertheless, our framework can also work with any other distributed tracing backend tool. Fig. 2 illustrates the architecture of the framework that implements our approach. The framework we propose comprises components of two types: monitoring libraries and kernel modules. We describe the role and relationship between these components in the following sections.

3.1 Monitoring libraries

Each monitoring library is in charge of three main tasks. First, it registers the threads executing the monitored service with the kernel modules. The registration is performed through an input/output

[1] A dedicated infrastructure layer for facilitating service-to-service communications

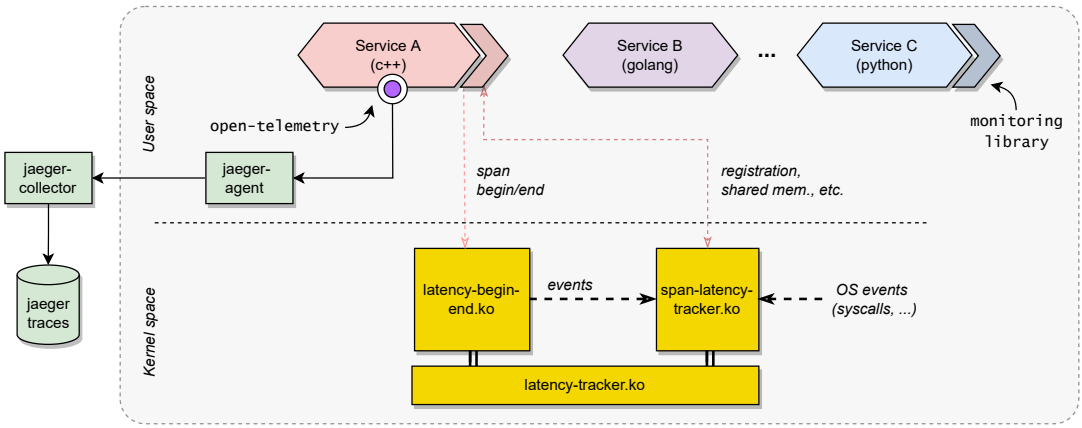

Figure 2: Architecture of the proposed framework

control (ioctl) system call. The ioctl system call is commonly used in Linux and other Unix-based operating systems to enable user-level programs to communicate with a device driver in order to control or query the device's state. The purpose of the registration phase is to allow kernel modules to identify the spans they need to track their latencies and annotate their corresponding traces. Considering that several services could be running on the same host machine[2], the registration allows choosing which services to monitor.

Second, the monitoring library intercepts the start and end of spans generated during the execution of services and forwards corresponding events to our kernel modules. Therefore, before running a service, we must dynamically link it with a monitoring library implemented in the same programming language. For instance, to run a service written in C++, we preload a library *X*, and to run a service written in Golang, we preload another library *Y*. Indeed, library preloading is a handy and efficient technique for hooking into functions and overriding specific symbols transparently. In Linux, we use *LD_PRELOAD* to preload monitoring libraries at runtime. *LD_PRELOAD* is an environment variable containing paths to shared libraries that the Linux loader (i.e., *ld.so*) will load before any other libraries at the application startup.

For instance, we leveraged this technique to hook into the function *set_tracer_provider()* in order to implement a monitoring library for services implemented in Python. Our goal is to instantiate a custom span processor and register it using the *add_span_processor()* method of the *trace_provider* object. The latter is received as a parameter of the function *set_tracer_provider()*. OTL allows hooks for span's start and end methods invocations through the span processor interface. It is worth noting that span processors are invoked in the same order they were registered, and their callbacks are not executed unless spans are sampled. Hence, whenever our monitored service calls an API method to start a span (e.g., the *start_span()* and *start_as_current_span()* methods), our custom span processor intercepts the invocation and injects a "start event" into the kernel space. Similarly, it injects an "end event" into the kernel whenever the scope of a given span is terminated. Since the application may use several span processors (e.g., span exporters), it is crucial to ensure that the tracing span processor is inserted at the head of the span processors list. This would ascertain that the callback functions provided by the tracing span processor are invoked first and, therefore, minimize the delay between the time the span is started/ended and the time the corresponding event is recorded. Our framework uses both the "trace id" and "span id" as a key to uniquely identify generated events.

Finally, the monitoring library uses the information received from kernel modules to annotate the traces. The kernel modules leverage a set of shared memory buffers to send annotation data to the monitoring libraries asynchronously. We implemented the shared memory using the relay interface provided by the Linux kernel [10]. This interface enables kernel modules to efficiently transfer large quantities of data to userspace applications via many relay channels. A relay channel is implemented as a set of per-CPU kernel buffers, each represented as a regular file. Our monitoring libraries retrieve annotation data from those files as it becomes available, using *mmap()* or *read()* functions. Then, they use the various API functions provided by OTL to annotate traces, such as *set_attribute()* and *add_event()*. Exceptionally, to annotate traces with the system calls executed by the service during a given span, it creates nested spans with the same names.

3.2 Kernel modules

Our framework uses three kernel modules for tracking the latencies of spans generated by registered services and calculating performance metrics. The operations of the *span-latency-tracker* and *latency-begin-end* kernel modules are based on the *latency-tracker* module, which was developed in our laboratory [8]. The latter provides a rich API enabling the efficient and scalable tracking of events in the kernel. It matches entry events to exit events based on a shared key. It then executes a user-provided callback if the delay between two matching events exceeds a predefined and dynamically configurable threshold. In our context, the entry and exit events are the *span_start* and *span_end* events triggered by our monitoring libraries.

Hence, the monitoring library intercepts calls to span begin or end methods and writes the span identifier in one of the two *proc*

[2]A physical or virtual machine

Table 1: Subset of the tracepoints monitored by the kernel module of the proposed tool

Tracepoint	Description
span_begin	Triggered when the monitored application starts a span. The main fields are trace_id and span_id.
span_end	Triggered when the monitored application closes a span.
task_newtask	Fired when a new thread is spawned. The main fields are PID and TGID (Thread Group ID).
sched_process_exit	Fired when a thread is terminated. PID is the main field of this event.
syscall_enter	Fired when a system call is executed. The identifier of the syscall is the main field of this event.
syscall_exit	Fired when a system call finishes its execution.

files [3] provided by the *latency-begin-end* module. The latter's role is to trigger kernel events that delimit spans latencies based on span identifiers written in the *proc* files. On the other hand, the *span-latency-tracker* module is responsible for two main tasks. First, it tracks the span begin and end events and calculates the span latencies using the API provided by the *latency-tracker* module. Similarly, it observes several other kernel events to calculate relevant performance metrics that characterize the execution state of the service and the system on which it is running (see Table 1). For example, the *span-latency-tracker* module monitors the *sys_enter* and *sys_exit* kernel events to record the system calls made by each thread of the monitored service. If the latency of a particular span exceeds a dynamically computed threshold, the module sends information about the system calls executed within that span to the monitoring library via relay channels. Since each span is associated with the thread that emitted it, only the system calls executed by that thread are annotated for that span. Besides, our kernel module uses the parent-child relationship between nested spans and the timestamps of system call execution to determine which span to annotate. The practitioner can configure the proposed framework to track only a subset of system calls of interest.

Furthermore, our tool can be customized to collect additional data and use it for annotating distributed traces. For instance, the *span-latency-tracker* kernel module can be configured to generate and export a kernel call stack whenever a span exceeds a preconfigured threshold. This kernel module leverages signals to notify monitoring libraries and enable them to annotate active spans with kernel and userspace call stacks. Moreover, it can be configured to export generic metrics denoting the load of the system on which the monitored service is running. One example of such metrics is the average wake-up latencies of threads. The wake-up latency denotes the time a thread waits in the running queue before it starts running on the CPU. When the monitored service threads exhibit increasing wake-up latencies, performance degradation can be explained by system overload or the presence of many higher-priority threads. We leverage the *sched_waking* and *sched_switch* kernel events for calculating the wake-up latency metric.

One intriguing challenge we had to address in our design was to prevent the kernel module from collecting data related to the threads spawned by the monitoring libraries. In other words, how can our tool only track the threads of the monitored service? Undoubtedly, tracking all the threads spawned by the application would affect monitoring accuracy and increase our tool overhead. From the operating system's point of view, it is infeasible to distinguish between the threads we spawn for annotating traces and those originally spawned by the service. The algorithm we devised to solve this issue comprises three steps. First, the monitoring library explicitly registers the main thread of the service at startup. Second, our kernel module leverages the *task_newtask* event to automatically register any thread whose TGID is the main thread PID. Finally, each spawned monitoring thread invokes the kernel function *unregister_thread()* to remove itself from the list of tracked threads. It is worth noting that the kernel thread uses the *sched_process_exit* event to detect threads that have finished their executions and unregister them automatically. This method allows tracking threads of interest, even if the monitored applications rely on thread spawning for load-based dynamic scaling, such as those using gRPC to implement services.

[3]These files are named *latency-tracker-begin* and *latency-tracker-end*.

4 RESULTS AND EVALUATION

We dedicate this section to evaluating the efficacy of our tool and estimating its overhead. We tested the functionalities of our tool with many multi-threaded applications using monitoring libraries written in C++ and Python. For instance, we assessed our tool using Astronomy Shop [7], a microservice-based application that simulates an e-commerce website. Fig. 3 illustrates how our tool annotates the long-lasting spans of monitored services with helpful information extracted from the kernel. We chose the Astronomy Shop application because it is already instrumented with OTL and is composed of fourteen services implemented in various programming languages. The use of various technologies to implement those services would allow us to compare the overhead costs induced by utilizing different monitoring libraries.

Table 2: Hardware and software experimental configurations

Host Environment	
CPU	Intel Core i7-7500U CPU @2.70GHz
Cores	4
RAM	16 GB
OS	Ubuntu 22.04
Kernel	Version 5.15.0-56
Opentelemetry-cpp	Version 1.8.0
Jaeger	Version 1.33.0

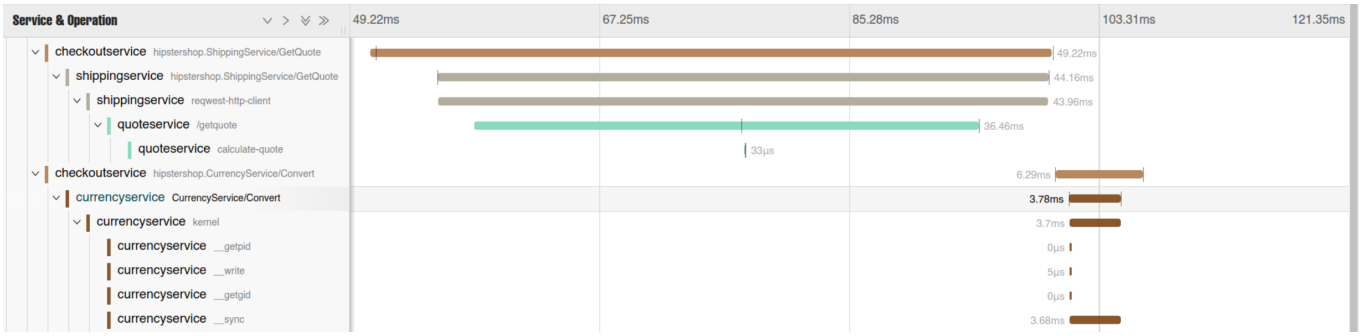

Figure 3: Snapshot of the Jaeger UI showing the span *CurrencyService/Convert* annotated by our tool with the system calls executed within it.

The overhead incurred by any performance analysis tool might obscure the causes of existing latencies and alter the normal operations of the monitored application if it is considerable. Hence, we conducted a set of experiments to assess the impact of annotating the traces generated by the "currency service". The latter uses gRPC to enable other services (e.g., the *frontend* and *checkout* services) to convert amounts of money between different currencies. Since the currency service is implemented in C++, our experiments require preloading a monitoring library written in the same programming language. For the sake of simplicity, we only measure the overhead of annotating traces with the system calls executed within spans. Table 2 presents the hardware and software experimental configuration we used to conduct these benchmarks.

In our first experiment, we aimed to measure the increase in overhead that occurs when our tool is coupled with a distributed tracer, like Jaeger. To achieve this, we developed a client program that can remotely invoke the "CurrencyConvert" procedure of the currency service multiple times. The currency service relies on spawning new threads to handle concurrent requests. Fig. 4 depicts the time required by the monitored service to fulfill the received requests when it is not traced, only traced with Jaeger, and traced with Jaeger and our tool. It is worth noting that our setup uses a trace sampling rate fixed at 10%. Besides, we configured our framework to annotate all spans regardless of their latencies (i.e., the threshold parameter of our kernel module is set to 1 nanosecond). The figure shows that our tool incurs negligible overhead if the number of requests is less than 10k. Nevertheless, when the number of requests reaches 100k, the overhead incurred by the usage of Jaeger combined with our tool is estimated at 7.55%. In other words, our tool imposes an additional overhead estimated at 4.16%, which is very acceptable.

The overhead of our tool is likely impacted by the quantity of tracing data collected from the kernel and injected into the trace. Drawing on this observation, we wanted to measure how this overhead varies according to the number of system calls executed per span. Hence, the second benchmark consists in inserting several innocuous system calls (i.e., *getpid()*) into the currency service and measuring the latency of processing received requests. Fig. 5 presents the results we got for this benchmark. First, this figure reveals that the number of system calls impacts the overhead of our tool. Second, it also shows that the difference in processing latencies is negligible when the workload is composed of 10k requests or less. Nevertheless, when the workload is composed of 100k requests or more, the processing latency increases by almost 4% for every five system calls added.

Results from the last experiment demonstrate that monitoring overhead increases linearly with the number of recorded system calls (precisely with the number of kernel events). They prove that our tool can scale well with the load and type of monitored services. Besides, it should be noted that most applications trace spans at a sampling rate less or equal to 1%. Furthermore, our tool allows practitioners to set a latency threshold for span annotation and specify a list of system calls to track, significantly reducing thus tracing overhead. In short, our tool presents minimal overhead, making its use with a distributed tracer very convenient.

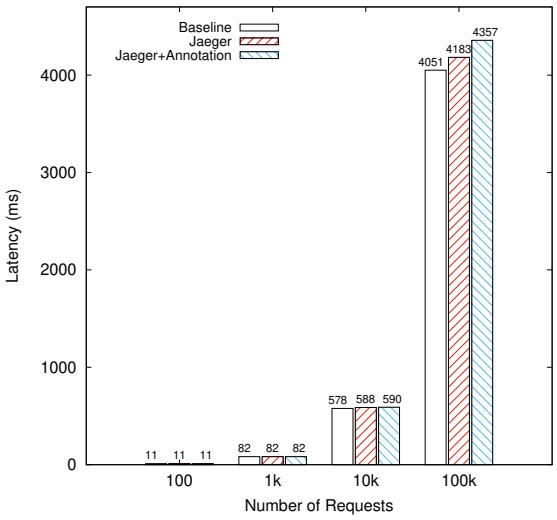

Figure 4: Execution time of the benchmarked microservice when tracing is not enabled, traced with Jaeger, and traced with our tool.

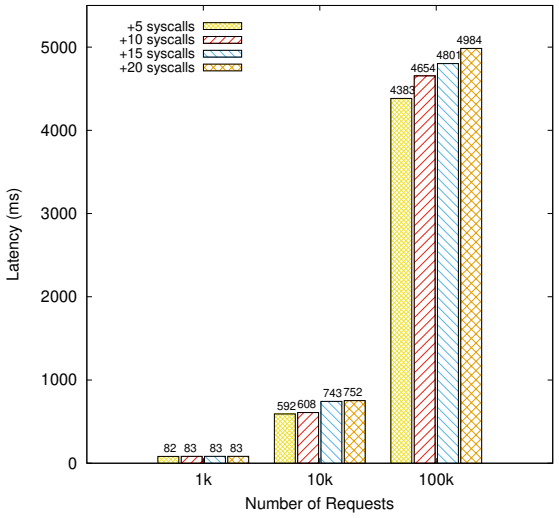

Figure 5: Execution time of the benchmarked microservice depending on the numbers of requests and injected system calls.

5 THREATS TO VALIDITY

Our approach is designed to transform existing distributed tracing frameworks into powerful performance diagnostic tools with minimal costs. However, it has two limitations that we will outline in the following paragraphs.

The proposed tool is compatible with compiled and interpreted programming languages like C++ and Python. However, it cannot annotate traces generated by microservices developed in bytecode-based programming languages (e.g., Java). Applications written in bytecode-based languages require specific Virtual Execution Environments (VEEs), like the Java Virtual Machine (JVM), to translate bytecode into machine code. Due to the complexity of the mechanisms implemented by the VEEs (e.g., garbage collection and JIT compilation), executing code at application startup can be challenging. Additionally, the threading models adopted by these VEEs make it difficult to associate generated spans with their underlying threads, complicating thus the task of adding annotations.

As a second limitation, our approach cannot intercept system calls of the virtual Dynamic Shared Object (vDSO) type. The vDSO is a mechanism that modern Linux kernels use to provide applications access to certain system calls (e.g., *clock_gettime()*) without requiring a context switch to kernel mode. As this allows a faster system call execution and less overhead, system calls accessed via vDSO are unlikely to induce considerable latencies or high resource consumption.

In short, our approach provides efficient tracing for microservices developed in numerous programming languages. Nevertheless, those developed in bytecode-based languages require additional considerations, mostly due to the complexity of their execution environments. Besides, our tool cannot intercept vDSO system calls, but this limitation is insignificant and does not diminish our approach's efficacy.

6 CONCLUSION

Microservices is a well-defined software architecture whose primary goal is to reduce the complexity of the development and deployment of distributed applications. Despite this architecture's numerous benefits, discovering the causes of unexpected long latencies of the application operations remains challenging. Distributed tracing, the most efficient method for detecting latency-related issues in microservice-based applications, enables practitioners to discover the unusual latencies of operations' executions quickly. Nevertheless, this method does not provide any means to enact analyses for debugging spotted latencies and pinpointing their root causes.

This paper proposes a novel approach for annotating span-based traces with meaningful data gathered from the kernel. It also introduces an open-source tool that implements this approach. This tool presents many advantages. First, it is entirely non-intrusive as it does not require the modification of the application, the distributed tracer, or the operating system. Second, it does not rely on any third-party library or tool like most related works. Finally, it is developed based on efficient algorithms and data structures, such as hash maps for matching *span_begin* and *span_end* events, and shared memory for exchanging data between kernel and monitoring libraries. Besides, as it is designed for microservice-based applications that exhibit high tail latencies, it only annotates long-lasting spans. As a result, the proposed tool presents minimal overhead.

In our future work, we aim to improve the proposed tool's functionalities by adding annotations that include further data and metrics derived from kernel events. We also plan to enhance our framework's annotation mechanism to support bytecode-based microservices. Furthermore, while this manuscript provides a coarse-grained overhead analysis based on request processing time, we intend to conduct a more detailed cost analysis that delves into our tool's CPU and memory overhead. Our ultimate goal is to analyze larger distributed systems and compare the overhead of our tool to alternative solutions. Additionally, we plan to gather practitioner feedback to improve the tool's usability. By achieving these goals, we hope to provide a more efficient and comprehensive performance diagnostic tool that addresses the limitations of existing distributed tracing systems.

DATA AVAILABILITY STATEMENT

The source codes of the proposed framework are available on the author's GitHub: https://github.com/adel-belkhiri.

ACKNOWLEDGMENTS

We would like to express our gratitude to the Humanitas Corporation and the Mathematics of Information Technology and Complex Systems (MITACS) organization for their support in funding this research and our heartfelt thanks to Maroua Ben Attia for her discussion of our framework design.

REFERENCES

[1] Amr S. Abdelfattah and Tomas Cerny. 2023. Roadmap to Reasoning in Microservice Systems: A Rapid Review. *Applied Sciences* 13, 3 (2023). https://doi.org/10.3390/app13031838

[2] Marcelo Amaral, Tatsuhiro Chiba, Scott Trent, Takeshi Yoshimura, and Sunyanan Choochotkaew. 2022. MicroLens: A Performance Analysis Framework for

Microservices Using Hidden Metrics With BPF. *IEEE 15th International Conference on Cloud Computing (CLOUD)* 00 (2022), 230–240. https://doi.org/10.1109/cloud55607.2022.00043

[3] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 405–417. https://www.usenix.org/conference/nsdi18/presentation/ardelean

[4] Adel Belkhiri, Martin Pepin, Mike Bly, and Michel Dagenais. 2023. Performance analysis of DPDK-based applications through tracing. *J. Parallel and Distrib. Comput.* 173 (2023), 1–19. https://doi.org/10.1016/j.jpdc.2022.10.012

[5] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. 2021. Automated Analysis of Distributed Tracing: Challenges and Research Directions. *Journal of Grid Computing* 19, 1 (2021), 9. https://doi.org/10.1007/s10723-021-09551-5

[6] Donghun Cha and Younghan Kim. 2021. Service Mesh Based Distributed Tracing System. *International Conference on Information and Communication Technology Convergence (ICTC)* 00 (2021). https://doi.org/10.1109/ictc52510.2021.9620968

[7] OpenTelemetry CNCF. 2022. Astronomy Shop, the OpenTelemetry Demo. https://github.com/open-telemetry/opentelemetry-demo

[8] Julien Desfossez, Mathieu Desnoyers, and Michel R. Dagenais. 2016. Runtime latency detection and analysis. *Software: Practice and Experience* 46, 10 (2016), 1397–1409. https://doi.org/10.1002/spe.2389

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

[10] The Linux Kernel documentation. 2022. Relay interface. https://docs.kernel.org/filesystems/relay.html

[11] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[12] Loïc Gelle, Naser Ezzati-Jivan, and Michel R Dagenais. 2021. Combining Distributed and Kernel Tracing for Performance Analysis of Cloud Applications. *Electronics* 10, 21 (2021), 2610. https://doi.org/10.3390/electronics10212610

[13] Brendan Gregg. 2017. Performance Superpowers with Enhanced BPF. USENIX Association, Santa Clara, CA.

[14] Chih-Cheng Hung, George A Papadopoulos, Matheus Santana, Adalberto Sampaio, Marcos Andrade, and Nelson S Rosa. 2019. Transparent tracing of microservice-based applications. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (2019). https://doi.org/10.1145/3297280.3297403

[15] Jaegertracing.io. 2022. Jaeger: Open Source, End-to-End Distributed Tracing. http://jaegertracing.io

[16] Madeline Janecek, Naser Ezzati-Jivan, and Seyed Vahid Azhari. 2021. Container Workload Characterization Through Host System Tracing. *IEEE International Conference on Cloud Engineering (IC2E)* 00 (2021), 9–19. https://doi.org/10.1109/ic2e52221.2021.00015

[17] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), 34–50. https://doi.org/10.1145/3132747.3132749

[18] Iman Kohyarnejadfard, Daniel Aloise, Seyed Vahid Azhari, and Michel R. Dagenais. 2022. Anomaly detection in microservice environments using distributed tracing data analysis and NLP. *Journal of Cloud Computing* 11, 1 (2022), 25. https://doi.org/10.1186/s13677-022-00296-4

[19] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* 00 (2020), 48–58. https://doi.org/10.1109/issre5003.2020.00014

[20] Harshal Sheth and Andrew Sun. 2018. Skua: Extending Distributed-Systems Tracing into the Linux Kernel. In *Proceedings of the DevConf.US*. 17–19.

[21] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[22] Meina Song, Qingyang Liu, and Haihong E. 2019. A Mirco-Service Tracing System Based on Istio and Kubernetes. *IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)* 00 (2019), 613–616. https://doi.org/10.1109/icsess47205.2019.9040783

[23] Zipkin.io. 2022. Zipkin. https://zipkin.io