electronics

Article

Combining Distributed and Kernel Tracing for Performance Analysis of Cloud Applications

Loïc Gelle [1], Naser Ezzati-Jivan [2,*] and Michel R. Dagenais [1]

[1] Computer Engineering and Software Engineering Department, Ecole Polytechnique Montreal, Montreal, QC H3T 1J4, Canada; loic.gelle@polymtl.ca (L.G.); michel.dagenais@polymtl.ca (M.R.D.)
[2] Computer Science Department, Brock University, St. Catharines, ON L22 3A1, Canada
* Correspondence: nezzati@brocku.ca

Abstract: Distributed tracing allows tracking user requests that span across multiple services and machines in a distributed application. However, typical cloud applications rely on abstraction layers that can hide the root cause of latency happening between processes or in the kernel. Because of its focus on high-level events, existing methodologies in applying distributed tracing can be limited when trying to detect complex contentions and relate them back to the originating requests. Cross-level analyses that include kernel-level events are necessary to debug problems as prevalent as mutex or disk contention, however cross-level analysis and associating events in the kernel and distributed tracing data is complex and can add a lot of overhead. This paper describes a new solution for combining distributed tracing with low-level software tracing in order to find the latency root cause better. We explain how we achieve a hybrid trace collection to capture and synchronize both kernel and distributed request events. Then, we present our design and implementation for a critical path analysis. We show that our analysis describes precisely how each request spends its time and what stands in its critical path while limiting overhead.

Keywords: performance evaluation; software analysis; multi-level analysis; execution tracing; critical path; root-cause analysis

Citation: Gelle, L.; Ezzati-Jivan, N.; Dagenais, M.R. Combining Distributed and Kernel Tracing for Performance Analysis of Cloud Applications. *Electronics* **2021**, *10*, 2610. https://doi.org/10.3390/electronics10212610

Academic Editor: Christos J. Bouras

Received: 10 September 2021
Accepted: 13 October 2021
Published: 26 October 2021

1. Introduction

Distributed applications have become an obvious choice for deploying services in the cloud. They offer better performance, flexibility and resilience than their monolithic counterparts. However, they often rely on increasingly numerous abstraction and software layers, such as communication and RPC (Remote Procedure Call) middle-ware, Linux containers, virtual machines and different threading and garbage collection mechanisms offered by programming languages. All of these make development, deployment, scaling and maintenance easier while making debugging more complex at the same time. Tracking user requests that span multiple services on different physical machines is also a challenge.

Distributed tracing is a solution that simplifies tracking user requests in distributed applications. It propagates a tracing context, relating to the original user request, along with the distributed nested requests. This allows for sampling, collecting and visualizing the trace information associated with each user request at the cost of instrumenting the chosen communication libraries to propagate the tracing context. However, distributed tracing primarily focuses on collecting high-level user-space events. Therefore, it will provide correct information about the flow and duration of requests but cannot explain why some subrequests are too long when this is caused by operating system level contention—waiting on CPU, disk, network or mutexes.

Software tracing is a better-suited, powerful technique for collecting both kernel-level and user-level events from a system. Software tracers record low-level events into trace files that can then be processed by dedicated analysis tools to diagnose bugs, contentions and latency. Because they collect precise, low-level events such as system calls, they are built for supporting a high throughput of events without disrupting the behaviour of the target application or machine. While this is desirable for the type of analysis targeted, it cannot easily support the more costly tracing context propagation, as distributed tracers would do.

A hybrid approach is necessary for obtaining the advantages of both distributed tracers and software tracers at the same time. In particular, a solution for collecting both distributed and kernel traces has to be devised, without sacrificing the performance of the target application. The traces have to be collected in a way that enables cross-level analysis, which is accurate and useful for performance characterization.

In this paper, we propose and implement a solution that combines distributed tracing with kernel tracing in order to analyze events and more effectively identify the root cause of performance problems. This approach uses the strength of each method: the ability to gain precise information about system interactions and high-level events from distributed tracing and the ability to collect precise and detailed low-level events from kernel and userspace tracing.

With our combined Kernel, userspace and distributed data, we then implement our critical path algorithm. A Critical Path is the longest sequence of execution that occurs during a program's runtime, which, in this case, is represented as a directed acyclic graph generated from events extracted from thread interactions both in Kernel and Userspace. Isolating and analyzing the critical path gives the user an efficient view of a thread/process and its interactions with the system resources and other threads/processes. This highlights the longest path of events and, as such, displays potential latency sources that are important in diagnosing performance problems. In particular, as we see the critical path of threads and processes on the system, we are able to observe contention for resources, for example, lock contention and cpu contention. Using a combined critical path algorithm, we can extend this to the distributed level and observe how competing requests can lead to performance degradations. It also allows us to view these same contentions as they occur between the two layers.

In this work, we used the open-source tools LTTng [1] as a kernel/software tracer, Jaeger [2] as a distributed tracer and Trace Compass (http://tracecompass.org, accessed on 10 May 2021) as a trace analysis tool.

In this paper, we propose a new solution for fine-grained performance characterization of cloud applications using both distributed and kernel tracing. Our contributions are as follows:

- First, we propose and demonstrate a new and efficient architecture for combined trace collection. We evaluate the impact on the performance of the target application.
- Second, we design and implement a new requests critical path analysis that can be used for precise root cause finding.

The rest of the paper is as follows: In Section 2, we review the related work. In Section 3, we describe the newly proposed solution, from trace collection to analysis. Then, in Section 4, we evaluate the overhead of our solution as well as the running time required for the analyses, and we illustrate the usefulness of our solution with a practical use case.

2. Related Work

Previous work on this subject includes studies on both the performance analysis of distributed applications and low-level software tracing.

2.1. Performance Analysis of Distributed Applications

The most generic approaches to performance analysis require no change in the source code of the application. They can either try to infer what the optimal configuration for an application is using machine learning [3] or assess its worst-case performance using controlled fault injection [4,5]. Fault injection can also be used to generate a database of labelled logs, and later, compare the logs corresponding to an unknown failure to that database to find the closest matching fault [6]. Reference [7] studies machine-level scheduling algorithms for improving the use of server resources that can be sub-optimal [8].

Those techniques are limited when it comes to understanding bugs that are more complex or less frequent. The unmodified log files produced by an application can be used for request profiling [9] or extraction of independent user transactions [10]. In a similar way, log files can be processed using machine learning algorithms for latency root cause analysis [11–13]. However, the techniques relying on log files are inherently limited by the quality and quantity of the data. Borderpatrol [14] overcomes this limitation by dynamically loading a patched version of `libc` for a more predictable quality of the log files at the cost of a 10–15% performance penalty.

Application-specific approaches require more or less direct instrumentation of the target application. Annotation techniques [15–17] allow for better log data generation while limiting the modifications to the source code. The annotations are usually destined to a compiler or profiler for generating instrumentation instructions. Dynamic instrumentation is also an alternative when recompiling the target application has to be avoided. Examples of solutions that use this technique include X-ray [18] for precise root cause analysis or Fay [19] for monitoring of distributed applications.

Distributed tracing focuses on end-to-end tracking of individual user transactions. Magpie [20] collects the events corresponding to remote procedure calls to identify causally related transactions and provide a consistent, end-to-end view of a user request. X-Trace [21] assigns a unique identifier to each request and propagates it along the data path while keeping the event collection path distinct. Dapper [22] emphasizes homogeneous systems and limits the instrumentation to a small amount of communication and threading libraries. It allows for extensive data collection while preventing changes in the application's source code. Dapper also implements probabilistic sampling of requests to lower the performance impact of data collection. While trace collection naturally covers the most frequent scenarios well, [23] proposes biased sampling to improve the diversity of the collected traces. The collection of distributed traces can be a basis for research on transaction profiling [24], workflow-specific resource management [25] or live monitoring [26] and probing [27].

2.2. Low-Level Software Tracing

The techniques described above are particularly well adapted to distributed systems. While they focus on context propagation and tracing at scale, they can only collect high-level events, which limits their ability to perform precise root cause analysis in practice. At a lower level, LTTng is a kernel and user-space tracer for Linux with an emphasis on performance, with a measured impact of 2% for typical intense workloads [1]. According to [28], it is the most performant kernel tracer for Linux. Unlike SystemTap [29,30], ftrace [31] and eBPF [32], it does not suffer from a significant increase in latency when used on multicore systems [33]. LTTng saves trace data to disk—or sends it through the network—for offline analysis and relies on the Common Trace Format [34], a standard format built for high trace event throughput. It offers a flight recorder mode that reduces the overhead by flushing the buffered events to the disk only when the user requests a snapshot. Desfossez et al. [35] uses this feature for efficient latency analysis of applications.

Low-level software tracing usually separates the event collection from the analysis that can be performed offline. LTTng traces can be analyzed to recover important metrics, such as CPU usage, with arbitrary precision and without resorting to monitoring and a fixed sampling rate [36]. Trace Compass is a trace analysis tool that relies on an efficient, query-able state system [37] structure. It is built to compute and display the results of complex trace analyses, such as a thread critical path analysis [38].

Some research tries to combine the flexibility of distributed tracing with the precision of kernel tracing. Reference [39] performs vertical context propagation to inject high-level request identifiers into the kernel. A system call is made every time a subrequest is started or finished in the target application, which can badly scale to high throughput applications.

Reference [40] proposes to patch LTTng so that it attaches a request context to every kernel event written in the trace file. The request context is injected into the kernel using a custom dynamic kernel module that adds the information to the current process structure `task_struct`. Both works emphasize the need for explicit synchronization, rather than simple interleaving, between the kernel and distributed tracing events.

Other related work includes two papers [41,42] in which they examined a way to find bottlenecks in multi-threaded applications. This is applicable because the focus was around waiting threads and how they impacted the throughput both locally and globally, and both papers propose different solutions to analyzing the data in a way which respects that. In a similar fashion, we need to keep in mind the waiting times of cloud applications and how they may be misleading as to the root cause of the problem. Another article with some relation to the problem at hand is on the subject of intrusion detection for Internet of Things (IOT) [36], where the goals are very different to that of this paper, but the process is similar, as the approach is based around multi-level tracing on multiple linked devices.

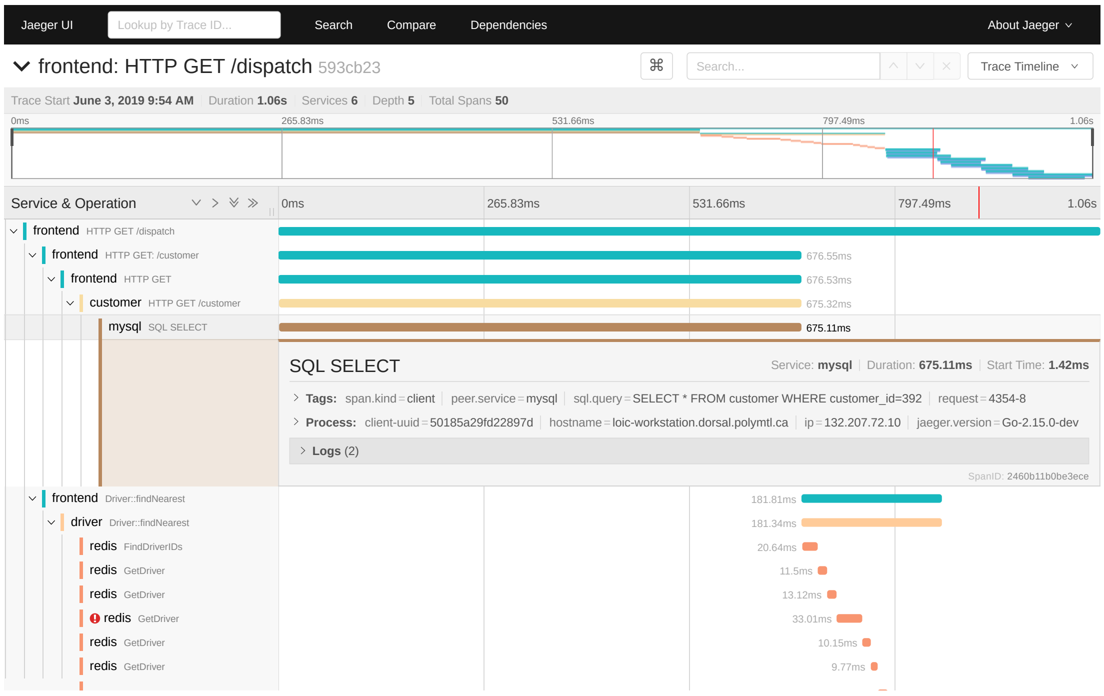
2.3. Critical Path Analysis

Existing work on critical paths analysis is primarily concerned with work on the Kernel and Userspace layers of a single system. Fournier et al. [34] demonstrate the use of critical paths analysis within web applications. This work combines multi-layered critical path analysis with a clustering algorithm to detect unusual events. Similarly, Doray et al. [43] also utilize critical path analysis at both the user and kernel level, with the goal of allowing developers with incomplete knowledge of a system to identify latency and performance issues. Finally, Ezzati-Jivan et al. [42] used critical path analysis in combination with a dependency graph model to analyze all related dependencies required to complete a task, rather than of individual threads/processes. We combine a similar approach to these methods with critical path analysis on distributed tracing to explore the interactions between threads at the distributed interaction level, allowing us to ascertain the location of contentions between two or more requests in a distributed system.

2.4. Discussion

Previous work on distributed systems focuses on performance analysis using high-level information collected at the application level. In particular, distributed tracing proposes a standard way of tracking end-to-end requests to identify bottlenecks among subrequests or machines. As illustrated in Figure 1, distributed tracers provide the user with a unified view of the flow of a given request. However, they cannot explain why a particular subrequest is longer than expected, especially when this is caused by operating system level contention.

The related work discussed above shows that kernel traces can be processed by specialized algorithms to identify complex bottlenecks and relate them to user processes and threads. Although some research work has focused on propagating distributed tracing events into kernel traces, no effort, to the best of our knowledge, has used distributed and kernel tracing together to improve root cause analysis. We differentiate our work by enhancing the root cause analysis with user-level traces and use it to capture low-level contention. We then propose an extended view of requests that can show and explain the low-level contention, improving the user's ability to diagnose and fix bottlenecks and other issues. In particular, we propose a critical path analysis of requests, as shown in Figure 2, and explain our solution to obtain this analysis in the next section of this article. Table 1 compares our contributions with a selection of related works.



frontend: HTTP GET /dispatch 593cb23

Trace Start **June 3, 2019 9:54 AM** Duration **1.06s** Services **6** Depth **5** Total Spans **50**

```
frontend  HTTP GET /dispatch
frontend  HTTP GET: /customer        676.55ms
frontend  HTTP GET                   676.53ms
customer  HTTP GET /customer         675.32ms
mysql     SQL SELECT                 675.11ms
```

SQL SELECT — Service: **mysql** | Duration: **675.11ms** | Start Time: **1.42ms**

Tags: span.kind = client | peer.service = mysql | sql.query = SELECT * FROM customer WHERE customer_id=392 | request = 4354-8

Process: client-uuid = 50185a29fd22897d | hostname = loic-workstation.dorsal.polymtl.ca | ip = 132.207.72.10 | jaeger.version = Go-2.15.0-dev

Logs (2)

SpanID: 2460b11b0be3ece

```
frontend  Driver::findNearest        181.81ms
driver    Driver::findNearest        181.34ms
redis     FindDriverIDs              20.64ms
redis     GetDriver                  11.5ms
redis     GetDriver                  13.12ms
redis     GetDriver                  33.01ms
redis     GetDriver                  10.15ms
redis     GetDriver                  9.77ms
```

Figure 1. Sample distributed trace shown by *Jaeger*. The events corresponding to a user request can be collected and processed by the distributed tracer *Jaeger*. The view shows the complete flow of subrequests, along with the causal links between them, their duration and their log events.

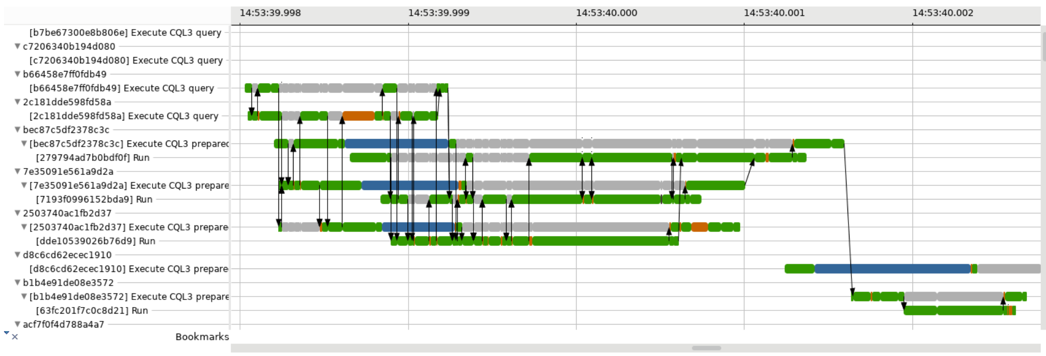

14:53:39.998
14:53:39.999
14:53:40.000
14:53:40.001
14:53:40.002
[b7be67300e8b806e] Execute CQL3 query
c7206340b194d080
[c7206340b194d080] Execute CQL3 query
b66458e7ff0fdb49
[b66458e7ff0fdb49] Execute CQL3 query
2c181dde598fd58a
[2c181dde598fd58a] Execute CQL3 query
bec87c5df2378c3c
[bec87c5df2378c3c] Execute CQL3 preparec
[279794ad7b0bdf0f] Run
7e35091e561a9d2a
[7e35091e561a9d2a] Execute CQL3 prepare
[7193f0996152bda9] Run
2503740ac1fb2d37
[2503740ac1fb2d37] Execute CQL3 prepare
[dde10539026b76d9] Run
d8c6cd62ecec1910
[d8c6cd62ecec1910] Execute CQL3 prepare
b1b4e91de08e3572
[b1b4e91de08e3572] Execute CQL3 prepare
[63fc201f7c0c8d21] Run
acf7f0f4d788a4a7
Bookmarks

Figure 2. View of our requests critical path analysis in Trace Compass. Each state has a different colour—green for `RUNNING`, orange for `PREEMPTED`, grey when blocked by another request, etc.—and the arrows indicate how the different requests intersect in their respective critical path. Instead of showing a single request only, our view includes all the requests that happened in parallel.

Table 1. Comparison with selected related works.

Work	Distributed Tracing	Kernel Tracing	Critical Path Analysis
[1]	no	yes	no
[3]	yes	no	no
[4,5]	yes	no	no
[18]	yes	yes	no
[19]	yes	yes	no
[34]	no	no	yes
[39]	yes	yes	no
[42]	no	yes	yes
This	yes	yes	yes

3. Proposed Solution

In this section, we present our methodology and contributions in greater detail. The newly proposed solution encompasses trace collection, analysis and visualization.

3.1. Instrumenting the Distributed Tracer

One of the key challenges in the problem studied is bridging the gap between:

- high-level, task-focused, microsecond-precise distributed traces on the one hand;
- low-level, event-focused, nanosecond-precise kernel traces on the other hand.

More specifically, integrating distributed traces into our analyses requires being able to improve the precision of the highest-level events. Let us suppose, for example, that we managed to get the event for the beginning of a distributed task right at the nanosecond precision. Then we can use the events from the kernel trace corresponding to context switches between threads to infer which thread had started the request. In a different scenario with lesser precision on the request start event, our analysis could be misled into inferring different thread information and then compute the wrong critical path for that request. Figure 3 illustrates the two scenarios. The correlation between a request and the executing thread could be obtained by other means, such as obtaining the thread identifier and including it as an event context field in the distributed tracer. This would require more changes to the distributed tracer and a higher overhead to obtain the thread identifier and store it in events.

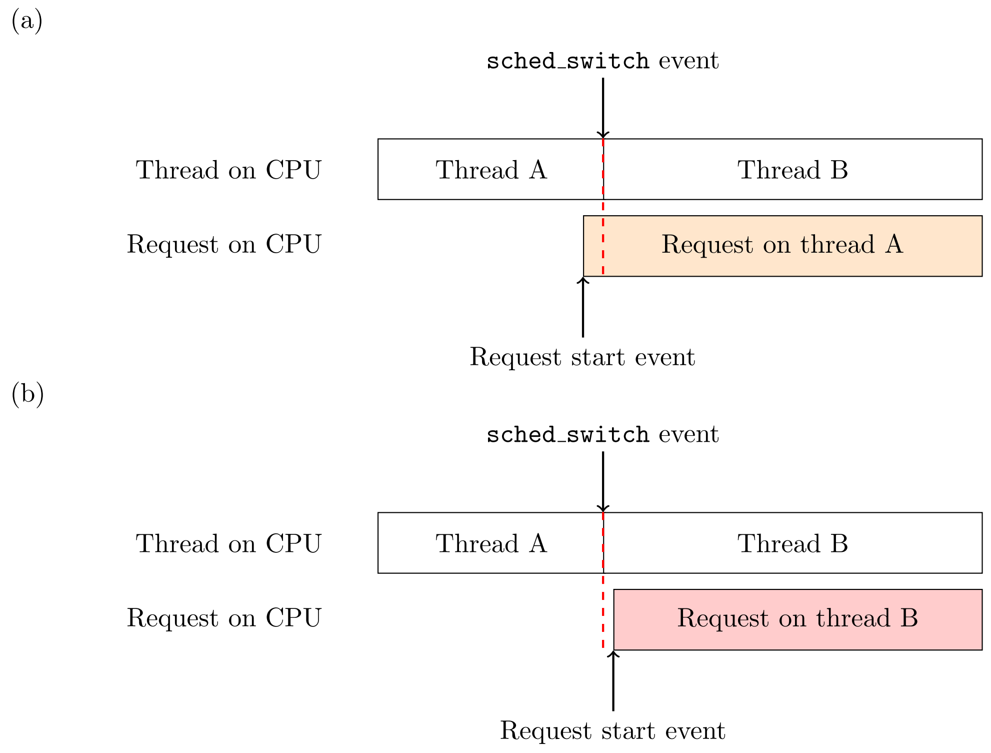


Figure 3. Impact of the precision of timestamps on trace analysis. The role of an event-driven analysis is to reconstruct the state of a machine at any given time using the sole trace events. Different event sequencing can yield different analysis results. In our case, insufficient precision on the distributed tracing events can impact the analysis depending on whether the request starts (**a**) right before or (**b**) right after a scheduling event.

The different techniques for traces synchronization discussed in Section 2 are based on vertical context propagation, and the cost of switching into kernel mode each time an event is emitted by the distributed tracer. Our solution avoids the overhead of vertical context propagation by collecting a minimal amount of synchronization events while staying in user-space mode. We use the LTTng tracer for collecting both kernel and user-space traces from applications. Because LTTng uses the same clock for kernel and user-space events, all the events emitted by a user application will be synchronized with the kernel trace without any additional work needed. Therefore, the only application that requires instrumentation is the distributed tracer in-process agent. Our technique relies on the instrumentation of the distributed tracer itself with LTTng so that it emits a user-space event each time a span is started or finished by the distributed tracer. The event carries the minimal information required for traces synchronization, namely the unique request identifier assigned by the distributed tracer. We implemented our solution on the distributed tracer Jaeger, more specifically for its Java and Go clients. Figure 4 describes how our solution can be deployed on a single host to collect at the same time, with distributed traces using Jaeger and kernel and user-space traces using LTTng. Figure 5 shows an example of synchronization events that are collected using our patched version of Jaeger.

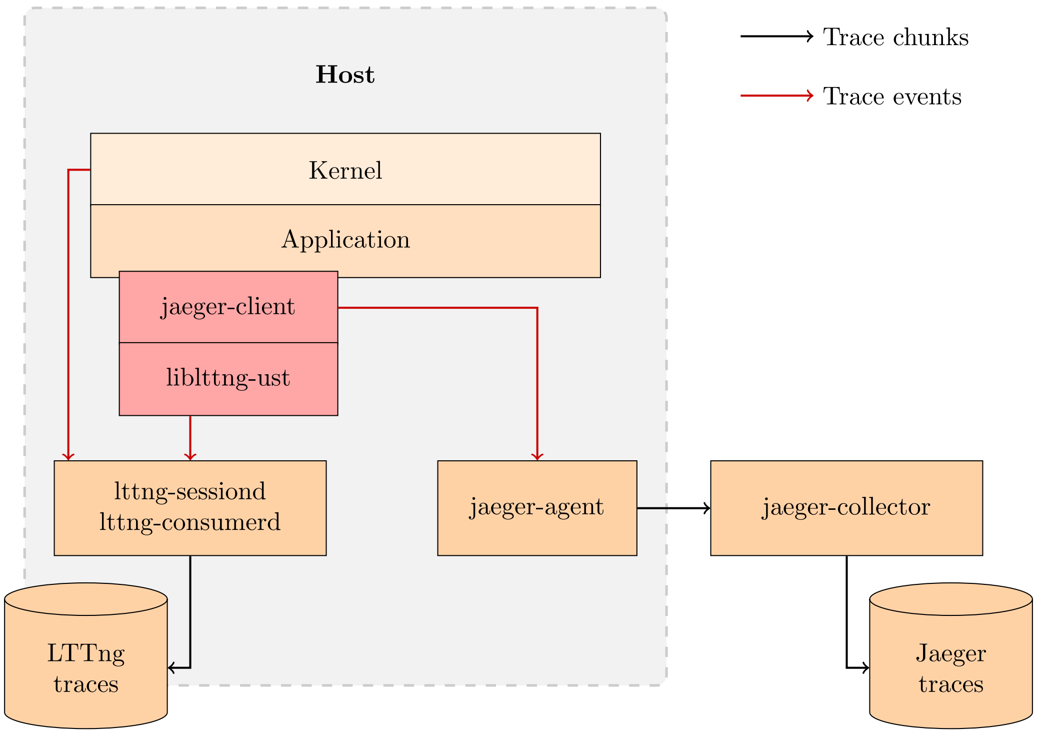


Figure 4. Design of our trace collection solution. Our solution targets a single host and is able to collect kernel traces using LTTng and distributed traces using Jaeger. The in-process Jaeger client is patched to emit LTTng user-space events and has to be compiled against `liblttng-ust`. The application code is unmodified, but it has to be compiled against the patched Jaeger client.

```
[12:46:59.671901514] (+0.000001098) loic-workstation jaeger_ust:start_span: cpu_id = 5,
trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4421066052755260657
[12:46:59.672049028] (+0.000019061) loic-workstation jaeger_ust:start_span: cpu_id = 5,
trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4146236183556049419
[12:47:00.069919803] (+0.000000349) loic-workstation jaeger_ust:end_span: cpu_id = 9,
trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4146236183556049419
[12:47:00.492785641] (+0.000000988) loic-workstation jaeger_ust:end_span: cpu_id = 1,
trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4421066052755260657
```

Figure 5. Sample synchronization events collected using our solution. Our solution collects events in the user-space mode for synchronization between distributed traces and kernel traces. Examples of such events are shown here, as displayed by the Babeltrace trace reader. Each event carries (i) nanosecond-precise absolute and relative timestamps, (ii) the name of the source server, (iii) the name of the event, (iv) the source CPU identifier and (v) the event payload.

It should be noted that some events will be present both in the distributed traces and the LTTng traces. This allows for more flexibility and adaptability to the targeted deployment. For example, in a large distributed application, it can be interesting to use Jaeger on every node but LTTng on specifically chosen targets. We then get self-sufficient distributed traces, regardless of LTTng tracing, but benefit from the analyses of complementary traces, as described below, on specific nodes of interest.

3.2. Synchronizing the Traces

The Trace Compass trace analysis tool can read traces from both LTTng and Jaeger. The Jaeger traces describe every request as JSON data, including their start timestamp, duration, attached logs and parent request reference. Using every LTTng event that corresponds to the beginning or end of a request, we can correct the timestamps present in the Jaeger traces into nanosecond-precise ones.

The challenging part is determining, for each request, the thread in which it is being executed at any given moment. It is fairly easy to infer the threads corresponding to the beginning and end of the request by using the scheduling events from the kernel trace. However, it is possible for a request to be handled by multiple threads in sequence. This is particularly true for asynchronous applications that rely on thread pools or pipelines to process queued tasks. For the sake of simplicity, we choose to associate a single thread to each sub-request, namely the thread that started the request. We then consider the critical path of that thread to be the critical path of the corresponding request. In the case of requests being migrated between threads, we would need more precise instrumentation to integrate the migrations into our analyses. For example: We could acquire a unique identifier that can be used to identify a thread (for example, a request id) and then construct the path after runtime from each machine or thread the request is detected on. As this would only require a change in the instrumentation and in trace collection, rather than the architecture of our system, we feel that the work is transferable to complex scenarios. See Section 5.

Conversely, we need to ensure that a thread is only associated with a single request at any given time. When there are multiple possible requests associated with the same thread, we enforce that rule by marking the request that started being active most recently. The full algorithm to associate threads to requests is shown in Algorithm 1. As shown on line 14, if the stack of associated requests is greater than zero, then the active or most recently active request is assigned by taking the top element of the request stack.

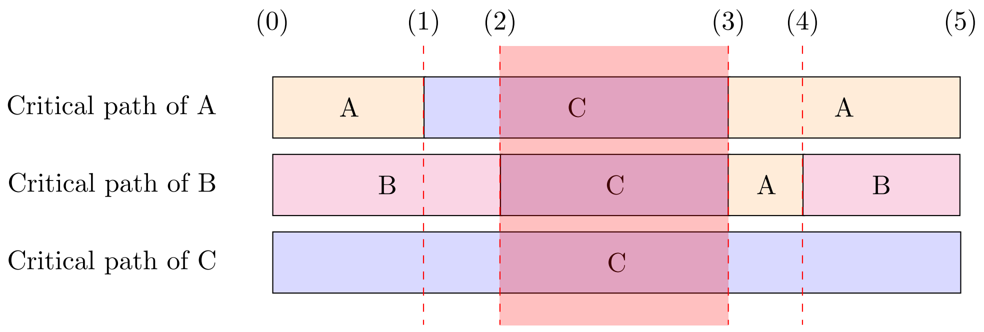
3.3. Algorithm for Multi-Level Critical Path Analysis

The Trace Compass includes a thread critical path analysis [38]. For any thread of interest to the user, the analysis computes its longest path of waiting after resources. It uses low-level events, such as context switches or locking of mutexes, to infer the cause of threads being blocked. The output of the analysis is a succession of the execution states that contribute to the latency of the target thread. Arrows in the output indicate blockage or wake-up links between threads.

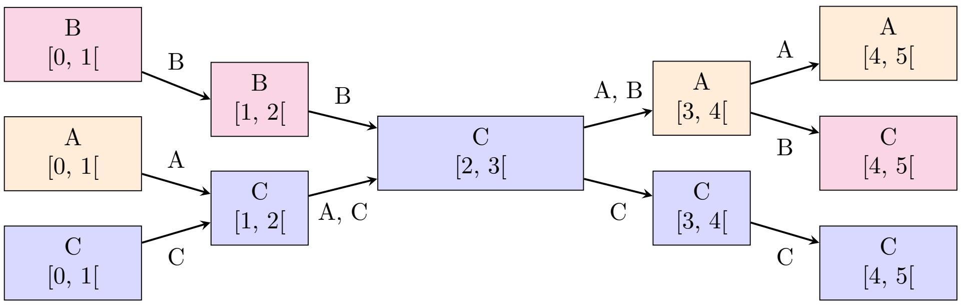
Since we aim at extending the critical path analysis to user-level requests, it is key to understand how the critical paths of different threads interact so that we can understand how the critical paths of different requests interact as well. It is indeed possible for two different threads to share a state, i.e., contend for the same resource at the same time, in their respective critical paths. That happens, for example, when two threads are waiting on the same mutex held by a third thread; in that case, the critical paths of all three threads could reflect that contention by having a shared state, as explained in Figure 6.

The critical path provided by the Trace Compass is actually a linked list, in which each node is either:

- an interval that has a starting time, duration, state and an associated thread identifier;
- an arrow that has a starting time, duration, state, source thread identifier and a target thread identifier.



(**a**)



(**b**)

Figure 6. Shared states between different thread critical paths. The critical paths of different threads can share one or more states (**a**). In this example, the thread C initially holds a mutex M and threads A, B and C are running. Each block indicates which thread is on the critical path of the thread of interest. (Solid lines indicate requesting, acquiring or releasing a lock on a resource). The critical paths are built according to the sequence of the following events: (0) trace begins, (1) A requests locking M, (2) B requests locking M, (3) C releases M and A locks M, (4) A releases M and B locks M, and (5) trace ends. (It is important to note that execution continues even after releasing a lock). The red overlay in subfigure (**a**) highlights the interval during which all threads have the same state in their respective critical path. Our critical path merging algorithm outputs the graph in subfigure (**b**). Each graph node indicates which thread the state refers to, as well as the time interval of the state. The critical path of a given thread can be recovered back from the graph by following the edges labelled with that thread name.

Depending on the status of a thread and the resources it could be waiting for at a given time, the Trace Compass will determine the state of a node to be `RUNNING`, `INTERRUPTED`, `PREEMPTED`, `TIMER`, `BLOCK_DEVICE`, `USER_INPUT` or `NETWORK`.

The previous work on thread critical path analysis is dedicated to kernel trace data only. The data structure chosen for the thread critical path is not adapted to cross-level analyses because it does not clearly identify states that are linked between different critical paths. The challenge in combining distributed traces with kernel traces in a critical path analysis lies in handling these shared states correctly.

We are addressing this problem by proposing a new algorithm that can identify contention on a shared resource between an arbitrary number of critical paths into a graph that includes them all. The goal is to have a single node in the output graph when it corresponds to contending for the same resource at the same time in different critical paths. A representation of the output of our algorithm is given in Figure 6. More specifically, the nodes are Java objects for which we provide a simplified structure in Figure 7.

The two hash tables in the node structure allow for the traversal of the graph in any direction. The key is the thread identifier of the critical path—an `Integer` object in Trace Compass—that connects the current node with the node associated with that key. The node also stores the node state—a `String`—which is the thread identifier the state refers to: the start timestamp and the end timestamp.

```
class Node {
        private HashMap<Integer, Node> incomingEdges;
        private HashMap<Integer, Node> outgoingEdges;
        private String state;
        private Integer stateTid;
        rivate long startTs;
        private long endTs;
}
```

Figure 7. Simplified version of the Java data structure used for our critical path nodes. A node from the graph obtained by merging critical paths has all the information needed to traverse the graph for any thread of interest.

Algorithm 1: Pseudo-code for the proposed synchronization algorithm. The algorithm associates each thread to the sequence of requests it is actively handling. Its output is used by the extended critical path view discussed in Section 3.4.

```
Input: requestEvents: a list of requestsSet
Output: allRequests: associates each TID to a mapping from intervals to a set of
  request UIDs corresponding to that interval;
activeRequests: associates each TID to a mapping from intervals to a unique active
  request corresponding to that interval
FUNCTION MAIN()
for e in requestEvents do do
    HANDLEEVENT(e, lastTimestamps, requestStacks)
end
FUNCTION HANDLEEVENT(e, lastTimestamps, requestStacks)
tid ← e.tid timestamp ← e.timestamp
requestUID ← e.requestUID
requestsStack ← requestsStacks[tid]
lastTimestamp ← lastTimestamps[tid]
if requestStack is not empty then
    activeRequests[tid][lastTimestamp → ts] ← top element of requestsStack
    allRequests[tid][lastTimestamp → ts] ← all elements of requestsStack
end
lastTimestamps[tid] ← timestam p
if e is a start event then
    remove requestsStack[requestUID]
end
else
    push spanUID to requestStack
end
```

A simplified version of our critical paths merging algorithm is shown in Algorithm 2. It gives a general idea of the approach that we are taking while staying relatively high-level. For the sake of completeness, it can be noted that most of the data structures that we actually use are `RangeMap` type in Java. A `RangeMap` is essentially a hash table in which keys are ranges, which allows for querying by intervals in logarithmic time. Those data structures get updated as the algorithm runs so that their state consistently reflects the current data in the graph being built. From a high-level perspective, the algorithm loops over the states of the critical paths of interest (line 9) to add each of them to the graph. Adding a state to the graph often requires splitting an existing state into two or three states while updating the edges of the graph accordingly and then merging the duplicate states (i.e., contending for the same resource). The complete version of our algorithm keeps track of the head of the graph for each thread—that is, the first state in chronological order to be in that thread critical path—allowing for targeted traversal of the graph.

Algorithm 2: Pseudo-code version of the critical paths merging algorithm.

```
Input: threadsOfInterest: list of integers
Output: graphNodes: set of graph nodes
FUNCTION MAIN()
for tidOfInterest in threadsOfInterest do
    criticalPath ← COMPUTECRITICALPATH(tidOfInterest)
    for state in criticalPath do
        MERGE(state)
    end
end
FUNCTION MERGE(state)
intersectingStates ← INTERSECT(graphNodes, state)
timestamp ← e.timestamp
for existingState in intersectingStates do
    MERGESTATES(existingState, state)
end
FUNCTION MERGESTATES(state1, state2)
if state1 starts before state2 then
    newState ← CREATESTATE(state1.start, state2.start)
    REPLACELEFT(state1, newState)
    MERGESTATES(state1, state2)
    return
end
else if state1 starts after state2 then
    newState ← CREATESTATE(state2.start, state1.start)
    REPLACELEFT(state2, newState)
    MERGESTATES(state1, newState)
    return
end
else if state1 finishes before state2 then
    newState ← CREATESTATE(state1.start, state1.start)
    REPLACELEFT(state2, newState)
    MERGESTATES(state1, newState)
    return
end
else if state1 finishes after state2 then
    newState ← CREATESTATE(state2.tart, state1.start)
    REPLACELEFT(state1, newState)
    MERGESTATES(state2, newState)
    return
end
FUNCTION REPLACELEFT(state1, state2)
state1.start ← state2.end
for tid in state1.incomingEdges do
    state1.outgoingEdges[tid] ← state2
end
state2.incomingEdges ← state1.incomingEdgesstate1.incomingEdges ← ()
for tid in state2.incomingEdges do
    state1.incomingEdges[tid] ← state2 state2.outgoingEdges[tid] ← state1
end
state2.state ← state1.state
```

3.4. Extending the Critical Path to the Requests

We propose an analysis that extends the critical paths of threads into critical paths of Jaeger requests. The goal is to have, for each request, a critical path that describes:

- how the critical path of the request intersects with the critical paths of other requests: at a high level, which would mean that the request is waiting on another request because of a contention between their underlying threads;
- the states of the threads handling the request at a lower level.

We achieve this goal by putting together the algorithms shown in Sections 3.2 and 3.3. After the traces are processed using these algorithms, we obtain the following elements:

- a graph representing the merged critical paths of all the threads of interest;
- for each thread, a data structure queryable by time range that maps time intervals to the unique identifier of the request active on that thread during that interval;
- for each thread, a data structure queryable by time range that maps time intervals to a set of unique identifiers of the requests processed by that thread during that interval.

Our analysis does a depth-first traversal of the critical paths graph. The critical paths of the requests are built incrementally by adding states to them while traversing the merged critical paths graph. For each state of the latter, we use our queryable map structures to determine the active request identifier corresponding to the time interval of the state. The state is simply copied to the critical path of that active request without modification. We then create a state in the critical path of each inactive request handled by the same thread. This state indicates that the request is currently blocked by the active request that we identified earlier.

3.5. Summarizing the Critical Paths

Because a kernel-level critical path is very detailed and low-level, it is sometimes useful to have a first look at a more synthetic piece of information. We propose a simple analysis that summarizes the critical path of a given request into the total time spent in various states: blocked by a concurrent request, waiting on CPU time, running, waiting on the network, etc. The summary fits into a pie chart and can help identify the root cause of a significant latency quickly.

Using the same approach, it is possible to extend our solution into additional views that rely on the analyses shown above. For example, one could find it interesting to fetch the number of failed system calls in a distributed request or the variation of the density of disk accesses during a transaction.

4. Evaluation

In this section, we evaluate the overhead of our trace collection solution, as well as the time required for running our analyses. We then demonstrate the usability of our implementation through a practical use case.

When analyzing any tracing solution, it is important to carefully cover the overhead introduced on the original system. This is because too much overhead introduced can both obscure the cause of existing latencies and introduce new ones. In order to evaluate the performance overhead of the individual systems and to see how they behave at different loads and configurations, we have conducted this required step.

We have implemented the analyses described above in Java in the trace analysis tool Trace Compass. We have instrumented the Java and Go clients of Jaeger to add LTTng tracepoints for synchronization purposes. Our solution is then tested on two representative distributed applications.

1. HotROD (https://github.com/jaegertracing/jaeger/tree/master/examples/hotrod accessed on 23/04/2021) is a demo application for Jaeger that is already instrumented. It consists of multiple services, including a web front end, a Redis server and a MySQL database. The user can make requests to order virtual car rides. We simulate loads

of concurrent requests using the Apache `ab` tool. We slightly modified HotROD to remove all simulated processing delays present in the demo application.

2. Cassandra (http://cassandra.apache.org/ accessed on 23/04/2021) is a performance-oriented, NoSQL distributed database. We simulate loads consisting of concurrent write or read requests using the `cassandra-stress` tool. We have instrumented Cassandra so that it emits Jaeger events for each processed request. Because Cassandra can handle a very high throughput of requests, it is likely to generate a lot more tracing data and thus be more impacted by our trace collection solution.

In both cases, the application, the Jaeger services and the LTTng daemon execute on the same physical machine to simplify the evaluation. The machine is equipped with an Intel® Core™ i7-7820X processor (8 physical cores, 16 virtual cores, 3.60 GHz base frequency) and 32 GB of RAM.

4.1. Trace Collection Overhead

We show that our solution adds a reasonable overhead when compared to only collecting Jaeger traces in an application while generating a significant amount of kernel and synchronization events. We measure the average request processing time for both HotROD and Cassandra in the five following scenarios.

- A-1. No tracing: Both Jaeger and LTTng are turned off: no traces are collected.
- B. Jaeger only: This is the baseline to which our solution will be compared: only Jaeger traces are collected. Sampling is off for HotROD—meaning that all the requests are traced—while only 10% of the requests are sampled for Cassandra. This is to avoid a huge overhead that would be very unlikely in a production setting for Cassandra. As all other scenarios use the following setup to collect trace data, and because this section is concerned with the overhead introduced by the tools in addition to Jaeger, rather than comparing the overhead of tracing vs. not tracing and their interactions, we use this as our Baseline rather than A-1.
- C-1. Full (snapshot mode): This is the first configuration for our solution. Jaeger traces are collected with a sampling configuration identical to the previous scenario. LTTng is also running to collect synchronization events and a subset of kernel events required for most analyses in Trace Compass. LTTng is running in *snapshot* mode, meaning that the events are kept in circular buffers in memory until the user requests them to be flushed to a trace file. This is a production-friendly mode of LTTng that allows collecting an actual trace only when an anomaly is detected. The performance is better than in standard mode, and the size of the trace files can be controlled more easily. In this scenario, we make sure to never trigger a trace flush and to simulate passive events recording with no anomaly. Because we never write the events to disk, they keep being overwritten by newer ones when the circular buffer is full.
- C-2. Full (standard mode): This is the second configuration for our solution. It is identical to the first one, except that LTTng is run in standard mode. The events that are collected are all written to a trace file.
- D-1. Fake syscall (snapshot mode): We implemented the solution proposed by [39]. Instead of instrumenting the distributed tracer with dedicated synchronization events collected in user-space, we modify it so that it emits a `getpid` system call for every request, beginning or end. Unlike a normal `getpid` system call, we add the request unique identifier as an argument that will be written to the trace buffers. LTTng is running in snapshot mode.
- D-2. Fake syscall (standard mode): This scenario is identical to the previous one, but LTTng is running in standard mode.

We include an additional scenario for Cassandra only:

- A-2. Integrated tracing: Cassandra is configured to trace all the requests using its built-in logging feature. All the tracing events are printed to a file.

The average request processing time is an important metric because it directly impacts the throughput of the target application. For distributed applications that are used in production, it is key that the trace collection does not significantly reduce the throughput of requests.

4.1.1. Results for HotROD

We executed a total of 10,000 requests using 10 client threads for each of the above scenarios. The average processing times are summarized in Figure 8. The relevant overheads between scenarios are given in Table 2.

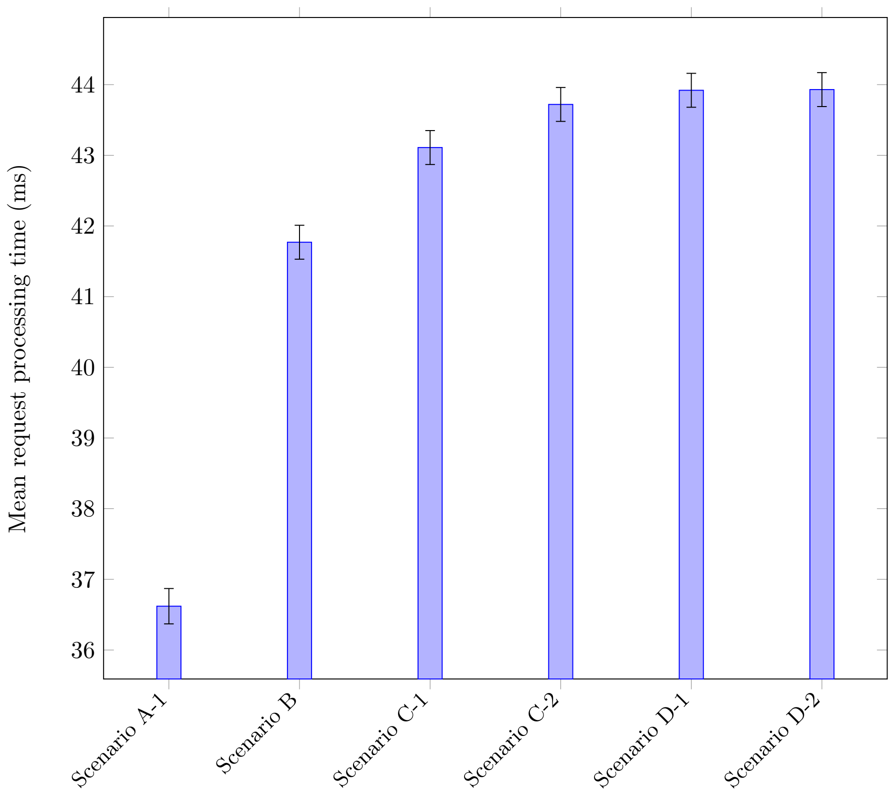


Figure 8. Request processing times for HotROD using different tracing scenarios. The two configurations corresponding to our solution are scenarios C-1 and C-2, while the baseline—tracing with Jaeger only—is scenario B. For reference, scenario A-1 corresponds to disabling all tracing. Scenarios D-1 and D-2 are two configurations for the fake syscall technique that we implemented for comparison.

Table 2. Overhead of our trace collection solution for HotROD. The overhead of our solution compared to the baseline is below 4% when using the snapshot mode in LTTng and around 5% when using the standard mode.

Scenario **Reference**	**B**	**C-1**	**C-2**	**D-1**	**D-2**
A-1	14.1 ± 0.6%	17.7 ± 0.6%	19.4 ± 0.7%	19.9 ± 0.7%	20.0 ± 0.7%
B (baseline)		3.2 ± 0.6%	4.7 ± 0.6%	5.2 ± 0.6%	5.2 ± 0.6%
C-1				1.9 ± 0.6%	
C-2					0.5 ± 0.5%

Even when tracing all the requests with Jaeger and LTTng, the overhead of our solution compared to tracing with Jaeger is only about 5% and is below 4% when using the snapshot mode from LTTng. This is comparable to other tools, such as Magpie [20], which achieves approximately 4%, or X-trace, which decreases the total system throughput by approximately 15% [21]. This overhead allows using our solution in a production environment without a huge impact on the application throughput. The fake syscall technique performs slightly worse than our solution, though without adding a significant penalty.

4.1.2. Results for Cassandra

The tool `cassandra-stress` can send read or write requests to a Cassandra server. Because they lead to workloads of different natures, it is interesting to benchmark our solution for both. We executed a total of 1,000,000 read requests and then 1,000,000 write requests using 20 client threads for each scenario. The average processing times are summarized in Figure 9. The relevant overhead for each scenario is given in Table 3 for read loads and in Table 4 for write loads.

Table 3. Overhead of our trace collection solution for high-throughput read workloads in Cassandra. The overhead of our solution compared to the baseline is below 18% when using the snapshot mode in LTTng and below 27% when using the standard mode.

Reference \ **Scenario**	**A-2**	**B**	**C-1**	**C-2**	**D-1**	**D-2**
A-1	41.0 ± 1.1%	117.7 ± 1.7%	156.4 ± 1.7%	175.0 ± 1.9%	199.7 ± 2.2%	222.8 ± 2.4%
A-2		54.3 ± 0.5%	81.8 ± 0.4%	95.0 ± 0.4%	112.5 ± 0.6%	128.9 ± 0.7%
B (baseline)			17.8 ± 0.2%	26.3 ± 0.3%	37.7 ± 0.4%	48.3 ± 0.4%
C-1					16.9 ± 0.17%	
C-2						17.4 ± 0.20%

Table 4. Overhead of our trace collection solution for high-throughput write workloads in Cassandra. The overhead of our solution compared to the baseline is below 27% when using the snapshot mode in LTTng and around 29% when using the standard mode.

Reference \ **Scenario**	**A-2**	**B**	**C-1**	**C-2**	**D-1**	**D-2**
A-1	61.8 ± 1.7%	139.0 ± 1.9%	202.3 ± 2.4%	208.7 ± 2.2%	231.8 ± 2.6%	241.4 ± 2.5%
A-2		47.7 ± 0.7%	86.8 ± 0.9%	90.7 ± 0.8%	105.0 ± 1.0%	111.0 ± 0.9%
B (baseline)			26.5 ± 0.3%	29.1 ± 0.3%	38.8 ± 0.3%	42.8 ± 0.3%
C-1					9.8 ± 0.23%	
C-2						10.6 ± 0.07%

This time, the performance impact of our solution is more significant, especially because Cassandra has a very high throughput and we are sampling 10% of the requests. Compared to tracing with Jaeger only, our solution adds a lot of kernel and synchronization events to the trace at the cost of a 18–27% performance penalty. Using our solution in a production setting would require lowering the sampling by several orders of magnitude and using LTTng in snapshot mode instead of standard mode.

Because of the throughput of tracing events that trigger a lot of switching into kernel mode, the performance is even worse with the fake syscall technique, which adds a 17–18% overhead to our solution.

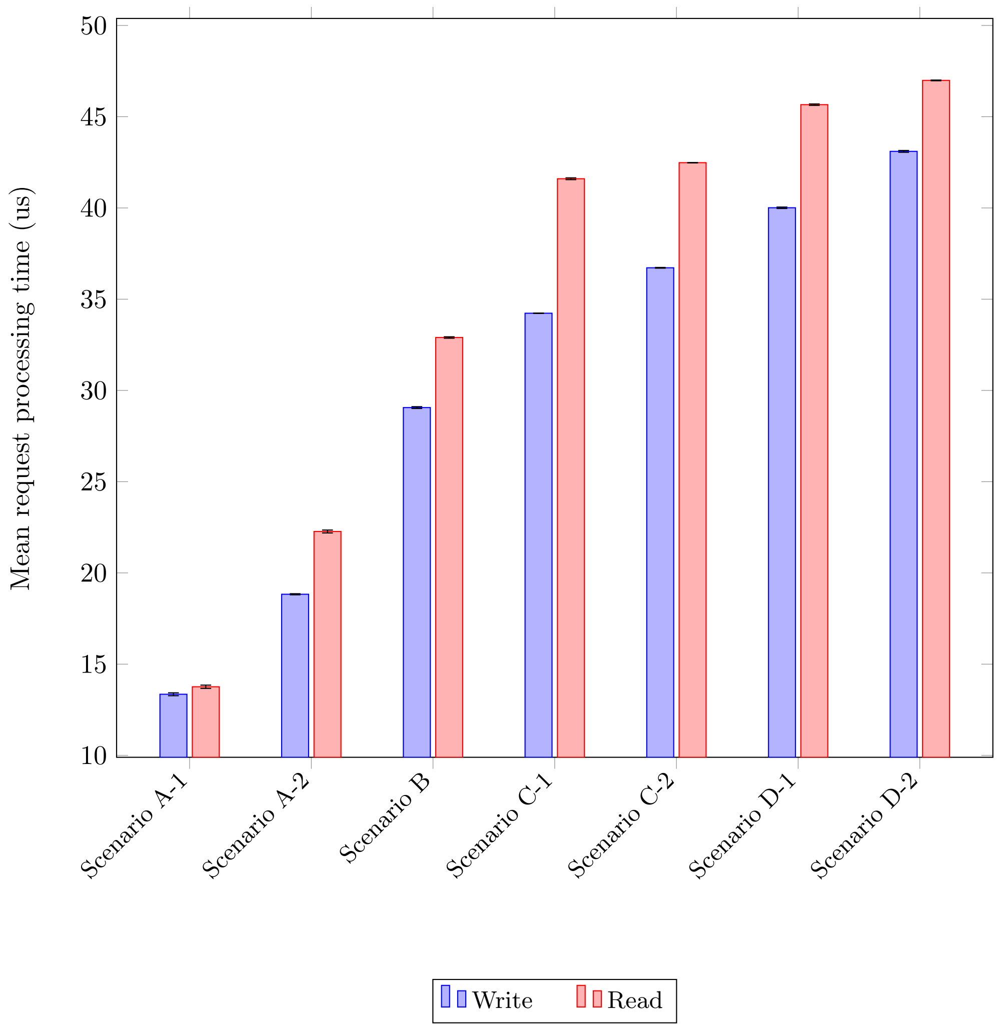


Figure 9. Request processing times for Cassandra using different tracing scenarios. The two configurations corresponding to our solution are scenarios C-1 and C-2, while the baseline—tracing with Jaeger only—is scenario B. For reference, scenario A-1 corresponds to disabling all tracing and scenario A-2 to using the built-in request logging feature in Cassandra. Scenarios D-1 and D-2 are two configurations for the fake syscall technique that we implemented for comparison. For each scenario, we distinguish between read and write workloads.

4.2. Trace Analysis Duration

It is interesting to get a sense of the time spent analyzing the traces. We created a set of test traces of different sizes and contents using `cassandra-stress`. We then run our analyses several times on each trace and record the average running time and confidence interval around the average, as given by the Java JUnit testing tool. The results are described in Figure 10.

Interestingly, the time required to perform our extended critical path analysis alone is very close to the full analysis time—that includes the kernel analysis from Trace Compass, along with our analysis—meaning that the parallelization in Trace Compass is almost perfect. The full analysis takes about 15 s for traces as big as 500 MB, which corresponds to 20–30 s of tracing data.

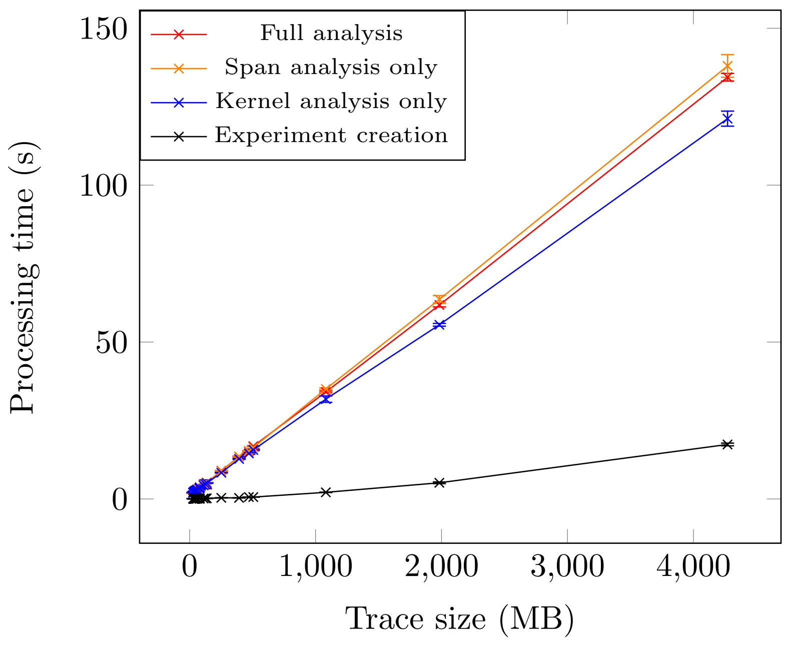


Figure 10. Execution time of our critical path analysis of requests. Considering the test traces used, our analysis seems to scale linearly with the trace size. The execution time is very reasonable for moderately large traces.

4.3. Use Case: Logical CPU Pressure Using Control Groups

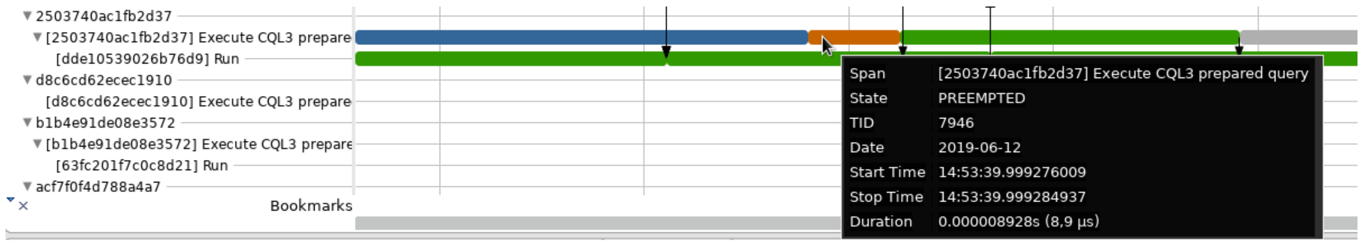
We demonstrate the usefulness of our solution on a practical use case, for which we generate a precise fault in a controlled environment. We show how our implementation can help identify the faults by combining distributed tracing events and kernel events in our analyses.

Our use case involves a bad configuration of a `cpu` control group. Control groups are a very popular mechanism, being an important building block for Linux containers. They allow for limiting the logical or physical resources allocated to a given group of processes. The `cpu` control group, for example, can limit the CPU usage for a set of processes to a fixed number of cycles per second. This use case is especially interesting since CPU contention is a frequent source of problems and is difficult to diagnose without kernel tracing. Furthermore, using control groups to create contention is convenient for experimentation while being more difficult to diagnose than more common sources of contention (e.g., competing threads).

We run Cassandra and put all its threads into a `cpu` control group while collecting trace events with Jaeger and LTTng. As we put Cassandra under a constant load, we limit the CPU usage for that control group to 1% only of the available CPU time and then waive the limit a few seconds later.

When we open the trace in Trace Compass and run our analyses, we end up getting the critical path for all of the requests. As shown in Figure 11, the tooltips in our analysis provide more information about a given state as well as state changes over time, allowing the user to track the state's progress through the critical path.

We can easily identify the abnormal latency in the trace. Figure 12 shows that some identical requests have very different processing times, despite each request being expected to take a similar amount of time to complete. During the time where the latency is huge, which we know corresponds to the control group limit that we set, we can observe in Figure 13 that the threads in Cassandra seem to share very scarce CPU resources, with each thread in the graph displayed as orange (waiting for cpu) or blocked and small areas of green where it is able to execute. Figure 14 shows that the CPU becomes significantly underused when the control group limit kicks in. When we zoom in on a longer request (see Figure 15), we can see that it gets preempted every 100ms for the same amount of time. This information is also available from our summarized critical path view shown in Figure 16, where the PREEMPTED state takes up the majority of the time.



Span	[2503740ac1fb2d37] Execute CQL3 prepared query
State	PREEMPTED
TID	7946
Date	2019-06-12
Start Time	14:53:39.999276009
Stop Time	14:53:39.999284937
Duration	0.000008928s (8,9 µs)

Figure 11. Use case: tooltips. The tooltips in our analysis allow for getting more information about a given critical path state.

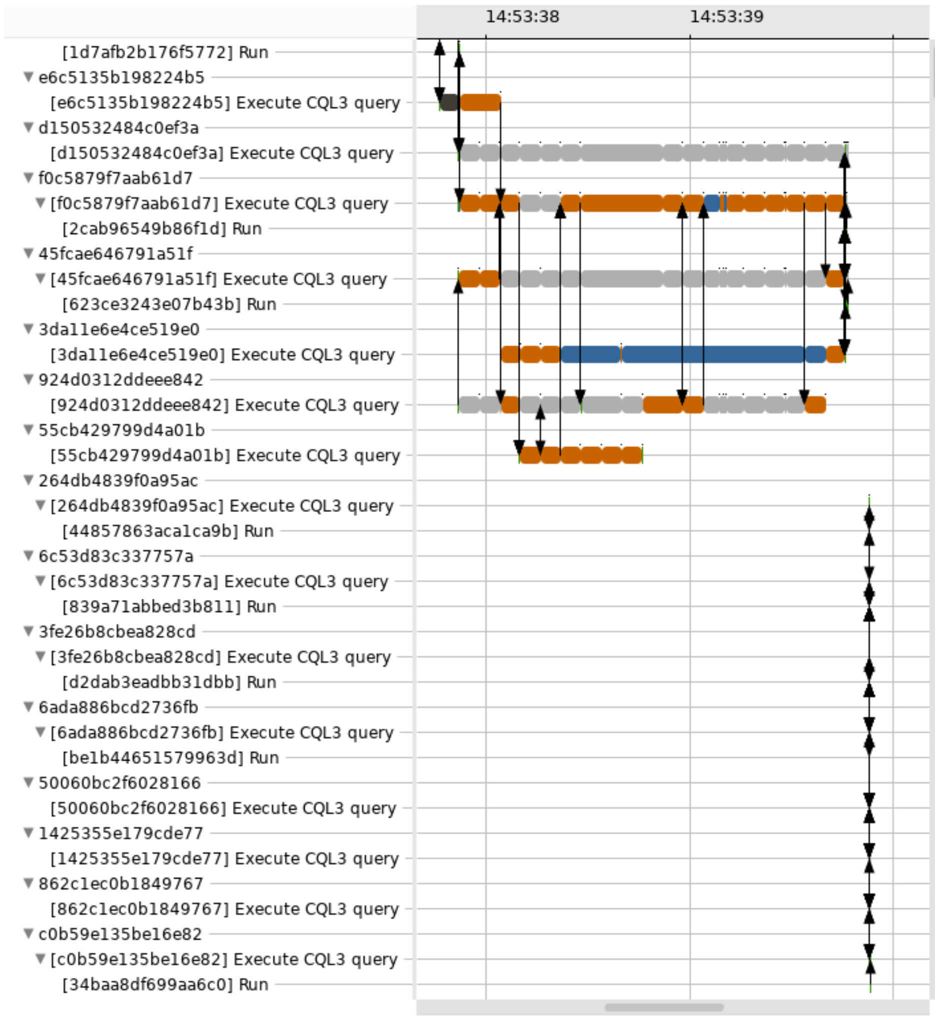
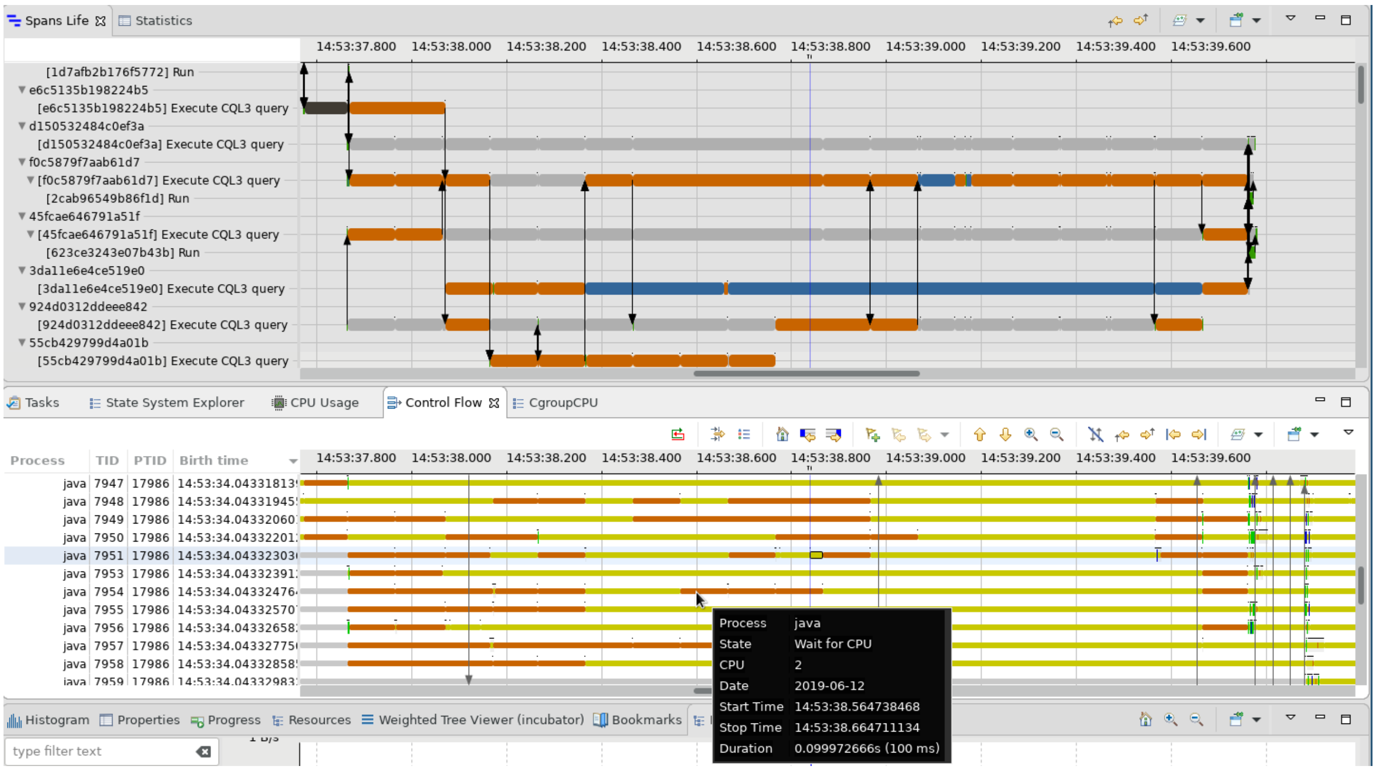

14:53:38
14:53:39
[1d7afb2b176f5772] Run
▼e6c5135b198224b5
[e6c5135b198224b5] Execute CQL3 query
▼d150532484c0ef3a
[d150532484c0ef3a] Execute CQL3 query
▼f0c5879f7aab61d7
▼[f0c5879f7aab61d7] Execute CQL3 query
[2cab96549b86f1d] Run
▼45fcae646791a51f
▼[45fcae646791a51f] Execute CQL3 query
[623ce3243e07b43b] Run
▼3da11e6e4ce519e0
[3da11e6e4ce519e0] Execute CQL3 query
▼924d0312ddeee842
[924d0312ddeee842] Execute CQL3 query
▼55cb429799d4a01b
[55cb429799d4a01b] Execute CQL3 query
▼264db4839f0a95ac
▼[264db4839f0a95ac] Execute CQL3 query
[44857863aca1ca9b] Run
▼6c53d83c337757a
▼[6c53d83c337757a] Execute CQL3 query
[839a71abbed3b811] Run
▼3fe26b8cbea828cd
▼[3fe26b8cbea828cd] Execute CQL3 query
[d2dab3eadbb31dbb] Run
▼6ada886bcd2736fb
▼[6ada886bcd2736fb] Execute CQL3 query
[be1b44651579963d] Run
▼50060bc2f6028166
[50060bc2f6028166] Execute CQL3 query
▼1425355e179cde77
[1425355e179cde77] Execute CQL3 query
▼862c1ec0b1849767
[862c1ec0b1849767] Execute CQL3 query
▼c0b59e135be16e82
▼[c0b59e135be16e82] Execute CQL3 query
[34baa8df699aa6c0] Run

Figure 12. Use case: identifying the latency. The requests are all equivalent in the expected execution time, but the ones on the left take 2 s to process, as opposed to about 5 ms for the ones on the right. The view shows that the slow requests spend most of their time waiting on CPU or on another request.

Further investigation can help diagnose the problem. The Resources view in Trace Compass clearly shows that the threads are preempted but not replaced on the CPU, as shown in Figure 17. In other words, the threads in Cassandra are not preempted because a higher priority process has to be scheduled. The scheduler event corresponding to Figure 17, as displayed by the Events view in Trace Compass, is shown in Table 5.



```
[1d7afb2b176f5772] Run
▾ e6c5135b198224b5
    [e6c5135b198224b5] Execute CQL3 query
▾ d150532484c0ef3a
    [d150532484c0ef3a] Execute CQL3 query
▾ f0c5879f7aab61d7
  ▾ [f0c5879f7aab61d7] Execute CQL3 query
      [2cab96549b86f1d] Run
▾ 45fcae646791a51f
  ▾ [45fcae646791a51f] Execute CQL3 query
      [623ce3243e07b43b] Run
▾ 3da11e6e4ce519e0
    [3da11e6e4ce519e0] Execute CQL3 query
▾ 924d0312ddeee842
    [924d0312ddeee842] Execute CQL3 query
▾ 55cb429799d4a01b
    [55cb429799d4a01b] Execute CQL3 query
```

Process	TID	PTID	Birth time
java	7947	17986	14:53:34.04331813
java	7948	17986	14:53:34.04331945
java	7949	17986	14:53:34.04332060
java	7950	17986	14:53:34.04332201
java	7951	17986	14:53:34.04332303
java	7953	17986	14:53:34.04332391
java	7954	17986	14:53:34.04332476
java	7955	17986	14:53:34.04332570
java	7956	17986	14:53:34.04332658
java	7957	17986	14:53:34.04332775
java	7958	17986	14:53:34.04332858
java	7959	17986	14:53:34.04332983

Process	java
State	Wait for CPU
CPU	2
Date	2019-06-12
Start Time	14:53:38.564738468
Stop Time	14:53:38.664711134
Duration	0.099972666s (100 ms)

Figure 13. Use case: CPU time sharing between worker threads. The Resources view in Trace Compass shows that the threads in Cassandra all share scarce CPU resources. They are waiting on the CPU or are blocked most of the time.

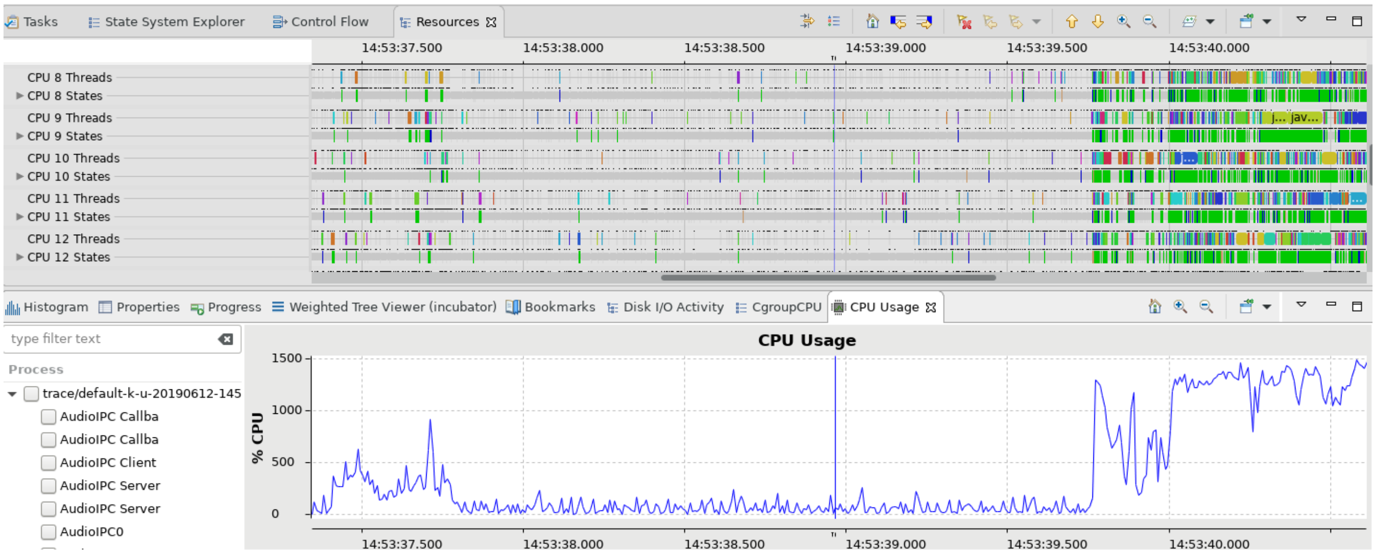


Figure 14. Use case: sub-optimal usage of the CPU. Heavy CPU limit can lead to the machine being underused. Once the CPU limit is waived, the usage is back to normal.

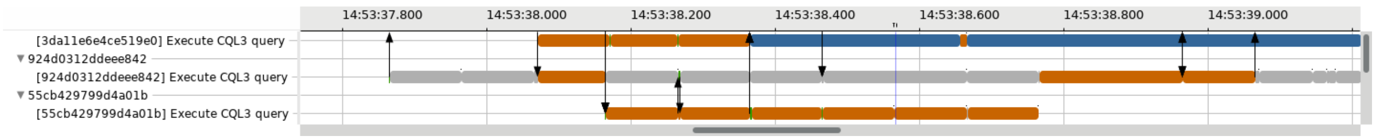
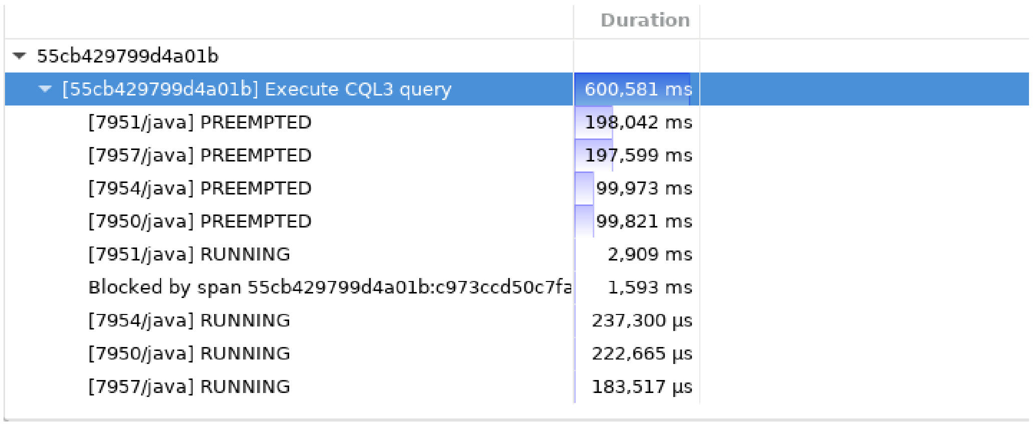


Figure 15. Use case: pattern of preemption. Request `55cb429799d4a01b` takes about 600 ms to process but shows a pattern of recurring preemption every 100 ms.



	Duration
▼ 55cb429799d4a01b	
▼ [55cb429799d4a01b] Execute CQL3 query	600,581 ms
[7951/java] PREEMPTED	198,042 ms
[7957/java] PREEMPTED	197,599 ms
[7954/java] PREEMPTED	99,973 ms
[7950/java] PREEMPTED	99,821 ms
[7951/java] RUNNING	2,909 ms
Blocked by span 55cb429799d4a01b:c973ccd50c7fa	1,593 ms
[7954/java] RUNNING	237,300 µs
[7950/java] RUNNING	222,665 µs
[7957/java] RUNNING	183,517 µs

Figure 16. Use case: summary of a critical path. The critical path summary shows that request `55cb429799d4a01b` spends most of its time in the `PREEMPTED` state.

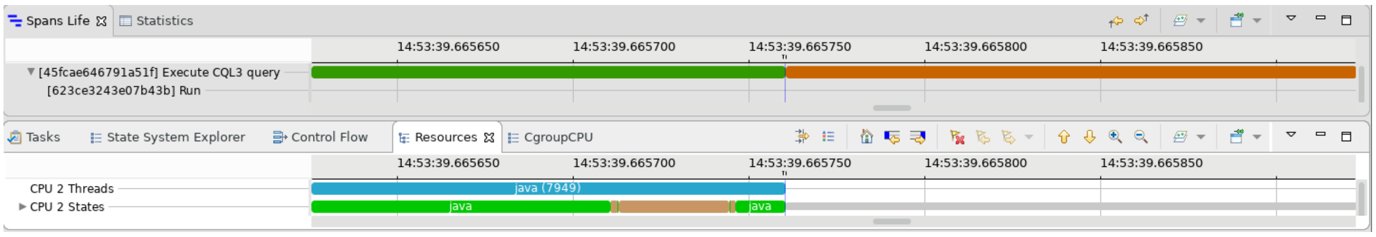


Figure 17. Use case: preemption without replacement. The Resources view in Trace Compass shows that the `java` thread used by Cassandra is preempted but not replaced.

Table 5. Use case: scheduling event for a worker thread. The worker thread is replaced by the virtual process `swapper`, which indicates that the CPU is becoming idle. The state of the worker thread is left to 0, meaning it could potentially be running.

CPU	Event Type	Contents	TID	Prio	PID
2	sched_switch	prev_comm=java, prev_tid=7949, prev_prio=20, prev_state=0, next_comm=swapper/2, next_tid=0, next_prio=20	7949	20	5322

Similarly, the kernel events corresponding to the control group limit being applied or waived can be identified. Along with the analyses and views included in Trace Compass, our solution allows for in-depth root cause finding since it relies on detailed kernel events collection.

4.4. Discussion

In this section, we explored and tested the implementation of our system on HotROD and Cassandra, allowing us to analyze the performance impact and the ability to detect performance issues. The results show that this is an effective solution for detecting and identifying causes of latency in distributed systems and that the overhead cost is acceptable for production environments with appropriate parameters.

Our solution has a variable overhead, depending on the nature of the target application. The overhead measured is about 5–6% for HotROD—with 100% of the requests being sampled—and close to 30% for Cassandra for a sampling frequency of 10%. The shorter duration of requests for Cassandra is impacted more by the added kernel tracing than the longer duration for HotROD requests. It is worth mentioning that a sampling frequency of 10% for high-throughput applications is acceptable for testing and benchmarking but unreasonable for production environments. A good trade-off between performance and trace collection can be achieved through lower sampling frequencies, as shown later on in Table 2.

Our full critical path analysis for requests requires about 15 s for processing moderately large traces of 20 s to 30 s. As this analysis is performed after runtime (and may be performed by a separate system than the one that is being analyzed), this is highly acceptable when analyzing a small number of problematic requests to diagnose a problem. The use case that we presented demonstrates the interest in multi-level trace data analysis. We argue that our analysis explains the root cause of actual latencies in a way that distributed tracers alone simply cannot achieve.

The results of this experiment are transferable to similar machine setups, such as un-grouped multi-core cpu systems. Although the architecture is slightly different and the method of introducing cpu contention is artificial rather than emerging from a natural scenario, the individual components (cpu, threads), their interactions and the contention introduced from their constraints are preserved.

As shown in Figure 5, each event contains information on which station it came from, allowing for traces to be collected and differentiated for each machine after runtime while also allowing us to track requests. By running an instance of Jaeger and LTTng on each machine, we could analyze the performance on multiple machines. This setup would allow for tracking the flow of requests as we could reconstruct the path that each event takes through the server.

5. Conclusions and Future Work

Detecting latency in distributed applications is complex as performance problems can often be obscured by being hidden behind the kernel layer. Existing methodologies are able to locate where and when a performance issue is occurring; however, they are often not able to ascertain why a slowdown occurs. Uncovering the source of these issues requires kernel tracing to collect low-level events; however, integrating kernel tracing and distributed tracing can impose a high overhead. In addition, to optimist the detection of performance failures, mapping the execution flow of requests and extracting the critical path is required to examine requests as they move through the layers of a distributed system and identify where latency is higher than expected.

In this paper, we developed a methodology to combine distributed tracing with kernel tracing to better characterize the performance of distributed applications. We proposed and demonstrated a solution for collecting and synchronizing the trace events using Jaeger as a distributed tracer and LTTng as a kernel and user-space tracer. Our solution for propagating the context from the distributed tracer into the kernel traces relies solely on the instrumentation of Jaeger in user-space. The results of our work show that extensive trace collection can be achieved without patching the application. The overhead is reasonable for average-throughput applications (and can be kept low enough for production environments on high-throughput applications) provided that trade-offs are made on the requests sampling frequency. Beyond this, we proposed analyses that compute the critical path of distributed requests and implemented them in the Trace Compass. We showed that the analyses could be run in a couple of seconds for moderately large traces. We illustrated how our analyses can provide useful information about precise kernel events or contentions and relate that information to the latency of given requests.

Further work should focus on integrating our solution to existing monitoring dashboards for better usability. Trace collection under the form of snapshots could be triggered when anomalies are detected by the monitoring application and lead to automated processing and reporting back to the dashboard for manual analysis. Additionally, user-level task schedulers, work queues and thread pools are getting more popular, which makes the analysis based on kernel scheduling events more difficult. Our algorithms will be accurate only if we can track requests being moved between threads and conversely determine which request is being actively handled by a given thread. That will imply better support by the distributed tracers for multi-threaded frameworks and languages. Lastly, while our algorithm can assist in discovering network issues via revealing network contention, a proper network analysis scenario would demonstrate the methodology's effectiveness in that regard.

Author Contributions: Conceptualization, L.G., N.E.-J. and M.R.D.; methodology, L.G., N.E.-J. and M.R.D.; validation, N.E.-J. and M.R.D.; investigation, L.G., N.E.-J. and M.R.D.; resources, L.G., N.E.-J. and M.R.D.; writing—original draft preparation, L.G., N.E.-J. and M.R.D.; writing—review and editing, N.E.-J. and M.R.D.; visualization, L.G.; supervision, N.E.-J. and M.R.D.; project administration, N.E.-J. and M.R.D.; funding acquisition, M.R.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, Google and EfficiOS.

Data Availability Statement: Source codes are available as an open-source project in the following GitHub repository: https://github.com/loicgelle/jaeger-go-lttng-instr, (accessed on 10 May 2021).

Acknowledgments: The authors of this paper are grateful to Ciena, EfficiOS, Ericsson, Google, the Natural Sciences and Engineering Research Council of Canada (NSERC) and Prompt for their financial support. The authors are grateful to Evan Galea for proofreading, formatting and updating the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Desnoyers, M.; Dagenais, M.R. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Ottawa Linux Symposium (OLS)*; Citeseer: Ottawa, ON, Canada, 2006; Volume 2006, pp. 209–224.
2. Sharma, R.; Singh, A. Logs and Tracing. In *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes*; Apress: Berkeley, CA, USA, 2020; pp. 259–279. [CrossRef]
3. Chen, H.; Jiang, G.; Zhang, H.; Yoshihira, K. Boosting the performance of computing systems through adaptive configuration tuning. In Proceedings of the 2009 ACM Symposium on Applied Computing, Honolulu, HI, USA, 8–12 March 2009; pp. 1045–1049. [CrossRef]
4. Bennett, C.; Tseitlin, A. Chaos monkey released into the wild. *Netflix Blog*, 30 July 2012.
5. Basiri, A.; Behnam, N.; De Rooij, R.; Hochstein, L.; Kosewski, L.; Reynolds, J.; Rosenthal, C. Chaos engineering. *IEEE Softw.* **2016**, *33*, 35–41. [CrossRef]
6. Pham, C.; Wang, L.; Tak, B.C.; Baset, S.; Tang, C.; Kalbarczyk, Z.; Iyer, R.K. Failure diagnosis for distributed systems using targeted fault injection. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 503–516. [CrossRef]
7. Leverich, J.; Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In Proceedings of the Ninth European Conference on Computer Systems, Amsterdam, The Netherlands, 14–16 April 2014; p. 4. [CrossRef]
8. Reiss, C.; Tumanov, A.; Ganger, G.R.; Katz, R.H.; Kozuch, M.A. *Towards Understanding Heterogeneous Clouds at Scale: Google Trace Analysis*; Technical Report ISTC-CC-TR-12-101; Intel Science and Technology Center for Cloud Computing: 2012; Volume 84. Available online: https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/ISTC-CC-TR-12-101.pd (accessed on 10 September 2021).
9. Aguilera, M.K.; Mogul, J.C.; Wiener, J.L.; Reynolds, P.; Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*; ACM: Bolton Landing, NY, USA, 2003; Volume 37, pp. 74–89.
10. Zhao, X.; Zhang, Y.; Lion, D.; Ullah, M.F.; Luo, Y.; Yuan, D.; Stumm, M. lprof: A non-intrusive request flow profiler for distributed systems. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, USA, 6–8 October 2014; pp. 629–644. [CrossRef]
11. Nagaraj, K.; Killian, C.; Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012; p. 26.

12. Kavulya, S.P.; Daniels, S.; Joshi, K.; Hiltunen, M.; Gandhi, R.; Narasimhan, P. Draco: Statistical diagnosis of chronic problems in large distributed systems. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), Boston, MA, USA, 25–28 June 2012; pp. 1–12. [CrossRef]
13. Tak, B.C.; Tang, C.; Zhang, C.; Govindan, S.; Urgaonkar, B.; Chang, R.N. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In Proceedings of the USENIX Annual Technical Conference, San Diego, CA, USA, 14–19 June 2009. [CrossRef]
14. Koskinen, E.; Jannotti, J. Borderpatrol: Isolating events for black-box tracing. In *ACM SIGOPS Operating Systems Review*; ACM: Bolton Landing, NY, USA, 2008; Volume 42, pp. 191–203. [CrossRef]
15. Ma, S.; Zhai, J.; Wang, F.; Lee, K.H.; Zhang, X.; Xu, D. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 6–18 August 2017; pp. 1111–1128.
16. Reynolds, P.; Killian, C.E.; Wiener, J.L.; Mogul, J.C.; Shah, M.A.; Vahdat, A. *Pip: Detecting the Unexpected in Distributed Systems*; NSDI: Santa Clara, CA, USA, 2006; Volume 6, p. 9. [CrossRef]
17. Huang, J.; Mozafari, B.; Wenisch, T.F. Statistical analysis of latency through semantic profiling. In Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, 23–26 April 2017; pp. 64–79.
18. Attariyan, M.; Chow, M.; Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. Presented at the Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), Hollywood, CA, USA, 8–10 October 2012; pp. 307–320.
19. Erlingsson, Ú.; Peinado, M.; Peter, S.; Budiu, M.; Mainar-Ruiz, G. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst. TOCS* **2012**, *30*, 13. [CrossRef]
20. Barham, P.; Donnelly, A.; Isaacs, R.; Mortier, R. Using Magpie for Request Extraction and Workload Modelling. In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, CA, USA, 6–8 December 2004; Volume 4, p. 18. [CrossRef]
21. Fonseca, R.; Porter, G.; Katz, R.H.; Shenker, S.; Stoica, I. X-trace: A pervasive network tracing framework. In Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation, Cambridge, MA, USA, 11–13 April 2007; p. 20. [CrossRef]
22. Sigelman, B.H.; Barroso, L.A.; Burrows, M.; Stephenson, P.; Plakal, M.; Beaver, D.; Jaspan, S.; Shanbhag, C. Dapper, A Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report dapper-2010-1. Available online: https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf (accessed on 24 May 2021).
23. Las-Casas, P.; Mace, J.; Guedes, D.; Fonseca, R. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In Proceedings of the ACM Symposium on Cloud Computing, Carlsbad, CA, USA, 11–13 October 2018; pp. 326–332.
24. Chanda, A.; Cox, A.L.; Zwaenepoel, W. Whodunit: Transactional profiling for multi-tier applications. In *ACM SIGOPS Operating Systems Review*; ACM: Bolton Landing, NY, USA, 2007, Volume 41, pp. 17–30. [CrossRef]
25. Mace, J.; Bodik, P.; Fonseca, R.; Musuvathi, M. Retro: Targeted resource management in multi-tenant distributed systems. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 589–603.
26. Kaldor, J.; Mace, J.; Bejda, M.; Gao, E.; Kuropatwa, W.; O'Neill, J.; Ong, K.W.; Schaller, B.; Shan, P.; Viscomi, B.; others. Canopy: An end-to-end performance tracing and analysis system. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; pp. 34–50.
27. Mace, J.; Roelke, R.; Fonseca, R. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst. TOCS* **2018**, *35*, 11. [CrossRef]
28. Gebai, M.; Dagenais, M.R. Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Comput. Surv. CSUR* **2018**, *51*, 26. [CrossRef]
29. Prasad, V.; Cohen, W.; Eigler, F.; Hunt, M.; Keniston, J.; Chen, J. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*; Citeseer: Ottawa, ON, Canada, 2005; pp. 49–64.
30. Eigler, F.C.; Hat, R. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium*; Citeseer: Ottawa, ON, Canada, 2006; pp. 261–268.
31. Bird, T. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*; Citeseer: Ottawa, ON, Canada, 2009; pp. 47–54.
32. Miano, S.; Bertrone, M.; Risso, F.; Tumolo, M.; Bernal, M.V. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), Bucharest, Romania, 18–20 June 2018; pp. 1–8. [CrossRef]
33. Desnoyers, M.; Dagenais, M.R. Lockless multi-core high-throughput buffering scheme for kernel tracing. *Oper. Syst. Rev.* **2012**, *46*, 65–81. [CrossRef]
34. Fournier, Q.; Ezzati-jivan, N.; Aloise, D.; Dagenais, M.R. Automatic Cause Detection of Performance Problems in Web Applications. In Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 27–30 October 2019; pp. 398–405. [CrossRef]

35. Desfossez, J.; Desnoyers, M.; Dagenais, M.R. Runtime latency detection and analysis. *Softw. Pract. Exp.* **2016**, *46*, 1397–1409. [CrossRef]
36. Gassais, R.; Ezzati-Jivan, N.; Fernandez, J.M.; Aloise, D.; Dagenais, M.R. *Multi-Level Host-Based Intrusion Detection System for Internet of Things*; Springer: Berlin/Heidelberg, Germany, 2020; Volume 9, pp. 1–16. [CrossRef]
37. Montplaisir-Gonçalves, A.; Ezzati-Jivan, N.; Wininger, F.; Dagenais, M.R. State history tree: An incremental disk-based data structure for very large interval data. In Proceedings of the 2013 International Conference on Social Computing, Alexandria, VA, USA, 8–14 September 2013; pp. 716–724. [CrossRef]
38. Giraldeau, F.; Dagenais, M. Wait analysis of distributed systems using kernel tracing. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 2450–2461. [CrossRef]
39. Ardelean, D.; Diwan, A.; Erdman, C. Performance analysis of cloud applications. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA, USA, 9–11 April 2018; pp. 405–417. [CrossRef]
40. Sheth, H.; Sun, A.; Sambasivan, R.R. Skua: Extending Distributed-Systems Tracing into the Linux Kernel. In Proceedings of the DevConf.US 2018, Boston, MA, USA, 17–19 August 2018.
41. Zhou, F.; Gan, Y.; Ma, S.; Wang, Y. wPerf: Generic Off-CPU analysis to identify bottleneck waiting events. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 527–543. [CrossRef]
42. Ezzati-Jivan, N.; Fournier, Q.; Dagenais, M.R.; Hamou-Lhadj, A. DepGraph: Localizing Performance Bottlenecks in Multi-Core Applications Using Waiting Dependency Graphs and Software Tracing. In Proceedings of the 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, Australia, 28 September–2 October 2020; pp. 149–159. [CrossRef]
43. Doray, F.; Dagenais, M. Diagnosing Performance Variations by Comparing Multi-Level Execution Traces. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 462–474. [CrossRef]