Using Performance Variation for Instrumentation Placement in Distributed Systems

Using Performance Variation for Instrumentation Placement in Distributed Systems

Lilian Sturmann

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2019

Abstract

Distributed systems are now ubiquitous in the infrastructures underpinning our everyday lives, yet diagnosing performance problems in these systems remains extremely challenging. The current state of the art for problem diagnosis in these systems relies on data from instrumentation in the system, but the placement of this instrumentation is an unsolved challenge in systems research and in production environments.

This work presents an implementation and evaluation of a performance variation-based tool that helps developers understand where instrumentation should be placed in a distributed system to better diagnose current and future performance problems. This tool identifies under-instrumented regions in these systems by localizing performance variation seen in system requests. Contributions of this work include the tool itself; implementations of several methods for localizing performance variation, including a method that prioritizes performance variation deeper in request call graphs; a conversion module that can also function as a stand-alone toolkit to allow the performance variation-based tool to be used across a variety of systems, including those instrumented using the Open Tracing model as well as those using a more general directed acyclic graph (DAG) models; and several experiments evaluating the tool and these methods on an open source distributed application.

They key insight informing this work is that similar workflows in the same system should perform similarly. Building on existing workflow-centric tracing tools

to profile system behavior, the tools and methods presented have the potential to significantly cut down on time spent diagnosing performance problems in distributed systems. The experiments evaluate their utility both for understanding where to place additional instrumentation for current problems in these systems as they arise, and for guiding informative placement of default system instrumentation to better handle future problems. Potentially, the tools and methods could also be adapted for use in a broader framework that seeks to dynamically tune instrumentation in running systems to the current system state.

Acknowledgements

I am deeply appreciative of the personal and professional support and guidance I have received during this research. First and foremost, I would like to thank my thesis director, Dr. Raja Sambasivan, whose expertise in distributed systems and the research process was of immense benefit to me and this thesis, and who continuously took the time to explore ideas with and present new opportunities to me and other students.

I am also grateful for all the support I have received from mentors at the Mass Open Cloud and the Boston University - Red Hat Collaboratory, and for the opportunity to work on fascinating and challenging projects throughout my internship, and to grow as an engineer and researcher in such a unique environment. Of course, I would like to give special thanks to my teammates on the tracing team at the MOC.

Further, I want to express my thanks to the Harvard professors who helped me discover more about the topics I love most in computer science, a prerequisite to the work presented here. I want to give special thanks to those professors and teaching fellows who stayed late answering questions at office hours, and to the advisors and administration who helped me navigate the research process.

Certainly not least, I am grateful for the encouragement and long-standing patience of my partner, who among many other things provided me with snacks throughout this process. I would also like to express gratitude to my parents for their long-distance words of support, and to my cat for his tolerance of my human activity.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

The desire to make systems more reliable is a powerful one; unfortunately, this addiction, if left unchecked, will inescapably lead to madness...

–James Mickens (Mickens, 2013)

Distributed computing systems form a core part of the infrastructures we rely on. They hold medical patient data (Raza, 2017; Rowe, 2017) and are present at the forefront of scientific discovery in areas like particle physics – the Worldwide LHC Computing Grid spanned 174 facilities in 2015 to support the Large Hadron Collider (Whyntie & Coles, 2015). They are increasingly mixed in with banking in the form of APIs to access bank accounts (Darrow, 2016; Whomes, 2018). Distributed systems also support the accessible data storage many people know and love as Google Drive (Google Drive, nd). Any time that people access the Internet, they are interacting with a distributed system when they route requests through DNS or load websites that are hosted on distributed content delivery networks like Akamai (Rollins, nd; Akamai, nd).

The definition of a distributed system as "multiple software components that are on multiple computers, but run as a single system" in which the multiple computers are potentially "geographically distant" (IBM, nd) and could number in the tens of thousands (Kozyrakis, 2013) applies to the systems supporting the majority of computing activities that individuals and businesses participate in today – from

communicating on social media to collaborating with teams in other companies via file sharing or VoIP services. For many people, the complex systems on which people and organizations store their data or interact with software as a service are known simply as "the cloud." Due to its prevalence in business, research, and other services, the "worldwide public cloud services market is projected to grow 17.3 percent in 2019 to total $206.2 billion" according to Gartner, with the fastest growing portion being infrastructure as a service (Costello & Hippold, 2018).

Yet despite the now heavy societal reliance on these systems and their subtle protrusion into most facets of daily life in industrialized nations, these systems' maintenance, in terms of diagnosis of performance problems and failures, remains a largely unsolved problem in systems research as well as production. By definition, distributed systems are comprised of multiple software or hardware components or both; in reality, these are often vast numbers of intricately interrelated components. Warehouse-scale computing, common today and used by Amazon and Google, is a class of distributed computing that can involve tens to hundreds of thousands of servers acting as one system (Barroso & Holzle, 2009; Kozyrakis, 2013). Especially as it becomes more common to interact with systems of this scale, it is difficult to know which of potentially thousands of servers, the applications they are running, their network connections, their lower stack levels, hardware, or myriad other moving parts could be responsible for a malfunction or performance degradation. It can be especially confounding when the malfunction lies not with any single component or application, but instead manifests only in the interactions between several components that may function well in isolation. In one example documented by the engineers at LightStep, a system failure was eventually tracked not to a single root cause, but to the interactions of two separate changes in the system, one relating to the code itself and the other to a new traffic pattern of requests to the service (Asemanfar, 2018).

In the face of such challenges, it is not surprising that debugging distributed systems is known to be notoriously time-consuming and expensive. A joint technical report from Lloyd's of London and AIR Worldwide estimated in 2018 that an outage in one of the "top cloud firms," such as Amazon Web Services, Microsoft's Azure, Google, IBM, and Alibaba (Ranger, 2018), could cost up to $19 billion in just six days (Nunns, 2018; Lloyd's, 2018). As distributed systems continue to increase in size and complexity (Ranganathan & Campbell, 2007), debugging these systems will continue to grow more challenging as well as more critical.

Today's solutions to aid engineers with problem diagnosis for performance degradation or other malfunctions rely on some of the many components that comprise a distributed system having the capability of recording metrics about the system's performance or behavior. This can take the form of code embedded in a distributed application that is external to and not necessary for the application's functionality. For a basic example, this code could record each time a system component is accessed and this record could be sent or stored in a specific place for later perusal. These externally inserted record-keeping mechanisms are commonly referred to as instrumentation points, logging points, or just metrics, instrumentation, logs, or logging in aggregate in a monitored system. These logs are typically used either to alert system operators to some unusual behavior that bears examination in case it reflects an urgent system problem, or to aid efforts in finding a root cause during some already known system problem or failure event. Instrumentation is critical in the quest to avoid system problems, repair current performance degradations or failures, and understand performance degradation or problems that have occurred in the past (Beyer et al., 2016, p. 56-57,138).

It is clear that well-placed instrumentation has a significant potential to cut down on financial and reputational costs of performance degradation and other mal-

functions in distributed systems and to directly or indirectly aid the many sectors of society relying on these systems today. However, the nature of these instrumentation points means that data about system behavior is only captured at the specific points in the system where the instrumentation is placed, and this presents difficulty. Unfortunately, it is not realistic to expect that developers placing instrumentation points in systems will be able to foresee all the areas that need to be instrumented, regardless of their domain expertise. This is especially true given that the "right" instrumentation for a system depends on that system's current state. This means that instrumentation must often be added to a system to diagnose a specific problem, as generalized instrumentation points have limited utility since they will not show enough detail in the parts of the system where a particular problem happens to be. As noted by Kaldor, J. et al: "diagnosing performance issues ... requires a global view, yet granular attribution, of performance" (Kaldor, 2017). Consequently, adding instrumentation in the right areas is a recurring challenege both during system development and also in the face of the many new problems the system could experience that need more fine-grained instrumentation in a certain area, above and beyond the instrumentation present in that area by default, to diagnose the root cause.

This work presents a tool that aims to identify under-instrumented regions of distributed systems, assuming some instrumentation is already in place, to help with these instrumentation efforts in these two sparate scenarios. First, the tool aims to help with default instrumentation placement, in which the goal is to have instrumentation in place to differentiate between differing requests in a normally-running system. This is important because well-placed instrumentation before problems are detected can ease the difficulty of figuring out where to add more instrumentation when problems do arise. Second, the tool aims to give engineers a better set of starting points in diagnosing the root cause of some immediate performance problem

that has arisen in the system. The same technique is applied in both cases, and this work shows the utility of examining placement of default instrumentation with the performance variation-based tool as well as the correspondence between performance variation and problems introduced into the test system.

The tool and its underlying technique and methods are based on the insight from previous work that performance variation in system requests is a marker of some unknown system behavior (Sambasivan et al., 2011; Sambasivan & Ganger, 2012), and indicative of an under-instrumented region. While this insight will be explained in more detail in the Approach section, it is crucial to understand that unknown system behavior and an under-instrumented region are two ways of saying the same thing about system activity, our knowledge of which is only as specific as the instrumentation present. More detailed instrumentation necessarily cuts down on unknowns about system behavior. It is the judicious placement of this instrumentation that allows for more detailed data to be collected in those system areas where it is more useful and less detailed data in areas where it is not. The ability to make this determination both cuts down on overhead and intrinsically passes a smaller amount of more relevant diagnosis-related data to the engineer.

Chapter 2: Problem Statement

The central question this work seeks to answer is: "Can under-instrumented regions in distributed systems that contribute to differing performance among similar requests be identified using performance variation?" This thesis will build on previous work (Sambasivan et al., 2011; Sambasivan & Ganger, 2012) to evaluate the potential of using performance variation as an indicator of under-instrumented regions in the system and of problematic system behavior. To do so, the thesis presents a tool for developers and engineers to guide instrumentation choices in distributed systems by distinguishing between requests with different performance characteristics. Given some instrumentation already present in the system, the tool helps identify any regions where a higher density of instrumentation could provide more useful information. This work also aims to validate the hypothesis that performance variation observed in distributed systems is tied to the root cause of performance degradation problems in these systems. As the approach relies on detecting differences in request latency or completion time, it will be applicable to system problems involving noticeable performance degradation, for example involving resource contention, improper load balancing, execution of an unexpected code path, or misconfiguration problems such as portions of a request being forwarded to the an incorrect component.

Chapter 3: Prior Work

The body of work on instrumentation in distributed systems has traditionally focused on failure cases and on improving or analyzing the information output of existing instrumentation points (Mariani & Pastore, 2008; Mariani et al., 2009; Xu et al., 2009; Yuan et al., 2010, 2012c,b). A popular approach in this research area is to focus on post-mortems, i.e. using logs to understand the root cause of a system failure. For example, research has been done on optimizing the location of logging points to differentiate different paths that an application takes (Zhao et al., 2017) to aid in failure diagnosis, as well as research on what to include in the logs (Yuan et al., 2012c,a).

This thesis differs in its alignment to the problem of informative distributed system instrumentation by focusing on performance degradation problems in a running system, such as those caused by resource contention or misconfiguration, and doing groundwork for adding new instrumentation points into the system by analyzing where such instrumentation may be useful. This question of where to place instrumentation in the system is more challenging, and impacts all tools and research focused on gaining visibility into the system, as these are necessarily limited by the instrumentation present.

Previous work that has a similar focus on locating where to place new instrumentation points includes Yuan et al. (2012a) and Zhao et al. (2017). The former, which presented the tool called ErrLog, focused on static analysis with a run-time

component and exclusively on exceptions. They analyzed the source code to identify unlogged exceptions and insert logs in these areas.

Zhao et al. (2017) and Ball & Larus (1996) have both taken the approach of differentiating code paths – Ball & Larus (1996) differentiates every possible code path, while Zhao et al. (2017) differentiates only those execution paths in the currently analyzed workload. While differentiating code paths is useful for post-mortem analysis, it is not necessarily the best approach for diagnosing performance problems. With performance problems, it is possible for requests experiencing performance degradation to have identical code-paths with requests that are not. Their work is similar in its aims but differs in its methodology from the work in this thesis.

Yuan et al. (2012c), presenting LogEnhancer, also focuses on adding new ways of collecting diagnostic information to the system, but does this by adding variables to existing instrumentation points to collect relevant data. Their work is not concerned with finding under-instrumented areas.

The work in this thesis is based on the approach of localizing problems in distributed systems by focusing on the variation of performance in similar requests proposed by Sambasivan et al. (Sambasivan & Ganger, 2012; Sambasivan et al., 2011). Later, Huang et al. (Huang et al., 2017) proposed a mathematical model for such analysis on single-node systems.

Chapter 4: Tool Goals and Requirements

The main goal of the tool is to provide guidance about where to add instrumentation in the system to aid in diagnosing both current and future problems. To meet this goal, one set of requirements is that the output of the tool should be human-meaningful data regarding which traces were processed, which of these could be expected to perform similarly, whether any had high performance variation, and to which part of the request the variation, if any, can be localized. To make the output easier to understand, high performance variation should be ranked by some meaningful characteristic, such as highest to lowest performance variation, in the output in cases where multiple regions of high performance variation exist. This output is meant to serve as a guide for where more instrumentation may be necessary.

A secondary goal is that the technique the tool is based on should be applicable to a variety of distributed systems and tracing infrastructures, and that the tool itself should be adaptable to different systems without changing its main mechanisms. This is why it was necessary for the tool to consume directed acyclic graphs as input, as these are the most expressive and general form that traces can take and will be applicable to the widest variety of systems and tracing infrastructures (see Section 6.2: Design Choices: The Performance Variation-Based Tool). This requirement also informed the decision for the tool to be able to consume both .JSON files, which are associated with span model traces, and .DOT files, which are associated with DAG model traces.

Chapter 5: Approach

The approach for this work is based on the key insight that performance variation can serve as a marker of unknown system behavior (Sambasivan et al., 2011; Sambasivan & Ganger, 2012). To observe, analyze, and localize performance variation, the approach relies on existing workflow-centric tracing tools. The insight and the workflow-centric tracing tools it relies on are explained in more detail in the below subsections.

5.1. Performance Variation Marks Unknown System Behavior

Any command or request sent to a system will trigger a series of actions the system takes to process that command within and among the system's components. This series of actions from beginning to end is known as the request's associated workflow. Two workflows are depicted in figure 5.1 (Ates et al., 2019). Each request's workflow has a total response time, also known as its overall latency, that is a measure of how long the request took to complete. This response time or latency is also referred to as the request's performance, in terms of time, ex. microseconds or milliseconds.

It is expected that requests that exhibit similar workflows—i.e., that are processed similarly within and among the nodes of a distributed application and by lower stack layers—should perform similarly in the same system because they are doing sim-

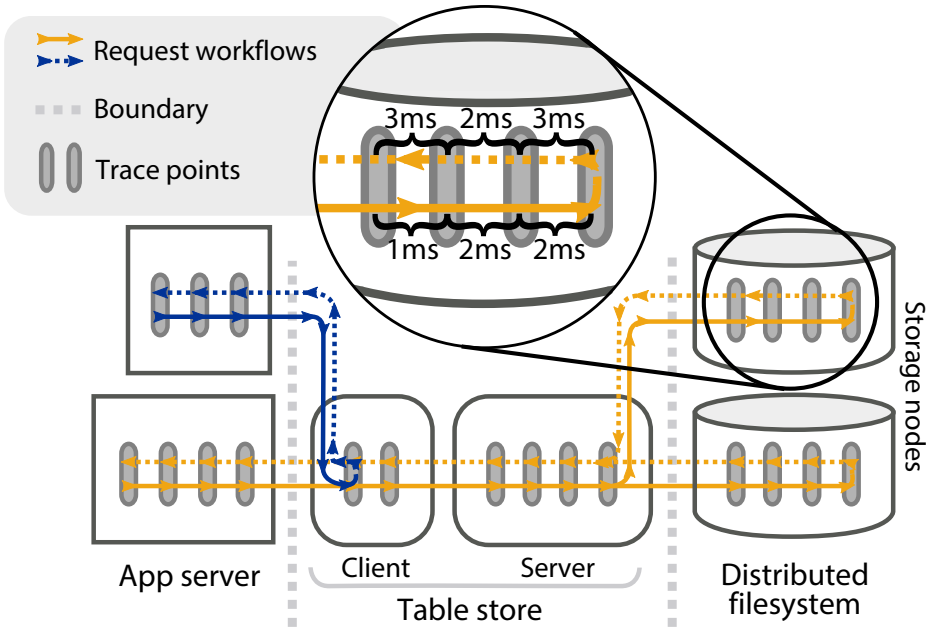

Figure 5.1: Two request workflows.

ilar work (Sambasivan et al., 2011; Sambasivan & Ganger, 2012). For example, one may expect that read requests in the same system are relatively similar to each other but somewhat different from write requests in the same system. This expectation would manifest in the response times or overall latencies of these requests.

When this expectation is not met and the performance of two seemingly similar workflows significantly differs, it means the expectation was incorrect and that the workflows were actually different in some way: there is some differentiating factor between the workflows that was not considered. This could be due to acceptable and expected heterogeneous behavior in the system, or could indicate a system problem, such as a misconfiguration. For example, consider two read requests that are issued to the same system. Because they are both read requests, one might expect that they are processed similarly in the system and are doing the same work, which would indicate that they should have similar performance. If one of the read requests is far slower than the other, it indicates that there must be some difference between

their workflows. Perhaps one of the read requests was able to retrieve the requested information from a cache and so was much faster than the other request, which missed in the cache and had to retrieve the information from storage on another node. Perhaps the information the requests needed to retrieve was held on different storage nodes and one could be accessed more quickly than the other.

This difference in performance, in terms of total response time, can be quantified by looking at performance variation among a group of request workflows. Performance variation can be assessed by calculating the variance or coefficient of variation values for a group of requests. These values quantify the range of difference from the mean response time for the group, or, intuitively, how closely the response times for the group cluster around a central value.

A significant benefit of the tool is that it goes beyond assessing whether a group of requests has high performance variation and localizes the source of the variation to a more specific system area within the path of the request, i.e., a specific system region that is involved in processing that type of request. These regions can be the components of a distributed system, such as compute or storage, or can refer to specific virtual or physical machines or even to functions or variables. When performance variation can be localized to a more specific area, the analysis is more informative, since localizing the source of the observed variation gives insight into where additional instrumentation may be needed to identify the unknown behavior. However, due to the way workflow data is obtained (described in Section 5.2 below), it is not always possible to localize variation to the most specific areas.

For meaningful performance variation values, only those requests which are expected to perform similarly should be assessed together; a group of requests performing many different actions in the system will inherently have high performance variation, and this is expected. Understanding how expectations are formed about

which requests should perform similarly, as well as understanding how performance variation in a group of requests can be localized, are both dependent on understanding representations of system workflows and what data is available about them from the instrumentation. This is described in the section below on workflow-centric tracing.

5.2. Workflow-Centric Tracing Records Request Behavior

In order to assess performance variation of requests in the system, one must have some way of capturing and reconstructing the work done in the system on behalf of each request. At minimum, a subset of work done on behalf of the request, order in which it was completed, and the latency between each recorded action must be available for analysis. For this data, this work relied on pre-existing workflow-centric tracing tools to monitor the system under observation.

The mechanisms of are explained through this excerpt from a paper recently submitted for review to a conference by our team (Ates et al., 2019):

> Workflow-centric tracing of distributed applications, also called end-to-end tracing, (Chen et al., 2004; Thereska et al., 2006; Reynolds et al., 2006; Fonseca et al., 2007, 2010; Sigelman et al., 2010; Sambasivan et al., 2016; Kaldor et al., 2017; Mace & Fonseca, 2018; OpenTracing website, nd; Jaeger, nd) makes it possible to capture requests' workflows to varying degrees of fidelity. The degree of fidelity depends on the amount of tracing instrumentation present in the application. Workflow-centric tracing works by propagating context (e.g., an ID unique to a request) with individual requests as the requests are executed by the nodes of a distributed application. This allows records of log points executed by requests to be tagged with requests' context, effectively turning the system's log points

into trace points. Sampling techniques can be used to keep tracing's overhead low enough (e.g., $< 1\%$) to be used in production, as is done at many companies today (Sigelman et al., 2010; Kaldor et al., 2017; OpenTracing website, nd; Jaeger, nd).

It is important to note that while tracing systems have surpassed logging in providing relevant, causal information about systems to make debugging more efficient, these tracing systems face the same severe placement-related limitations as logging due to their reliance on instrumentation placed a priori within the distributed system.

After the data about a request's workflowis captured by the tracing system, this data is stored in a back end from which a trace reconstructor can gather trace point records from different machines and stitch together those with related context to create traces of requests' workflows. This reconstructor can be triggered by a querying system in which a request ID is provided to the back end and the back end returns an ordered set of records associated with the request from the trace points that were on the path of the relevant request.

Chapter 6: Design

6.1. System Overview

The performance variation-based tool is comprised of several logical units of work, or phases, that process sets of workflow-centric traces and emit output indicating to which regions of these traces any performance variation can be localized. These phases are:

- Phase 1: Input parsing
- Phase 2: DAG conversion
- Phase 3: Critical path extraction
- Phase 4: Grouping based on expectation of performance
- Phase 5: Analysis and localization of performance variation
- Phase 6: Output

It will be apparent that the first four phases in this pipeline can be considered pre-processing. Any time the tool is invoked, the set of traces forming its input go through each of these logical units of work, with the exception of the DAG conversion, which is optional to cover the case in which the traces are already DAGs, and the extraction of the critical path, which is necessary in systems with large amounts of

parallel processing, like Ceph, but not otherwise. Additionally, several parts of this pipeline can be used as stand-alone tools as well; for example, the DAG conversion can be applied to a single trace when called from the command line, without invoking the larger framework of the tool itself.

Section 6.2 below highlights the design choices that arose in relation to several of the phases listed above, while section 6.3 describes the design considerations of the test environment in which the tool was run.

6.2. Design Choices: The Performance Variation-Based Tool

This section highlights some of the design considerations regarding building the tool itself. These highlights are related to Phases 2 (DAG conversion), 4 (grouping), and 5 (analysis and localization) above, as they were the most involved and most interesting design considerations.

6.2.1 The DAG As a More Expressive Tracing Model

The design choice of which tracing model to use relates to Phase 2: converting traces to DAGs in the System Overview.

A request's reconstructed workflow is known as a trace of the request and can take several graph-like forms. The Open Tracing model (OpenTracing website, nd), also known as the span or swim lane model, prioritizes showing call graph relationships at the expense of happened-before-relationships (Lamport, 1978) that track causality between events. By contrast, the more expressive DAG model is able to record both call graph relationships as well as happened-before-relationships. This becomes important especially when requests' workflows in the traces exhibit any of these behaviors:

- Concurrent behavior, in which multiple tasks executed on behalf of the request overlap in time on what is referred to as different branches of the trace;
- Synchronization points, in which some part of a request must wait for two or more concurrent branches to complete before joining them into one;
- Asynchronous behavior, in which concurrency exists with no following synchronization point.

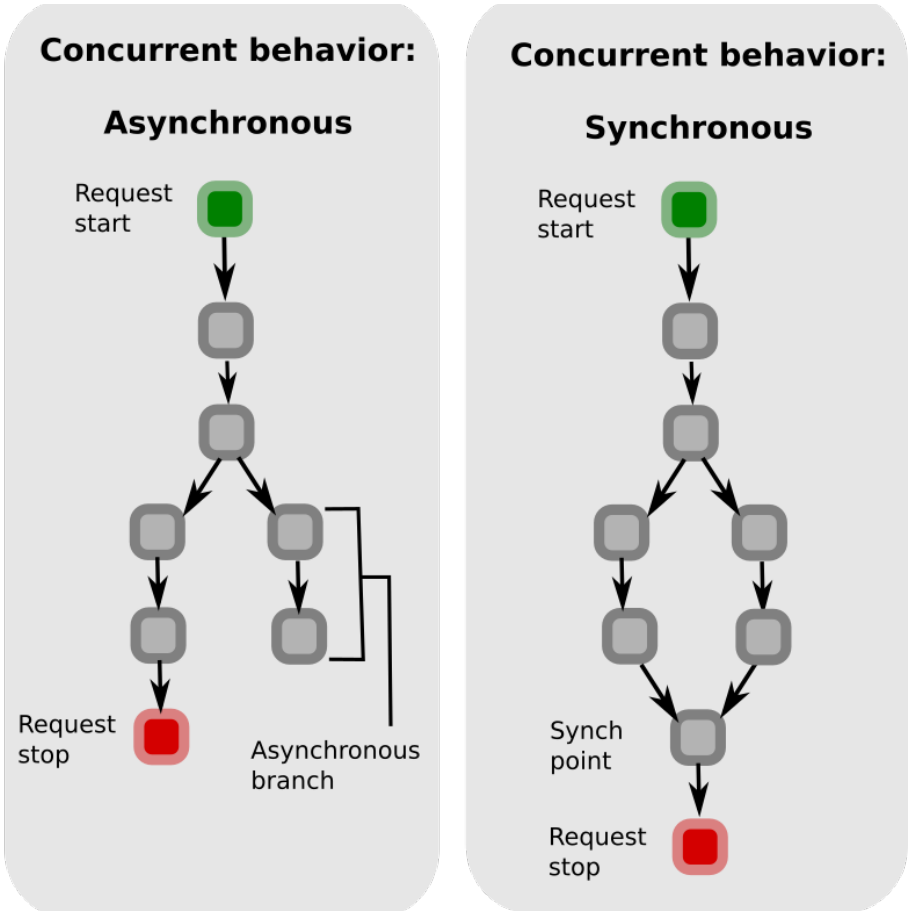

Figure 6.1: Concurrent behavior in traces: asynchronous vs. synchronous.

In figure 6.2, the same request is shown in the contrasting diagrams, with the span model on the left and the DAG model on the right. This request consists of a main function that calls three other functions: a read() function, followed by

a write() function, and lastly an execute function(). Each span or bracket in the span model represents a semantic unit of work, a human-meaningful and arbitrarily designated set of system tasks, that takes a span of time to execute. The nested structure corresponds to parent-child relationships between these units of work in which a span nested inside another is called by the other. These are highlighted by the dotted arrows from the outer main() function to its children. Of note is the fact that while the span model explicitly shows these call graph relationships – that is, the main() function's relationship as a parent function to the read(), write(), and execute() functions – it does not have any explicit way of notating the relationships of the read(), write(), and execute() functions to each other. They may have occurred in any order; they may have all been sequential, or two or all three of them could have happened concurrently.

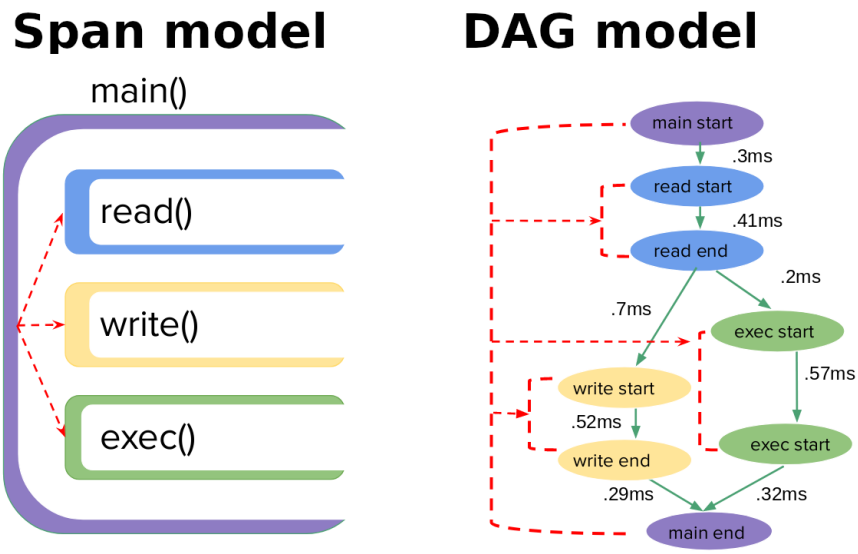

Figure 6.2: Example of trace in both span and DAG models.

The DAG model on the right in figure 6.2 shows this same trace of the same request in a different form. This model is explicit about the causal relationships,

or happened-before-relationships, between each of the events and function activities. The "spans" from the span model are broken down into more the more specific events of start and end times for each of the functions. In this model, these start and end events are captured as the nodes of the graph, while the edges of the graph, represented with continuous lines, represent causal or happened-before-relationships between them and generally are labeled with latency values. In the example, the following relationships are evident: 1) the read() function finishes before, i.e. has a happened-before-relationship with, the write() and execute() functions; 2) the write() and execute() functions are concurrent with respect to each other; 3) read(), write(), and execute() are all called by main() as shown by the nested brackets super-imposed over the DAG. It is also evident that each event that is "downstream" of another has a causal relationship with it, even if they are not adjacent; the start of the read() event has a happened-before-relationship with the end of the write() event, for example. Because causal as well as call graph relationships can be explicitly noted as part of the graph, it is also easier to use this model for programmatic analysis as is done with the performance variation tool in this thesis.

The explicit causal relationships of the DAG model also allow for the critical path of the request to be extracted. The critical path of a request is the sequential set of work that must be done in order for the request to complete, and each part of the critical path is directly responsible for some part of the latency of the request. Therefore, the critical path is exactly the portion of the request's workflowthat will impact performance variation. In practice, the critical path is the entire request in cases with no concurrency, and is the temporally longest branch of the request in cases with concurrency that is followed by synchronization. In cases where there is asynchronous behavior in the workflow, that is some concurrency without a synchronization point, the asynchronous branches are not considered part of the critical path

because no part of the workflow waits for them to complete.

In this work, only the DAG model was used to represent traces because only this model is able to express causal relationships in systems, like Ceph or Hadoop, that exhibit concurrency and synchronization points in processing workloads. Therefore in order to make this work broadly applicable to a variety of systems, a conversion process had to be implemented for span model traces to get them into a DAG format. (For more details on the conversion process, see Section 7.2.1 in Implementation.) However, since our workflows in the OpenStack test environment did not always exhibit concurrency, the process of extracting the critical path was omitted in these cases.

6.2.2 Effects of Instrumentation Granularity in Traces

The design necessity of grouping certain traces together, and considerations about what characteristics to use to do so, relate to Phase 4: grouping traces, shown in the System Overview at the beginning of this chapter. This step is crucial to understanding how performance variation and instrumentation are related.

As the basis of the analysis is a set of expectations about which requests should perform similarly, this expectation must be formalized in some way before the tool can conduct any analysis. The tool uses trace structure to formalize these expectations and differentiate between workflows that have different performance expectations. The utility of this design choice is highlighted by the way workflow-centric tracing reconstructs the work done in the system by requests.

As described in Section 5.2, the trace as a record of a request's workflow depends on context propagation through existing instrumentation points in the distributed system. The density or sparseness of detail in this record corresponds to the density or sparseness of instrumentation along the request's path of processing

through the system. In this sense, there is no such thing as a "fully instrumented" trace – there are only more and less expressive records of the workflow. It is in this context that one must understand how performance variation can serve as a marker for under-instrumented regions and guide instrumentation placement in the system.

Since the same request can leave traces of differing levels of detail depending on the instrumentation present in the system, adding instrumentation on the path of a request can reveal differentiating factors in the workflows of requests that previously manifested as identical or similar. Some of these differentiating factors, at the right level of granularity, can reveal or point toward the root causes of performance degradation. For example, a group of requests that each create a virtual server, but half of which perform much more slowly than the other half, may all seem similar until an additional trace point is added that reveals that the slow requests are being processed on a misconfigured node. At this time, the differentiating trace point makes clear that these requests are not all performing the same work in the system. Therefore there is no expectation that they will all perform similarly, and the performance variation from earlier is explained.

The example above illustrates a key feature of how this work defines those requests that are expected to perform similarly: the trace structure. The trace structure in this case specifically refers to the presence and ordering of trace points and trace data within a request's trace. Trace data can refer to a particular parameter captured by a trace point, for example a variable value at the time that the request execution passed through that trace point. Since the idea that workflows should perform similarly is based on their execution of similar work in the system, and since this work is captured as a series of trace points in a request's trace, the structure of the trace is a reasonable basis for forming expectations about which requests should have similar performance.

By extension, altering the structure of the trace for a request by adding or removing trace points also refines the expectations of which workflows should perform similarly, since this structure represents the known work done in the system on behalf of the request. Adding a trace point that differentiates request behavior previously thought to be alike or similar thus inherently refines expectations for which workflows should have similar performance. When performance variation is assessed only for those workflows expected to perform similarly, the addition of a trace point that differentiates these workflows has the effect of decreasing performance variation. Conversely, the expectation is that the presence of high performance variation will be indicative of an under-instrumented area.

The process of forming expectation groups from traces and assessing their performance variation will be discussed in more detail in the Implementation section.

6.2.3 Localization, Causality, and the Call Graph

Design decisions about how to localize performance variation to more specific portions of request workflows relate to Phase 5: analyzing performance variation in the System Overview.

When a group of requests that are expected to perform similarly actually show high variation in their performance, it is not enough to indicate this as output with the tool. Since the traces are DAGs, each edge of the graph can be investigated for high performance variation using the same method that is applied to the entire trace. In fact, the tool checks these edges for each group of traces regardless of whether the entire trace has high performance variation. The edges are ranked from highest to lowest performance variation in the output of the tool. In the best case in which verbose information is provided from the trace points in the system, any high variation edges can be mapped back to regions in the system or lines of code based

on the information contained in the nodes comprising the starting and ending points of the edge.

In the design of the tool, both raw variance and coefficient of variation were used to measure the performance variation of requests and their edges. These values quantify the range of difference from the mean response time for the group, or, intuitively, how closely the response times for the group cluster around a central value. The equation for variance σ^2 is

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

where for a group of response times of size n, the distance from the mean of each value 1 to n is first squared (to avoid negative values) and then summed, with this sum finally divided by the number of values in the group. A higher variance value means the response times in the group are less predictable, in which case they are said to have higher performance variation. The standard deviation, which is perhaps more familiar to some readers, is the square root of the variance value. A group with a standard deviation of $2ms$ will have a variance of $4ms^2$.

The coefficient of variation is a unitless ratio that takes the standard deviation and divides it by the mean:

$$CV = \frac{\sigma}{\mu}$$

In this case, the standard deviation of each group's response time is divided by the mean response time for each group of traces from requests that are expected to perform similarly. Because the coefficient of variation relates a group's variation to its mean, it makes it possible to compare variation across groups with different means. This value is also useful because it addresses the assumption that requests with high

variation compared to their mean are more significant. It should be noted that in this work, the coefficient of variation was calculated based on the population standard deviation, rather than the sample standard deviation, as no sampling was used in the experiments.

One of the challenges with localization of performance variation is in considering which edges of a DAG to analyze and how to prioritize among them. The most significant advantage of including call graph-related edges, rather than those related strictly to causal relationships, is that this gives the tool the ability to prioritize high variation found in edges that are deeper inside a call graph, which conforms to both the intuition and the finding in previous work (Huang et al., 2017) that information obtained from deeper in the call graph is often more useful in diagnosing a root cause since it refers to more specific events than the parents encompassing them. Knowing the depth of an edge in the call graph hierarchy may be especially useful in cases of concurrency, in which edges in two separate branches may exhibit similar performance variation values, but knowing their relative depths can help prioritize one over the other in the ranking.

6.3. Design Choices: The Testing Environment

In addition to designing the tool itself, design choices arose regarding which system to use for testing; which tracing infrastructure to pair with this system to capture request workflows; and how to drive workloads, or sets of requests, through the system in order to obtain data.

OpenStack (OpenStack, nd) was a natural choice for the system on which to test the tool. OpenStack is a popular open source distributed application that serves as a cloud management platform and has a microservices architecture. This architecture makes it suitable for the use case of analyzing performance variation in

a distributed system with multiple interrelated components executing each request. The fact that it is open source makes it a great candidate for research, as its code can be examined or altered to suit the needs of the experiments. It is also popular enough and has enough developer momentum behind it to be a relatively stable and well-documented environment. OpenStack is already successfully being used to run an Open Cloud Exchange model (Desnoyers et al., 2015) production public cloud and to provide Infrastructure as as Service at the Mass Open Cloud (Mass Open Cloud, nd). [1] Specifically, a single-node devstack implementation of OpenStack was used as the experimental environment for ease of deployability and for reproducibility. Devstack is "a series of extensible scripts used to quickly bring up a complete OpenStack environment" (Devstack, nd).

The choice of workflow-centric tracing infrastructure to capture and collect request workflows from the OpenStack system was more complex. For the same reasons applicable to the testing system described above, the tracing tool chosen needed to be open source. More problematic was the fact that while the DAG model is the most expressive and versatile model for traces, especially in cases of programmatic processing as is the case in this work, the tracing tools currently available today tend to use the span model for historical reasons. There was no tracing tool available that would natively emit DAG model traces. With this limitation, the choice was between the tracing tool most readily available for the OpenStack system, OSProfiler (OSProfiler, nd) or another, more fully-featured open source tracing tool originally developed at Uber, named Jaeger (Jaeger, nd). OSProfiler was chosen because it is designed to work with OpenStack and cuts down on the engineering effort necessary for altering OpenStack to be able to work with Jaeger, which may have been a

[1] The Mass Open Cloud is the joint effort of several Boston-area universities, including Harvard University and Boston University, and the research in this thesis took place as part of its tracing team.

generally useful effort but was orthogonal to the goals of this thesis. As OSProfiler and the other available choices for tracing infrastructure were unable to meet the requirements described in Section 7.1.1: Requirements on Tracing Tools, the emitted traces needed to be converted to a DAG format by the tool before analysis. This will be described more fully in the Implementation.

Another consideration was how to drive the workloads, or sets of requests, through the test OpenStack system to obtain traces for analysis. As testing on a production system was not viable, the test system did not have native workloads running on it and these workloads had to be created and automated to run in addition to creating and running the tool itself. Several choices were available for how to create and run these workloads, including benchmarking tools such as Rally (Rally, nd). After some exploration with Rally, it became clear that while a benchmarking tool will drive workloads through the system as a side effect, its primary purpose is assessing how quickly large volumes of basic operations can run in the system and it had severe drawbacks in terms of customizability of the workloads. For the purposes of this work, its constraints on workload types and request batching and its lack of compatibility with OSProfiler led to its rejection in favor of the more direct approach of writing bash scripts (Rendek, 2017) to drive the workloads. These bash scripts could be fully customized for any experiments and posed no constraints on which system operations could be batched together or how the tracing infrastructure's functionality was invoked at the time of running the workloads.

Chapter 7: Implementation

This section first covers some interface constraints between the instrumentation and the tool, which inform several of the realities of implementation described afterward. Next, detail is provided on some highlighted algorithmic challenges in creating the tool and running the experiments. Each of these challenges also has its place in the general workflow. The workflow is described in Section 7.3 to give an overview of how each component from the System Overview works together, and will detail any functionality that is not highlighted in the Algorithmic Challenges.

Python was chosen as the language in which to implement the performance variation-based tool. This would make it easily compatible with OpenStack and OSProfiler, which are written in Python. Python is also a common language for engineers and students collaborating on the Mass Open Cloud, so this choice of language made for more maintainable and understandable code for others who may become interested in the project. The current implementation is roughly 2,000 lines of Python code, about one quarter of which relates to converting traces to DAGs, half to grouping and analysis, and one quarter for other miscellaneous functionality, e.g. parsing input.

7.1. Adapters to Work With Tracing Infrastructure

The most significant requirement imposed upon the system under study by the performance variation-based tool is that it is outfitted with some tracing infras-

tructure. As described in Section 5.2, the tool must be able to consume traces of request workflows for its analysis. This means that there must be some minimal instrumentation already present in the system under study, that it must be capable of propagating metadata, and that the system must be outfitted with a tracing infrastructure that allows data to be reconstructed about each workflow. Many of today's systems meet this requirement (Sigelman et al., 2010; Kaldor et al., 2017).

Additionally, there are several other requirements concerning how the system's instrumentation must behave in order to produce traces that can be used for a rigorous analysis. These characteristics are generally useful in tracing systems and tracing instrumentation, and would help engineers more easily understand the system behavior that the traces represent. They are described below in Subsection 7.1.1 both to give a fuller picture of the dependencies of the performance variation-based tool as well as to serve as an indirect contribution to the study of the limitations and opportunities of current workflow-centric tracing tools and instrumentation practices.

7.1.1 Requirements on Tracing Tools

The following are required from the tracing infrastructure and monitored system in order to do the analysis proposed by this work.

- Well-defined beginning and end markers for traces. It is difficult to gain information from a trace if it is unclear where the request it represents begins or ends. Specifically, one problem that can arise from the lack of an end marker is that it is not possible know when the data associated with a request has finished propagating to the tracing back end, which results in the danger of querying for a trace too early and getting a malformed trace.

- Clear instrumentation of concurrency and synchronization in systems that ex-

hibit these behaviors. As described in Section 6.2.1: The DAG As a More Expressive Tracing Model, a clear record of concurrency and synchronization points in requests allows for a straightforward understanding of happened-before-relationships in the request as well as of the request's critical path. Everything one could want to know about the cause of a request's poor performance lies on its critical path.

- Systematized, human-meaningful trace point names that map to specific components, nodes, and/or lines of code. Ideally, boundaries of components, VMs, etc. are clearly marked. Analysis of request and system behavior can only be as clear and meaningful as the data that is recorded about it.

- Trace output that conforms to the DAG model. For the reasons outlined above in Section 6.2.1, the DAG format is especially useful for understanding causality and the critical path of requests. Having support for an option in the tracing system to emit the trace in DAG format would greatly aid any effort relying on this type of data.

Where these requirements were not met by the tracing tools available, the implementation of the performance variation-based tool had to bridge the gaps, as will be described in more detail in Section 7.2: Algorithmic Challenges below.

7.2. Highlighted Algorithmic and Parsing Challenges

Several of the most complex challenges in this work arose not from analyzing the performance variation of requests, but rather in the pre-processing phases preparing the traces of the requests for such analysis. Moreover, bridging the gaps between what the performance variation tool required of the tracing infrastructure

and instrumentation versus what the available tracing tools and instrumentation provided accounts for several of these challenges. Specifically, the need to convert span model traces into a DAG format and the need to build a check in the workloads to attempt to ensure that trace data had finished propagating to the tracing back end are both challenges that arose from the limitations of the tracing infrastructure and instrumentation, the burden of which necessarily fell onto the performance variation tool in this context.

7.2.1 Conversion of Traces to DAG Model

For reasons described in Subsection 6.2.1: The DAG As a More Expressive Tracing Model, the DAG model is a more expressive and versatile model for traces, and is able to represent concurrency and synchronization in the system. This model, unlike the span model that is emitted by most tracing tools, is suitable as input to the performance variation-based tool as it holds all the information this tool will need for its analysis. Being able to convert the native span-based traces from tracing tools like OSProfiler into DAGs was therefore indispensable, and a module to handle this conversion as a pre-processing step was created. This module is a tool in its own right that can be used to convert a span-model trace in a .JSON file into a DAG-model trace. Depending on the command line arguments with which the module is invoked, the output is emitted to stdout or as a .DOT file that resides in the same directory as the input file. Two of the main challenges with this conversion are highlighted below.

Inferring causality. The utility of the DAG model comes largely from its ability to indicate happened-before causal relationships among events. Because concurrency and synchronization points were not explicitly recorded by the tracing infrastructure, the algorithm to do this conversion had to make a best guess about concurrent events and synchronization points based on parsing the timestamps in-

cluded in the native span model traces. It is an inherent limitation that these guesses cannot be guaranteed to be accurate, since the presence or absence of concurrent activities and synchronization points was not preserved and cannot be verified. Causal relationships are partially preserved in the span model traces; the events for which causality is not explicitly preserved are the sibling spans within the same nested parent span (see figure 6.2 for reference). For the purposes of representing where concurrency and synchronization are likely to have occurred along a request path among sibling spans, the algorithm used the following definitions:

- A span whose starting timestamp is later than a sibling's ending timestamp has a sequential or **happened-before causal relationship** with its sibling and will remain on the same branch of the DAG as its sibling.
- When the happened-before-relationship occurs in the context of multiple existing branches in the DAG, where a span s has a starting timestamp that is later than the ending timestamps of multiple siblings on multiple branches, a **synchronization point** is establised at s.
- A span whose starting timestamp is earlier than a sibling's ending timestamp has a **concurrent relationship** with its sibling and will form a separate branch in the DAG.

The sibling span with the earliest starting time stamp had its start event attached to the parent span's start event in the DAG; the others followed the above formula to determine how they would be attached. After all siblings' start and end events, and those of any of their children, were attached in the DAG, the original parent's end event was appended.

Preserving call graph relationships. Another consideration was whether and how to preserve non-causal information present in the span model, such as call

graph relationships specifying which units of work, e.g. functions, called others to do work on their behalf. As DAGs and their related .DOT files have only two elements, nodes and directed edges, the preservation of any additional information not related to specifying events, their ordering, and their latencies had the potential to be challenging and necessitated the careful exercise of judgment. Preserving call graph relationships was determined to be worthwhile in order to have a full picture of which edges' latencies have the potential to impact others, which is determined not only by causal relationships but caller-callee relationships as well. For example, a parent span's high latency and high latency variation are directly related to any non-asynchronous child span's high latency since the parent's duration encompasses the child's.

The most straightforward way to preserve the caller-callee relationship in the DAG was pushing this information into the node or event names. The DAG model requires that any event with a duration, such as the spans that are first class values in the span model, is broken down into its component events. For example, a single span from a span model trace becomes a start event and an end event (each represented by its own node) in the DAG model. This inherently breaks up the "span" as an object or first class value in the DAG. However, a naming scheme that gives corresponding names to start and end events derived from the same span preserves these associations. In combination with the conversion logic (omitted for brevity) that dictates that the events in a DAG located along the path between a corresponding $< start, end >$ pair are events derived from the children of the span from which the $< start, end >$ pair was derived, it becomes possible to precisely track not only which events in a DAG formed a span in the span model, but also which events are related to that span's parent or child spans. This is the caller-callee relationship.

7.2.2 Checks for Trace Completeness

Another challenge resulted from the lack of end markers for traces in OpenStack with OSProfiler (see Subsection 7.1.1). This end marker, had there been one, would serve as a definitive marker that the request has completed, that all synchronous parts of its workflow have finished, and that there is no more work being completed on behalf of the request. Without this information, there is no way to have any signal that the tracing back end or storage is ready to be queried for the trace of a particular request. This is a general problem that affects users of the system regardless of whether or not any external tool, such as the performance variation-based tool, is used in conjunction. Earlier in the course of this research, an ongoing concern was the inability to understand why some traces were malformed or incomplete, until this lack of end marker was discovered. It would be reasonable if tracing infrastructures and the trace points in monitored systems adopted an instrumentation discipline to ensure that end markers exist for traces so that this type of malformed trace is not returned when a query is made to the back end.

In lieu of such guarantees, the workloads created in this work to generate trace data in OpenStack have a built in check that essentially attempts to wait for trace data related to a request to finish propagating to a back end. It should be noted that this check also does not guarantee completeness; however, it did seemingly help avoid the malformed traces encountered earlier. In the bash script that retrieves traces related to the automated workloads created, a line count is obtained for the retrieved trace at least twice. If the newer line count is larger than the older line count, this is taken as evidence that the trace data is still propagating and that the trace is still growing and not yet complete. In this case, the script sleeps for 5 seconds and tries again. Once the new line count matches the old one, it is assumed that no more data

is propagating and the trace is dumped to a directory for use by the performance variation tool.

7.2.3 Extracting the Critical Path

When a request's workflow shows concurrent activity, extracting the critical path of the request becomes important for analysis of its latency for reasons described in Subsection 6.2.1. When there is no concurrency, the request's critical path is the entire sequential workflow of the request and extracting it is not necessary.

To handle cases where concurrency may exist in the workflow, and by so doing make the tool more widely applicable to a variety of systems, first a check must be done for whether concurrency is present in the workflow. If it is not, the rest of the critical path extraction can be skipped. If it is, then one of two methods can be employed: in the ideal case, where an end marker is present in the trace (see Subsection 7.2.2 above), finding this end marker and processing backward until the starting point of the trace is reached is a reliable way to discard asynchronous paths, since those paths never join back up with the end marker (see figure 6.1 for reference). Where multiple paths exist between the ending and starting markers, the longest is considered the critical path (Cormen et al., 2009, p.594). Since a synchronization point has a happened-before-relationship with all the branches it synchronizes, the longest of these will be responsible for the request's overall latency.

Since the traces used in this work did not have end markers, and this was not something the performance variation-based tool could supply, the second and less ideal method of choosing the longest path without knowing which branch would have contained an end marker for request completion had to be employed. Like causality inference, this is not guaranteed to be correct but is instead a "best guess" method considered adequate for the use cases here.

7.2.4 Grouping by Expectation Based on Trace Structure

Analysis would not be possible if traces of requests could not be batched or grouped according to which ones are expected to perform similarly. This grouping process is based on the trace's structure as outlined in Subsection 6.2.2: Effects of Instrumentation Granularity in Traces, since this structure represents the work done on behalf of the request with a certain level of granularity. Accordingly, one algorithmic challenge was the pre-processing step of analyzing each trace's structure to see if it could be grouped with any others from the set of input traces. (If not, then no analysis was performed on the trace. Groups were required to have 3 or more member traces for analysis to be mathematically meaningful.)

To characterize the structure of each trace, a depth-first traversal was used to ascertain the nodes of the trace in a predictable order. This ordering was stored in memory as a string of concatenated node names and other markers indicating relationships, e.g. parent-to-child. This essentially acted as a hash on each complex structure, mapping it to a string that could be compared with others. When the strings matched, the structure of the traces was deemed to be similar and their performance was expected to be similar. In reality, the string representations of the graph structures became cumbersome, so they in turn were mapped to shorter hash values uniquely representing each string representation.

7.2.5 Parsing Input

One challenge that was not algorithmic but nevertheless significant was the parsing of input files. Both .JSON files (the typical file type of the span model trace) and .DOT files (the file type of DAG traces) needed to be parsed, as the traces serving as input to the tool could come in either format. Furthermore, the tool needed to

detect which format was passed to it in order to handle the trace appropriately by doing a conversion to a DAG in the former case.

A pre-existing Python module that loads and handles JSON objects as iterable Python data structures was indispensable here, as were many helper functions that needed to be created based on Python regular expressions to break each text file into a traversable graph of nodes and edges. Roughly 100 of the tool's lines of code are dedicated to parsing the input text from these trace files.

7.3. Workflow Overview

Below is a brief overview of how the pieces of the tool's workflow, some aspects of which are highlighted above in Algorithmic Challenges, fit together.

7.3.1 Input Parsing

The tool is given a a directory as input, from which it ingests a set of files, each of which contains the workflow information for one trace:

```
python main.py <directory storing a set of traces>
```

The files may be in .JSON or .DOT format, which indicate that the traces are in span or DAG formats, respectively. The tool detects the file extension and converts the trace to a DAG (see Subsection 7.3.2 below) in the former case. Otherwise, the DAG from the .DOT file is read in and kept in memory.

7.3.2 DAG Conversion

This step is optional and is only invoked if the current input file is a .JSON file holding a span model trace. While the conversion module can be used alone if called from the command line with a file as an argument, it can also be called

from within the tool's main program. As described in Algorithmic Challenges above, the conversion tool makes necessary guesses about concurrency and synchronization points based on parsing the timestamps of the input trace and emits a DAG. When called from the main program, this DAG is also kept in memory.

7.3.3 Critical Path Extraction

In case the DAG format trace is not strictly sequential (that is, it exhibits concurrency), the critical path is extracted by taking the longest branch as described in Algorithmic Challenges.

7.3.4 Expectation Grouping

For each trace read into memory, a depth first traversal of its graph structure produces a string that is unique to the structure. These strings are mapped to shorter hash values (pseudo-random six-character strings) for easier handling and are compared to hash values associated with the other traces from the input set. A globally accessible hash table using these hash values as keys holds the grouping information. Each key is associated with a list of specific traces that belong to the group, those traces' metadata, and metadata about the group as a whole. The latter category includes analysis information, such as the group's variance and average response time. When a hash value does not match any key in this hash table, it is added to the hash table as a new key and thus forms its own group.

7.3.5 Performance Variation: Detection and Localization

For each trace group that has three or more members, the latency of each member of the group is stored for future computation as part of the associated trace metadata in the hash table described in Subsection 7.3.4 above. Each time a new

member is added to a group, computation is done to find the new average response time and response time variance for the group. These are also computed for each edge in the traces belonging to the group. If any of these values exceed a threshold that can be set at the invocation of the tool, a flag is set and the groups and edges exhibiting high performance variation are marked for separate handling in the output.

While the edges between events in the DAG are straightforward enough to include in the analysis, making use of the call-graph information stored in the DAG proved more complex. While this information is completely retrievable from the DAG, it relies on superimposing extra edges onto the DAG and deciding whether to give these edges the same priority as the causal edges. (See figure 6.2 for reference, where the DAG model on the right side shows causal edges that are represented with solid arrows and call-graph edges that are represented with dashed arrows.)

Two different algorithms were implemented to analyze only the causal edges or also the superimposed edges related to the call graph, referred to as "flat" and "hierarchical" localization respectively. While the edges considered in flat localization are a strict subset of those considered in hierarchical localization, the aim was to observe whether the ranking and prioritization among the edges were more straightforward in one model versus the other. For example, it is worth seeing whether the hierarchical edges contain relevant data that is not shown in the flat edges, or whether they merely introduce noise to the output. The hierarchical edges were useful for finding the root cause of an experimental problem described in Section 8.2.

7.3.6 Output

The output prints to STDOUT in the terminal from which the tool is invoked. No GUI was deemed necessary, as this tool is meant for engineers who are comfortable with the command line. The output prints the traces processed and performance

variation thesholds used, as well as information about the expectations for which traces should perform similarly. It then prints a ranked list of any edges to which performance variation was localized as well as the group and trace associated with that edge and whether the group itself was marked as having high performance variation. Finally, it emits a ranked list of all edges from all groups from highest to lowest variation. See figure 7.1 for a screen shot of the beginning portion of this output.

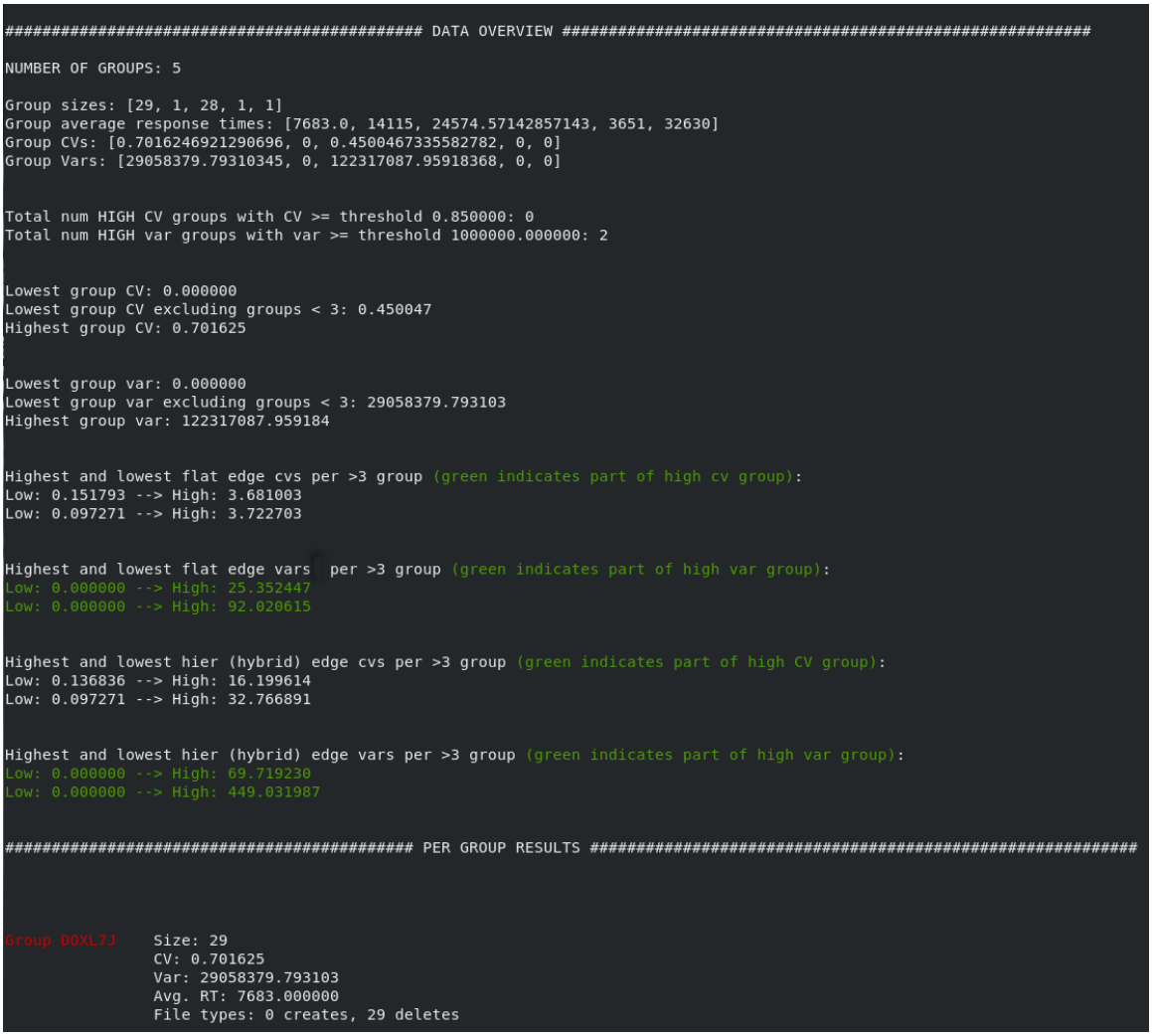

Figure 7.1: Example output from performance variation tool.

Chapter 8: Experimental Set-Up and Results

The experiments outlined below served multiple purposes. First, they collectively serve to test how well the tool works in terms of its ability to do preprocessing steps such as grouping traces accurately according to expectations of performance variation. Second, they serve to answer specific questions and validate hypotheses particular to each experiment. The first experiment analyzes performance variation in OpenStack to discover whether there is an opportunity to guide instrumentation choices with the tool, or whether instrumentation in such a system is already optimal. The second experiment aims to see whether there is an opportunity for the tool to assist with debugging a problem in the system and localizing its source to the root cause. The third experiment uses a more complex problem to further test the hypotheses of the second experiment.

As a preliminary assessment, the time the tool takes to run on a data set of 60 traces was tested using the wall-clock time of the Unix TIME command. On average across 25 runs of 60 traces each, the tool took 4.12 seconds with a high of 4.248 seconds and a low of 4.029 seconds. All of the experiments below were performed on a single-node deployment of an OpenStack system. Below, a more detailed description of the purpose, specific parameters, and the results are included for each experiment.

8.1. Experiment 1: Baseline Performance Variation

All systems innately have some performance variation. Instrumenting non-problematic regions that exhibit high performance variation that is caused by acceptable, heterogeneous system behavior is important in a way that is perhaps more subtle than instrumenting such regions to find the cause of an active problem; instrumenting a non-problematic region can be beneficial for understanding the general system state and consequently for having a foundation of understanding when a problem does arise in the future. When a system is instrumented in such a way that there is relatively low performance variation observed during times when the system is running normally, the contrasting performance variation observed when a problem does arise can make diagnosis more targeted and meaningful. The performance variation associated with system problems will be more obvious and less likely to be lost in the "noise" of the general performance variation of a less carefully instrumented system. Therefore, observing unevenly distributed performance variation across a normally running system would indicate that default instrumentation in the system is itself not distributed well. Adding instrumentation in these high-variation areas would cut down on the noise in the system and make future diagnosis less cumbersome.

This experiment seeks to understand whether the instrumentation placement in our example system, OpenStack, is already sufficient, or whether more instrumentation would be useful. High performance variation in this case would be indicative not of any system problem but instead of instrumentation that is not detailed enough to distinguish between dissimilar requests while the system is running normally. The hypothesis for this experiment is that the instrumentation in OpenStack will show unevenly distributed performance variation by default. If this is the case, it indicates that adding instrumentation in the regions with high baseline performance variation

could be useful for diagnosing future problems. Adding in this instrumentation is left to future work.

8.1.1 Baseline Variation: Set-up

Two VMs were created in the Mass Open Cloud environment, running devstack modified to include OSProfiler as the tracing tool. The two VMs were used in the same ways to run the same workloads, mirroring each other to make it less likey that there was an unexpected condition on one VM that would skew the results. On each VM, the following custom bash workload (see Section 6.3 on design choices for workloads) executed 30 of this set of operations:

- With OSProfiler tracing enabled, Create a server of FLAVOR M1.TINY from IMAGE CIRROS-0.3.5-X86_64-DISK and wait for its state to become ACTIVE (indicating the build has finished).
- With OSProfiler tracing enabled, delete the server that was just created.

It then executed these operations:

- Sleep for 30 seconds to allow for tracing data to propagate to the back end.
- Query the back end for the request IDs for each of the requests executed above and dump the traces in a specified directory.

The steps outlined above were considered to be one "run" of the workload and produced one data set. The data set emitted by each run of the workload included 30 traces of SERVER CREATE requests and 30 traces of corresponding SERVER DELETE requests. This workload was run three times on each of the two VMs, for a total of six data sets.

For the analysis in this experiment, the data sets were collectively run through the performance variation-based tool, which converted each traceto a DAG, grouped traces expected to perform similarly together based on their structure, and analyzed each group's traces and their edges for performance variation (see: System Overview and Implementation). This is a streamlined version of the implementation of the entire tool, avoiding the critical path extraction that is more useful in systems with high rates of concurrent activity like Hadoop and Ceph.

8.1.2 Baseline Variation: Results

The first result of note is that the grouping based on structure corresponded with the originating command types across all data sets: there were no groups in which SERVER CREATE traces were mixed with SERVER DELETE traces, and traces from each command type generally clustered in one large group with several outliers. All but a handful of traces associated with a specific command type were grouped together. Outliers could be explained by deviations in the workflow of the request.

This indicates that grouping based on structure is a valid way to form expectations of performance, since the groups reflected the work done in the system on behalf each request. It also indicates that there is enough instrumentation in the OpenStack system by default to differentiate SERVER CREATE from SERVER DELETE traces, even without any explicit data captured by trace points to preserve the issuing command of the request (the issuing command was preserved outside of the trace for this experiment for verification purposes).

For this experiment, the ranking of trace edges from highest to lowest variance across all groups of traces was examined. There were 2,214 edges in all. Table 8.1 shows the percentage of total edge variance accounted for by the top 1, 5, and 10 edges in the ranking, and figure 8.1 shows this data for all edges.

Percent of edge variance accounted for highest variance edges
The top 1 edge (0.02% of edges) accounted for 52% of the variance.
The top 5 edges (0.2% of edges) accounted for 84% of the variance.
The top 10 edges (0.4% of edges) accounted for 91% of the variance.

Table 8.1: Variance distribution in trace edges.

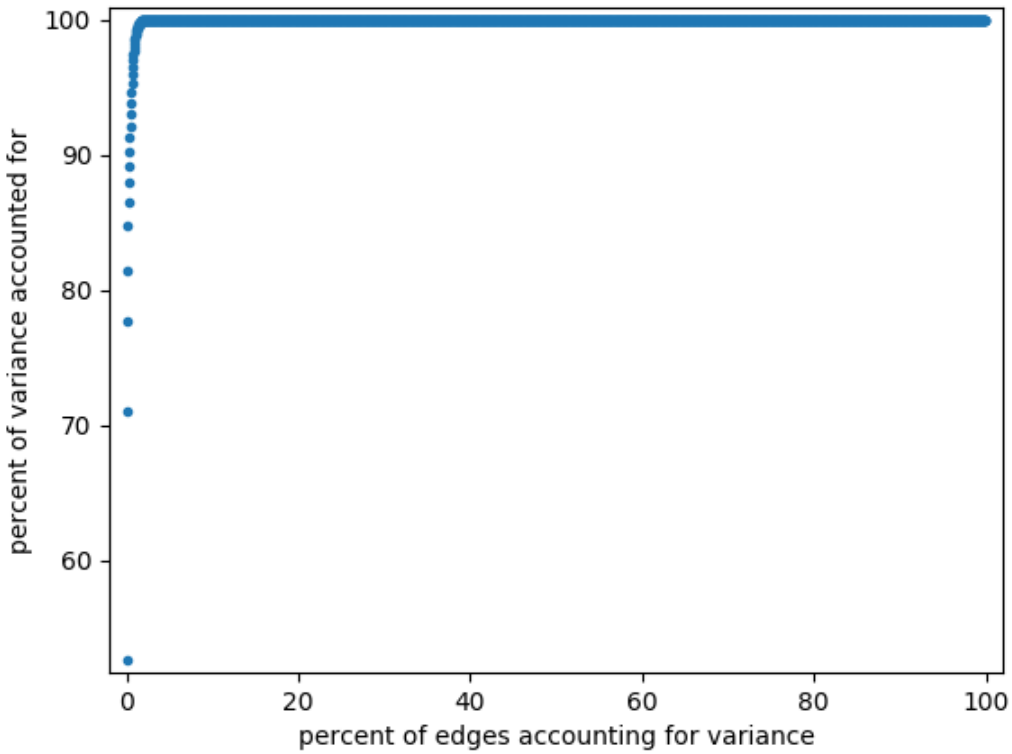

Figure 8.1: Cumulative distribution function showing variance for trace edges in OpenStack.

A small number of edges account for a disproportionate amount of the edge variance, making the amount of performance variation inconsistent throughout the system as a whole. The inconsistency of performance variation across these request edges indicates that there are under-instrumented regions in OpenStack, that they can be detected by the tool, and that the tool can contribute to a more deliberate default instrumentation placement in the system that is more informative since it is more likely to differentiate dissimilar system activity. These results suggest that the hypothesis was correct and that instrumenting these edges could help distribute performance variation more evenly in the system and make diagnosis easier in a future performance degradation scenario.

8.2. Experiment 2: Adding sleep() Function

In this experiment, a resource contention issue was simulated through a SLEEP() function that was placed along the request path for SERVER CREATE requests and set to execute arbitrarily 33% of the time (based on a pseudo-random value between 0 and 100 generated for this purpose). The SLEEP() function simulated the request needing to wait on a shared resource at arbitrary times. The Eventlet Networking Library's (Eventlet Networking Library, nd) EVENTLET.SLEEP() function was chosen to respect the threading models in OpenStack. SERVER CREATE requests were chosen over SERVER DELETE requests because they had less innate performance variation comparatively.

Broadly, this experiment tests the underlying insight the tools is based on, namely that a performance degradation problem in the system will exhibit some associated performance variation. More specifically, it tests the utility of the tool. The performance variation-based tool is useful insofar as it can 1) identify performance variation in the system and 2) localize it to a narrower system region than the entire

path of the set of requests experiencing the problem. This experiment tested the tool's achievement of these two goals by using it to analyze traces that have a simple, specific performance problem injected some portion of the time. Because it is known which request path has this problem and what the problem is, it is possible to tell whether performance variation showed up in this region in the analysis. The hypotheses for this experiment were the following:

- Higher performance variation will be found along the path of requests that are experiencing the problem compared to other requests that are not;
- The performance variation can be localized to a relevant region along the request path by using the tool;
- Performance variation decreases for affected requests when relevant instrumentation is added to the system;
- Variables are useful for capturing relevant trace information and differentiating among dissimilar requests.

8.2.1 Adding sleep() Function: Set-up

For this experiment, the same workload from Experiment 1 was used, in which 30 servers were created and immediately deleted (see Subsection 8.1.1 for specifics). The differentiating factor was a 20-second EVENTLET.SLEEP() function added to the OpenStack system along the critical path of the OPENSTACK SERVER CREATE request. This endeavor involved inspecting the OpenStack code for classes and functions likely to be involved in the creation of a server and testing to make sure that a line of code added to the path executed when running this command.

The EVENTLET.SLEEP() function suspends the current green thread, or user-level thread, for a specified number of seconds before resuming operation. While

the thread is suspended, other user-level threads have a chance to process. This was inserted in the NOVA COMPUTE MANAGER's code in the BUILD_SUCCEEDED function, which must execute for each successful SERVER CREATE request. Additional code around this function ensured that it would execute 33% of the time by generating a pseudo-random value between 0 and 100 and triggering the function execution if the number was less than 33. Python's RANDOM.RANDINT(), which returns an integer greater than or equal to the first specified value and less than or equal to the second, was used for this purpose.

3 phases were run of this experiment, adding new trace points to OpenStack for the latter two variations to determine whether adding instrumentation around the manufactured problem would affect performance variation in the expected ways outlined above. These 3 variations aim for escalating specifitity of the data gathered from the trace points:

(a) **Phase 1:** Add the SLEEP() function, but do not add any trace points in OpenStack.

(b) **Phase 2:** Add custom trace points around the SLEEP() function along the path of the request.

(c) **Phase 3:** Modify the custom trace points around the SLEEP() function to include a variable recording whether SLEEP() executes.

In Phase 2, the trace points inserted around the EVENTLET.SLEEP() function did not track anything about the sleep function itself:

```
def _build_succeeded(self, node):
        rt = self._get_resource_tracker()
        rt.build_succeeded(node)
```

```
profiler.start("sleep_tracepoint", {})
        if random.randint(0,100) < 33:
            eventlet.sleep(20)
            slept = "True"

profiler.stop({})
```

In Phase 3, the same trace points included a variable to record one value if EVENTLET.SLEEP() executed and another value if it did not:

```
def _build_succeeded(self, node):
        rt = self._get_resource_tracker()
        rt.build_succeeded(node)
        slept = "False"

profiler.start("sleep_tracepoint", {"slept": slept})
        if random.randint(0,100) < 33:
            eventlet.sleep(20)
            slept = "True"

profiler.stop({"slept": slept})
```

These variable values were preserved as strings, not booleans, for compatibility with the textual nature of the .DOT file that would contain this information.

8.2.2 Adding sleep() Function: Results

The data indicates that the hypotheses were correct, in that performance variation did increase after adding the SLEEP() function to CREATE requests, and performance variation did decrease again after adding a differentiating trace point that captured the occurrence of this problem.

In Phase 1, adding the SLEEP() function to the CREATE request path resulted in both higher response times for requests, compared to their baseline response times without the SLEEP() function, and also higher variance for CREATE requests compared to their baseline. Specifically, the response times were 63.32% higher, and the variance was 271.44% higher for these requests compared with baseline. The change in variance for these requests suggests that performance variation is correlated with a performance degradation problem and can be used as a marker to detect this type of problem. It should be noted that response time and variance also rose for DELETE requests compared with the baseline, but these values were not as high compared to their baseline as those for the CREATE requests, as can be seen in the bar and line graphs in figures 8.2, 8.3 and 8.4 below.

See Tables 8.2 and 8.3 for the exact values of the average variance and average response times for these requests across the different experimental phases.

Average Variance and RT for Create Requests				
	No problem	Problem	Problem and TP	Problem and TP w/ Var
Avg. var in ms^2	31,431,341.49	116,748,095.03	159,023,759.42	52,840,433.48
Avg. RT in ms	10,700.69	17,476.91	22,266.27	24,127.43074

Table 8.2: Create requests: Average variance and response time.

In Phase 2, when adding trace points near the SLEEP() function that did not track whether SLEEP() occurred, the values for CREATE requests' variance remained similar to Phase 1, as expected. The performance variation seen in Phase

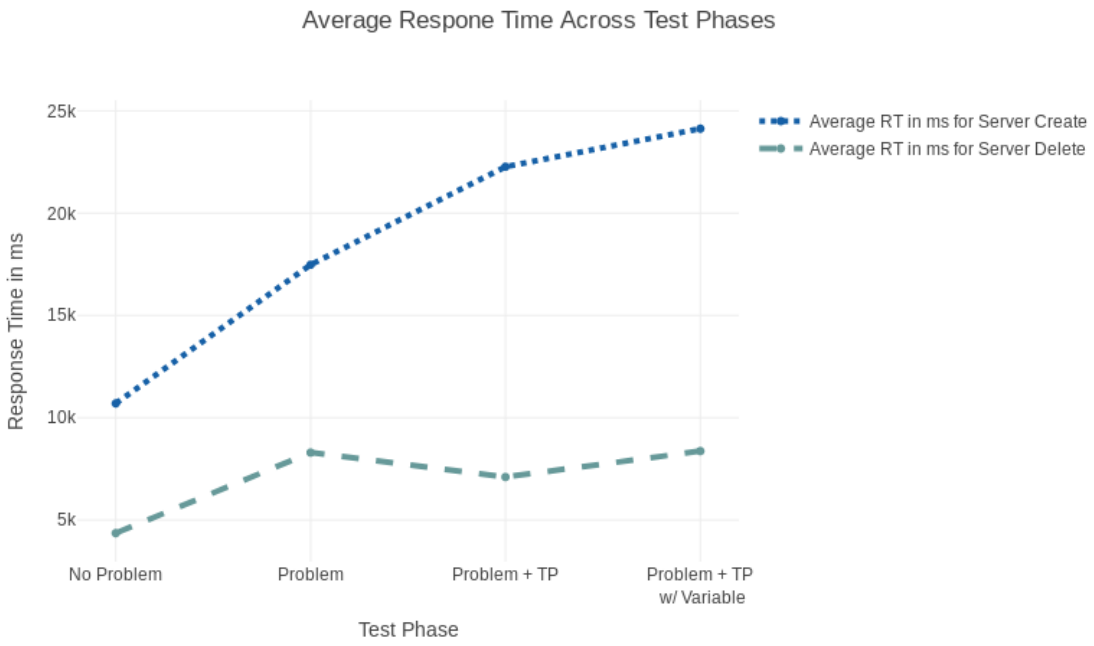

Figure 8.2: Average response times for create and delete requests, line graph.

Average Variance and RT for Delete Requests				
	No problem	Problem	Problem and TP	Problem and TP w/ Var
Avg. Var in ms^2	16,371,940.89	36,234,882.92	33,768,383.00	33,059,106.44
Avg. RT in ms	43,60.08	8,299.37	7,103.35	8,371.61

Table 8.3: Delete requests: Average variance and response time.

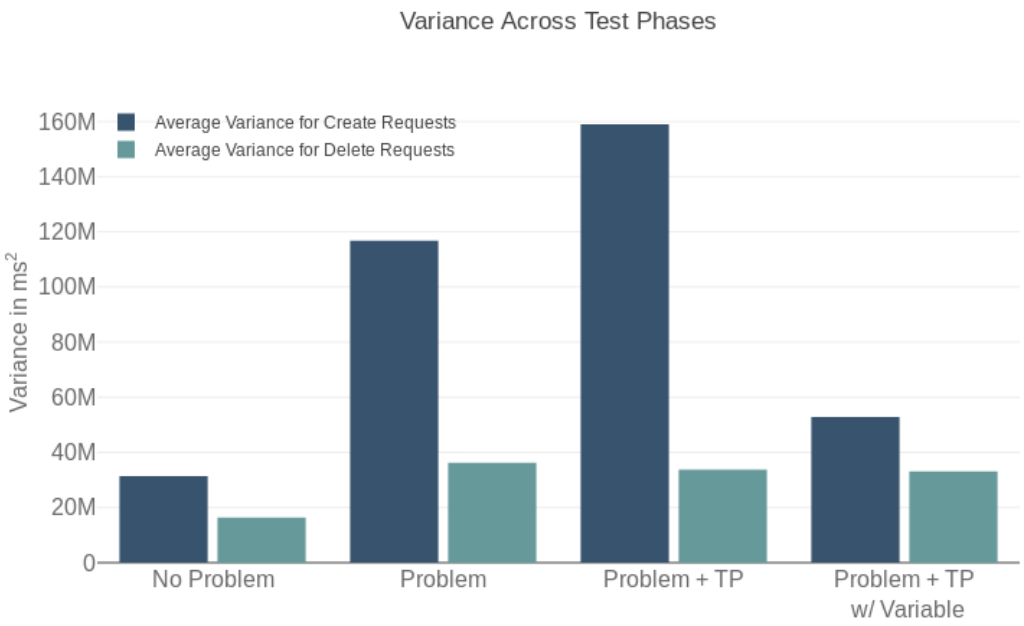

Figure 8.3: Average variance for create and delete requests, bar graph.

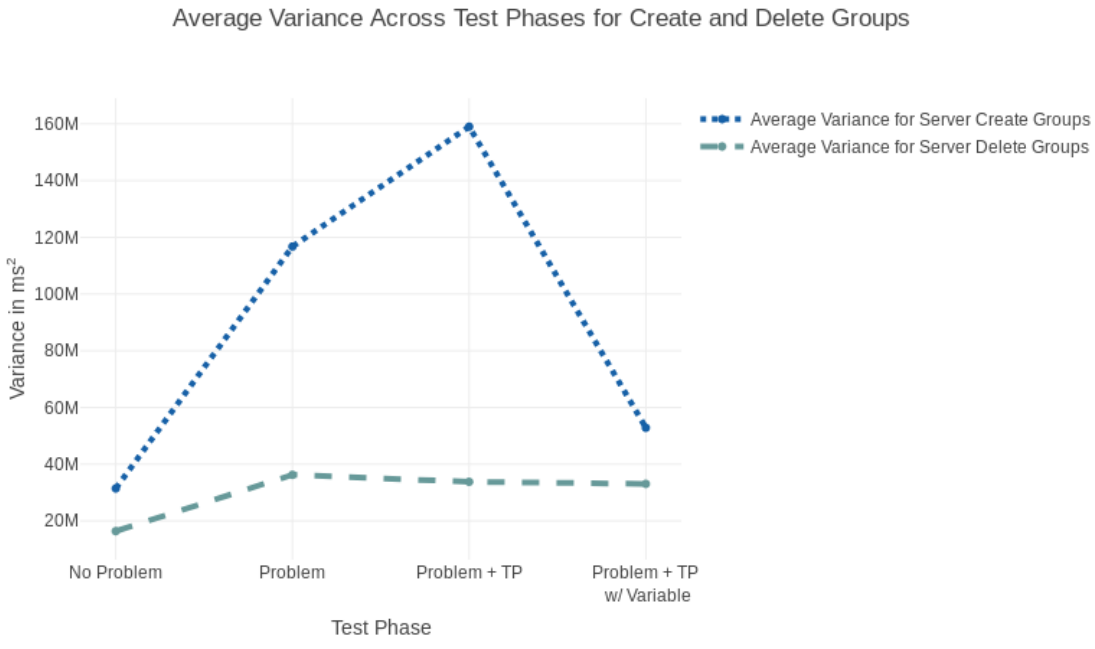

Figure 8.4: Average variance for create and delete requests, line graph.

2 for the group of CREATE requests was also localized to the trace points inserted to track the SLEEP() function. Figure 8.5 shows that the highest variance edge, from the localization method using the call graph, is related to the (helpfully named) SLEEP_TRACEPOINT around the SLEEP() event. This results speaks to the utility of using information from the call graph to rank edges, specifically by giving a higher weight to edges deeper in the call graph. Only the localization method using this weight resulted the edge associated with the SLEEP_TRACEPOINTs receiving the highest ranking.

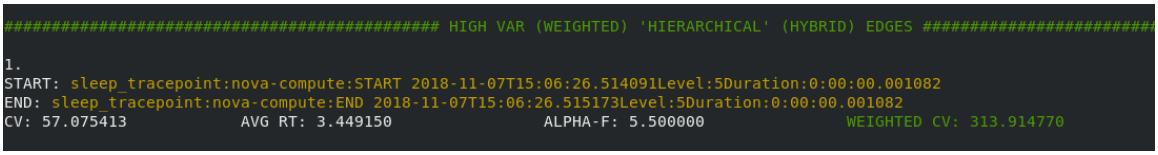

Figure 8.5: Highest variance edge from Phase 3 shows the region with SLEEP() added.

In Phase 3, while the average response time remained high for CREATE requests, as expected since the problem remained in place along the request's path roughly one-third of the time, the variance for these requests dropped again and resembles the variance of the baseline group (see figures 8.2, 8.3, and 8.4). This speaks to the importance of adding instrumentation in high performance variation regions for differentiating dissimilar workflows and also the importance of using descriptive variables that capture relevant information in trace points. Furthermore, the drop in performance variation indicates that requests that are not performing similar work in the system were identified, and that the recent change in request grouping is based on a factor involved in the root cause of the performance degradation. Indeed, the grouping of the CREATE requests changed dramatically in this phase compared with the others. The average number of groups for CREATE requests rose from 1 in the previous phase to to 2 in this phase, accounting for differentiation between requests in which SLEEP() executed versus those in which it did not, and the average group

size dropped accordingly from 28.35 traces to 14.1 traces, resulting from the smaller and more specific groupings based on the differentiating trace point's variable.

Thus all of the hypotheses were met for this experiment: higher performance variation was found along the path of CREATE requests compared to DELETE requests; this variation was able to be localized to a system region associated with the "problem", as shown by the high variance edge formed by the SLEEP_TRACEPOINTS above; the variation decreased when relevant instrumentation was added to the system; and this instrumentation included the use of a variable to capture information about a specific system event, which is more informative than trace points capturing only the request path. This problem and its solutions are analogous to, for example, a resource contention problem in which a variable captures the length of a particular shared queue in the request path, which is the source of the contention.

8.3. Experiment 3: Increasing Resource Contention

Similar to Experiment 2, this experiment aimed to mimic a resource contention problem. In this case, the problem centered around a misconfiguration that would introduce long queue lengths in the NOVA COMPUTE service, creating a bottleneck for SERVER CREATE requests. [1] Accordingly, the root cause of the system problem was that the number of concurrent servers that can be created within OpenStack's NOVA service was limited to ten in the configuration options, while additional SERVER CREATE requests were forced to wait on a semaphore.

The purpose of this experiment was to see if the tool could detect a correlation between the bottleneck and the SERVER CREATE requests' performance, among multiple request types. High performance variation is expected for these requests, since the SERVER CREATE requests received during periods of low concurrency would

[1] This experiment was also included in a recent paper submission (Ates et al., 2019).

execute immediately, whereas others would have to wait varying amounts of time for the semaphore. The first hypothesis was that the tool would detect higher performance variation for SERVER CREATE requests than for other types of requests, and the second hypothesis was that this variation could be localized to a particular edge that would give insight into the cause of this problem, corresponding to a system region having to do with the bottleneck.

8.3.1 Increasing Resource Contention: Set-up

The resource contention problem was created by issuing multiple concurrent requests to the system. It used three workloads consisting of combinations of SERVER CREATE SERVER DELETE, SERVER LIST, FLOATING IP LIST, and VOLUME LIST requests, which are various commands in OpenStack. A variable was introduced in OpenStack's trace points to record queue lengths around a semaphore in the NOVA COMPUTE service, as this would serve as the point of resource contention, and NOVA's configuration of MAX CONCURRENT BUILDS was set to 10 to provide the bottleneck around the semaphore.

20 instances of each workload were started simultaneously and multiplexed among 8 vCPUs on the same devstack system. Performance variation was assessed for the traces resulting from these workloads to see if a correspondence between the concurrency and variation could be found. In this case, the traces were grouped based on request type as well as their structure.

8.3.2 Increasing Resource Contention: Results

The grouping obtained is shown in Figure 8.6. As the group with the highest variance is the SERVER CREATE group, the first hypothesis was validated: the tool was able to discover the group with more resource contention by looking at the

performance variation.

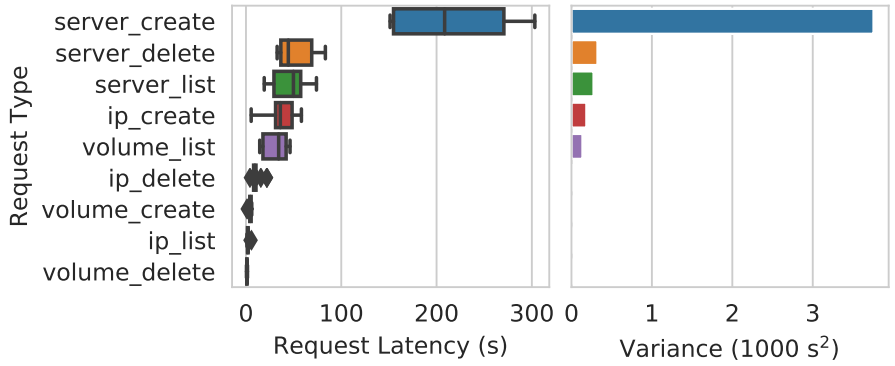

Figure 8.6: High variance in SERVER CREATE traces.

Figure 8.7 shows the variance of edge latencies for the SERVER CREATE group. It should be noted that edges related to the call graph were not used for the localization in this case. In the figure, 6 trace point pairs measure 95% of the variance. The x axis shows the trace points on the critical path, with each assigned a reference number. The variance is highly localized, and inspecting the edge with the highest variance aids in finding the root cause of the contention problem. This edge corresponds to 7 lines of code, 4 of which are comments. In the remaining 3 lines, a semaphore (NOVA.COMPUTE.MANAGER.COMPUTEMANAGER._BUILD_SEMAPHORE) is acquired. Inspecting the initialization of this semaphore shows that the configuration option MAX CONCURRENT BUILDS determines the maximum number of simultaneous VM creations within a single host, explaining the root cause of the high variance in simultaneous VM creations as opposed to other requests.

The information yielded by the trace points around the semaphore, including the fact that semaphore queue lengths are correlated with response times, would give engineers a strong starting point in a real life scenario to identify the problem's root cause. In this specific case, engineers would be able to identify the root cause by examining the semaphore's initialization routine in which the default concurrency

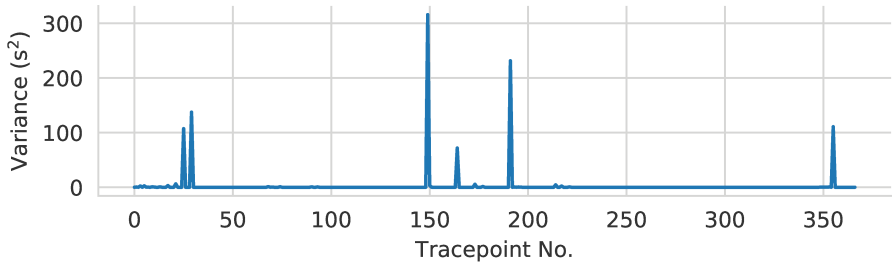

Figure 8.7: Variance of edge latencies for SERVER CREATE group.

value is specified, validating the second hypothesis.

Chapter 9: Summary and Conclusion

This work explored the use of a performance variation-based tool to locate areas in a distributed system where more instrumentation may be beneficial, both for differentiating between requests in a well-functioning system to make future problem diagnosis easier, or to narrow down the root cause of a current performance problem by identifying and localizing the performance varitiation resulting from that problem.

The tool used workflow-centric tracing to capture workflows of requests in an OpenStack test environment to test various aspects of using performance variation as a marker for regions of the system in which instrumentation does not sufficiently differentiate between requests doing different work. It found that a few trace edges between trace points are currently responsible for a disproportionate amount of the variance seen in the test system, suggesting that adding instrumentation in these areas could be useful for differentiating requests and understanding system behavior. The tool was subsequently able to identify high performance variation in a group of requests with an injected problem meant to simulate resource contention, and was able to localize this performance variation to the two trace points surrounding the problem. In the last experiment, the tool was also able to localize performance variation to highlight a system area affected by a misconfiguration causing actual resource contention.

While there is much opportunity for further work on this topic, these findings are promising for using performance variation to optimize instrumentation placement

in distributed systems. This holds true both for problem scenarios and general instrumentation placement, e.g. in system development. As instrumentation must frequently be added to systems to diagnose a large range of specific problems as they arise, performance variation-based techniques for identifying where this instrumentation is needed, based on the current system state, have the potential to significantly cut down on the amount of guesswork that often comes with adding instrumentation to these massively complex systems.

9.1. Limitations

Using latency variance to diagnose performance problems in a distributed system makes the assumption that a performance problem will cause a slowdown in the system at least part of the time. This slowdown shows up as latency variance across time for a group of requests that should have similar performance. This assumption carries with it the limitation that this technique cannot be used to help diagnose performance problems for requests that are consistently slow. Problems of this type would have to be addressed with some other method for diagnosis.

Another limitation is the dependence on some minimal instrumentation already present in the system under study. This technique only works for distributed systems which can and do have tracing systems enabled. This excludes many systems that rely heavily on databases.

More concretely, this work is limited in the generalizability of its results because the testing done for the experiments was not done at a large enough scale or on a wide enough variety of systems to merit statistical significance. This should be one of the next steps addressed in any future work expanding on this technique.

9.2. Future Work

There are many opportunities to expand on the scope of this work, in myriad directions. In the immediate future, this same work could be applied to other systems, such as Hadoop and Ceph, to see if the findings hold true beyond OpenStack. It would be especially interesting to apply it to cross-layer problems rather than just application-level. Moreover, the tests carried out in this thesis should be done on a much larger scale in order to have results that are reliable and statistically significant. The data gathered from the performance variation-based tool should be put through rigorous statistical and data science-based analyses. One unexplored but extremely relevant statistical concept to apply to this work is covariance: when performance variation rises or falls in one region of the system, does it rise or fall anywhere else, and what practical knowledge can be derived from this? Another missing piece to fill in would be to add default instrumentation to areas suggested by the tool in a system like OpenStack and see if it's useful to engineers running a production system. Ideally, if this is the case, these new instrumentation points in the system could be merged upstream to benefit everyone who uses the system, assuming the system is open source.

Longer-term, the question of how to set thresholds for marking "high" performance variation must be addressed. Is this based on the particular system, or can more widely applicable guidelines be used? Additionally, this work may be useful in a larger framework for automating dynamic instrumentation of a running system, such as the work being done currently on Pythia, a just-in-time instrumentation framework for distributed systems (Ates et al., 2019). It would also be interesting and perhaps powerful to address the role of regular expressions in controlling the granularity of expectation grouping for the traces. If each trace's structure is represented as a string,

regular expressions could group traces more flexibly based on some relevant subset of structural features, rather than based on the entire structure.

References

Akamai (n.d.). What is a cdn?

Asemanfar, A. (2018). The myth of the server's terrible, horrible, no good, very bad day.

Ates, E., Sturmann, L., Toslali, M., Krieger, O., Coskun, A., & Sambasivan, R. R. (2019). An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Manuscript submitted for publication.*

Ball, T. & Larus, J. R. (1996). Efficient path profiling. In *Proceedings of MICRO-29.*

Barroso, L. A. & Holzle, U. (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool.

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, Inc.

Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., & Brewer, E. (2004). Path-based failure and evolution management. In *NSDI '04: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation.*

Cormen, T. H., Lelserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* MIT Press, 3 edition.

Costello, K. & Hippold, S. (2018). Gartner forecasts worldwide public cloud revenue to grow 17.3 percent in 2019. online.

Darrow, B. (2016). Pssst, amazon cloud is not really new to banks. *Fortune*.

Desnoyers, P., Hennessey, J., Holden, B., Krieger, O., Rudolph, L., & Young, A. (2015). Using Open Stack for an Open Cloud Exchange(OCX). In *2015 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 48–53).

Devstack (n.d.). Openstack docs: Devstack.

Eventlet Networking Library (n.d.). Eventlet networking library.

Fonseca, R., Freedman, M. J., & Porter, G. (2010). Experiences with tracing causality in networked services. In *INM/WREN '10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., & Stoica, I. (2007). X-Trace: a pervasive network tracing framework. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*.

Google Drive (n.d.). Google drive.

Huang, J., Mozafari, B., & Wenisch, T. F. (2017). Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17 (pp. 64–79). New York, NY, USA: ACM.

IBM (n.d.). What is distributed computing.

Jaeger (n.d.). Jaeger website.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., & Song,

Y. J. (2017). Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

Kaldor, J. e. a. (2017). Canopy: An end-to-end performance tracing and analysis system. In *Proc. ACM Symposium on Operating Systems Principles (2017)* Shanghai, China: ACM.

Kozyrakis, C. (2013). *Resource Efficient Computing for Warehouse-scale Datacenters*. Technical report, Computer Systems Laboratory at Stanford University.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7).

Lloyd's (2018). *Emerging Risk Report 2018: Technology*. Technical report, Lloyd's of London and AIR Worldwide.

Mace, J. & Fonseca, R. (2018). Universal context propagation for distributed system instrumentation. In *Eurosys '18: Proceedings of the 13th ACM SIGOPS European Conference on Computer Systems*.

Mariani, L. & Pastore, F. (2008). Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 117–126).

Mariani, L., Pastore, F., & Pezze, M. (2009). A toolset for automated failure analysis. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)* (pp. 563–566).

Mass Open Cloud (n.d.). Mass open cloud website. `http://massopen.cloud`.

Mickens, J. (2013). The saddest moment.

Nunns, J. (2018). Cloud outage could cost up to $19bn in just six days. *Computer Business Review*.

OpenStack (n.d.). Openstack website.

OpenTracing website (n.d.). OpenTracing website. `http://opentracing.io/`.

OSProfiler (n.d.). Read the docs: Osprofiler.

Rally (n.d.). Openstack docs: Rally.

Ranganathan, A. & Campbell, R. H. (2007). What is the complexity of a distributed computing system? In *Complexity*, volume 12 (pp. 37–45).

Ranger, S. (2018). What is cloud computing? everything you need to know about the cloud, explained. *ZDNET*.

Raza, K. (2017). How the cloud is transforming healthcare. *Forbes*.

Rendek, L. (2017). Bash scripting tutorial for beginners.

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M., & Vahdat, A. (2006). Pip: detecting the unexpected in distributed systems. In *NSDI '06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*.

Rollins, S. (n.d.). Introduction to cs-682.

Rowe, J. (2017). Why software-as-a-service is taking off in healthcare. *Healthcare IT News Cloud Decision Center*.

Sambasivan, R. R. & Ganger, G. R. (2012). Automated diagnosis without predictability is a recipe for failure. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing* (pp. 21–21).: USENIX Association.

Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., & Ganger, G. R. (2016). Principled workflow-centric tracing of distributed systems. In *SoCC '16: Proceedings of the Seventh Symposium on Cloud Computing*.

Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., & Ganger, G. R. (2011). Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report dapper-2010-1, Google.

Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., & Ganger, G. R. (2006). Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*.

Whomes, R. (2018). Don't panic, it's on the mainframe: financial services and cloud computing. *Global Banking and Finance*.

Whyntie, T. & Coles, J. (2015). Number-crunching higgs boson: meet the world's largest distributed computer grid. *The Conversation*.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. (2009). Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*.

Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., & Pasupathy, S. (2010). SherLog: error diagnosis by connecting clues from run-time logs. In *ASPLOS '10: Proceedings*

of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems.

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., & Savage, S. (2012a). Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI' 12: Proceedings of the 10th conferences on Operating Systems Design & Implementation*.

Yuan, D., Park, S., & Zhou, Y. (2012b). Characterizing logging practices in open-source software. In *ICSE' 12: Proceedings of the 34th International Conference on Software Engineering*.

Yuan, D., Zheng, J., Park, S., Zhou, Y., & Savage, S. (2012c). Improving software diagnosability via log enhancement. *ACM SIGPLAN Notices*, 47(4), 3–14.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., & Zhou, Y. (2017). Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

Appendix A: Glossary

asynchronous As part of a request's workflow, an asynchronous activity is one that other parts of the request do not wait for. Consequently, it does not affect the overall response time of the request. An asynchronous task often executes concurrently with other parts of the request, and often has its own branch in the visualization of the workflow available in the request's trace. 19, 21, 34, 36

coefficient of variation The standard deviation of a set of values divided by the mean of that set of values. Like variance, coeffient of variation is a useful way of understanding how closely clustered a set of values is around its mean. Unlike variance, coefficient of variation is normalized for the mean of the set, which makes it possible to meaningfully compare different sets' coefficients of variation. 14, 25, 26

concurrent As part of a request's workflow, a concurrent activity overlaps in time with another part(s) of the request's workflow. Concurrency introduces complexity for understanding the workflow's critical path. 19, 20, 32, 33, 36, 45, 54, 55

critical path The temporally longest path or branch in a request's workflow when multiple concurrent operations are involved. The critical path determines the request's overall response time. 17, 21, 22, 31, 36, 39, 45, 48

DAG A directed acyclic graph (DAG) is a a graph with nodes and directional edges (to indicate, for example, the flow of data) in which no cycles are present. This work uses DAGs to represent traces of request workflows. iii

DAG directed acyclic graph. iii, vi, vii, 11, 17–22, 24, 26–28, 31–34, 37–40, 45, 68, *Glossary:* DAG

default instrumentation Instrumentation that is present in a system during periods of time when the system is assumed to be operating normally, as opposed to instrumentation added usually temporarily in order to diagnose a specific problem. Default instrumentation often collects data at a lower granularity across the system, compared with problem-related instrumentation that is concentrated on the region assumed to hold the problem's root cause. 43

distributed system A network of many computers as component parts, ex. public and private clouds. iii, iv, 1–3, 6, 10, 14, 16, 26

happened-before-relationship Causal relationships originally defined by Leslie Lamport, which can be established in distributed systems under two conditions: 1. if one event occurred earlier than another in the same process, or if one event is a sending of a message and the other is the receipt of a message in different processes. DAGs represent happened-before relationships with directed edges. 18, 21, 31, 33, 36

instrumentation Code inserted into a system for the purpose of revealing something about the system's performance. iii, iv, 3–6, 10, 14–16, 22, 23, 29, 30, 32, 42–44, 46–49

latency The length of time it takes the system to process something. Some latency always exists, but high latency is undesirable. 6, 12, 13, 15, 21, 34, 36, 39, 56

localize To determine which part of a system is responsible for some observed behavior. 12, 14, 15, 17, 24, 41, 47, 48, 54, 56

OpenStack A popular open source distributed application used for managing cloud environments and infrastructure. 22, 26–29, 35, 42, 43, 46–49, 54, 55

OpenTracing An API specification for open source tracing tools, which relies on the span model. *Glossary:* span model

OSProfiler A tracing tool built for OpenStack. 27–29, 32, 35, 44

performance variation Variation in response times or overall latencies of requests or their component parts. iii, 4–6, 10, 12, 14, 15, 17, 21–26, 29–32, 35, 36, 40–43, 45–49, 54–56, *Glossary:* variance

response time The total time it takes for a request to complete execution. 12–14, 25, 40, 57

similar Of a request or its workflow structure: Requests are similar when they are processed simliarly within and among the components of a system, that take comparable paths through the system, that have similar workflow structure in their trace representations, and that can be expected to have similar performance in terms of their overall response times. iv, 6, 12, 13, 23, 24, 37, 43

span A component of the span model of tracing, a span represents a semantic unit of work that is supposed to be human-meaningful and corresponds to the time between a starting instrumentation point and a stopping instrumentation point.

There is no standard definition of what type of work or what system boundaries (function, component, etc.) a span should represent. 20, 21, 34, 38

span model Also known as the swim lane model or Open Tracing model, the span model is a way of representing trace data in a nested structure of "parent spans" and corresponding "child spans" whose work is initiated by the parent. This model does not show any dependency or causal relationships among spans with the same parent. 11, 19–22, 27, 32–34, 37, 38, *Glossary:* span

synchronization point A part of a request's workflow in which several concurrent branches "join" back together into one branch; a part of a workflow that waits for several other activities in the workflow to complete before proceeding. 19, 22, 31–33, 36, 39

trace A record showing the workflow associated with a request. 10, 11, 16–20, 22–24, 27, 28, 30–41, 44, 45, 47, 55, *Glossary:* workflow-centric tracing

trace point An instrumentation point in the system that propagates contextual metadata, ex. a request ID, in order for its data to be associated with that of other trace points to later reconstruct the workflow of a request. 16, 23, 24, 31, 35, 45, 47, 49, 50, 55, 57, *Glossary:* workflow-centric tracing

variable Some data that is collected within a trace point and can be examined later as part of a request's trace. 14, 23, 49, 55

variance Standard deviation squared, showing the average "distance" of the data from a certain meaningful value, ex. the mean. 14, 25, 56, 57

workflow The work done within and among system components on behalf of a specific user command or request. Every system request has an associated workflow

that can be revealed with tracing tools. iv, 12–16, 18, 21–24, 26, 27, 29, 30, 35, 36, 38

workflow-centric tracing A means of instrumenting a distributed system that uses metadata propagation to capture request workflows and produces traces showing the path a request took through the system and any addiitonal information available at each trace point. iv, 12, 15, 22, 27, 30

workload A set of requests and their associated workflows. 22, 26, 28, 32, 35, 44, 48, 55