tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces

Lexiang Huang
The Pennsylvania State University

Timothy Zhu
The Pennsylvania State University

Abstract

The traditional approach for performance debugging relies upon performance profilers (e.g., gprof, VTune) that provide average function runtime information. These aggregate statistics help identify slow regions affecting the entire workload, but they are ill-suited for identifying slow regions that only impact a fraction of the workload, such as tail latency effects. This paper takes a new approach to performance profiling by utilizing distributed tracing systems (e.g., Dapper, Zipkin, Jaeger). Since traces provide detailed timing information on a per-request basis, it is possible to group and aggregate tracing data in many different ways to identify the slow parts of the system. Our new approach to trace aggregation uses the structure embedded within traces to hierarchically group similar traces and calculate increasingly detailed aggregate statistics based on how the traces are grouped. We also develop an automated tool for analyzing the hierarchy of statistics to identify the most likely performance issues. Our case study across two complex distributed systems illustrates how our tool is able to find multiple performance issues that lead to 10× and 28× performance improvements in terms of average and tail latency, respectively. Our comparison with a state-of-the-art industry tool shows that our tool can pinpoint performance slowdowns more accurately than current approaches.

CCS Concepts

• Software and its engineering → Software testing and debugging; **• General and reference → Performance**.

Keywords

performance debugging, distributed systems tracing

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486994

ACM Reference Format:
Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3472883.3486994

1 Introduction

Diagnosing performance issues is a slow and labor-intensive process. While correctness issues are more easily identified due to crashes or erroneous output, performance issues manifest themselves as slowdowns, which are hard to detect. For example, using inefficient serialization/deserialization doesn't break functionality, but it adds unnecessary delays that can surprisingly be a major slowdown [45]. Furthermore, it is often difficult to determine whether the speed of a component is normal or slow since processing times are typically variable. Especially when issues involve tail latency (i.e., the time to process the slowest fraction of requests), identifying the root cause(s) often requires significant debugging time and instrumentation. As a result, companies often ignore or defer non-critical performance issues that can be hidden by adding more resources [68]. Even so, performance bugs are still a widespread problem – a recent cloud study of over 3,000 bugs identifies 23% of them as being performance related [24], and another study of Google Cloud Platform incident reports show that 34.7% of them are related to undesirable slowdowns [63].

The traditional approach for performance diagnosis is performance profiling. Performance profilers (e.g., gprof [23], VTune [17]) automatically instrument code to measure the average computation time within functions in the code base. By providing a sorted list of the most time-consuming functions, developers can quickly identify candidates for performance optimization. While this provides useful information for identifying slow code in the average case, these tools lack the ability to capture the parts contributing to tail latency as well as pinpoint problems that only affect a fraction of the traffic. These tools are also primarily designed for single programs rather than distributed systems, which are growing in popularity and usage in modern system designs. Our goal is to build a better performance profiler that can provide useful aggregate statistics while overcoming these limitations.

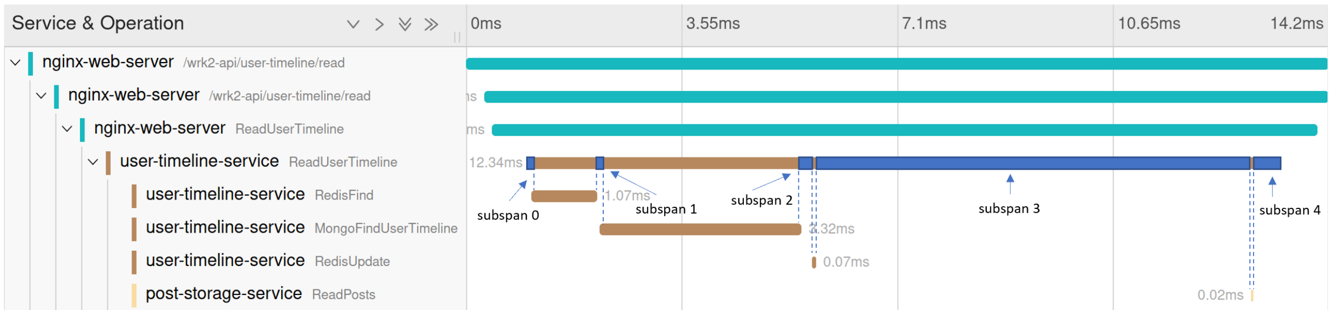

Service & Operation
0ms
3.55ms
7.1ms
10.65ms
14.2ms
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server ReadUserTimeline
user-timeline-service ReadUserTimeline
user-timeline-service RedisFind
user-timeline-service MongoFindUserTimeline
user-timeline-service RedisUpdate
post-storage-service ReadPosts
12.34ms
1.07ms
.32ms
0.07ms
0.02ms
subspan 0
subspan 1
subspan 2
subspan 3
subspan 4

Figure 1: Example trace from Jaeger [26] with annotations to illustrate subspans in our novel subspan analysis.

To address these issues, especially in the context of complex distributed systems, we propose a radically different approach to performance profiling by using distributed tracing. Distributed tracing systems (e.g., Dapper [53], Zipkin [44], Jaeger [26]) are being integrated into many of today's distributed systems as a fundamental tool for debugging [34]. In the context of this work, a *trace* consists of detailed timing information about how a single request traverses the system. A *request* represents the work that a user submits to the system, such as a web request. Fig. 1 illustrates an example trace of a request traversing a social network web service. Each of the time ranges is known as a *span*, which represent the starting/ending time of operations being performed.

Research has shown that distributed tracing is a powerful tool for debugging [1, 4, 15, 18, 26, 29–31, 35, 36, 44, 49, 53, 56, 66], but the standard practice for trace analysis is a slow process where engineers manually look through a large number of traces [5, 46, 60, 69]. Without looking at many traces, it is easy to be misled about performance bottlenecks – the advantage of traditional performance profilers is their ability to present aggregate information over a large amount of data. However, current tracing tools (e.g., Jaeger [26]) are primarily designed to present detailed information about a single trace at a time. The state of the art in using traces for performance analysis is based on comparing a "good" trace with a "bad" trace [5, 16, 26, 32, 50]. However, this relies upon a user to identify the right pair of traces to compare, which can be difficult with the variability that is often seen in complex systems. For identifying performance regressions or anomalies, it's possible to identify an interesting or representative trace, but this is not the right approach for general performance profiling.

What's needed for performance profiling is methodologies for aggregating traces to reveal performance problems. Recent systems like Canopy [29] provide a framework for efficiently analyzing many traces, but users are still responsible for deciding how to analyze the trace data. The difficulty in aggregating traces lies in deciding (i) what group of traces should be aggregated, and (ii) what data should be aggregated. For example, aggregating all the traces that communicate with a cache and database would yield different results from aggregating the traces that only communicate with the database. Furthermore, when aggregating all traces, one is limited to aggregating data that is common across the traces (e.g., all database accesses). However, when aggregating a group of traces that all share the same trace structure, one can aggregate more detailed information about individual spans (e.g., first database access vs. second database access). In practice today, trace aggregation relies upon a lot of human effort and expertise to craft the appropriate queries for aggregating traces, and the resulting aggregated data is often limited to coarse-grained statistics at the granularity of overall latency or operation latency. Even with some of the most advanced tracing tools available in industry (e.g., Lightstep [32], Canopy [29]), they only allow for coarse-grained grouping and filtering at the granularity of trace features (e.g., HTTP status code, span latency), and the grouping/filtering is user-driven. Our goal is to design new automated aggregation methods that take advantage of the trace structure to enable a more detailed analysis of performance problems.

In this paper, we introduce a new trace aggregation methodology and tool, tprof, that hierarchically groups traces together and provides aggregate information at each layer of the hierarchy. At the top layer, all the traces are summarized together with overall statistics (avg latency, tail latency, etc.). At the second layer, the traces are organized by request type. For example, requests that post a new message to a social network will be separated from requests that read the posts. At the third layer, traces are partitioned based on the trace structure and what services/operations are used. For example, traces that have a cache miss and query the database would be separated from traces that have a cache hit.

At the fourth layer, traces are grouped such that each span has the same child spans in the same order. Since this layer can assume that traces (within a group) have the same ordering and structure of spans, this allows for a more detailed trace analysis, which we call subspan analysis. We define the *subspans* of a span as the time periods where the span is likely doing work (e.g., time between first child span finishing and second child span starting). For example, Fig. 1 illustrates the subspans of the user-timeline-service ReadUserTimeline span. One advantage of this novel subspan analysis is the ability to automatically extract more detailed subspan information without any additional tracing instrumentation. Another key benefit is that we can use this subspan information to synthesize an "aggregate" trace that represents the average span and subspan durations. This hypothetical aggregate trace allows users to visualize the average behavior of a group of traces.

Once we've calculated all of these statistics, a key challenge is in navigating the large number of statistics. So to help users quickly identify performance problems, we design an automated diagnosis tool that analyzes the hierarchy and generates reports of potential issues. Starting from the top of the hierarchy, it narrows into a smaller set of traces that exhibits slow behavior identified in the upper layers. The benefit of a hierarchical approach is that each layer provides increasingly more detailed information, which is only possible due to how the traces are aggregated at each layer. For example, it is only possible to aggregate information at the operation level for the top layers, but the bottom layer is able to perform a subspan analysis since all traces in a group have the same trace structure and span ordering.

We evaluate tprof on both the recent TrainTicket benchmark [69, 70] and DeathStarBench benchmark [21] to illustrate how our tool can reveal multiple subtle performance issues. In our results, we show that fixing these issues leads to significant performance benefits, improving overall average latency by 10× and tail latency by 28×.

We make three primary research contributions:

- We design a novel performance profiler, tprof, that hierarchically aggregates distributed systems traces. At each layer of the hierarchy, tprof partitions traces into groups that are aggregated and analyzed, where lower layers are finely grouped to enable more detailed analyses. Our tool is open-sourced at https://github.com/lexiangh/tprof.
- We propose a new subspan analysis technique for extracting timing information about time periods between child spans of a span (Sec. 3.4). We use average subspan and span data to generate a synthetic "aggregate" trace that is representative of the average aggregate behavior (Sec. 3.5).
- We introduce a novel automated performance diagnosis that analyzes the aggregated statistics and generates a report of potential performance issues (Sec. 3.7).

2 Related work and background

Existing profilers can be divided into two categories. On the one side, there are traditional *coarse-grained* performance profilers that accumulate function runtimes and provide average durations. These are good for identifying issues that affect the overall behavior for all the requests, but they are ill-suited to finding slowdowns that affect only a fraction of the workload. On the other side, there are distributed tracing systems that provide *fine-grained* timing information about each request. This allows for detailed analysis of where a specific request is taking all of its time, but analyzing many traces is difficult for engineers due to the large quantity of data. Our research introduces a new hierarchical approach for spanning the space from coarse-grained to fine-grained performance profiling by providing aggregate statistics across progressively smaller groups of traces. Our automated analysis starts with a coarse-grained overview of all the traces and traverses the hierarchy to identify subgroups of traces that exhibit slow behavior.

2.1 Coarse-grained: Performance profilers

Traditional performance profiling tools [17, 23, 27, 37, 54, 55, 58, 59], developed over decades, are essential for diagnosing performance issues. These tools are typically programming language specific and automatically instrument the code to collect information about the cumulative runtimes of functions in the code base. Instrumentation is the process of adding monitoring code for the purpose of debugging. For example, gprof [23] integrates with the compiler to automatically insert monitoring code at the beginning and end of functions to measure function statistics. For high-level languages such as Python, some profilers (e.g., cProfile [55]) hook into language-specific profiling and tracing infrastructure to automatically track function performance.

There are many enhancements to traditional performance profiling techniques. First, it is possible to calculate both the time spent within a function as well as time spent within a function excluding calls to other functions. For example, if f() calls g() and f takes a long time, then it is possible to determine whether this is due to g or code within f. In Python, the cumtime metric represents the cumulative time spent in a function and subfunctions, whereas the tottime metric represents the total time spent in a function excluding subfunctions. Second, some profilers (e.g., [17, 23]) are able to collect and display call graph information to provide structural information on how functions are called. For example, VTune [17] shows a graphical call hierarchy and highlights the critical path and most time-consuming functions.

While these coarse-grained tools are great at providing overall aggregate information without the need for manual instrumentation, they are fundamentally limited by design to only provide average information for the entire workload. Issues that only affect a fraction of the workload may be hidden in the averages. For example, a performance bug where data is unnecessarily copied between buffers multiple times may be hidden if it only occurs on a cache hit code path since averages are often heavily skewed by the slower cache miss code paths. Additionally, slowdowns that cause high tail latency for services may not show up in averages.

2.2 Fine-grained: Tracing systems

Modern distributed tracing systems [4, 18, 26, 36, 44, 53] have become mainstream debugging tools for complex distributed systems [13, 34]. Tracing systems work by either manual instrumentation or using libraries with built-in tracing support [41–43]. These tracing infrastructures log fine-grained information about how requests traverse the services and components within a system. Thus, one is able to investigate details at the granularity of single requests rather than overall averages. Additionally, a key benefit of tracing systems is the ability to correlate how a request accesses services and components in the distributed system. By contrast, coarse-grained performance profilers were originally designed for single programs and are not suited for tracking a request's performance across many servers and services.

Tracing systems provide mechanisms for uniquely identifying time periods within a request, known as *spans*. Spans belonging to the same request are grouped together into a *trace*, which provides a complete view of the time spans within a request. Additionally, parent and child span information is tracked to capture the trace graph structure, similar to a call graph structure in a single program.

Fig. 1 shows a graphical representation (known as a Gantt chart) of an example trace for a request traversing a social network web service. The x-axis provides a timeline, and the y-axis shows the various spans, arranged hierarchically based on the parent/child information. The rectangles correspond to the starting/ending time of the spans, which typically correspond to operations such as Remote Procedure Calls (RPCs). However, users are able to track any time range, so one common approach to debugging is to add additional spans to investigate more detailed timing information for suspicious parts of the system. Additionally, users are able to tag information with spans such as RPC parameters, debugging info, request type, etc. [36, 53]. This makes tracing a powerful tool not only for performance issues, but also for general debugging and monitoring. Thus, many papers and systems have been developed to make the trace collection process scalable and practical [1, 2, 4, 15, 18, 26, 29–31, 35, 36, 44, 49, 53, 56, 66]. As research has lowered the overhead costs of tracing, production systems often have some tracing enabled for general monitoring of system health [34].

While much research has focused on tracing infrastructure, the trace analysis process is still mostly human-driven, where engineers visually inspect traces to identify problems [5, 46, 60, 69]. Trace analysis systems like Canopy [29] and Lightstep [32] provide mechanisms for filtering and sorting to identify interesting traces, but the standard approach still involves looking at traces. Not only is this slow and unscalable for engineering time, but developers can also be misled into investigating false positives; it is sometimes hard to distinguish between normal system variability and performance problems. Computing the critical path within a trace (e.g., [10, 11, 32]) is helpful when inspecting traces, but users can still be misled as shown in Sec. 4.4.2. Rather than inspecting individual traces, performance profiling needs to aggregate and summarize trends across many traces.

Recent systems from industry like Canopy [29], Lightstep [32], and Pintrace [14] have basic support for aggregation, but only at the coarse granularity of operations and overall latency. Furthermore, these systems still require human effort and expertise in deciding what relevant features to query (e.g., which request type to investigate). Our work advances the state of the art in trace analysis by aggregating traces via a hierarchical multi-pronged approach including fine-grained trace aggregation based on the structure embedded in traces. We also develop an automated approach for analyzing and traversing the hierarchy of aggregated statistics to identify slow regions for users to investigate.

2.3 Other related work

Besides distributed tracing, there is a variety of other performance debugging approaches. A few papers such as Zeno [63], lprof [68], and Stitch [67] use logging data to reconstruct code execution paths for performance debugging. However, logs may not contain all the events for a request, so it does not work as well as tracing for performance profiling.

A number of papers present methods for debugging anomalies [4, 6, 8, 9, 12, 12, 19, 22, 25, 33, 38–40, 51, 64, 65]. Anomaly debugging focuses on identifying outliers, whereas our performance profiler's goal is to identify the most relevant parts of the system that exhibit consistently slow performance. This includes tail latency issues, which are not necessarily caused by anomalies.

A few papers provide a root cause diagnosis of tail latency, but since they do not build upon tracing systems, their diagnosis is mostly centered around identifying problematic servers and anomalous system metrics rather than narrowing in on slow code regions that contribute to tail latency. For example, CloudPD [52], CloudRanger [57], and Roots [28] assume the system design and implementation is good and identify the faulty or slow server/VM/container/service that is causing high tail latency. SEER [22] and Sage [20] take a step further and react to slowdowns by provisioning extra resources to avoid high tail latency. However, these approaches are not designed to diagnose root cause issues with the system design and implementation itself. CauseInfer [7], ϵ-Diagnosis [51], X-ray [3], and FChain [40] take another approach in root cause diagnosis by identifying the system attributes and metrics (e.g., high CPU usage) that are correlated with tail latency. This is helpful for developers to understand the factors leading to tail latency but cannot pinpoint the code regions where further investigation is needed.

3 tprof

tprof is our new performance profiling tool that uses distributed tracing systems to provide aggregate statistics about subsets of traces. The key challenge is in aggregating traces with potentially different request structures and operations. This involves answering two primary research questions:

(1) Which traces should be grouped together so they can be aggregated in a meaningful way?
(2) What information (e.g., spans vs. operations vs. subspans) can be aggregated within a group?

There is a trade-off when aggregating traces. Aggregating many diverse traces provides a broad system view, but one cannot aggregate in a detailed fashion since traces may not have the same structure. Aggregating similar traces allows for a more precise analysis based on trace structure, but there are fewer traces to aggregate in each group.

Our solution is to design four different methods of grouping and analyzing traces where each method defines two functions: 1) a grouping function that partitions a set of traces into subsets based on trace properties (e.g., structure), and 2) an analysis function that calculates aggregate statistics about the subsets of traces. These methods are organized hierarchically as a four-layer tree of traces where the top of the tree aggregates all traces and the lower layers progressively partition the traces into smaller groups with more detailed aggregation techniques. Of note, the bottom layer enables our new subspan analysis (Sec. 3.4), which we use to generate our novel "aggregate" trace representing the average behavior of the group of traces (Sec. 3.5). tprof also supports separating tail latency requests from non-tail latency requests to help in identifying issues with the slowest requests (Sec. 3.6). Lastly, tprof introduces an automated performance diagnosis that generates a report of potential performance issues (Sec. 3.7).

3.1 Definitions

In this work, we focus on distributed systems composed of multiple services. A *service* corresponds to a program running within one or more containers or virtual machines (VMs) that provides some functionality to other services or end-users. Services are instrumented with tracing code to label spans within a service. *Spans* correspond to time periods (start, end) within a service, and are labeled with an *operation* name that describes the span. For example, spans are typically created to represent the time to complete RPC operations. Production systems often have some spans already instrumented for basic system monitoring, and many common libraries have built-in support for tracing [41–43]. Developers can easily insert additional spans as well to help in the debugging process.

Spans are typically created in a hierarchical manner with parent/child relationships. This provides a mechanism to correlate how services utilize other services. That is, an end-user submits a high-level *request* to the system to perform some work (e.g., generate a report). This request starts at a service (e.g., web server) that will contact multiple other services to help in gathering data and computing the desired output. The parent/child relationships allow for the tracing system to correlate how the services communicate and serve the request. All the span data associated with a request is identified in the tracing system as a *trace*, and we often use the terms request and trace synonymously in this paper.

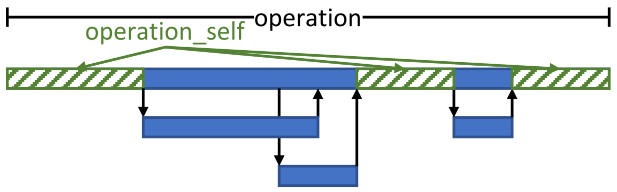


Figure 2: Illustrates layer 1/2 operation_self metric.

3.2 Layers 1/2: Operation analysis

In the first layer, all the available traces are grouped together to form one group that represents the overall behavior of the system. This allows us to analyze the system similarly to existing coarse-grained performance profilers. In the second layer, traces are grouped based on the request type. For example, this could correspond to a get vs. put request in a key-value store, or a browse vs. checkout request in an e-commerce website. By default, tprof uses the root span in the trace, which is typically an API endpoint, to group the trace, but request type identification can be user-defined (e.g., based on an attribute). The benefit of the second layer is the ability to calculate aggregate statistics for each request type whereas traditional performance profilers only have enough information to calculate averages across all requests.

Since the first and second layers make no assumptions on trace structure, we can only aggregate based on the names of the operations/services. This mimics traditional performance profilers analyzing programs at the granularity of functions. Currently, tprof calculates latency averages, std deviations, and 50th/99th percentiles for each operation as well as the overall latency, though data is available to calculate other statistics as needed.

In addition to calculating the statistics for operation durations, tprof also analyzes the operation durations excluding time periods where one or more of the operation's child spans is running, which we label operation_self. That is, operation_self represents the amount of time where an operation is definitely not waiting on a child span to finish. Fig. 2 illustrates the difference between the operation duration and operation_self metric. This idea is similar to mainstream performance profilers such as python's profiling with the tottime and cumtime metrics as well as gprof's call graph profiling with the self and children metrics.

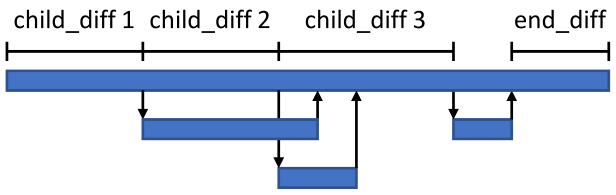


Figure 3: Illustrates layer 3 child_diff/end_diff metric.

3.3 Layer 3: Span and child analysis

In the third layer, we consider the structure of a trace's spans in the grouping process. Traces are grouped such that they have the same spans and the same parent/child span relationships. That is, two traces belong to the same group if they have the same spans, and each matching span has the same corresponding child spans. For example, if a trace consists of a span performing operation A, with two child spans performing operations B and C, then it is grouped with any other trace that consists of one operation A span that has exactly two child spans performing operations B and C.

In the third layer, we do not consider the order in which the child spans occur. The purpose of selecting this grouping methodology is to capture the structure of the trace in a way where parallel fanout operations would not be separated into different groups. So in our example, the first child span could perform operation C and the second child span could perform operation B or vice versa, and both types of traces would be grouped together.

Since layer 3's groups have the same structure, we can perform a more detailed aggregation based on spans rather than operations. In our running example, suppose operation C also has a child operation B. If we calculate statistics based on operations, then the two operation B durations would all be averaged together. By contrast, calculating statistics based on spans would separate the operation B that's a child of A from the operation B that's a child of C.

In addition to analyzing the spans, we also introduce a new metric for analyzing the start times of a span's children, which we label as child_diff (Fig. 3). Specifically, we calculate the time from (start of span → start of first child), (start of first child → start of second child), (start of second child → start of third child), and so forth. This information can help narrow down where a problem occurs within a span. For example, if the time till the first child span is large, then one might consider inspecting the initial code for that operation. If the child spans are executed in parallel, but the time between the start of the first and second child spans is large, then there might be a thread-related problem with launching the child operations. We also track the last part of the span after all child spans are complete, which we label as end_diff.

3.4 Layer 4: Subspan analysis

In the fourth layer, we consider both the structure and order of a trace's spans in the grouping process. That is, two traces belong to the same group if they have the same spans, and each matching span has the same corresponding child spans that begin and end in the same order. For example, Fig. 4 illustrates an example of the six possible orderings for a trace with an operation A that has child operations B and C. Depending on the code being debugged, many of the orderings may not show up. If operation C is performed after operation B completes, then in the example, only the first ordering would be present in the traces.

The purpose of ensuring the identical ordering of both the begin and end times is to enable our novel *subspan analysis* technique. Our goal is to define time periods, which we call *subspans*, within a span that corresponds to time periods where the operation is definitely performing some work. Fig. 4 illustrates the subspans A0, A1, and A2 within span A. Since operation A invokes both operations B and C, then it must be performing some work to launch the operations, and this is represented by subspans A0 and A1. At the end of the operation after both B and C are complete, A must be performing some work to finish up the operation, and this is represented by subspan A2. We do not define subspans in the other time periods where A might be waiting for operations B and/or C to complete.

A benefit of the subspan analysis is to provide a more detailed breakdown of time periods without instrumenting or collecting additional information. But to aggregate the subspans between traces in a meaningful way (i.e., have the first subspan corresponds to the same time period), the traces must have the same subspans. This is precisely why our grouping function ensures that traces within a group have the same ordering of begin/end times of child spans. Thus, the subspan analysis produces identical subspans for traces in the same group. With identical subspans, tprof then analyzes the traces by aggregating all of the subspans to produce statistics such as the average, std deviation, 50th/99th percentiles, etc.

Alg. 1 presents our subspan analysis algorithm for calculating the subspans of a span, which we recursively apply to all spans within a trace. For a given span, we collect all the events of which there are four possible types: span_start (start of span), child_start (start of child span), child_end (end of child span), span_end (end of span). We sort the events by time and then define subspans where (i) the subspan end time is determined by a child_start or span_end event, and (ii) the subspan start time is determined by the previous event (in time). For example, in the first case in Fig. 4, three subspans are defined that end at the events corresponding to the start of child B (A0), the start of child C (A1), and the end of span (A2). These subspans start at the event prior to these events (i.e., start of span occurs prior to start of child B, end of child B occurs prior to start of child C, end of child C occurs prior to end of span). We label subspans numerically

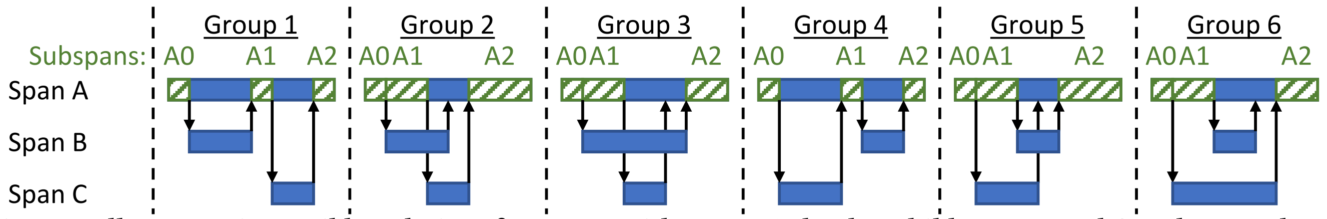


Figure 4: Illustrates six possible orderings for a trace with a span A that has child spans B and C. In layer 4, these would be separate into separate groups, with the corresponding subspan analysis for subspans A0, A1, and A2.

```
Func generate_subspans(span s):
    subspans = []
    prev_event_time = s.start_time
    subspan_number = 0
    events = get_sorted_events(s)
    foreach e in events do
        if e == "child_start" or e == "span_end" then
            new_subspan = Subspan()
            new_subspan.end_time = e.time
            new_subspan.start_time =
              prev_event_time
            new_subspan.span = s
            new_subspan.number = subspan_number
            subspans.append(new_subspan)
            subspan_number++
            if e == "span_end" then
                break
            end
        end
        prev_event_time = e.time
    end
    return subspans
```

Algorithm 1: Algorithm for generating subspans.

to have a consistent numbering scheme to allow us to easily aggregate traces within the same group (where traces are assumed to have the same structure and span ordering).

We find the algorithm is relatively fast, so we haven't spent any time optimizing our prototype. In terms of scalability, we find that tprof scales linearly with respect to the number of traces, taking seconds to minutes to analyze thousands of traces. Note that tprof doesn't need to analyze every trace, and sampling is a typical approach in tracing. Furthermore, tprof is easily parallelizable between layers, though this engineering is left to future work.

3.5 Aggregate trace visualization

A key benefit of subspan analysis is the ability to synthesize an "aggregate" trace based on average span and subspan durations. Specifically, each span's duration is set to its average span duration, and the start times of child spans are determined based on the average subspan durations. For example, consider the first case in Fig. 4, and suppose span A is on average 9ms, subspans A0, A1, and A2 are on average 1ms each, and spans B and C are on average 3ms each. Then span A, B, and C durations would be 9ms, 3ms, and 3ms, and they would start at 0ms, 1ms, and 5ms, respectively. This can then be visualized with existing tools (e.g., Jaeger [26]).

This synthetic aggregate trace gives users a "representative" view of an average trace that combines all the information across the traces within a layer 4 group. Prior work has aggregated coarse-grained trace statistics at the granularity of operation durations and overall latency [46], but we are the first to present the idea of an aggregate trace. This is because the aggregate trace is only suitable with our layer 4 grouping and subspan aggregation technique.

Theorem 1. *Creating an aggregate trace using average span and subspan durations always produces a valid trace.* [1]

The proof works by decomposing a span into a sum of subspans and child spans and then applying the linearity of expectation. The formal proof is included in Appendix A.

By contrast, aggregating based on non-average statistics does not always work. For example, consider the 99th percentile in the prior example where half of the traces have span B and C durations of 5ms and 1ms, respectively, and the other half of the traces have 1ms and 5ms durations, respectively. Assume all subspans are 1ms and span A is 9ms in all traces. Thus, the 99th percentile metric would be 9ms, 5ms, and 5ms for spans A, B, and C, respectively. However, there's no way to create a synthetic trace with non-overlapping child spans B and C each of duration 5ms while being contained within a 9ms span A duration.

3.6 Tail latency support

Another benefit of using tracing data over performance profiling averages is the ability to diagnose tail latency issues. tprof specifically takes advantage of this by providing an explicit tail latency analysis at each layer in the hierarchy. For each group of traces being analyzed, tprof further subdivides the group into "normal" traces and "tail" traces based on a user-defined threshold (e.g., 90%). The slowest requests above the threshold percentile are marked as tail requests and the rest are marked as normal requests. tprof then performs the same analysis (depending on the layer) on the normal traces, tail traces, and all the traces collectively.

[1] A valid aggregate trace maintains the trace structure and span averages.

3.7 Automated performance diagnosis

To aid users in performance diagnosis, tprof analyzes the aggregated statistics and auto-generates a report of the most relevant subspans to investigate. In the first layer, tprof identifies the slowest operations across all of the available traces by sorting based on the average operation_self duration multiplied by the count of executions. This provides a global perspective to identify important operations to focus on when debugging, which mimics the information presented in traditional coarse-grained performance profilers.

In the second layer, tprof narrows into the request type where the slowdown seen in layer 1 has the largest impact. Specifically, we iterate through all groups in layer 2 and rank them based on the operation_self duration multiplied by the count similar to in layer 1. In addition, tprof classifies whether the slowdown is tail-specific or not by comparing tail and normal traces (Sec. 3.6). If the ratio of tail/normal operation_self durations exceeds a threshold, tprof labels it as a tail issue. Empirically, we find that the specific threshold isn't very sensitive (anywhere between 3.5× and 8× works in our experiments), so we default to 4×.

Since an operation may be utilized in multiple spans within a trace (e.g., multiple calls to database), tprof uses the layer 3 data to narrow into the specific problematic span as well as the slow region within the span. Specifically, tprof sorts the child_diff metric multiplied by the count of invocations across all spans of the slow operation to identify the slow part of the span. We use the index of the child that starts late to identify the problematic subspan in the layer 4 group.

Finally, tprof analyzes the layer 4 data to identify which layer 4 group best exhibits the problematic subspan. We sort the groups by the average subspan duration multiplied by the count and fraction of time of the subspan within its span. This pinpoints slow subspans that are frequent and constitute a large fraction of its span. Once the layer 4 group is identified, tprof synthesizes an aggregate trace to visually point out the problematic subspan in the context of the average behavior of the layer 4 group. With this problematic subspan, a user can then correlate the slow subspan with the code. Once a user has narrowed the scope of the problem to a specific part of the code, the user can add additional spans to instrument more detailed regions of code. Our tool can then be used again to narrow down which parts of the code is slow. Additionally, one can run A/B tests and use tprof to evaluate whether a proposed fix helps performance.

Since each layer in the automated report generation is based on rankings, we actually generate multiple performance diagnosis reports in a hierarchy so that even if the top issue is not a bug, a user can look into the second or third ranked issues. Some slowdowns may be due to natural variability or heavy computation, and it is up to the user to decide whether it is a bug or not as with traditional performance profilers. As a performance profiler, tprof is also not designed to fix issues, but rather to provide helpful information for users to diagnose performance slowdowns. Nevertheless, as shown in Sec. 4.2.1, tprof saves users a lot of engineering time in identifying important areas to focus on to find performance bugs and/or optimize performance.

Fig. 5 in Sec. 4.3 shows an example report generated by tprof. The report is dynamically generated via a simple web application so that the different parts of the report hierarchy can be easily collapsed and navigated in a web interface.

4 Evaluation

To show the benefits and generality of tprof, we conduct evaluations on two systems, TrainTicket [70] and DeathStarBench [21]. We evaluate tprof to answer these key questions:
(1) How effective is tprof in detecting slowdowns? (Sec. 4.2)
(2) Can tprof expose unknown slowdowns? (Sec. 4.3)
(3) How does tprof compare to the state of the art? (Sec. 4.4)

4.1 Experimental setup

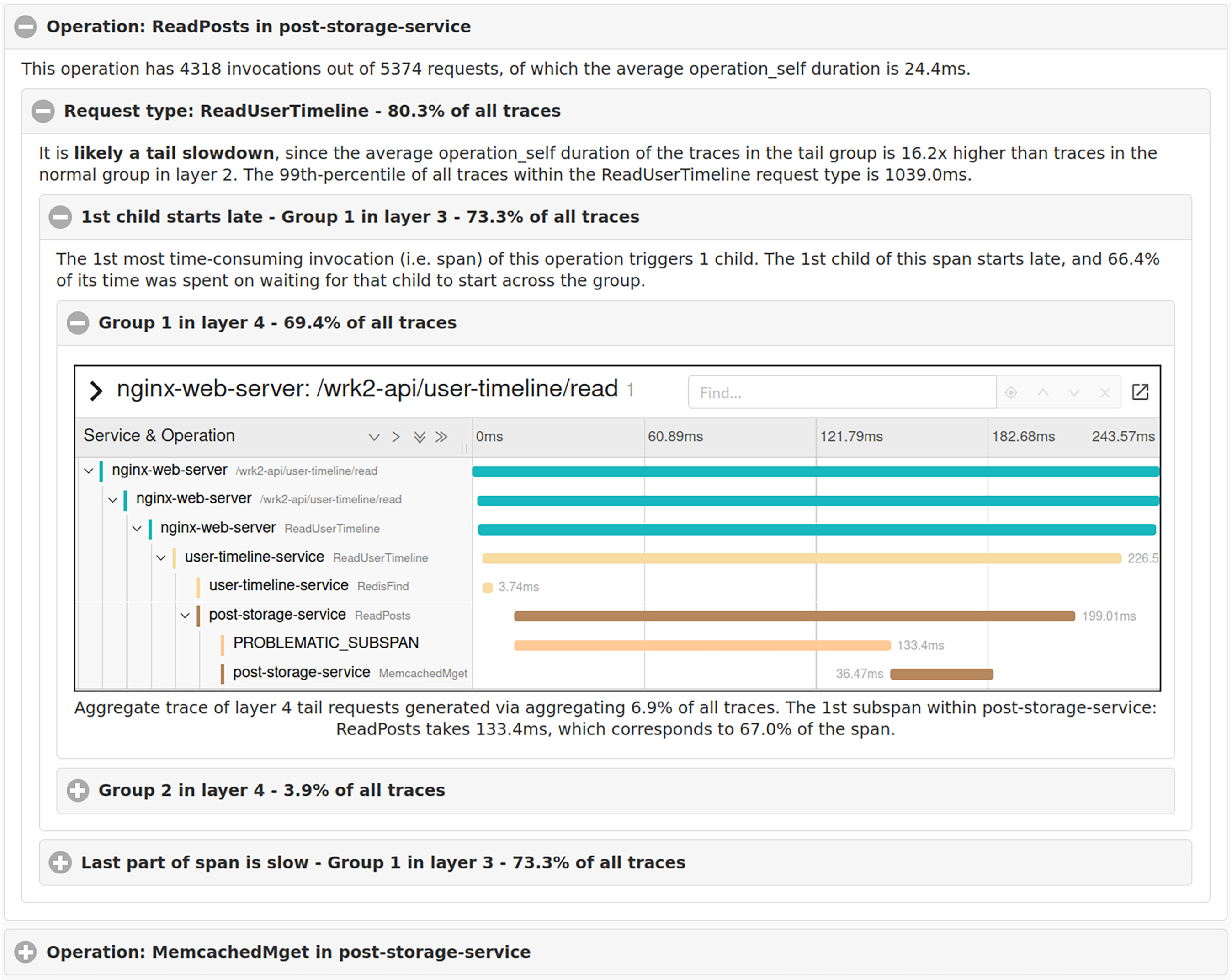
DeathStarBench: DeathStarBench [21] is a suite of microservice benchmark applications, and we use the social network application, which represents an example of a modern distributed systems design using microservices with 30 services including databases, caches, web servers, and application logic. Each microservice is designed to be simple and modular, but performance diagnosis is complex by virtue of having many services all working in conjunction to serve requests.

We focus on the ComposePost (CP) and ReadUserTimeline (RUT) workloads provided by the benchmark. After initializing the social network with a sample Facebook dataset [47], the CP requests insert random messages (i.e., posts) for random users. These posts show up in the users' timeline views, which are read by the RUT requests. In our system, we send CP and RUT requests in a 1:4 ratio under a 90% load.

TrainTicket: TrainTicket [70] is a complex microservice benchmark system that currently consists of 41 services excluding databases and caches. It is designed to mimic the microservice systems seen in industry.

A key difference between the systems is that TrainTicket often directly accesses the different microservices through REST APIs in the javascript code whereas DeathStarBench funnels requests through a frontend nginx web server. This architectural difference in the workloads results in traces that look very different. For example, the DeathStarBench traces start with a root nginx span, whereas the TrainTicket traces have a variety of root spans. Thus, our evaluation with both systems demonstrates that tprof generalizes to different types of systems and use cases.

Cluster configuration: Our experiments are conducted in our dedicated experimental cluster consisting of 20 dual-socket servers with 12 cores, 16-32GB of memory, and 10GbE networking. The cluster is set up with CloudStack [48] to



Operation: ReadPosts in post-storage-service

This operation has 4318 invocations out of 5374 requests, of which the average operation_self duration is 24.4ms.

Request type: ReadUserTimeline - 80.3% of all traces

It is **likely a tail slowdown**, since the average operation_self duration of the traces in the tail group is 16.2x higher than traces in the normal group in layer 2. The 99th-percentile of all traces within the ReadUserTimeline request type is 1039.0ms.

1st child starts late - Group 1 in layer 3 - 73.3% of all traces

The 1st most time-consuming invocation (i.e. span) of this operation triggers 1 child. The 1st child of this span starts late, and 66.4% of its time was spent on waiting for that child to start across the group.

Group 1 in layer 4 - 69.4% of all traces

nginx-web-server: /wrk2-api/user-timeline/read 1
Find...
Service & Operation
0ms
60.89ms
121.79ms
182.68ms
243.57ms
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server ReadUserTimeline
user-timeline-service ReadUserTimeline
226.5
user-timeline-service RedisFind
3.74ms
post-storage-service ReadPosts
199.01ms
PROBLEMATIC_SUBSPAN
133.4ms
post-storage-service MemcachedMget
36.47ms

Aggregate trace of layer 4 tail requests generated via aggregating 6.9% of all traces. The 1st subspan within post-storage-service: ReadPosts takes 133.4ms, which corresponds to 67.0% of the span.

Group 2 in layer 4 - 3.9% of all traces

Last part of span is slow - Group 1 in layer 3 - 73.3% of all traces

Operation: MemcachedMget in post-storage-service

Figure 5: tprof's auto-generated performance diagnosis report, which includes the aggregate trace combining average subspan and span durations. The report is structured hierarchically, matching tprof's 4 layers. The most relevant statistics are summarized in each layer, but the best information to initially investigate is the PROBLEMATIC_SUBSPAN within the aggregate trace.

support Ubuntu 18.04 virtual machines (VMs) using KVM under default settings. We launch VMs on each node and place microservices in Docker containers provided by the benchmarks to achieve load balancing in the initial setting. We also launch a VM for sending requests to the system. Once an experiment is completed, we run tprof to collect traces from Jaeger and generate a report of the performance.

4.2 Effectiveness in detecting slowdowns

4.2.1 Real-world slowdowns

Zhou et al. [69] conduct an industry survey on real-world bugs and perform a detailed user study evaluating the amount of human effort required to find the real-world bugs when they're injected into the TrainTicket benchmark [70]. We compare tprof against this work and demonstrate that tprof successfully pinpoints both of the performance slowdowns described in the paper without any human effort.

SSL slowdown: The first performance bug studied in [69] is due to the SSL protocol slowing down requests. Using the same fault injection code as in that paper, we replicate this bug in our cluster by injecting the fault into one of the frequently used services in the benchmark (ts-station-service). tprof successfully identifies the ts-station-service : queryForStationId operation as the top performance problem. Even though the operation is short (6.14ms) compared to other operations (100s of ms), it is executed many times and contributes significantly to the overall latency. tprof correctly identifies that it is not a tail-specific bug, indicating

Slowdowns	Fixes	Average Latency Improvement	99-percentile Latency Improvement
1. User Timeline ->Nginx network delay	Service migration: User Timeline-> Nginx	Subspan: 54.1ms -> 10.1ms (5.36x) RUT: 164ms -> 109ms (50%faster)	Subspan: 450ms -> 27.1ms (16.6x) RUT: 1190ms->1070ms (Timed out)
2. Post Storage ->User Timeline network delay	Service migration: Post Storage ->User Timeline	Subspan: 34.8ms -> 14ms (2.49x) RUT: 109ms->66.9ms (63% faster)	Subspan: 432ms -> 90.1ms (4.79x) RUT: 1070ms->1040ms (Timed out)
3. Post Storage - ReadPosts' child late start	Code Modification: Remove *memcached_quit()*	Subspan: 16.4ms->0.276ms (59.4x) RUT: 66.9ms->32.9ms (103% faster)	Subspan: 1010ms -> 4.16ms (243x) RUT: 1040ms -> 88.8ms (11.7x)
4. Compose Post - UploadText's child late start	Service migration: Compose Post ->Post Storage	Child_diff: 90.7ms->8.15ms (11.1x) CP: 82.6ms -> 49.9ms (66% faster)	Child_diff: 239ms->9.44ms (25.3x) CP: 273ms -> 94.1ms (190% faster)
5. Post Storage - MemcachedMget slow	Code Modification: Use native BSON format instead of JSON	MemcachedMget span: 9.35ms -> 2.42ms (3.86x) MongoFindPost span: 6.46ms -> 2.14ms (3.02x) RUT: 34.5ms -> 12.6ms (174% faster)	MemcachedMget span: 45.9ms -> 10.2ms (4.50x) MongoFindPost span: 37.4ms -> 10.4ms (3.60x) RUT: 98.8ms -> 31.5ms (214% faster)
6. Post Storage - MongoFindPost slow			
7. Write HomeTimeline - RedisUpdate inefficiency	Library usage change: Asynchronous commit instead of synchronous commit	Span: 18.8ms -> 0.549ms (34.2x) CP: 47.8ms -> 39.7ms (20% faster)	Span: 43.3ms -> 1.94ms (22.3x) CP: 164ms -> 64.9ms (153% faster)
8. User Timeline - Mongo Insert inefficiency	Library usage change: Replace *mongoc_ collection_find_and_ modify()* with *mongoc _collection_update()*	Span: 17.7ms -> 1.91ms (9.27x) CP: 39.7ms -> 24.9ms (59% faster)	Span: 24.1ms -> 5.20ms (4.63x) CP: 64.9ms -> 48.0ms (35% faster)
9. User Timeline - ReadUserTimeline's child late start	Library usage change: *std::async()* in deferred instead of async mode	Subspan: 3.62ms->0.508ms (7.13x) RUT: 12.7ms->10.9ms (17% faster)	Subspan: 10.7ms -> 3.50ms (3.06x) RUT: 31.0ms -> 27.1ms (14% faster)
Summary	ComposePost (CP)	58ms -> 25.2ms (2.30x)	296ms -> 47.1ms (6.28x)
	ReadUserTimeline (RUT)	164ms -> 10.9ms (15.0x)	1190ms -> 27.1ms (43.9x)
	All request types	142ms -> 13.7ms (10.4x)	1130ms -> 40.6ms (27.8x)

Table 1: Summary of performance issues and improvements in DeathStarBench case study. Relevant metrics that most accurately reveal the slowdowns are presented in their corresponding Latency Improvement cells.[2]

that the slowdown affects typical requests. As the amount of code in this operation is relatively small, one would consider environmental factors such as SSL, network overhead, etc., which would lead to the root cause. As a performance profiler, tprof is not designed to reveal the SSL root cause or fix the bug, but it does reveal the right location to investigate further.
Slow SQL queries: The second performance bug studied in [69] is due to using too many nested "select" and "from" clauses in a SQL statement. The authors reproduce this effect by injecting a delay into the ts-voucher-service to simulate the slow SQL queries, and we reproduce the problem in the same way. tprof successfully identifies the slow operation as the top performance problem. It also identifies it as a non-tail-specific problem since the slowdown affects all requests.

We extend this bug into a tail bug by only injecting the delay 2% of the time. tprof successfully reports the operation as the top performance problem, but this time it is marked as a tail bug, which indicates a user should be investigating a rarer phenomenon than the typical behavior.
Debugging effort comparison: Zhou et al. [69] conduct an industry survey that both identifies these bugs and also tracks the debugging effort. In the first SSL bug, the survey reports that experienced engineers spent 56 hours debugging and fixing this bug, with 22 hours spent on identifying,

[2]Slowdowns evaluated on spans are detected by operation_self metric.

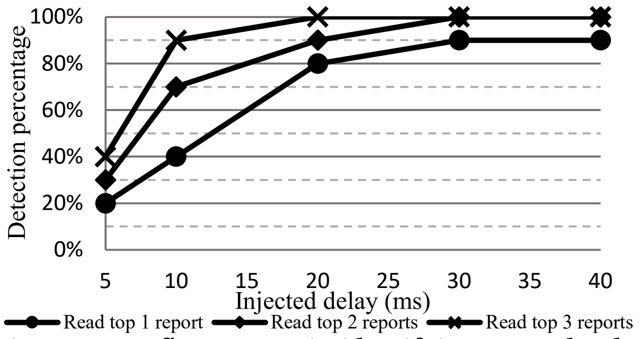


Figure 6: tprof’s accuracy in identifying 10 randomly injected slowdowns. When a slowdown is large, tprof is able to pinpoint the subspan corresponding to the injected slowdown.

scoping, and localizing the fault [69]. In the second slow SQL query bug, the survey reports 8 hours of engineering time, with 2 hours spent on localizing the fault [69]. Zhou et al. [69] also conduct an in-house evaluation with industrial debugging practices and the state-of-the-art trace visualization ShiViz [5] tool. In the first bug, they failed to find the bug using any of the techniques. In the second bug, they found the bug in 1.7-2.6 hours. By comparison, tprof is able to identify the slow operations in both cases automatically. Additionally, tprof is able to identify the more complex tail version of the second bug.

4.2.2 tprof accuracy

To evaluate tprof’s accuracy in identifying slowdowns, we randomly pick 10 locations within the DeathStarBench system and inject delays to mimic slowdowns. Fig. 6 shows tprof’s accuracy in identifying the correct subspan where the delays are injected. Our injected delays range from 5ms to 40ms, which is roughly half of the average ReadUserTimeline (RUT) request type latency (10.7ms) to 50% larger than the average ComposePost (CP) request type latency (25.5ms). Among the 10 randomly selected locations, 7 are related to the more complex CP request type and the other 3 are related to the RUT request type. When the injected delay is 5ms (20% of CP/50% of RUT), tprof reports other subspans as being the dominant slow region to investigate. But as the delay increases and becomes a major slowdown, tprof correctly identifies the subspan with the delay in all but one of the cases. In that case, the injected delay was in an infrequently executed code path comprising only 7% of the traces. tprof is able to identify it as a (tail) slowdown, though it is the second ranked issue.

4.3 Exposing unknown performance slowdowns

To evaluate how well tprof works in realistic settings where we do not have prior knowledge about the performance problems, we conduct a case study using DeathStarBench to discover entirely new issues. We first provision and configure the system as best as we could to avoid bottlenecks and then proceed to search for performance issues. We run tprof over a benchmark experiment and read the top-ranked performance report. The auto-generated report points out a problematic subspan, and we look at the corresponding code to figure out how to resolve the issue.

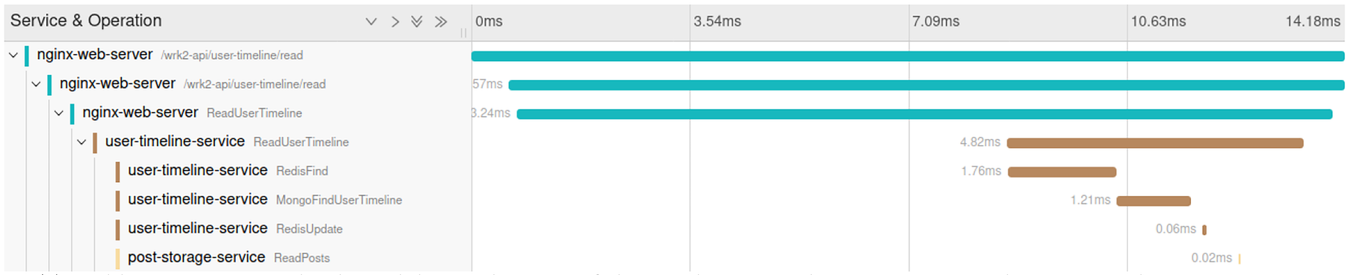
#	tprof rank	Lightstep rank	
		Operation	Operation Tail
1	1	2	2
2	1	2	5
3	1 (tail)	7	4
4	1 (tail)	3	1
5&6	2	13	N/A
7	1	9	N/A
8	2	5	N/A
9	2	5	4

Table 2: Comparing tprof with state-of-the-art approaches (coarse-grained aggregation with Lightstep) by ranking slowdowns from Tbl. 1. tprof identifies all slowdowns as the top-ranked issue excluding previously identified issues.

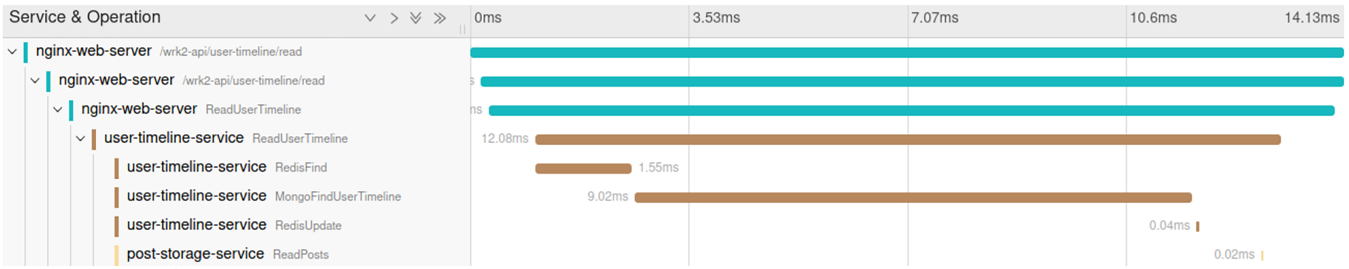
Tbl. 1 presents the summary of the slowdowns, fixes, and performance improvements that we identified. Performance issues in complex systems are notoriously difficult to identify, and we find a variety of problems ranging from code bugs and design inefficiencies to deployment/resource management issues (e.g., suboptimal VM placement). Compared to the original code base with a manually optimized configuration, we see a 10× average latency improvement and 28× 99th percentile tail latency improvement. This demonstrates the potential of tprof in identifying issues that can significantly impact the overall system performance.

Fig. 5 shows an example report for slowdown 3 to illustrate tprof’s automated report generation. Since tprof operates in a hierarchical fashion, the report is also hierarchical where each layer is collapsible to facilitate navigation. From the report, we see that the ReadPosts operation in post-storage-service is slow for the ReadUserTimeline request type. In particular, the tail latency traces (top 10%) are 16.2× higher than the other traces using the operation_self metric (Fig. 2), so tprof classifies this as a tail-specific issue. tprof then identifies that the first part of the span (i.e., before the first child is launched) is slow, which corresponds to the first subspan as marked by PROBLEMATIC_SUBSPAN. Our synthetically generated aggregate trace is not from any single trace, but instead represents an aggregation of many traces (6.9% of all traces in this case). Thus, we have confidence that this subspan is slow on average across many traces.

After looking at the relevant code, we discover a bug in the code where connections are not properly reused due to an erroneous memcached_quit() invocation. After fixing the


Service & Operation
0ms
3.54ms
7.09ms
10.63ms
14.18ms
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server /wrk2-api/user-timeline/read
57ms
nginx-web-server ReadUserTimeline
3.24ms
user-timeline-service ReadUserTimeline
4.82ms
user-timeline-service RedisFind
1.76ms
user-timeline-service MongoFindUserTimeline
1.21ms
user-timeline-service RedisUpdate
0.06ms
post-storage-service ReadPosts
0.02ms

(a) Tail latency trace with a long delay at the start of the ReadUserTimeline operation in the nginx-web-server service.


Service & Operation
0ms
3.53ms
7.07ms
10.6ms
14.13ms
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server /wrk2-api/user-timeline/read
nginx-web-server ReadUserTimeline
user-timeline-service ReadUserTimeline
12.08ms
user-timeline-service RedisFind
1.55ms
user-timeline-service MongoFindUserTimeline
9.02ms
user-timeline-service RedisUpdate
0.04ms
post-storage-service ReadPosts
0.02ms

(b) Tail latency trace with a slow MongoFindUserTimeline operation in user-timeline-service.

Figure 7: Along with Fig. 1, three tail latency traces near the 99th percentile. The traces show very different behavior so looking at any one of them can lead to misleading conclusions. Aggregating many more traces, as is done in tprof, will provide a clearer picture of the system performance characteristics and potential bugs.

issue, we find that performance is drastically improved with the subspan's 99th percentile latency improving by 243× (1010ms -> 4.16ms), which corresponds to an 11.7× 99th percentile reduction for ReadUserTimeline requests.

4.4 Comparison with state-of-the-art approaches

We compare against the state-of-the-art approaches and tools used in practice today using the slowdowns in Sec. 4.3.

4.4.1 Coarse-grained aggregation

Trace analysis systems such as Lightstep [32], Canopy [29], and Pintrace [14] support basic aggregation of tracing data at the granularity of operations. We use Lightstep to group by the operation name and sort to identify the slowest operations (Lightstep rank - Operation). We repeat this process with an initial filtering step for tail spans (top 10%) to identify tail latency issues (Lightstep rank - Operation Tail). We also try other aggregation approaches suggested by Lightstep's documentation and tutorials such as aggregating by service, but these approaches perform worse.

Tbl. 2 shows the ranking of each slowdown identified in Sec. 4.3. tprof identifies all issues as the top-ranked issue (excluding previously identified issues), whereas the coarse-grained aggregation ranks the issue much lower in most cases. This is because the slowest operations are often parents of the actual slowdown. Furthermore, tprof identifies the part of the span (i.e., subspan) that is slow, whereas Lightstep can only denote an entire operation as being slow.

In some of the Lightstep rank - Operation Tail cases, the operation is not in the ranking since the non-tail issues were all filtered out from the tail spans. For the tail slowdowns (3 & 4), Lightstep rank - Operation Tail has a higher ranking than Lightstep rank - Operation, but without knowing in advance whether an issue is a tail issue or not, it is difficult to know which issue one should spend time investigating.

4.4.2 Debugging individual traces

A common and well-established approach to debugging with traces is to manually look through "interesting" or slow traces. For example, one can use Lightstep to identify the slowest spans and look at the corresponding traces [62], or one can use Jaeger directly to view/compare the high latency traces [16]. The problem with this approach is that individual traces can be misleading when investigating slowdowns.

Fig. 1 shows an example trace of a 99th percentile request, and Fig. 7 shows two other traces near the 99th percentile. The three traces show completely different results. In Fig. 7a, one would think to investigate the ReadUserTimeline operation in nginx-web-server. In Fig. 7b, one would think to investigate MongoFindUserTimeline in user-timeline-service. In Fig. 1, one would think to investigate subspan 3 in the ReadUserTimeline operation in user-timeline-service. Looking at any one of these traces could lead to a lot of wasted time chasing slowdowns that only affect an insignificant fraction of requests. Looking at all three of these traces could lead to confusion or a conclusion that it's all random noise. Users experience the same issues even when applying critical path analysis. It turns out that only Fig. 1 is an actual bug, which tprof identifies as the last slowdown in Tbl. 1.

We also use an advanced correlation feature in Lightstep to identify operations that are correlated with high latency following their tutorial [60]. The approach works by using their correlation metric to identify the operation to filter by, and then manually investigating traces where the operation is slow. While the correlation metric performs reasonably well in identifying the slow operation, the resulting traces are similarly misleading. Like with Jaeger, actual problematic traces are sometimes among these traces, but there is so much noise across traces that it's hard to pick out the consistently slow regions without aggregation.

5 Discussion

5.1 Generality of layers

In practice, users may sometimes want to partition or filter tracing data based on date/time and other attributes (e.g., HTTP status code). This can be accomplished by performing a user-defined pre-filtering step before using tprof.

Users may also desire to compare traces based on features other than request types, such as geographic locations of clients, traces traversing through a particular service, requests with warning/error messages, etc. In these cases, those features can be used alternatively to partition in layer 2.

5.2 Handling complex trace structures

Complex trace structures may result in many layer 4 groups, and tprof employs two design principles to address this challenge. First, the automated analysis can swiftly pinpoint the relevant trace group that manifests the performance issues. Our evaluation shows that tprof is able to accurately point out the problematic group from dozens to over one hundred layer 4 groups. Second, we use a hierarchical design to provide different granularities of aggregation. For example, slowdown 4 in Tbl. 1 is most accurately detected via the child_diff metric in layer 3, which aggregates nearly 9x more traces than the most relevant layer 4 group.

5.3 Subspans not on the critical path

Critical path analysis on traces, which has been studied in academia [10] and is commercially available [61], is orthogonal to tprof's contributions. Those approaches can be easily applied to the aggregate traces generated by tprof. Users can choose whether to consider subspans not on the critical path of the trace. Nevertheless, our evaluation in Sec. 4.3 shows that tprof is able to unveil many unknown performance issues without critical path information.

6 Conclusion

We present tprof as a new performance profiler that aggregates distributed systems traces to diagnose performance bugs and inefficiencies. It uses four different analyses to hierarchically group and aggregate traces based on the trace structure. Our novel subspan analysis enables the creation of aggregate traces for easy visualization. Our automated performance diagnosis tool analyzes the aggregated statistics to report the most likely performance issues. Our evaluation demonstrates how tprof is able to accurately identify both tail latency and average latency bugs, leading to 28× and 10× improvements, respectively.

Appendix

A Proof of aggregate trace: Theorem 1

Proof. We begin by showing that (i) spans can always be decomposed into a sum of subspans and child spans; (ii) the time between the beginning of a span till the start of any child span can also be decomposed into a sum of subspans and child spans. It then follows from a linearity of expectation argument that the aggregate traces are valid.

To give some concrete examples on decomposing a span (i), consider the example traces in Fig. 5. In a group 1 trace, A = A0 + B + A1 + C + A2. In a group 2 trace, A = A0 + A1 + C + A2. In a group 3 trace, A = A0 + B + A2. It is always possible to find this decomposition due to how subspans are defined. Starting at the end of the span, we find the prior event in time, and this period refers to the last subspan, which is part of the decomposition. The prior event must be one of the other events (*start of span*, *end of child span*, *start of child span*). If it is the *start of span*, then we are done. If it is the *end of child span*, then the next component in our decomposition would be that child, and we would repeat this decomposition process from the start of that child's span. If it is the *start of child span*, then the next component in our decomposition would be the subspan ending at this event and starting at the prior event, and we would repeat this decomposition process from the prior event. In other words, we can recursively define decomposition as follows:

```
decompose(event) =
  if (event is start of span)
    return None
  if (event is end of span or
      event is start of child span)
    return subspan(prior event, event)
         + decompose(prior event)
  if (event is end of child span)
    return child_span(start of child span, event)
         + decompose(start of child span)
```

This decomposition also applies to the time between the beginning of a span till the start of any child span (ii).

We can now treat each component in the decomposition as a random variable. We do not need to make any assumptions about the distribution of the random variables or even whether they are independent or not. By the linearity of expectation, the expected value of a sum of random variables can always be decomposed as a sum of expected values. Therefore, by using averages for each component (i.e., subspan or child span), we maintain the overall average span duration as well as the average start time of each child span, and thus our aggregate traces are always valid. We note that this only applies to averages since other metrics do not decompose linearly, as explained in Sec. 3.5. □

References

[1] 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

[2] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. 2019. An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 165–170. https://doi.org/10.1145/3357223.3362704

[3] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 307–320. https://www.usenix.org/system/files/conference/osdi12/osdi12-final-33.pdf

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling

[5] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging Distributed Systems. *Commun. ACM* 59, 8 (July 2016), 32–37. https://doi.org/10.1145/2909480

[6] Pengfei Chen, Yong Qi, and Di Hou. 2015. InvarNet-X : A Black-Box Invariant-Based Approach to Diagnosing Big Data Systems. *IEEE Transactions on Emerging Topics in Computing* 5, 4 (nov 2015), 450–465. https://doi.org/10.1109/tetc.2015.2497143

[7] Pengfei Chen, Yong Qi, and Di Hou. 2016. CauseInfer : Automated End-to-End Performance Diagnosis with Hierarchical Causality Graph in Cloud Environment. *IEEE Transactions on Services Computing* 12, 2 (sep 2016), 214–230. https://doi.org/10.1109/tsc.2016.2607739

[8] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2009. Automated anomaly detection and performance modeling of enterprise applications. *ACM Transactions on Computer Systems* 27, 3 (nov 2009), 1–32. https://doi.org/10.1145/1629087.1629089

[9] L. Cherkasova, K. Ozonat, Ningfang Mi, J. Symons, and E. Smirni. 2008. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 452–461. https://doi.org/10.1109/DSN.2008.4630116

[10] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[11] Charlie Curtsinger and Emery D. Berger. 2016. COZ: Finding Code that Counts with Causal Profiling. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/atc16/technical-sessions/presentation/curtsinger

[12] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing - SOCC '14*. ACM Press, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670987

[13] Distributed Systems Tracing with Zipkin 2012. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin

[14] Pinterest Engineering. 2017. Analyzing distributed trace data. https://medium.com/pinterest-engineering/analyzing-distributed-trace-data-6aae58919949

[15] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems* 30, 4 (2012), 311–326. https://doi.org/10.1145/2382553.2382555

[16] Joe Farro. 2018. Trace comparisons arrive in Jaeger 1.7. https://medium.com/jaegertracing/trace-comparisons-arrive-in-jaeger-1-7-a97ad5e2d05d

[17] Fix Performance Bottlenecks with Intel® VTune™ Profiler Septermber 2021. https://software.intel.com/en-us/vtune.

[18] 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

[19] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[20] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

[21] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[22] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 19–33. https://doi.org/10.1145/3297858.3304004

[23] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

[24] Haryadi S. Gunawi, Vincentius Martin, Anang D. Satria, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, and Jeffrey F. Lukman. 2014. What Bugs Live in the Cloud?. In *Proceedings of the ACM Symposium on Cloud Computing - SOCC '14*. 1–14. https://doi.

org/10.1145/2670979.2670986

[25] Dan Gunter, Brian L. Tierney, Aaron Brown, Martin Swany, John Bresnahan, and Jennifer M. Schopf. 2007. Log summarization and anomaly detection for troubleshooting distributed systems. In *2007 8th IEEE/ACM International Conference on Grid Computing*. IEEE, 226–234. https://doi.org/10.1109/GRID.2007.4354137

[26] Jaeger: open source, end-to-end distributed tracing 2021. https://www.jaegertracing.io/.

[27] Java Profiler - JProfiler 2021. https://www.ej-technologies.com/products/jprofiler/overview.html.

[28] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, New York, NY, USA, 469–478. https://doi.org/10.1145/3038912.3052649

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

[30] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 326–332. https://doi.org/10.1145/3267809.3267841

[31] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19* (2019), 312–324. https://doi.org/10.1145/3357223.3362736

[32] Lightstep: The DevOps observability platform 2021. https://lightstep.com/.

[33] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 48–58. https://doi.org/10.1109/ISSRE5003.2020.00014

[34] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases*. Survey. Brown University.

[35] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 18 pages. https://doi.org/10.1145/3190508.3190526

[36] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[37] Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient Flow Profiling for Detecting Performance Bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2931037.2931066

[38] S. Nedelkoski, J. Cardoso, and O. Kao. 2019. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 241–250. https://doi.org/10.1109/CCGRID.2019.00038

[39] S. Nedelkoski, J. Cardoso, and O. Kao. 2019. Anomaly Detection from System Tracing Data Using Multimodal Deep Learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 179–186. https://doi.org/10.1109/CLOUD.2019.00038

[40] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. 2013. FChain: Toward black-box online fault localization for cloud systems. In *Proceedings - International Conference on Distributed Computing Systems*. 21–30. https://doi.org/10.1109/ICDCS.2013.26

[41] OpenTelemetry auto-instrumentation and instrumentation libraries for Java 2021. https://github.com/open-telemetry/opentelemetry-java-instrumentation.

[42] OpenTelemetry instrumentation for Python modules 2021. https://github.com/open-telemetry/opentelemetry-python-contrib.

[43] OpenTelemetry JavaScript Client 2021. https://github.com/open-telemetry/opentelemetry-js.

[44] OpenZipkin · A distributed tracing system Septermber 2021. https://zipkin.io/.

[45] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[46] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated. https://books.google.com/books?id=fgfIyAEACAAJ

[47] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[48] Navin Sabharwal and Ravi Shankar. 2013. *Apache CloudStack Cloud Computing*. Packt Publishing.

[49] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16*. ACM Press, New York, NY, USA, 401–414. https://doi.org/10.1145/2987550.2987568

[50] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/diagnosing-performance-changes-comparing-request-flows

[51] Huasong Shan, Yunpeng Zhang, Yuan Chen, Xiao Xiao, Haifeng Liu, Xiaofeng He, Min Li, and Wei Ding. 2019. ϵ-Diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In *The Web Conference 2019 - Proceedings of the World Wide Web Conference, WWW 2019*. 3215–3222. https://doi.org/10.1145/3308558.3313653

[52] Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. 2013. CloudPD: Problem determination and diagnosis in shared dynamic clouds. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Vol. 1. IEEE, 1–12. https://doi.org/10.1109/DSN.2013.6575298

[53] 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

[54] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies via Lightweight Variance Profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 19 pages. https://doi.org/10.1145/3295500.3356167

[55] The Python Profilers 2021. https://docs.python.org/3/library/profile.html.

[56] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. 2006. Stardust: Tracking Activity in a Distributed Storage System. *SIGMETRICS Perform. Eval. Rev.* 34, 1 (June 2006), 3–14. https://doi.org/10.1145/1140103.1140280

[57] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. CloudRanger: Root Cause Identification for Cloud Native Systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 492–502. https://doi.org/10.1109/CCGRID.2018.00076

[58] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 47–61. https://doi.org/10.1145/3037697.3037729

[59] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. *SIGPLAN Not.* 53, 2 (March 2018), 332–347. https://doi.org/10.1145/3296957.3177159

[60] Robin Whitmore. 2019. Find Correlated Areas of Latency and Errors | Lightstep Learning Portal. https://docs.lightstep.com/docs/find-correlated-areas-of-latency

[61] Robin Whitmore. 2019. See the Critical Path | Lightstep Learning Portal. https://docs.lightstep.com/docs/view-traces#see-the-critical-path

[62] Robin Whitmore. 2019. View Traces | Lightstep Learning Portal. https://docs.lightstep.com/docs/view-traces

[63] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[64] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. ACM Press, New York, NY, USA, 117. https://doi.org/10.1145/1629575.1629587

[65] Xu Zhang, Qingwei Lin, Yong Xu, Si Qin, Hongyu Zhang, Bo Qiao, Yingnong Dang, Xinsheng Yang, Qian Cheng, Murali Chintalapati, Youjiang Wu, Ken Hsieh, Kaixin Sui, Xin Meng, Yaohai Xu, Wenchi Zhang, Furao Shen, and Dongmei Zhang. 2019. Cross-dataset Time Series Anomaly Detection for Cloud Systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1063–1076. https://www.usenix.org/conference/atc19/presentation/zhang-xu

[66] Zhihong Zhang, Jianfeng Zhan, Yong Li, Lei Wang, Dan Meng, and Bo Sang. 2009. Precise request tracing and performance debugging for multi-tier services of black boxes. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 337–346. https://doi.org/10.1109/DSN.2009.5270321

[67] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 603–618. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao

[68] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 629–644. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao

[69] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 14, 8 (2018), 1–18. https://doi.org/10.1109/TSE.2018.2887384

[70] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. *Proceedings - International Conference on Software Engineering* (2018), 323–324. https://doi.org/10.1145/3183440.3194991