Diagnosing performance changes in distributed systems by comparing request flows

CMU-PDL-13-105

Submitted in partial fulfillment for the requirements for
the degreee of
Doctor of Philosophy
in
Electrical & Computer Engineering

Raja Raman Sambasivan

B.S., Electrical & Computer Engineering, Carnegie Mellon University
M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2013

To my parents and sister.

Keywords: distributed systems, performance diagnosis, request-flow comparison

Abstract

Diagnosing performance problems in modern datacenters and distributed systems is challenging, as the root cause could be contained in any one of the system's numerous components or, worse, could be a result of interactions among them. As distributed systems continue to increase in complexity, diagnosis tasks will only become more challenging. There is a need for a new class of diagnosis techniques capable of helping developers address problems in these distributed environments.

As a step toward satisfying this need, this dissertation proposes a novel technique, called *request-flow comparison*, for automatically localizing the sources of *performance changes* from the myriad potential culprits in a distributed system to just a few potential ones. Request-flow comparison works by contrasting the workflow of how individual requests are serviced within and among every component of the distributed system between two periods: a non-problem period and a problem period. By identifying and ranking performance-affecting changes, request-flow comparison provides developers with promising starting points for their diagnosis efforts. Request workflows are obtained with less than 1% overhead via use of recently developed end-to-end tracing techniques.

To demonstrate the utility of request-flow comparison in various distributed systems, this dissertation describes its implementation in a tool called Spectroscope and describes how Spectroscope was used to diagnose real, previously unsolved problems in the Ursa Minor distributed storage service and in select Google services. It also explores request-flow comparison's applicability to the Hadoop File System. Via a 26-person user study, it identifies effective visualizations for presenting request-flow comparison's results and further demonstrates that request-flow comparison helps developers quickly identify starting points for diagnosis. This dissertation also distills design choices that will maximize an end-to-end tracing infrastructure's utility for diagnosis tasks and other use cases.

Acknowledgments

Many people have helped me throughout my academic career and I owe my success in obtaining a Ph.D. to them. Most notably, I would like to thank my advisor, Greg Ganger, for his unwavering support, especially during the long (and frequent) stretches during which I was unsure whether my research ideas would ever come to fruition. His willingness to let me explore my own ideas, regardless of whether they ultimately failed or succeeded, and his ability to provide just the right amount of guidance when needed, have been the key driving forces behind my growth as a researcher.

I would also like to thank the members of my thesis committee for their time and support. My committee consisted of Greg Ganger (Chair), Christos Faloutsos, and Priya Narasimhan from Carnegie Mellon University, Rodrigo Fonseca from Brown University, and Ion Stoica from the University of California, Berkeley. Rodrigo's insights (often conveyed over late-night discussions) on tracing distributed-system activities have proven invaluable and helped shape the latter chapters of this dissertation. Christos's insights on machine learning and mechanisms for visualizing graph differences have also been invaluable. Priya and Ion's detailed feedback during my proposal and defense helped greatly improve the presentation of this dissertation.

Of course, I am greatly indebted to my many friends and collaborators with whom I shared many long hours, both at CMU and at internships. At CMU, Andrew Klosterman, Mike Mesnier, and Eno Thereska's guidance when I was a new graduate student was especially helpful. I learned how to rigorously define a research problem, break the problem down into manageable pieces, and make progress on solving those pieces from them. I often find myself looking back fondly on the 3 A.M. Eat & Park dinners Mike and I would go on to celebrate successful paper submissions—never have I been so exhausted and so happy at the same time. Alice Zheng helped me realize the tremendous value of inter-disciplinary research. Working with her convinced me to strengthen my background in statistics and machine learning, a decision I am grateful for today. At Google, Michael De Rosa and Brian McBarron spent long hours helping me understand the intricacies of various Google services in the span

of a few short months.

Elie Krevat, Michelle Mazurek and Ilari Shafer have been constant companions during my last few years of graduate school, and I am happy to say that we have collaborated on multiple research projects. Their friendship helped me stay sane during the most stressful periods of graduate school. I have also greatly valued the friendship and research discussions I shared with James Hendricks, Matthew Wachs, Brandon Salmon, Alexey Tumanov, Jim Cipar, Michael Abd-El-Malek, Chuck Cranor, and Terrence Wong. Of course, any list of my Parallel Data Lab cohorts would not be complete without mention of Karen Lindenfelser and Joan Digney. Joan was invaluable in helping improve the presentation of my work and Karen provided everything I needed to be happy while in the lab, including pizza, administrative help, and good conversations.

Though I have been interested in science and engineering as far back as I can remember, it was my high school physics teacher, Howard Myers, who gave my interests substance. His obvious love for physics and his enthusiastic teaching methods are qualities I strive to emulate with my own research and teaching.

I am especially grateful to my parents and sister for their unconditional help and support throughout my life. Their unwavering confidence in me has given me the strength to tackle challenging and daunting tasks.

My research would not have been possible without the gracious support of the members and companies of the Parallel Data Laboratory consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, and Western Digital). My research was also sponsored in part by two Google research awards, by the National Science Foundation, under grants #CCF-0621508, #CNS-0326453, and #CNS-1117567, by the Air Force Research Laboratory, under agreement #F49620-01-1-0433, by the Army Research Office, under agreements #DAAD19-02-1-0389 and #W911NF-09-1-0273, by the Department of Energy, under award #DE-FC02-06ER25767, and by Intel via the Intel Science and Technology Center for Cloud Computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern clouds and datacenters are rapidly growing in scale and complexity. Datacenters often contain an eclectic set of hardware and networks, on which they run many diverse applications. Distributed applications are increasingly built upon other distributed services, which they share with other clients. All of these factors cause problem diagnosis in these environments to be especially challenging. For example, Google engineers diagnosing a distributed application's performance often must have intricate knowledge of the application's components, the many services it depends upon (e.g., Bigtable [28], GFS [55], and the authentication mechanism), its configuration (e.g., the machines on which it's running and its resource allocations), its critical paths, the network topology, and the many co-located applications that might be interfering with it. Many researchers believe increased automation is necessary to keep management tasks in these environments from becoming untenable [38, 49, 50, 80, 95, 120, 127].

The difficulty of diagnosis in cloud computing environments is borne out by the many examples of failures observed in them over the past few years and engineers' herculean efforts to diagnose these failures [11, 107, 133]. In some cases, outages were exacerbated because of unaccounted for dependencies between a problematic service and other services. For example, in 2011, a network misconfiguration in a single Amazon Elastic Block Store (EBS) cluster resulted in an outage of all EBS clusters. Analysis eventually revealed that the thread pool responsible for routing user requests to individual EBS clusters used an infinite timeout and thus transformed a local problem into a global outage [129].

Though completely automated diagnosis is the eventual goal, many intermediary steps—each requiring less diagnostician[1] (or engineer) involvement—lie on the path to achieving it for most modern applications and services. Expert diagnosticians possess vast amounts of knowledge and insight, which they synthesize to diagnose complex problems, so attempting to replace them immediately with automation is not a feasible approach. Instead, a logical first step is to create techniques that use systems knowledge, machine learning, statistics, and visualization to help diagnosticians with their efforts—for example, by automatically localizing the root cause of a new problem from the myriad components and dependencies in the system to just a few potential culprits. As Figure 1.1 shows, problem localization is one of the three key steps of diagnosis, so automating it can vastly reduce the effort needed to diagnose new problems. Once shown to be effective, these localization techniques can serve as excellent building blocks on which to layer further automation.

The main contribution of this dissertation is a problem localization technique, called *request-flow comparison*, for automatically localizing the root cause of unexpected *performance changes* (i.e., degradations), an important problem-type for which better diagnosis techniques are needed. Such changes are common in both traditional dedicated-machine environments and shared-machine environments, such as modern datacenters. They arise due to issues including (but not limited to) code regressions, misconfigurations, resource contention, hardware problems, and other external changes. An analysis of bugs reported for the Ursa Minor distributed storage service [1], running in a dedicated-machine environment, shows that more than half the reported problems were unexpected performance changes [117]. I observed similar ratios for shared-machine environments by analyzing

[1] In this dissertation, the term *diagnostician* refers to software developers, first-response teams (e.g., Google's site-reliability engineers), and any other group tasked with diagnosing complex problems in distributed systems and datacenters.

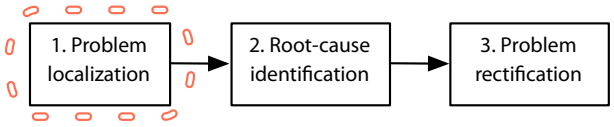


Figure 1.1: The problem diagnosis workflow. This diagram illustrates the three key steps of diagnosing problems in distributed systems. First, diagnosticians must localize the source of the problem from the numerous components in the system and its dependencies to just a few potential culprits. Second, they must identify the specific root cause. Third, they must affect a fix. This dissertation describes a novel technique for automating the first step: problem localization.

Hadoop File System (HDFS) [134] bug reports, obtained from the JIRA [135] and conversations with developers.

Request-flow comparison works by comparing how a distributed service processes client and system-initiated requests (e.g., "read a file") between two periods of operation: one where performance was acceptable (the non-problem period) and one where performance has degraded (the problem period). Each request has a corresponding workflow representing the execution order and timing of the application's components, sub-components (e.g., functions within a component), and dependencies involved in processing the request. These request workflows (also called request flows) can be represented as graphs and comparing corresponding ones between both periods to identify performance-affecting changes focuses diagnosis efforts to just the changed areas. Figure 1.2 illustrates the request-flow comparison process.

To obtain request workflows, request-flow comparison builds on *end-to-end tracing*, a powerful mechanism for efficiently (i.e., with less than 1% runtime overhead [117, 125]) capturing causally-related activity within and among the components of a distributed system [3, 15, 27, 29, 47, 48, 66, 78, 110, 111, 117, 125, 130, 131, 141, 142, 152]. End-to-end tracing's workflow-centric approach contrasts with machine-centric approaches to tracing and monitoring, such as DTrace [26] and Ganglia [92], which cannot be used to obtain a complete and coherent view of a service's activity. As distributed services grow in complexity and continue to be layered upon other services, such coherent tracing methods will become increasingly important for helping developers understand their end-to-end behaviour.

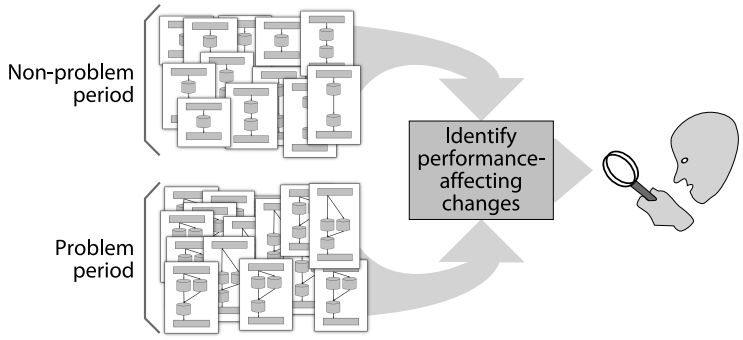


Figure 1.2: Request-flow comparison. This technique takes as input request workflows that show how individual requests were processed during two periods: a non-problem period and a problem period. Since even small distributed systems can process hundreds of requests per second, the number of input workflows can be very large. Request-flow comparison localizes the source of the problem by comparing corresponding request workflows between both periods to identify performance-affecting changes, which represent good starting points for diagnosis.

Indeed, there are already a growing number of industry implementations, including Google's Dapper [125], Cloudera's HTrace [33], Twitter's Zipkin [147], and others [36, 143]. Looking forward, end-to-end tracing has the potential to become the fundamental substrate for providing a global view of intra- and inter-datacenter activity in cloud environments.

1.1 Thesis statement and key results

This dissertation explores the following thesis statement:

Request-flow comparison is an effective technique for helping developers localize the source of performance changes in many request-based distributed services.

To evaluate this thesis statement, I developed Spectroscope [117], a tool that implements request-flow comparison, and used it to diagnose real, previously undiagnosed performance changes in the Ursa Minor distributed storage service [1] and select Google services. I also explored Spectroscope's applicability to the Hadoop File System [134] and identified needed extensions. Since problem localization techniques like request-flow comparison do not directly identify the root cause of a problem, but rather help diagnosticians in their efforts, their effectiveness depends on how well they present their results. To identify good presentations, I explored three approaches for visualizing Spectroscope's results and ran a 26-person user study to identify which ones are best for different problem types [116]. Finally, based on my experiences, I conducted a design study illustrating the properties needed from end-to-end tracing for it to be useful for various use cases, including diagnosis. My thesis statement is supported by the following:

1. I demonstrated the usefulness of request-flow comparison by using Spectroscope to diagnose six performance changes in a 6-component configuration of Ursa Minor, running in a dedicated-machine environment. Four of these problems were *real* and *previously undiagnosed*—the root cause was not known a priori. One problem had been previously diagnosed and was re-injected into the system. The last problem was synthetic and was injected into the system to evaluate a broader spectrum of problem types diagnosable by request-flow comparison. The problems diagnosed included those due to code regressions, misconfigurations, resource contention, and external changes.

2. I demonstrated that request-flow comparison is useful in other environments by using

a version of Spectroscope to diagnose two real, previously undiagnosed performance changes within select Google services, running in a shared-machine environment. The Google version of Spectroscope was implemented as an addition to Dapper [125], Google's end-to-end tracing infrastructure. I also explored Spectroscope's applicability to HDFS and identified extensions needed to the algorithms and heuristics it uses for it to be useful in this system.

3. I further demonstrated that diagnosticians can successfully use request-flow comparison to identify starting points for diagnosis via a 26-person user study that included Google and Ursa Minor developers. I also used this user study to identify the visualization approaches that best convey request-flow comparison's results for different problem types and users.

4. Via a design study, I determined how to design end-to-end tracing infrastructures so that they are maximally effective for supporting diagnosis tasks. In the study, I distilled the key design axes that dictate an end-to-end tracing infrastructure's utility for different use cases, such as diagnosis and resource attribution. I identified good design choices for different use cases and showed where prior tracing implementations fall short.

1.2 Goals & non-goals

The key goal of this dissertation is to develop and show the effectiveness of a technique for automating one key part of the problem diagnosis workflow: localization of problems that manifest as steady-state-performance changes (e.g., changes to the 50^{th} or 60^{th} percentile of some important performance metric's distribution). In doing so, it aims to address three facets of developing an automation tool that helps developers perform management tasks. First, it describes algorithms and heuristics for the proposed automation and evaluates them. Second, it presents a user study evaluating three approaches for effectively presenting the automation tool's results to diagnosticians. Third, it describes the properties needed from the underlying data source (end-to-end tracing) for success.

By describing an additional application for which end-to-end tracing is useful (diagnosing performance changes), this dissertation also aims to further strengthen the argument for implementing this powerful data source in datacenters and distributed systems. Though end-to-end tracing has already been shown to be useful for a variety of use cases, includ-

ing diagnosis of certain correctness and performance problems [29, 47, 48, 110, 117, 125], anomaly detection [15, 29], profiling [27, 125], and resource usage attribution [15, 141], additional evidence showing its utility is important so as to prove the significant effort required to implement and maintain it is warranted.

A significant non-goal of this dissertation is localizing performance problems that manifest as anomalies (e.g., requests that show up in the 99th percentile of some important performance metric's distribution). Other techniques exist for localizing such problems [15]. This dissertation also does not aim to help diagnose correctness problems, though request-flow comparison could be extended to do so.

Another important non-goal of this dissertation is complete automation. Request-flow comparison automatically localizes the root cause of an performance change; it does not automatically identify the root cause or automatically perform corrective actions. Much additional research is needed to understand how to automate other other phases of the diagnosis workflow.

This dissertation also does not attempt to *quantitatively* show that request-flow comparison is better than other automated problem localization techniques. Such an evaluation would require a user study in which many expert diagnosticians are asked to diagnose real problems observed in a common system using different localization techniques. But, it is extraordinarily difficult to obtain a large enough pool of expert diagnosticians for the length of time needed to complete such a study. Instead, this dissertation demonstrates request-flow comparison's effectiveness by illustrating how it helped diagnosticians identify the root causes of *real*, *previously undiagnosed* problems. It also qualitatively argues that request-flow comparison is more useful than many other localization techniques because it uses a data source (end-to-end traces) that provides a coherent, complete view of a distributed system's activity.

A final non-goal is this dissertation is online diagnosis. Instead, this dissertation focuses on demonstrating the effectiveness of request-flow comparison and leaves implementing online versions of it to future work.

1.3 Assumptions

To work, request-flow comparison relies on four key assumptions. First, it assumes that diagnosticians can identify two distinct periods of activity, one where performance was acceptable (the non-problem period) and another where performance has degraded (the

problem period). Doing so is usually easy for steady-state-performance changes. Since request-flow comparison simply identifies changes in request workflows, it is indifferent to the number of distinct problems observed in the problem period, as long as they do not exist in the non-problem period. Also, the problem period need not be a tight bound around the performance degradation being diagnosed, but simply needs to contain it. However, for many implementations, the effectiveness of request-flow comparison will increase with tighter problem-period bounds.

Second, request-flow comparison assumes that the workload (e.g., percentage of different request types) observed in both the non-problem and problem periods are similar. If they are significantly different, there is no basis for comparison, and request-flow comparison may identify request workflow changes that are a result of the differing workloads rather than problems internal to the distributed system. To avoid such scenarios, diagnosticians could use change-point detection algorithms [31, 99] to determine whether the non-problem and problem period workloads differ significantly before applying request-flow comparison.

Third, request-flow comparison assumes that users possess the domain expertise necessary to identify a problem's root cause given evidence showing how it affects request workflows. As such, request-flow comparison is targeted toward distributed systems developers and first-response teams, not end users or most system administrators. Chapters 3.6 and 8.2.2 suggest that future work should consider how to incorporate more domain knowledge into request-flow comparison tools, so as to reduce the amount of domain knowledge users must possess to make use of their results.

Fourth, request-flow comparison assumes the availability of end-to-end traces (i.e., traces that show the workflow of how individual requests are serviced within and among the components of the distributed system being diagnosed). Request-flow comparison will work best when these traces exhibit certain properties (e.g., they distinguish concurrent activity from sequential activity), but will work with degraded effectiveness when some are not met.

1.4 Dissertation organization

This dissertation contains eight chapters. Chapter 2 introduces request-flow comparison, provides a high-level overview of it, and describes the algorithms and heuristics used to implement request-flow comparison in Spectroscope. Chapter 3 describes case studies of using Spectroscope to diagnose real, previously undiagnosed changes in Ursa Minor [1]

and in select Google services. It also describes extensions needed to Spectroscope for it to be useful for diagnosing problems in HDFS [134]. Most of the content of these chapters was published in NSDI in 2011 [117], William Wang's Master's thesis in 2011 [150], and HotAC in 2007 [118].

Chapter 4 describes the user study I ran to identify promising approaches for visualizing request-flow comparison's results. Most of this chapter was previously published in a technical report in 2013 (CMU-PDL-13-104) [116].

Chapter 5 highlights a stumbling block on the path to automated diagnosis: lack of predictability in distributed systems (i.e., high variance in key metrics). It shows how Spectroscope and other diagnosis tools are limited by high variance and suggests a framework for helping developers localize high variance sources in their systems. In addition to increasing the effectiveness of automation, eliminating unintended variance will increase distributed system performance [16]. Most of this chapter was published in HotCloud in 2012 [115].

Chapter 6 presents related work about performance diagnosis tools. Chapter 7 describes properties needed from end-to-end tracing for it to be useful for various use cases, such as diagnosis and resource attribution. Finally, chapter 8 concludes.

Chapter 2

Request-flow comparison

This chapter presents request-flow comparison and its implementation in a diagnosis tool I built called Spectroscope. Section 2.1 presents a high-level overview of request-flow comparison and the types of performance problems for which it is useful. Section 2.2 provides relevant background about end-to-end tracing and describes the properties request-flow comparison needs from it. Section 2.4 describes the workflow that any tool that implements request-flow comparison will likely have to incorporate. Different tools may choose different algorithms to implement the workflow, depending on their goals. Section 2.5 discusses request-flow comparison's key limitations. Section 2.6 describes the algorithms and heuristics used by Spectroscope, which were chosen for their simplicity and ability to limit false positives.

2.1 Overview

Request-flow comparison helps diagnosticians localize performance changes that manifest as *mutations* or changes in the workflow of how client requests and system-initiated activities are processed. It is most useful in *work-evident* distributed systems, for which the work necessary to service a request, as measured by its response time or aggregate throughput, is evident from properties of the request itself. Examples of such systems include most distributed storage systems (e.g., traditional filesystems, key/value stores, and table stores) and three-tier web applications.

Request-flow comparison, as described in this dissertation, focuses on helping diagnose steady-state-performance changes (also known as behavioural changes). Such changes differ from performance problems caused by anomalies. Whereas anomalies are a small

number of requests that differ greatly from others with respect to some important metric distribution, steady-state changes often affect many requests, but perhaps only slightly. More formally, anomalies affect only the tail (e.g., 99th percentile) of some important metric's distribution, whereas steady-state changes affect the entire distribution (e.g., they change the 50th or 60th percentile). Both anomalies and steady-state performance changes are important problem types for which advanced diagnosis tools are needed. However, most existing diagnosis tools that use end-to-end traces focus only on anomaly detection for performance problems [15] or correctness problems [29, 78].

Since request-flow comparison only aims to identify performance-affecting mutations, it is indifferent to the actual root cause. A such, many types of problems can be localized using request-flow comparison, including performance changes due to configuration changes, code changes, resource contention, and failovers to slower backup machines. Request-flow comparison can also be used to understand why a distributed application performs differently in different environments (e.g., different datacenters) and to rule out the distributed system as the cause of an observed problem (i.e., by showing request workflows have not changed). Though request-flow comparison can be used in both production and testing environments, it is exceptionally suited to regression testing, since such tests always yield well-defined periods for comparison.

2.2 End-to-end tracing

To obtain graphs showing the workflow of individual requests, request-flow comparison uses end-to-end tracing, which identifies the flow of causally-related activity (e.g., a request's processing) within and among the components of a distributed system. Many approaches exist, but the most common is *metadata-propagation-based tracing* [27, 29, 46, 47, 48, 110, 117, 125, 141]. With this approach, *trace points*, which are similar to log statements, are automatically or manually inserted in key areas of the distributed system's software. Usually, they are automatically inserted in commonly-used libraries, such as RPC libraries, and manually inserted in other important areas. During runtime, the tracing mechanism propagates an unique ID with each causal flow and uses it to stitch together executed trace points, yielding end-to-end traces. The runtime overhead of end-to-end tracing can be limited to less than 1% by using head-based sampling, in which a random decision is made at the start of a causal flow whether or not to collect any trace points for it [117, 125].

Other approaches to end-to-end tracing include *black-box-based tracing* [3, 21, 78, 83, 111,

130, 131, 152] and *schema-based tracing* [15, 66]. The first is used in un-instrumented systems, in which causally-related activity is inferred based on externally observable events (e.g., network activity). With the second, IDs are not propagated. Instead, an external schema dictates how to stitch together causally-related trace points. Chapter 7 further describes different end-to-end tracing approaches and discusses design decisions that affect a tracing infrastructure's utility for different use cases. The rest of this section focuses on properties needed from end-to-end tracing for request-flow comparison.

2.3 Requirements from end-to-end tracing

Any end-to-end tracing approach can be used with request-flow comparison, as long as the resulting traces can be represented as graphs annotated with detailed (per-component, per-function, or between trace point) timing information. The most meaningful results will be obtained if *request-flow graphs*—directed acyclic graphs that show the true structure of individual requests (i.e., components or functions accessed, concurrency, forks. and joins)—are used. Figure 2.1 shows an example of a request-flow graph obtained for the Ursa Minor distributed storage service [1]. Call graphs or other types of graphs that do not preserve true request structure can also be used with degraded effectiveness. Request-flow graphs can be constructed from end-to-end traces if they satisfy the two properties listed below:

1. **The traces distinguish concurrent activity from sequential activity**: Doing so is necessary to differentiate requests with truly different structures from those that differ only because of non-determinism in service order. This requirement can be satisfied by including information necessary to establish happens-before relationships [85].
2. **The traces capture forks and joins**: Without trace points indicating forks, traces would not be able to identify the thread responsible for initiating a set of concurrent activity. Without trace points indicating joins, traces would not be able to distinguish requests that wait for a specific thread or RPC versus those that wait for a subset.

Additionally, to be useful for diagnosis, traces should show trace points executed on the critical path of individual requests. To allow graphs to be annotated with timing information, traces should include timestamps that are comparable across machines or should include information necessary to establish happens-before relationships for inter-machine accesses. Finally, the amount of instrumentation affects the utility of request-flow comparison. At a

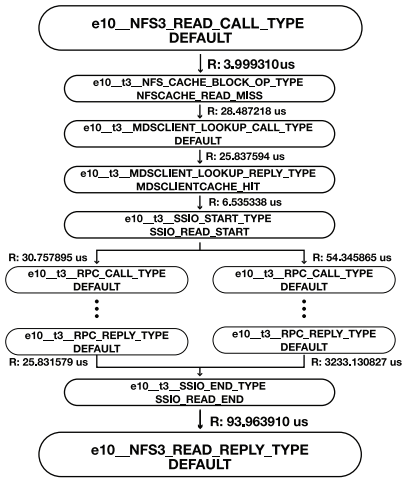


Figure 2.1: A request-flow graph. Nodes of such graphs show trace points executed by individual requests and edges show latencies between successive trace points. Fan-outs represent the start of concurrent activity and fan-ins represent synchronization points. The graph shown depicts how one READ request was serviced by the Ursa Minor distributed storage service [1] and was obtained using the Stardust tracing infrastructure [141]. For this system, nodes names are constructed by concatenating the machine name (e.g., e10), the component name (e.g., NFS), the trace-point name (e.g., READ_CALL_TYPE) and an optional semantic label. Ellipses indicate omitted trace points, some of which execute on different components or machines.

minimum, traces should show the components or machines accessed by individual requests.

2.4 Workflow

Figure 2.2 shows request-flow comparison's workflow. In the first step, it takes as input request-flow graphs from a non-problem period and a problem period. In most cases, graphs from two periods of actual system execution are used. For example, graphs from two executions of a benchmark could be compared when diagnosing performance regressions for a system under test. Alternatively, graphs from different parts of the same benchmark execution could be compared. For live production systems, graphs from two distinct time ranges could be used, with the problem period representing an interval during which clients complained, a pager alert was received, or a service-level agreement violation occurred. For example, at Google, I often used daylong non-problem and problem periods, which I selected based on a Google dashboard that shows per-day average response times for major services. Though graphs obtained from a model describing expected performance could be used for the non-problem period, these are hard to create and not widely available for

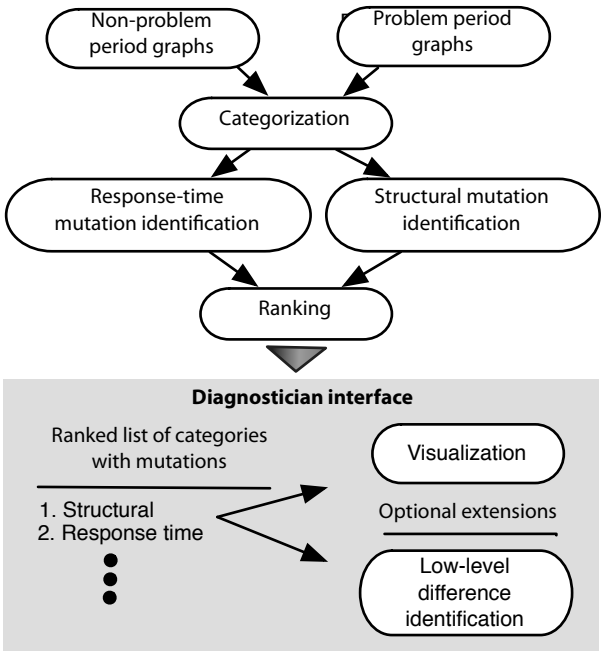


Figure 2.2: The request-flow comparison workflow. Request-flow graphs from a non-problem period and a problem period serve as the inputs to the workflow. Similar requests from both periods are grouped into the same category and those that contain performance-affecting mutations are identified. Categories with mutations are ranked according to their effect on the overall performance change. Ranked categories are presented to the diagnostician for further analysis.

complex distributed systems [132].

Even small distributed systems can service hundreds to thousands of requests per second, so comparing all of them individually is not feasible. As such, the second step of request-flow comparison's workflow involves grouping the input request-flow graphs from both periods into *categories*, which are used as the fundamental unit for comparing request flows. Categories should contain requests that are expected to perform similarly, the choice of which may differ depending on the distributed system under analysis. A basic expectation, valid in many work-evident systems, is that similarly-structured or identically-structured requests (i.e., those that visit the same components and functions and exhibit the same amount of concurrency) should perform similarly. However, additional information may be useful. For example, for web servers, the URL could help guide categorization. For distributed storage systems in which the amount of data retrieved can vary greatly (e.g., GFS [55]), the size of data requested could be used. Sometimes additional aggregation is useful to reduce the number of categories. For example, in load-balanced systems, requests

that exhibit different structures only because they visit different replicas should be grouped into the same category.

The third step of request-flow comparison's workflow involves identifying which categories contain mutations and precursors. *Mutations* are expensive requests observed in the problem period that have mutated or changed from less expensive requests observed in the non-problem period. Conversely, *precursors* are requests from the non-problem period that have mutated into more expensive requests in the problem period. Two very common types of mutations are response-time mutations and structural mutations. *Response-time mutations* are requests whose structure remains identical between both periods, but whose response times have increased. Comparing them to their precursors localizes the problem to the components or functions responsible for the slowdown and focuses diagnosis efforts. *Structural mutations* are requests whose performance has decreased because their structure or workflow through the distributed system has changed. Comparing them to their precursors identifies the changed substructures, which are good locations to start diagnosis efforts.

The fourth step of request-flow comparison's workflow is to focus diagnosticians' effort on the most performance-affecting mutations by ranking them according to their contribution to the performance change. In the fifth and final final step, ranked categories are presented to the diagnostician for further analysis.

Many additional workflow steps are possible. For example, additional localization may be possible by identifying the low-level parameters (e.g., function variables, client-sent parameters) that best differentiate mutations and their precursors. As such, Spectroscope includes an optional workflow step for such identification. For two case studies, this additional step immediately revealed the root cause (see the case studies described in Chapter 3.3.1 and 3.3.2). Also, Spectroscope could be extended to detect anomalies by identifying requests in the tails of intra-category, intra-period, response-time distributions.

2.5 Limitations

Despite its usefulness, there are five key limitations to request-flow comparison's workflow. Most notably, request-flow comparison's method of localizing performance changes by identifying mutations may not identify the most direct problem sources. For example, in one case study, described in Chapter 3.3.1, my implementation of request-flow comparison, Spectroscope, was used to help diagnose a performance degradation whose root cause was

a misconfiguration. Though Spectroscope was able to localize the problem to a change in the storage node used by one component of the service and to specific variables used in the codebase, it did not localize the problem to the problematic configuration file. Similarly, when diagnosing problems such as contention, request-flow comparison may localize the problem by identifying a change in per-component timing, but will not localize it to the specific competing process or request. I believe additional workflow steps could be used to increase the directness of localization. This is exemplified by Spectroscope's additional workflow step, which seeks to localize performance changes to low-level parameter differences between mutations and precursors.

The number of categories identified by request-flow comparison as containing mutations will likely greatly exceed the number of problems present in the system. This is partly because a single problem will often affect many different request types, each with different performance expectations. For example, a single performance problem may affect both attribute requests and write requests. Since these two request types are expected to exhibit fundamentally different performance characteristics, they will be assigned to different categories and identified as separate mutations, even though both show the same underlying problem. To cope, after fixing the problem identified by a highly-ranked mutation category, diagnosticians should always re-run the request-flow comparison tool to see if other mutations were caused by the same problem. If so, they will not appear in the new results.

The granularity at which request-flow comparison can localize problems is limited by the granularity of tracing. For example, if the traces only show inter-component activity (e.g., RPCs or messages passed between components), then request-flow comparison will be limited to localizing problems to entire components. Component-level localization is also possible for black-box components (i.e., components that export no trace data whatsoever) if traces in surrounding components document accesses to them.

Request-flow comparison, as presented in this dissertation, will be of limited value in dynamic environments (i.e., ones in which applications can be moved while they are executing or between comparable executions). To be useful in such environments, extra workflow steps would be needed to determine whether an observed performance change is the expected result of an infrastructure-level change or is representative of a real problem [119].

Finally, many distributed systems mask performance problems by diverting additional resources to the problematic application or job. For example, many map-reduce implementations, such as Hadoop [8], limit the performance impact of stragglers by aggressively

duplicating or restarting slow-to-complete tasks [6, 155]. Request-flow comparison will be less useful in these systems, since they may exhibit no discernible performance slowdown in some cases.

2.6 Implementation in Spectroscope

I implemented request-flow comparison in two versions of Spectroscope. The first was used to diagnose problems in Ursa Minor [1] and to explore the utility of Spectroscope's algorithms and heuristics in HDFS [134]. It was written using a combination of Perl, Matlab, and C. The second was used to diagnose select Google services and was implemented in C++ as an extension to Google's end-to-end tracing infrastructure, Dapper [125]. Specifically, the Google version of Spectroscope uses Dapper's aggregation pipeline to obtain call graphs. Both versions implement request-flow comparison's workflow using the same algorithms and heuristics, which were chosen for their simplicity, ability to use unlabeled data, and ability to regulate false positives. As such, they are not differentiated in this dissertation unless necessary. The second constraint is necessary because labels indicating whether individual requests performed well or poorly are uncommon in many systems. The period label (problem or non-problem) is not sufficient because both periods may contain both types of requests. I included the third constraint because Facebook and Google engineers indicated that false positives are the worst failure mode of an automated diagnosis tool because of the amount of diagnostician effort wasted [107]. However, when running user studies to evaluate visualizations for Spectroscope (see Chapter 4), I found that false negatives are just as troublesome, because they reduce diagnosticians' confidence in automation.

In addition to comparing request flows between two periods, Spectroscope can also be used to categorize requests observed in a single period. In such cases, Spectroscope will output the list of categories observed, each annotated with relevant metrics, such as average response time and response time variance. Categories can be ranked by any metric computed for them. The remainder of this section describes the algorithms and heuristics used by Spectroscope for request-flow comparison.

2.6.1 Categorization

Spectroscope creates categories composed of identically-structured requests. It uses a string representation of individual request-flow graphs, as determined by a depth-first traversal, to identify which category to assign a given request. Though this method will incorrectly

assign requests with different structures, yet identical string representations, to the same category, I have found it works well in practice. For requests with parallel substructures, Spectroscope computes all possible string representations when determining the category in which to bin them. The exponential cost is mitigated by imposing an order on parallel substructures (i.e., by always traversing children in lexographic order) and by the fact that most requests we observed in Ursa Minor and Google exhibited limited parallelism.

Creating categories that consist of identically-structured requests may result in a large number of categories. To reduce the number, I explored using unsupervised clustering algorithms, such as those used in Magpie [15], to group similar, but not identical, requests into the same category [118]. But, I found that off-the-shelf clustering algorithms created categories that were too coarse-grained and unpredictable. Often, they would assign requests with important differences into the same category, masking their existence. Though there is potential for improving Spectroscope's categorizing step by using clustering algorithms, improvements are needed, such as specialized distance metrics that better align with diagnosticians' notions of request similarity.

2.6.2 Identifying response-time mutations

Since response-time mutations and their precursors have identical structures, they will be assigned to the same category, but will have different period labels. As such, categories that contain them will have significantly different non-problem and problem-period response times. To limit false positives, which result in wasted diagnostician effort, Spectroscope uses the Kolmogorov-Smirnov two-sample hypothesis test [91] instead of raw thresholds to identify such categories.

The Kolmogorov-Smirnov test takes as input two empirical distributions and determines whether there is enough evidence to reject the null hypothesis that both represent the same underlying distribution (i.e., they differ only due to chance or natural variance). The null hypothesis is rejected only if the expected false-positive rate of declaring the two distributions as different—i.e., the p-value—is less than a pre-set value, almost always 5%. Spectroscope runs the Kolmogorov-Smirnov hypothesis test on a category's observed non-problem and problem-period response-time distributions and marks it as containing response-time mutations if the test rejects the null hypothesis. Since the test will yield inaccurate results if the following condition is true: $\frac{non\text{-}problem\ period\ requests \cdot problem\ period\ requests}{non\text{-}problem\ period\ requests + problem\ period\ requests} \leq 4$, such categories are summarily declared not to contain mutations. The Kolmogorov-Smirnov test's runtime

is $O(N)$, where N is the number samples in the distributions being compared (i.e., the number of non-problem and problem-period requests).

The Kolmogorov-Smirnov test is non-parametric, which means it does not assume a specific distribution type, but yields increased false negatives compared to tests that operate on known distributions. I chose a non-parametric test because I observed per-category response-time distributions are not governed by well-known distributions. For example, inadequate instrumentation might result in requests with truly different structures and behaviours being grouped into the same category, yielding multi-modal response-time distributions.

Once categories containing response-time mutations have been identified, Spectroscope runs the Kolmogorov-Smirnov test on the category's edge latency distributions to localize the problem to the specific areas responsible for the overall response time change.

2.6.3 Identifying structural mutations and their precursors

Since structural mutations are requests that have changed in structure, they will be assigned to different categories than their precursors during the problem period. Spectroscope uses two steps to identify structural mutations and their precursors. First, Spectroscope identifies all categories that contain structural mutations and precursors. Second, it determines a mapping showing which precursor categories are likely to have donated requests to which structural-mutation category during the problem period.

Step 1: Identifying categories containing mutations and precursors

To identify categories containing structural mutations and their precursors, Spectroscope assumes that the loads observed in both the non-problem and problem periods were similar (but not necessarily identical). As such, structural-mutation categories can be identified as those that are assigned significantly *more* problem-period requests than non-problem-period ones. Conversely, categories containing precursors can be identified as those that are assigned significantly *fewer* problem-period requests than non-problem ones.

Since workload fluctuations and non-determinism in service order dictate that per-category counts will always vary slightly between periods, Spectroscope uses a raw threshold to identify categories that contain structural mutations and their precursors. Categories that contain `SM_P_THRESHOLD` more requests from the problem period than from the non-problem period are labeled as containing mutations and those that contain `SM_P_THRESHOLD`

fewer are labeled as containing precursors. Unlike for response-time mutations, Spectroscope does not use non-parametric hypothesis tests to identify categories containing structural mutations because multiple runs of the non-problem and problem period would be needed to obtain the required input distributions of category counts. Also, Google engineers indicated that request structures change frequently at Google due to frequent software changes from independent teams, discouraging the use of approaches that require long-lived models.

Choosing a good threshold for a workload may require some experimentation, as it is sensitive to both the number of requests issued and the sampling rate. Fortunately, it is easy to run Spectroscope multiple times, and it is not necessary to get the threshold exactly right. Choosing a value that is too small will result in more false positives, but the ranking scheme will assign them a low rank, and so will not mislead diagnosticians.

Mapping structural mutations to precursors

Three heuristics are used to identify possible precursor categories for each structural-mutation category. Algorithm 1 shows pseudocode for these heuristics. Figure 2.3 shows how they are applied.

The first heuristic involves pruning the total list of precursor categories to eliminate ones with a different root node than requests in the structural-mutation category under consideration. The root node describes the overall type of a request, for example READ, WRITE, or READDIR, and requests of different high-level types should not be precursor/mutation pairs.

The second heuristic removes from consideration precursor categories that have decreased in request count less than the increase in request count of the structural-mutation category. This "1:unique N" assumption reflects the common case that one problem will likely generate many different mutations due to the way various resources will be affected. For example, a problem that causes extra requests to miss in one cache may also affect the next level cache, causing more requests to miss in it too. Since this assumption will not alway hold, it can be optionally disabled. However, I have found it to work sufficiently well in practice.

The third heuristic ranks the remaining precursor categories according to their likelihood of having donated requests to the structural-mutation category under consideration. It assumes that precursor categories that are structurally similar to the structural-mutation category are more likely to have donated requests. As such, the precursor categories are ranked

Input : List of precursor categories and structural-mutation categories
Output: Ranked list of candidate precursor categories for each structural-mutation category

```
for i ← 1 to length(sm_list) do
    candidates [i] ← find_same_root_node_precursors(precursor_list, sm_list [i]);
    if apply_1_n_heuristic then
        candidates [i] ← prune_unlikely_precursors( candidates [i], sm_list [i]);
    end
    // Re-arrange in order of inverse normalized string-edit distance
    candidates [i] ← rank_by_inverse_nsed( candidates [i], sm_list [i]);
end
```

Algorithm 1: Heuristics used to identify and rank precursor categories that could have donated requests to a structural-mutation category. The "1:unique N" heuristic assumes that one precursor category will likely donate requests to many structural-mutation categories and eliminates precursor categories that cannot satisfy this assumption. The final output list is ranked by inverse normalized string-edit distance, which is used to approximate the probability that a given precursor category donated requests. The cost of these heuristics is dominated by the string-edit distance calculation, which costs $O(N^2)$, where N is the length of each string.

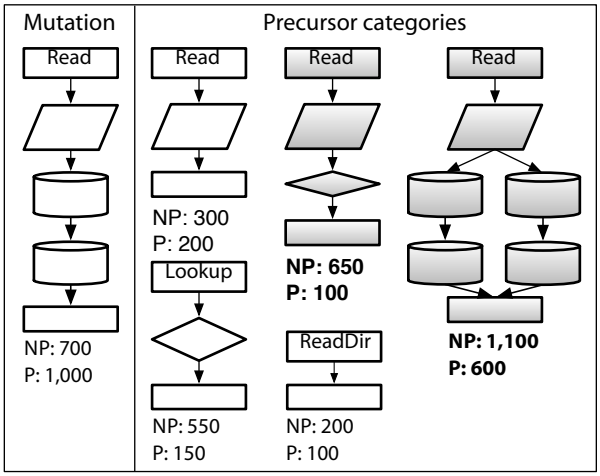


Figure 2.3: How the precursor categories of a structural-mutation category are identified. One structural-mutation category and five precursor categories are shown, each with their corresponding request counts from the non-problem (NP) and problem (P) periods. For this case, the shaded precursor categories will be identified as those that could have donated requests to the structural-mutation category. The precursor categories that contain LOOKUP and READDIR requests cannot have donated requests, because their constituent requests are not READS. The top left-most precursor category contains READS, but the 1:N heuristic eliminates it. Due to space constraints, mocked-up graphs are shown in which different node shapes represent the types of components accessed.

by the inverse of their normalized Levenshtein string-edit distance from the structural-mutation category. The string-edit distance implementation used by the Ursa Minor/HDFS version of Spectroscope has runtime $O(N^2)$ in the number of graph nodes. As such, it dominates runtime when input request-flow graphs do not exhibit much concurrency. Runtime could be reduced by using more advanced edit-distance algorithms, such as the $O(ND)$ algorithm described by Berghel and Roach [19], where N is the number of graph nodes and D is the edit distance computed. Sub-polynomial approximation algorithms for computing edit distance also exist [7].

2.6.4 Ranking

Spectroscope ranks categories containing mutations according to their contribution to the overall performance change. The contribution for a category containing response-time mutations is calculated as the number of non-problem-period requests assigned to it multiplied by the change in average response time between both periods. The contribution for a category containing structural mutations is calculated as the number of extra problem-period requests assigned to it times the average change in response time between it and its precursor category. If more than one candidate precursor category exists, an average of their response times are used, weighted by structural similarity. Figure 2.4 shows the equations used for ranking.

Spectroscope will split categories containing structural mutations and response-time

$$\textit{Response time}_i = (NP_{\textit{requests}_i})\dot{(}P_{\textit{response time}_i} - NP_{\textit{response time}_i})$$

$$\textit{Structural mutation}_i = (P_{\textit{requests}_i} - NP_{\textit{requests}_i}) \sum_{j=1}^{C} P_{\textit{response time}_i} - \frac{\frac{1}{nsed(i,j)}}{\sum_{j=1}^{C} \frac{1}{nsed(i,j)}} P_{\textit{response time}_j}$$

Figure 2.4: Contribution to the overall performance change by a category containing mutations of a given type. The contribution for a category containing response-time mutations is calculated as the number of non-problem-period requests it contains times the category's change in average response time between both periods. The contribution for a category containing structural mutations is calculated as the number of extra problem-period requests it contains times the change in average response time between it and its precursor category. If multiple candidate precursor categories exist, an average weighted by structural similarity is used. For the equations above, *NP* refers to the non-problem period, *P* refers to the problem period, *nsed* refers to normalized string-edit distance, and *C* refers to the number of candidate precursor categories.

mutations into separate virtual categories and rank each separately. I also explored ranking such categories based on the combined contribution from both mutation types. Note that category ranks are not independent of each other. For example, precursor categories of a structural-mutation category may themselves contain response-time mutations, so fixing the root cause of the response-time mutation will affect the rank assigned to the structural-mutation category.

2.6.5 Visualization

When used to diagnose problems in Ursa Minor and Google services, Spectroscope showed categories containing mutations using PDF files created via Graphviz's dot program [61], which uses a ranked layout [51] to draw directed graphs. For structural-mutation categories, Spectroscope showed both the mutation's graph structure and the precursor's graph structure side-by-side, allowing diagnosticians to identify changed substructures. For response-time mutations, one graph was shown with the edges responsible for the overall response-time change highlighted in red. For both types of mutations, aggregate statistics, such as average response time, average edge latencies, contribution to performance change, and number of requests assigned to the category from both periods were overlaid on top of the output graphs. Figure 2.5 shows a mocked-up example of this visualization for two mutated categories. Spectroscope also allows diagnosticians to view categories using a train-schedule

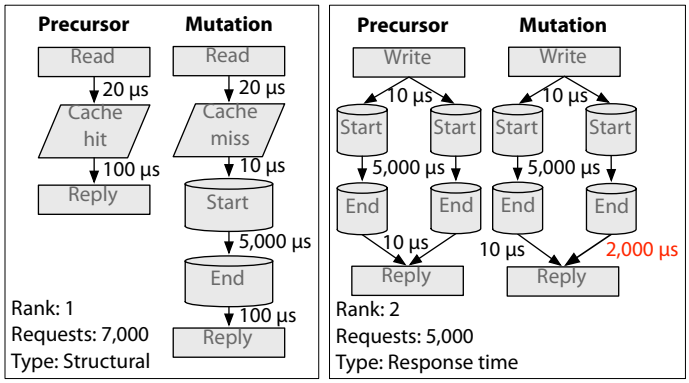


Figure 2.5: Example of how initial versions of Spectroscope visualized categories containing mutations. Structural-mutation categories and their precursor categories were shown side-by-side so that diagnosticians could manually identify where their structures started to differ. Response-time-mutation categories were shown with the interactions responsible for the overall timing change highlighted in red.

visualization (also called a swimlane visualization) [146].

For Spectroscope to be useful to diagnosticians, it must present its results as clearly as possible. My experiences using Spectroscope's initial visualizations convinced me they are not sufficient. Chapter 4 describes a user study I ran with real distributed systems developers comparing three advanced approaches for presenting Spectroscope's results.

2.6.6 Identifying low-level differences

Identifying the differences in low-level parameters between a mutation and precursor can often help developers further localize the source of the problem. For example, the root cause of a response-time mutation might be further localized by identifying that it is caused by a component that is sending more data in its RPCs than during the non-problem period.

Spectroscope allows developers to pick a mutation category and candidate precursor category for which to identify low-level differences. Given these categories, Spectroscope induces a regression tree [20] showing the low-level parameters that best separate requests in these categories. Each path from root to leaf represents an independent explanation of why the mutation occurred. Since developers may already possess some intuition about what differences are important, the process is meant to be interactive. If the developer does not like the explanations, he can select a new set by removing the root parameter from consideration and re-running the algorithm.

The regression tree is induced as follows. First, a depth-first traversal is used to extract a template describing the parts of request structures that are common between both categories, up until the first observed difference. Portions that are not common are excluded, since low-level parameters cannot be compared for them.

Second, a table in which rows represent requests and columns represent parameters is created by iterating through each of the categories' requests and extracting parameters from the parts that fall within the template. Each row is labeled as precursor or mutation depending on the category to which the corresponding request was assigned. Certain parameter values, such as the `thread ID` and `timestamp`, must always be ignored, as they are not expected to be similar across requests. Finally, the table is fed as input to the C4.5 algorithm [108], which creates the regression tree and has worst-case runtime of $O(MN^2)$, where M is the number of problem and non-problem requests in the precursor and mutations categories and N is the number of parameters extracted. To reduce runtime, only parameters from a randomly sampled subset of requests are extracted from the database,

currently a minimum of 100 and a maximum of 10% of the total number of input requests. Parameters only need to be extracted the first time the algorithm is run; subsequent iterations can modify the table directly.

2.6.7 Limitations of current algorithms & heuristics

This section describes some current limitations with the algorithms and heuristics implemented in Spectroscope.

Load observed in both periods must be similar: Though request-flow comparison expects that the workloads (i.e., percentage of request types) observed in both periods will be similar, Spectroscope's algorithms could be improved to better tolerate differences in load. For example, Spectroscope could be extended to use queuing theory to predict when a latency increase is the expected result of a load change.

Utility is limited when diagnosing problems due to contention: Such problems will result in increased variance, limiting the Kolmogorov-Smirnov's tests to identify distribution changes and, hence, Spectroscope's ability to identify response-time mutations. Chapter 3.2 further describes how Spectroscope is affected by variance.

Utility is limited for systems that generate very large graphs and those that generate many unique graph structures: Spectroscope's algorithms, especially the $O(N^2)$ algorithm it uses for calculating structural similarity between structural mutations and precursors, will not scale to extremely large graphs. Also, systems that generate many unique request structures may result in too many categories with too few requests in each to accurately identify mutations. Chapter 3.5 describes how these two problems occur for HDFS [134] and describes the two extensions to Spectroscope's algorithms that address them.

Requests that exhibit a large amount of parallelism will result in slow runtimes: Spectroscope must determine which request-flow graphs (i.e., directed acyclic graphs) are isomorphic during its categorization phase. Doing so is trivial for purely sequential graphs, but can require exponential time for graphs that contain fan-outs due to parallelism. Unless the cost of categorization can be reduced for them (e.g., by traversing children in lexographic order), Spectroscope will be of limited use for workloads that generate such graphs.

Chapter 3

Evaluation & case studies

This chapter evaluates Spectroscope in three ways. First, using workloads run on Ursa Minor [1] and Google's Bigtable [28], it evaluates the validity of Spectroscope's expectation that requests with the same structure should perform similarly. Second, it presents case studies of using Spectroscope to diagnose real and synthetic problems in Ursa Minor and select Google services. Not only do these case studies illustrate real experiences of using Spectroscope to diagnose real problems, they also illustrate how Spectroscope fits into diagnosticians' workflows when debugging problems. Third, it explores Spectroscope's applicability to even more distributed systems by describing extensions necessary for it's algorithms to be useful in helping diagnose problems in HDFS [134]. Section 3.1 describes Ursa Minor, the Google services used for the evaluation, and the workloads run on them. Section 3.2 evaluates the validity of Spectroscope's same structures/similar performance expectation. Section 3.3 describes the Ursa Minor case studies, and Section 3.4 describes the Google ones. Section 3.5 describes extensions needed for Spectroscope to be useful for HDFS and Section 3.6 concludes.

3.1 Overview of Ursa Minor & Google services

The Ursa Minor distributed storage service: Ursa Minor is an object-based storage service and is based on the network attached secure disk architecture (NASD) [57]. An Ursa Minor instance (called a "constellation") consists of potentially many NFS servers (which serve as proxies for unmodified NFS clients and, as such, are responsible for translating NFS requests to native Ursa Minor client ones), storage nodes (SNs), metadata servers (MDSs), and end-to-end trace servers. To access data, the NFS server (or a native Ursa Minor client) must first

send a request to the metadata server to obtain the appropriate permissions and the names of the storage nodes that contain the data. Once the NFS server knows the names of the storage nodes, it is free to access them directly. Ursa Minor was actively developed between 2004 and 2011 and contains about 230,000 lines of code. More than 20 graduate students and staff have contributed to it over its lifetime. More details about its implementation can be found in Abd-El-Malek et al. [1]. It is normally run in a dedicated-machine environment, with different components run on different machines.

Ursa Minor's tracing services are provided by a version of Stardust [141] revised to be useful for diagnosis. Stardust's architecture is similar to other metadata propagation-based tracing infrastructures [27, 29, 46, 47, 48, 110, 117, 125, 141]. The revised version outputs request-flow graphs, which are the type of graphs Spectrocope works best with. Stardust's default head-based sampling percentage is 10% and Ursa Minor itself contains about 200 trace points, 124 manually inserted as well as automatically generated ones for each send and receive RPC.

Figure 3.1 shows the 5-component Ursa Minor constellation used to evaluate Spectroscope. It consists of one NFS server, one metadata server, two storage nodes, and one end-to-end tracing server. Each component runs on its own dedicated machine in a homogeneous cluster, which consists of machines with 3.0GhZ processors and 2GB of RAM. Though Ursa Minor could be run with an arbitrary number of components in either shared machine or dedicated machine environments, the configuration listed above is the one used during its nightly regression tests, during which all of the case study problems were

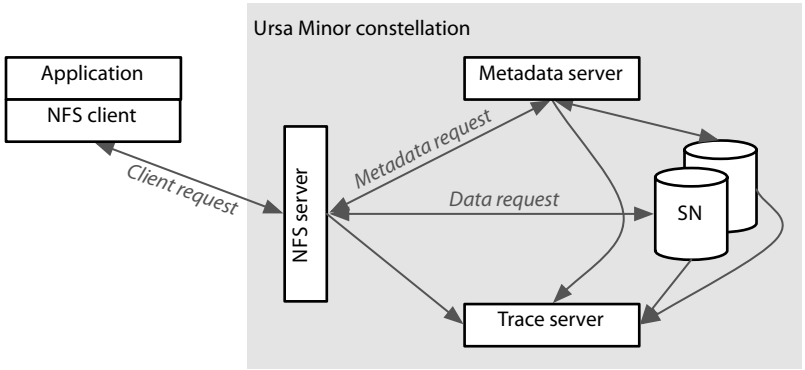


Figure 3.1: 5-component Ursa Minor constellation used during nightly regression tests. The constellation consists of one NFS server, one metadata server, two storage nodes (one reserved for data and one for metadata), and one end-to-end tracing server.

observed. The workloads used for the regression tests are listed below. The maximum size of the request-flow graphs generated for these workloads on the 5-component instance of Ursa Minor was a few hundred nodes. Table 3.1 further describes these workloads and request-flow graph sizes.

`Postmark-large`: This synthetic workload evaluates the small file performance of storage systems [77]. It utilizes 448 subdirectories, 50,000 transactions, and 200,000 files and runs for 80 minutes.

`Linux-build`: and `ursa minor-build`: These workloads consist of two phases: a copy phase, in which the source tree is tarred and copied to Ursa Minor and then untarred, and a build phase, in which the source files are compiled. `Linux-build` (of 2.6.32 kernel) runs for 26 minutes. `Ursa Minor-build` runs for 10 minutes.

`SPEC SFS 97 V3.0 (SFS97)`: This synthetic workload is the industry standard for measuring NFS server scalability and performance [126]. It applies a periodically increasing load of NFS operations to a storage system's NFS server and measures the average response time. It was configured to generate load between 50 and 350 operations/second in increments of 50 ops/second and runs for 90 minutes.

`IoZone`: This workload [100] sequentially writes, re-writes, reads, and re-reads a 5GB file in 20 minutes.

Google services: At Google, Spectroscope was applied to an internal-facing service

			Graph size (# of nodes)			
			Individual		Per-category	
Workload	**Sampled**	**Categories**	**Avg.**	**SD**	**Avg.**	**SD**
`Postmark`	131,113	716	66	65	190	81
`Linux-build`	145,167	315	12	40	160	500
`Ursa Minor-build`	16,073	63	9	20	96	100
`SFS 97`	210,669	1,602	30	51	206	200
`IoZone`	134,509	6	6	6	61	82

Table 3.1: Requests sampled and average request-flow graph sizes for Ursa Minor workloads. 10% of observed requests were sampled when obtaining traces for these workloads, yielding a few hundred thousand samples for each. The category counts and graph sizes shown represent results for one run of the corresponding workload. Multiple identical runs of the same workload will all vary slightly in category counts and graph sizes; non-problem and problem runs can differ significantly in them. The difference between the average request-flow graph size and the average graph size of each category occurs because most graphs are small, but a few are very large.

(used by Google employees to maintain Google's infrastructure), an external-facing service (representing a major Google product), and Bigtable [28]. The names of the internal- and external-facing services are anonymized. All the services run in a shared-machine environment. The workloads applied to the internal-facing service and Bigtable were generated by load tests that applied a constant rate of incoming requests. The workloads applied to the external-facing service and Bigtable were the result of real user requests.

Unlike Stardust [141], Dapper's aggregation pipeline does not generate request-flow graphs, but only call graphs that are aggressively collapsed so as to merge identical sibling and descendant nodes whenever possible. Also, many Google applications only contain enough instrumentation to show inter-component activity (e.g., RPCs between machines). As such, the maximum graph size observed for Google workloads was very small and never more than a hundred nodes. Since the workloads seen within Google are much larger than Ursa Minor's workloads, Dapper uses a head-based sampling percentage of less than 0.1%.

3.2 Do requests w/the same structure have similar costs?

To group requests that should perform similarly into the same category, Spectroscope relies on the common expectation that requests with the same structure (i.e., those that visit the same components and functions and exhibit the same amount of concurrency) should exhibit similar performance costs (e.g., response times). Categories that exhibit high intra-period variance in response times and edge latencies contain requests that do not satisfy this expectation. Such high-variance categories will increase the number of false negatives Spectroscope outputs, as they will decay the Kolmogorov-Smirnov's test to identify response-time mutations and edge latency changes. Spectroscope's ability to effectively rank mutations will also be affected. Chapter 5 further explores how high variance (i.e., unpredictability) affects Spectroscope and automation tools in general.

Though categories may exhibit high intra-period variance intentionally (for example, due to a scheduling algorithm that minimizes mean response time at the expense of variance), many do so unintentionally, as a result of unexpected resource contention or poorly written code. For example, workloads run on early versions of Ursa Minor always yielded several high-variance categories because one of its hash tables always binned items into the same bucket, resulting in unneeded contention and slowly increasing access times.

Figures 3.2 and 3.3 quantify how well categories meet the same structure/similar costs expectation. They show CDFs of the squared coefficient of variation (C^2) in response time

for *large categories* output by Spectroscope when run on graphs generated by three Ursa Minor workloads and three Bigtable [28] workloads. Each Bigtable workload represents requests sent to an instance of Bigtable over a 1-day period. Each instance runs in its own datacenter and is shared among all applications running in that datacenter. C^2 is a normalized measure of variance and is defined as $(\frac{\sigma}{\mu})^2$. Distributions with C^2 less than one exhibit low variance, whereas those with C^2 greater than one exhibit high variance. *Large categories* contain more than 10 requests; Tables 3.2 and 3.3 show that they account for only 15–45% of all categories, but contain more than 98% of all requests. Categories containing fewer requests are not included, since their smaller sample size makes the C^2 statistic unreliable for them.

For the workloads run on Ursa Minor, at least 88% of the large categories exhibit low variance. C^2 for all the categories generated by `postmark-large` is small. More than 99% of its categories exhibit low variance and the maximum C^2 value observed is 6.88. The results for `linux-build` and SFS97 are slightly more heavy-tailed. For `linux-build`, 96% of its categories exhibit low variance, and the maximum C^2 value is 394. For SFS97, 88% exhibit C^2 less than 1, and the maximum C^2 value is 50.3. Analysis of categories in the large tail of these workloads show that part of the observed variance is a result of contention for locks in the metadata server.

Unlike Ursa Minor, Bigtable is run within a shared-machine environment, which often results in increased performance variance compared to dedicated-machine environments. Also, the call graphs created for Bigtable by Dapper's aggregation pipeline show only inter-component activities (e.g., RPCs) and are heavily collapsed. As such, they cannot differentiate many unique request structures and artificially increase C^2 for categories created using them. However, even with these issues, 47–69% of categories created for the workloads run on Bigtable exhibit low variance. Increased performance isolation [63, 148], using request-flow graphs instead of call graphs, adding more instrumentation to better disambiguate different request structures, and using more sophisticated graph collapsing techniques (e.g., ones that aim to preserve one-to-one mappings between unique request structures and collapsed graphs or ones that collapse graph substructures only when their performance is similar) would all serve to considerably reduce C^2.

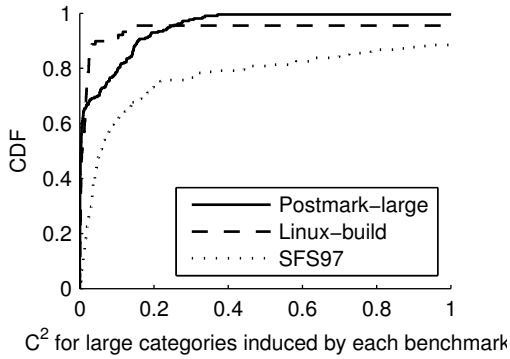


Figure 3.2: CDF of C^2 for large categories induced by three workloads run on Ursa Minor. At least 88% of the categories induced by each benchmark exhibit low variance ($C^2 < 1$). The results for `linux-build` and SFS are more heavy-tailed than `postmark-large`, partly due to extra lock contention in the metadata server.

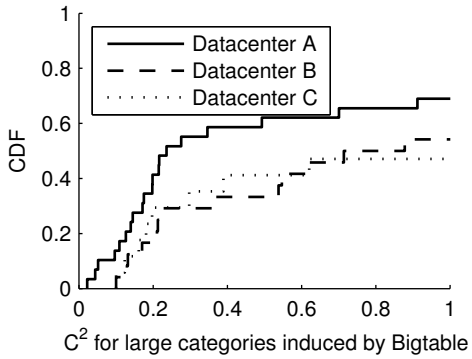


Figure 3.3: CDF of C^2 for large categories induced by Bigtable instances in three Google datacenters. Bigtable is run in a shared-machine environment, which often results in increased performance variance. Also, the call graphs generated for Bigtable by Dapper's aggregation pipeline are relatively sparse and heavily collapsed. As such, many unique request structures cannot be disambiguated and have been merged together in the observed categories. Despite these issues, 47–69% of large categories exhibit low variance.

	Workload		
	P	L	S
Categories	716	351	1,602
Large categories (%)	29.9	25.3	14.7
Requests sampled	131,113	145,167	210,669
In large categories (%)	99.2	99.7	98.9

Table 3.2: Distribution of requests in the categories induced by three Ursa Minor workloads. Here, P refers to `Postmark-large`, L to `Linux-build`, and S to SFS97. *Large categories*, which contain more than 10 requests, account for between 15–29% of all categories generated, but contain over 99% of all requests.

	Google datacenter		
	A	B	C
Categories	29	24	17
Large categories (%)	32.6	45.2	26.9
Requests sampled	7,088	5,556	2,079
In large categories (%)	97.7	98.8	93.1

Table 3.3: Distribution of requests in the categories induced by three instances of Bigtable over a 1-day period. Fewer categories and requests are observed than for Ursa Minor, because Dapper samples less than 0.1% of all requests. The distribution of requests within categories is similar to Ursa Minor.

3.3 Ursa Minor case studies

I used Spectroscope to diagnose five real performance problems and one synthetic problem in Ursa Minor. All of the real problems were observed during nightly regression tests, which perhaps represent the best use case for Spectroscope due to the well-defined problem and non-problem periods they create. Four of the real problems were previously undiagnosed, so the root cause was not known a priori. The remaining real problem had been diagnosed previously using traditional diagnosis methods and was re-injected into Ursa Minor to gauge Spectroscope's usefulness for it. The sole synthetic problem was injected into Ursa Minor to explore a wider range of problems diagnosable using Spectroscope. Three quantitative metrics were used to evaluate the quality of the ranked list of categories containing mutations that Spectroscope outputs:

The percentage of the 10 highest-ranked categories that are relevant: This is the most important metric, since diagnosticians will naturally investigate the highest-ranked categories first. Relevant categories are ones that were useful in helping diagnose the problem.

The percentage of false-positive categories: This metric evaluates the quality of the entire ranked list by identifying the percentage of all results that are *not* relevant.

Request coverage: This metric provides an alternate means for evaluating ranked list quality. It shows the percentage of all requests affected by the problem contained in the ranked list's categories.

Table 3.4 summarizes Spectroscope's performance using these metrics.[1] Unless otherwise noted, I used a default value of 50 as the threshold at which a category was deemed to contain structural mutations or associated precursors (SM_P_THRESHOLD). When necessary, I lowered it to further explore the space of possible structural mutations. For all of the case studies, I used the default sampling percentage of 10%.

3.3.1 MDS configuration change

After a particular large code check-in, performance of `postmark-large` decayed significantly, from 46tps to 28tps. To diagnose this problem, I used Spectroscope to compare request flows between two runs of `postmark-large`, one from before the check-in and one from after. The results showed many categories that contained structural mutations.

[1] The results shown in Table 3.4 have been slightly revised from those presented in my NSDI'11 paper due to a minor bug I found in Spectroscope after the paper's publication.

#	Name	Manifestation	Root cause	# of results	Quality of results		
					Top 10 rel. (%)	FPs (%)	Cov. (%)
3.3.1	MDS config.	Structural	Config.	96	100	3	77
3.3.2	RMWs	Structural	Env.	1	100	0	97
3.3.3	MDS Prefetch. 50	Structural	Internal	7	29	71	93
	MDS Prefetch. 10			14	70	50	96
3.3.4	Create behaviour	Structural	Design	11	40	64	N/A
3.3.5	100μs delay	Response time	Internal	17	0	100	0
	500μs delay			166	100	6	92
	1ms delay			178	100	7	94
3.3.6	Periodic spikes	No change	Env.	N/A	N/A	N/A	N/A

Table 3.4: Overview of the Ursa Minor case studies. This table shows information about each of six problems diagnosed using Spectroscope. For most of the case studies, quantitative metrics that evaluate the quality of Spectroscope's results are included. The problems described in Sections 3.3.1, 3.3.2, 3.3.3, 3.3.4, and 3.3.6 were real problems observed in Ursa Minor. The problem described in Section 3.3.5 was synthetic and injected into Ursa Minor to more thoroughly explore the range of problems for which Spectroscope is useful.

Comparing them to their most-likely precursor categories (as identified by Spectroscope) revealed that the storage node utilized by the metadata server had changed. Before the check-in, the metadata server wrote metadata only to its dedicated storage node. After the check-in, it issued most writes to the data storage node instead. I also used Spectroscope to identify the low-level parameter differences between a few structural-mutation categories and their corresponding precursor categories. The regression tree found differences in elements of the data distribution scheme (e.g., type of fault tolerance used).

I presented this information to the developer of the metadata server, who told me the root cause was a change in an infrequently-modified configuration file. Along with the check-in, he had mistakenly removed a few lines that pre-allocated the file used to store metadata and specify the data distribution. Without this, Ursa Minor used its default distribution scheme and sent all writes to the data storage node. The developer was surprised to learn that the default distribution scheme differed from the one he had chosen in the configuration file.

Summary: For this real problem, comparing request flows helped diagnose a performance change caused by modifications to the system configuration. Many distributed systems contain large configuration files with esoteric parameters (e.g., hadoop-site.xml)

that, if modified, can result in perplexing performance changes. Spectroscope can provide guidance in such cases by showing how modified configuration options affect system behaviour.

Quantitative analysis: For the evaluation in Table 3.4, results in the ranked list were deemed relevant if they included metadata accesses to the data storage node and their most-likely precursor category included metadata accesses to the metadata storage node.

3.3.2 Read-modify-writes

This problem was observed and diagnosed before development on Spectroscope began; I re-injected it in Ursa Minor to show how Spectroscope could have helped diagnosticians easily debug it.

A few years ago, performance of `IoZone` declined from 22MB/s to 9MB/s after upgrading the Linux kernel from 2.4.22 to 2.6.16.11. I originally discovered the root cause by manually examining Stardust traces. I found that the new kernel's NFS client was no longer honouring the NFS server's preferred READ and WRITE I/O sizes, which were set to 16KB. The smaller I/O sizes used by the new kernel forced the NFS server to perform many *read-modify-write*s (RMWs), which severely affected performance. To remedy this issue, support for smaller I/O sizes was added to the NFS server and counters were added to track the frequency of RMWs.

To show how comparing request flows and identifying low-level parameter differences could have helped developers quickly identify the root cause, I used Spectroscope to compare request flows between a run of `IoZone` in which the Linux client's I/O size was set to 16KB and another during which the Linux client's I/O size was set to 4KB. All of the categories output by Spectroscope in its ranked list were structural-mutation categories that contained RMWs.

I next used Spectroscope to identify the low-level parameter differences between the highest-ranked result and its most-likely precursor category. The resulting regression tree showed that the `count` parameter perfectly separated precursors and mutations. Specifically, requests with `count` parameter values less than or equal to 4KB were classified as precursors.

Summary: Diagnosis of this problem demonstrates how comparing request flows can help diagnosticians identify performance problems that arise due to a workload change. It also showcases the utility of highlighting relevant low-level parameter differences.

Quantitative analysis: For Table 3.4, results in the ranked list were deemed relevant if they contained RMWs and their most-likely precursor category did not.

3.3.3 MDS prefetching

A few years ago, several Ursa Minor developers and I tried to add *server-driven metadata prefetching [67]* to Ursa Minor. This feature was supposed to improve performance by prefetching metadata to clients on every metadata server access, in hopes of minimizing the total number of accesses necessary. However, when implemented, this feature actually reduced overall performance. The developers spent a few weeks (off and on) trying to understand the reason for this unexpected result but eventually moved on to other projects without an answer.

To diagnose this problem, I compared two runs of `linux-build`, one with prefetching disabled and another with it enabled. I chose `linux-build`, because it is more likely to see performance improvements due to prefetching than the other workloads.

When I ran Spectroscope with `SM_P_THRESHOLD` set to 50, several categories were identified as containing mutations. The two highest-ranked results immediately piqued my interest, as they contained WRITEs that exhibited an abnormally large number of lock acquire/release accesses within the metadata server. All of the remaining results contained response-time mutations from regressions in the metadata prefetching code path, which had not been properly maintained. To further explore the space of structural mutations, I decreased `SM_P_THRESHOLD` to 10 and re-ran Spectroscope. This time, many more results were identified; most of the highest-ranked ones now exhibited an abnormally high number of lock accesses and differed only in the exact number.

Analysis revealed that the additional lock/unlock calls reflected extra work performed by requests that accessed the metadata server to prefetch metadata to clients. To verify this as the root cause, I added instrumentation around the prefetching function to confirm that it was causing the database accesses. Altogether, this information provided me with the intuition necessary to determine why server-driven metadata prefetching decreased performance: the extra time spent in the DB calls by metadata server accesses outweighed the time savings generated by the increase in client cache hits.

Summary: This problem demonstrates how comparing request flows can help developers account for unexpected performance loss when adding new features. In this case, the problem was due to unanticipated contention several layers of abstraction below the feature

addition. Note that diagnosis with Spectroscope is interactive—for example, in this case, I had to iteratively modify SM_P_THRESHOLD to gain additional insight.

Quantitative analysis: For Table 3.4, results in the ranked list were deemed relevant if they contained at least 30 LOCK_ACQUIRE → LOCK_RELEASE edges. Results for the output when SM_P_THRESHOLD was set to 10 and 50 are reported. In both cases, response-time mutations caused by decay of the prefetching code path are conservatively considered false positives, since these regressions were not the focus of this diagnosis effort.

3.3.4 Create behaviour

Performance graphs of postmark-large showed that the response times of CREATEs were increasing significantly during its execution. To diagnose this performance degradation, I used Spectroscope to compare request flows between the first 1,000 CREATEs issued and the last 1,000. Due to the small number of requests compared, I set SM_P_THRESHOLD to 10.

Spectroscope's results showed categories that contained both structural and response-time mutations, with the highest-ranked one containing the former. The response-time mutations were the expected result of data structures in the NFS server and metadata server whose performance decreased linearly with load. Analysis of the structural mutations, however, revealed two architectural issues, which accounted for the degradation.

First, to serve a CREATE, the metadata server executed a tight inter-component loop with a storage node. Each iteration of the loop required a few milliseconds, greatly affecting response times. Second, categories containing structural mutations executed this loop more times than their precursor categories. This inter-component loop can be seen easily if the categories are zoomed out to show only component traversals and plotted in a train schedule, as in Figure 3.4.

Conversations with the metadata server's developer led me to the root cause: recursive B-Tree page splits needed to insert the new item's metadata. To ameliorate this problem, the developer increased the page size and changed the scheme used to pick the created item's key.

Summary: This problem demonstrates how request-flow comparison can be used to diagnose performance degradations, in this case due to a long-lived design problem. Though simple counters could have shown that CREATEs were very expensive, they would not have shown that the root cause was excessive metadata server/storage node interaction.

Quantitative analysis: For Table 3.4, results in the ranked list were deemed relevant if

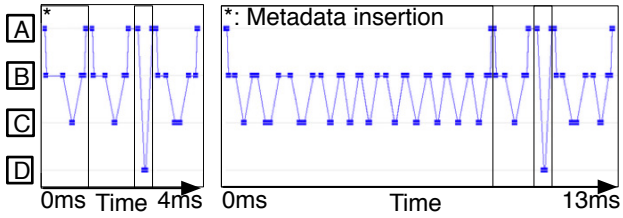


Figure 3.4: Visualization of create behaviour. Two train-schedule visualizations are shown, the first one a fast early create during `postmark-large` and the other a slower create issued later in the benchmark. Messages are exchanged between the NFS server (A), metadata server (B), metadata storage node (C), and data storage node (D). The first phase of the create procedure is metadata insertion, which is shown to be responsible for the majority of the delay.

they contained structural mutations and showed more interactions between the NFS server and metadata server than their most-likely precursor category. Response-time mutations that showed expected performance differences due to load are considered false positives. Coverage is not reported as it is not clear how to define problematic CREATEs.

3.3.5 Slowdown due to code changes

I injected this synthetic problem into Ursa Minor to show how request-flow comparison can be used to diagnose slowdowns due to feature additions or regressions and to assess Spectroscope's sensitivity to changes in response time.

For this case study, I used Spectroscope to compare request flows between two runs of `SFS97`. Problem period runs included a spin loop injected into the storage nodes' WRITE code path. Any WRITE request that accessed a storage node incurred this extra delay, which manifested in edges of the form $\star \rightarrow$ `STORAGE_NODE_RPC_REPLY`. Normally, these edges exhibit a latency of 100μs.

Table 3.4 shows results from injecting 100μs, 500μs, and 1ms spin loops. Results were deemed relevant if they contained response-time mutations and correctly identified the affected edges as those responsible. For the latter two cases, Spectroscope was able to identify the resulting response-time mutations and localize them to the affected edges. Of the categories identified, only 6–7% are false positives and 100% of the 10 highest-ranked ones are relevant. The coverage is 92% and 94%.

Variance in response times and the edge latencies in which the delay manifests prevent Spectroscope from properly identifying the affected categories for the 100μs case. It identifies 11 categories that contain requests that traverse the affected edges multiple times

as containing response-time mutations, but is unable to assign those edges as the ones responsible for the slowdown.

3.3.6 Periodic spikes

`Ursa minor-build`, which is run as part of the nightly regression test suite, periodically shows a spike in the time required for its copy phase to complete. For example, from one particular night to another, copy time increased from 111 seconds to 150 seconds, an increase of 35%. Ursa Minor's developers initially suspected that the problem was due to an external process that periodically ran on the same machines as Ursa Minor's components. To verify this assumption, I compared request flows between a run in which the spike was observed and another in which it was not.

Surprisingly, Spectroscope's output contained only one result: GETATTRs, which were issued more frequently during the problem period, but which had not increased in average response time. I ruled this result out as the cause of the problem, as NFS's cache coherence policy suggests that an increase in the frequency of GETATTRs is the result of a performance change, not its cause. I probed the issue further by reducing `SM_P_THRESHOLD` to see if the problem was due to requests that had changed only a small amount in frequency, but greatly in response time, but did not find any such cases. Finally, to rule out the improbable case that the problem was caused by an increase in variance of response times that did not affect the mean, I compared distributions of intra-category variance between two periods using the Kolmogorov-Smirnov test; the resulting p-value was 0.72, so the null hypothesis was not rejected. These observations convinced me that the problem was not due to Ursa Minor or processes running on its machines.

I next suspected the client machine as the cause of the problem and verified this to be the case by plotting a timeline of request arrivals and response times as seen by the NFS server (see Figure 3.5). The visualization shows that during the problem period, response times stay constant but the arrival rate of requests decreases. The other Ursa Minor developers and I now suspect the problem to be backup activity initiated from the facilities department (i.e., outside of our system).

Summary: This problem demonstrates how comparing request flows can help diagnose problems that are not caused by internal changes. Informing developers that nothing within the distributed system has changed frees them to focus their efforts on external factors.

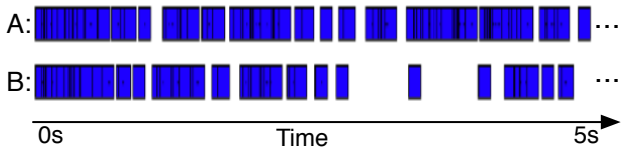


Figure 3.5: Timeline of inter-arrival times of requests at the NFS Server. A 5s sample of requests, where each rectangle represents the process time of a request, reveals long periods of inactivity due to lack of requests from the client during spiked copy times (B) compared to periods of normal activity (A).

3.4 Google case studies

I used the Google version of Spectroscope to diagnose two real problems observed in two services. Unlike for the Ursa Minor case studies, my experiences diagnosing problems at Google are purely qualitative.

3.4.1 Inter-cluster performance

A team responsible for an internal-facing service at Google observed that load tests run on their software in two different datacenters exhibited significantly different performance. However, they believed that performance should be similar because the machines configurations they were using in both datacenters were almost the same. They wanted to know if the performance difference was an artifact of their software or was due to some other problem.

I used Spectroscope to compare request flows between the two load test instances. The results showed many categories that contained response-time mutations; many were caused by latency changes not only within the service itself, but also within RPCs and within several dependencies, such as the shared Bigtable instance running in the lower-performing cluster. This led me to hypothesize that the performance difference was due to a pervasive problem in the slower load test's datacenter and that the team's software was not at fault. Google engineers confirmed this hypothesis by determining that the Bigtable instance in the slower load test's datacenter was not working properly. This experience is a further example of how comparing request flows can help developers rule out the distributed system (in this case, a specific Google service) as the cause of the problem.

3.4.2 Performance change in a large service

To help identify performance problems, Google keeps per-day records of average request latencies for major services. I used Spectroscope to compare two day-long periods for one such external-facing service, which exhibited a significant performance deviation, but only a small difference in load, between the periods compared. Though many interesting mutations were identified, I was unable to identify the root cause due to my limited knowledge of the service, highlighting the importance of domain knowledge in interpreting Spectroscope's results.

3.5 Extending Spectroscope to HDFS

To understand how the algorithms and heuristics used by Spectroscope would have to change for different distributed systems, I also tried applying Spectroscope to HDFS [134]. The traces obtained for HDFS from X-Trace [47] are much more detailed than those obtained for GFS [55] from Dapper's aggregation pipeline. As such, they capture more of the unique semantics of HDFS/GFS-like filesystems. HDFS's write semantics, in particular, cause two problems for Spectroscope's algorithms: very large request-flow graphs and an extremely large number of categories. Section 3.5.1 describes HDFS, its write semantics, and the workloads used for this exploration. Section 3.5.2 describes the large graph problem and how their size can be reduced by collapsing them. Section 3.5.3 describes the category explosion problem and presents ways Spectroscope could be extended to handle it.

3.5.1 HDFS & workloads applied

HDFS is the filesystem used by the open source implementation of Map Reduce [40], Hadoop [8]. It is modeled after the Google File System (GFS) [55], which is itself based on the NASD [57] architecture. As such, it consists of a metadata server (called a namenode) and many storage nodes (called datanodes). HDFS is usually run in both shared and dedicated-machine environments. Unlike Ursa Minor, whose block size—i.e., the granularity at which data is read and written—is relatively small, HDFS's block size is 64MB. Larger sizes are possible via a configuration parameter. When writing a block (i.e., when servicing a NEW_BLOCK or APPEND_BLOCK request), HDFS ensures reliability by using a pipeline to replicate each 64KB portion to a configurable number of datanodes. The first datanode in the pipeline is the one that receives the write operation and the remaining ones are

randomly chosen.

To obtain request-flow graphs, I instrumented HDFS using a modified version of X-Trace [150]. I added about 60 trace points, including send/receive tracepoints for all communication between clients, namenodes, and datanodes. The test HDFS cluster I used for this exploration uses a homogeneous set of 10 machines running in a dedicated environment. All of them are configured with dual Intel 3.0GhZ processors and 2GB of RAM. Each machine runs a datanode, and one of them runs both the namenode and a backup namenode. The replication factor was set to three and the block size was 64MB. Table 3.5 describes the workloads used. A head-based sampling percentage of 10% was used.

3.5.2 Handling very large request-flow graphs

As a result of HDFS's large block size and the pipeline used to replicate data in 64KB portions, request-flow graphs of HDFS write operations are often extremely large. Table 3.6 shows request-flow graphs of write operations often contain 15,000 or more nodes. Such gargantuan graphs are almost impossible for diagnosticians to understand. Also, Spectroscope's heuristics for mapping structural mutations to precursors, which currently costs $O(N^2)$ time in number of nodes, cannot scale enough to handle them.

Figure 3.6a shows a request-flow graph for a small 320KB APPEND_BLOCK request. Each staggered column in the middle of the graph represents the flow of a single 64KB portion through the replication pipeline. Edges between columns represent dependencies between when a portion can be sent and received by datanodes (e.g., the second portion cannot be sent to the next datanode in the pipeline until it has acknowledged receipt for the first). Note that the replication pipeline's structure depends only on the amount of data written. Its structure for any write operation of the same size will be identical. Also, its structure for write operations of different sizes will differ only in the number of fan-outs present.

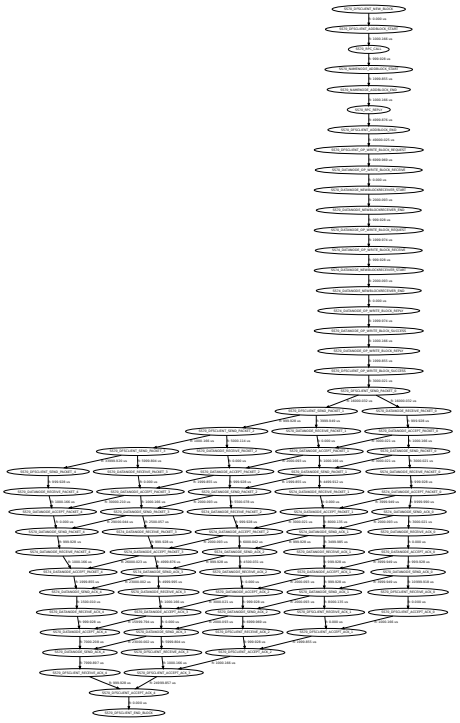
The above insight implies that HDFS write graphs can be effectively collapsed before

Benchmark	Description
`Rand sort`	Sorts 10GB/datanode of random data generated by `RandomTextWriter`
`Word count`	Counts 10GB/datanode of random data generated by `RandomTextWriter`
`Grep`	Searches for a non-existent word in 10GB/datanode of random data

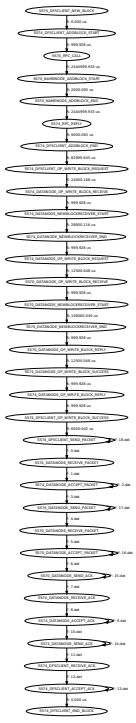
Table 3.5: The three workloads used for my HDFS explorations. These workloads are distributed with Apache Hadoop [8].

Benchmark	Graph size	Write graph size	Collapsed graph size	Collapsed write graph size
	Avg. # of nodes/standard deviation			
`Rand sort`	1390/4400	15,200/1,920	10.2/9.77	41.0/0.00
`Word count`	128/1370	15,500/0.00	7.52/3.12	41.0/0.00
`Grep`	7.42/2.42	56.0/21.2	7.39/1.75	41.0/0.00

Table 3.6: Request-flow graph sizes for the HDFS workloads. The leftmost data columns show that most request-flow graphs are small, except for write graphs, which often contain more than 15,000 nodes. Because `grep` does not write much data, its write graphs are smaller than the other two workloads. The rightmost column show that collapsing write request-flow graphs' flows through the replication pipeline into logical flows reduces their sizes from 15,000 nodes to only 41 nodes.



(a) Request-flow graph of a 320KB NEW_BLOCK request



(b) Collapsed graph of a 320KB NEW_BLOCK request

Figure 3.6: Graph of a small 320KB write operation in HDFS. The left-most graph (Figure 3.6a) shows how a NEW_BLOCK request is serviced. Each staggered column of nodes near the middle of the graph represents the flow of 64KB of data through the replication pipeline. Since this graph writes only 320KB of data, its maximum breadth is only five. A full 64MB write would generate a graph with breadth 1,024. The graph on the right (Figure 3.6b) is a collapsed version of the graph on the left. Every flow through the replication pipeline has been collapsed into a single logical flow, with filenames of files containing edge latency distributions listed on the edges.

being input into Spectroscope's categorization phase. Specifically, individual flows through the replication pipeline can be collapsed into a single logical flow. A one-to-one correspondence between the collapsed graphs and unique request structures can be maintained by adding the write size to the collapsed versions' root node labels. Figure 3.6b shows the collapsed version of the 320KB APPEND_BOCK request. Edges of the replication pipeline's logical flow show distributions of the edge latencies observed for individual 64KB portions, so no latency information is lost. Table 3.6 shows that collapsing write graphs in his manner reduces their size from upwards of 15,000 nodes to 41 nodes.

3.5.3 Handling category explosion

Since the datanodes to which replicas are written during write operations are chosen randomly, HDFS will generate an extremely large number of categories, each with too few requests to properly identify mutations. The maximum number of WRITE categories that can be generated using a HDFS cluster with N datanodes and a replication factor of R is a permutation of all the possible orderings of datanodes in the replication pipeline: ${}_NP_R$. In practice, the expected number of write categories generated by a workload will depend on the number of write requests issued. A workload that issues k such requests will generate approximately $C - C(1 - \frac{1}{C})^k$ categories, where C = ${}_NP_R$. Figure 3.7 shows a graph of the number of write categories expected as a function of write requests issued for various HDFS cluster sizes.

Experiments conducted on the test HDFS cluster (N=10, R=3) bear out this category explosion problem. The results are shown in Table 3.7. `Rand sort` is write heavy; of the 259 categories it generates, 132 are write categories that contain an average of 1.05 requests—to few for identifying those that contain mutations. Since they are less write heavy, `word count` and `grep` generate few write categories, but each of these categories also contain too few requests for identifying mutations.

To reduce the number of categories, Spectroscope must be extended to group similar, but not identical, requests into the same category while keeping intra-period response-time variance low. The latter constraint is needed because high intra-period variance within categories will reduce Spectroscope's ability to both rank mutations and identify response-time mutations. I previously explored using unsupervised clustering algorithms to group similar, but not identical requests into the same category, but found them unsatisfactory [117, 118]. The remainder of this section discusses two alternate categorization policies.

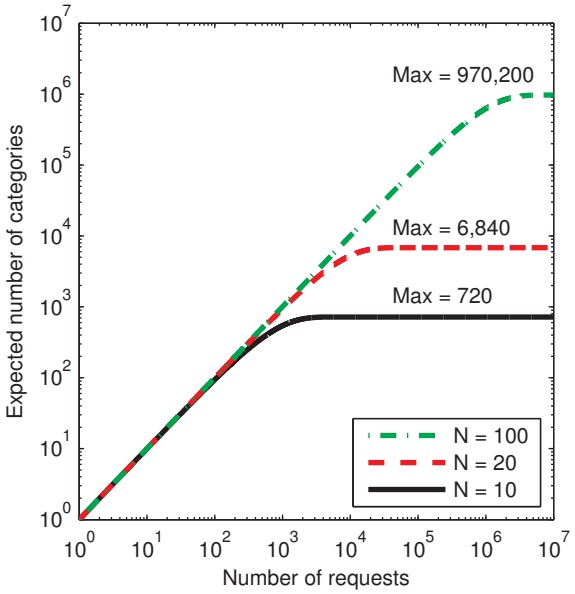


Figure 3.7: Expected number of categories that will be generated for different-sized HDFS clusters. The formula $C - C\dot{(}1 - \frac{1}{C})^k$ describes the expected number of write categories. C is the maximum possible number of write categories that can be generated given the machine set configuration, and k is the number of write requests observed. In general, $C =_N P_R$, where N is the number of datanodes and R is the replication factor. For the graphs shown in this figure, N was varied, while R was fixed at three. They show that the expected number of categories grows with number of requests at an almost imperceptibly decreasing rate, until a saturation point equal to the maximum number of categories is reached.

Categorizing by omitting machine names from node labels: Creating categories by using request-flow graphs with machine names omitted from their node labels would greatly reduce the number of categories. This simple policy may be adequate for HDFS clusters comprised solely of homogeneous machines running in a dedicated environment. For example, for the test HDFS cluster, omitting machine names from the graphs generated by `rand sort` reduces the number of write categories generated from 132 to 16. Almost all write requests are assigned to a single category, which exhibits a low C^2 value of 0.45.

Categorizing by clustering similar-performing HDFS nodes: Despite the simplicity of the above policy, it is insufficient for HDFS clusters running in heterogeneous environments or shared-machine environments, for which different HDFS nodes will often exhibit different performance characteristics. Problematic machines may also increase performance variance. This policy accounts for such environments by clustering datanodes into equivalence classes of similar-performing ones and using equivalence class names in-

Benchmark	# of requests	# of categories	Avg. # of requests	# of write requests	# of write categories	Avg. # of requests
`Rand sort`	1,530	259	5.90	139	132 (126)	1.05
`Wordcount`	1,025	116	8.84	8	7 (8)	1.14
`Grep`	1,041	116	8.97	2	2 (2)	1.00

Table 3.7: Category sizes and average requests per category for the workloads when run on the test HDFS cluster. HDFS's write semantics, which specifies that data be replicated to three randomly chosen datanodes, generates a large number of write categories. These categories contain too few requests to accurately identify response-time or structural mutations. For example, `rand sort`, the most write intensive of the benchmarks above generates 132 write categories with an average of 1.05 requests in each. Numbers listed in parentheses indicate the expected number of write categories for the given benchmark.

stead of machine names in individual request-flow graphs. The specific clustering algorithm used for this approach will depend on the amount of performance variance that is expected. For clusters that exhibit only a small amount of variance, perhaps due to a small number of problematic machines in a homogeneous dedicated-machine environment, simple approaches, such as peer comparison [75, 76] may be sufficient. For clusters that exhibit more variance, such as those running within Amazon's EC2, more advanced techniques may be necessary. One promising technique is *semi-supervised* learning, in which developer feedback is incorporated in choosing what category to assign a request.

3.6 Summary & future work

By showing how real problems were diagnosed using Spectroscope in both Ursa Minor and select Google services, this chapter demonstrates the utility of request-flow comparison and the efficacy of the algorithms and heuristics Spectroscope uses to implement it. By exploring how Spectroscope could be extended to work with HDFS, this chapter also demonstrates how graph collapsing, applied carefully, can be effective in reducing the graph sizes input to Spectroscope so as to reduce its runtime overhead.

The experiences described in this chapter also reveal promising avenues for future work. Most the Google case studies show that even with Spectroscope's automation, domain knowledge is still an important factor when diagnosing problems. However, for distributed services built upon many other distributed services, each owned by a different set of developers, no single person may possess the required knowledge. To help, future work should

focus on incorporating more domain knowledge into diagnosis tools, such as Spectroscope For example, machine learning techniques could be used to learn the semantic meaning of request-flow graphs and such meanings could be displayed along with Spectroscope's output. My HDFS explorations also reveal the need for future work to mitigate category explosion in systems that can generate many unique request-flow graphs.

Chapter 4

Advanced visualizations for Spectroscope

My experiences diagnosing performance problems using Spectroscope's [117] initial Graphviz-based [61] visualizations convinced me that they were inadequate. For example, the Graphviz-based visualizations required diagnosticians to manually and painstakingly compare structural mutations and corresponding precursors to identify differences. For response-time mutations and general timing changes, dot's ranked layout could not be easily used to draw edges proportional to observed latencies, complicating analyses of where time is spent. Unfortunately, existing literature does not provide guidance on how to improve Spectroscope's visualizations. Though much systems research has focused on techniques for automated problem localization [14, 15, 29, 72, 76, 88, 98, 103, 117, 122, 144], apart from a few select instances [86, 90], little research has focused on how best to present their results. Likewise, in the visualization community, little research has focused on identifying visualizations that are useful for distributed systems developers and diagnosticians.

To address the need for more research on visualizations for distributed systems diagnosis tasks, this chapter describes a 26-person user study that compares three promising approaches for helping developers compare graphs of precursor categories (called "before graphs" in this chapter) and graphs of mutation categories (called "after graphs" in this chapter). The user study comprises 13 professionals (i.e., developers of Ursa Minor and software engineers from Google) and 13 graduate students taking distributed systems classes. I did not expect a single approach to be best for every user study task, so my goal with this study was to identify which approaches work best for different usage modes and for diagnosing different types of problems.

The approaches compared in this chapter were chosen based on my intuition and the

recommendations of those developers who helped me diagnose problems in Ursa Minor [1] and Google services. The side-by-side approach is nearly a "juxtaposition," which presents independent layouts of both before and after graphs. Diff is an "explicit encoding," which highlights the differences between the two graphs. Animation is closest to a "superposition" design that guides attention to changes that "blink." All of the approaches automatically identify correspondences between matching nodes in both graphs so as to focus diagnosticians on relevant structural difference—i.e., those without correspondences.

Despite the large body of work on comparing graphs [9, 10, 44], I found no existing implementations of side-by-side, diff, and animation suitable for request-flow comparison's domain-specific needs. For example, since correspondences between nodes of before-and-after graphs are not known a priori, they must be identified algorithmically. Therefore, I built my own interfaces. For this study, categories containing both structural mutations and response-time mutations were not split into multiple virtual categories. Rather, the resulting edge latency and structural changes are displayed together in the same graph.

Overall, the user study results show that side-by-side is the best approach for helping diagnosticians obtain an overall understanding of an observed performance change. Animation is best for helping diagnose problems that are caused by structural mutations alone (e.g., by a change in the amount of concurrent activity or by a slower thread of activity replacing a faster thread). Diff is best for helping diagnose problems caused by response-time mutations alone.

The rest of this chapter is organized as follows. Section 4.1 describes related work. Section 4.2 describes the interfaces. Section 4.3 describes the user study methodology and Section 4.4 presents results. Based on my experiences with this study, Section 4.5 describes lessons learned and opportunities for future work. Finally, Section 4.6 concludes.

4.1 Related work

Recent research has explored a number of approaches, including some akin to side-by-side, diff, and animation, to help users identify differences in graphs, but no single one has emerged as the clear winner [9, 10, 113]. The choice depends on the domain, the types of graphs being compared, and the differences of interest, thus motivating the need for the study presented in this chapter.

Archambault et al. [9] suggest that animation may be more effective for helping users understand the evolution of nodes and edges in a graph, whereas small multiples (akin to

the side-by-side approach) may be more useful for tasks that require reading node or edge labels. In contrast, Farrugia et al. [44] find that static approaches outperform animation for identifying how connections evolve in social networks. Both evolution-related and label-related comparisons are necessary to understand differences in request-flow graphs. Robertson et al. [113] compare animation's effectiveness to that of small multiples and one other static approach for identifying trends in the evolution of Gapminder Trendalyzer [53] plots (3-dimensional data plotted on 2-D axes). They find that animation is more effective and engaging for presenting trends, while static approaches are more effective for helping users identify them. For unweighted, undirected graphs, Archambault et al. [10] find that a "difference map" (akin to the diff view) is significantly preferred, but is not more useful. Melville et al. develop a set of general graph-comparison questions and find that for small graphs represented as adjacency matrices, a superimposed (akin to diff) view is superior to a juxtaposed (side-by-side) view [93].

In addition to user studies comparing different approaches, many tools have been built to identify graph differences. Many use domain-specific algorithms or are built to analyze specific graph structures. For example, TreeJuxtaposer [97] uses domain knowledge to identify node correspondences between trees that show evolutionary relationships among different species. TreeVersity [60] uses a diff-based technique to identify differences in node attributes and structure for trees with unweighted edges and known correspondences. G-PARE [121] presents overlays to compare predictions made by machine-learning algorithms on graph-based data. Visual Pivot [114] helps identify relationships between two trees by using a layout that co-locates a selected common node. Donatien [64] uses a layering model to allow interactive matching and comparison of graphs of document collections (i.e., results from two search engines for the same query). Beck and Diehl [17] use a matrix representation to compare software architectures based on code dependencies.

In contrast, in this study, I attempt to identify good graph comparison techniques for helping developers identify performance-affecting differences in distributed system activities. These are intuitively represented as directed, acyclic, weighted graphs, often with fan-ins and fan-outs, and for which node correspondences are not known. These graphs may differ in structure and edge weight. I also believe the intended audience—those familiar with distributed systems development—will exhibit unique preferences distinct from the general population.

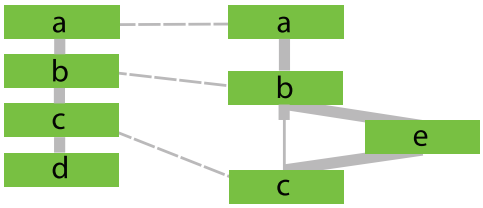
In the systems community, there has been relatively little research conducted on visual methods for diagnosis. Indeed, a recent survey of important directions for log analysis

concludes that because humans will remain in the diagnosis loop for the foreseeable feature, visualization research is an important next step [102]. One project in this vein is NetClinic, which considers root-cause diagnosis of network faults [86]. The authors find that visualization in conjunction with automated analysis [72] is helpful for diagnosis. As in this study, the tool uses automated processes to direct users' attention, and the authors observe that automation failures inhibit users' understanding. In another system targeted at network diagnosis, Mansmann et al. observe that automated tools alone are limited in utility without effective presentation of results [90]. Campbell et al. [25] conduct an in-situ study of Hadoop [8] users to identify inefficiencies in their manual diagnosis workflows. They find that many inefficiencies occur because users must often switch between multiple monitoring tools (e.g., Ganglia [92] and the Hadoop job management interface) and because important data is presented poorly, thus giving them the insight necessary to design an improved diagnosis interface. Along these lines, Tan et al. [131] describe simple visualizations for helping Hadoop users diagnose problems.

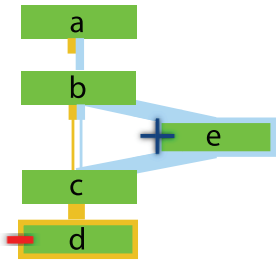
4.2 Interface design

Figure 4.1 shows the side-by-side, diff, and animation interfaces used for the user study. Their implementations were guided by an initial pilot study that helped eliminate obvious deficiencies. The interfaces are implemented in JavaScript, and use modified libraries from the JavaScript InfoVis Toolkit [18]. This section further describes them.

Side-by-side: The side-by-side interface (illustrated with a simplified diagram at right and in Figure 4.1(a,d)) computes independent layered layouts for the before and after graphs and displays them beside each other. Nodes in the before graph are linked to corresponding nodes in the after graph by dashed lines. This interface is analogous to a parallel coordinates visualization [69], with coordinates given by the locations of the nodes in the before and after graphs. Using this interface, latency changes can be identified by examining the relative slope of adjacent dashed lines: parallel lines indicate no change in latency, while increasing skew is indicative of longer response time. Structural changes can be identified by the presence of nodes in the before or after graph with no corresponding node in the other graph.

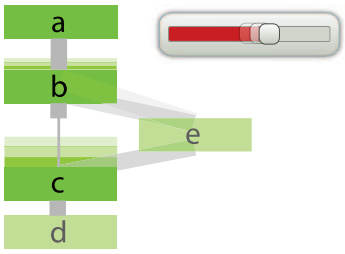


Diff: The diff interface (shown at right and in Figure 4.1(b,e)) shows a single static image in an explicit encoding of the differences between the before and after graphs, which are associated with the colors orange and blue respectively. The layout contains all nodes from both the before and after graphs. Nodes that exist only in the before graph are outlined in orange and annotated with a minus sign; those that exist only in the after graph are outlined in blue and annotated with a plus sign. This approach is akin to the output of a contextual diff tool [87] emphasizing insertions and deletions.



I use the same orange and blue scheme to show latency changes, with edges that exist in only one graph shown in the appropriate color. Edges existing in both graphs produce a per-edge latency diff: orange and blue lines are inset together with different lengths. The ratio of the lengths is computed from the ratio of the edge latencies in before and after graphs, and the next node is attached at the end of the longer line.

Animation: The animation interface (at right and in Figure 4.1(c,f)) switches automatically between the before and after graphs. To provide a smooth transition, it interpolates the positions of nodes between the two graphs. Nodes that exist in only one graph appear only on the appropriate terminal of the animation. They become steadily more transparent as the animation advances, and vanish completely by the other terminal. Independent layouts are calculated for each graph, but non-corresponding nodes are not allowed to occupy the same position. Users can start and stop the animation, as well as manually switch between terminal or intermediate points, via the provided slider.



4.2.1 Correspondence determination

All of the interfaces described above require determining *correspondences* between the before and after graphs, which are not known a priori. They must determine which nodes in the before graph map to which matching nodes in the after graph, and by extension which nodes in each graph have no match in the other. This problem is not feasible using graph structure and node names alone, because many different nodes can be labeled with the same name (e.g., due to software loops). Fortunately, the converse is guaranteed to be true—if a node in the before graph matches a node in the after graph, their node names will

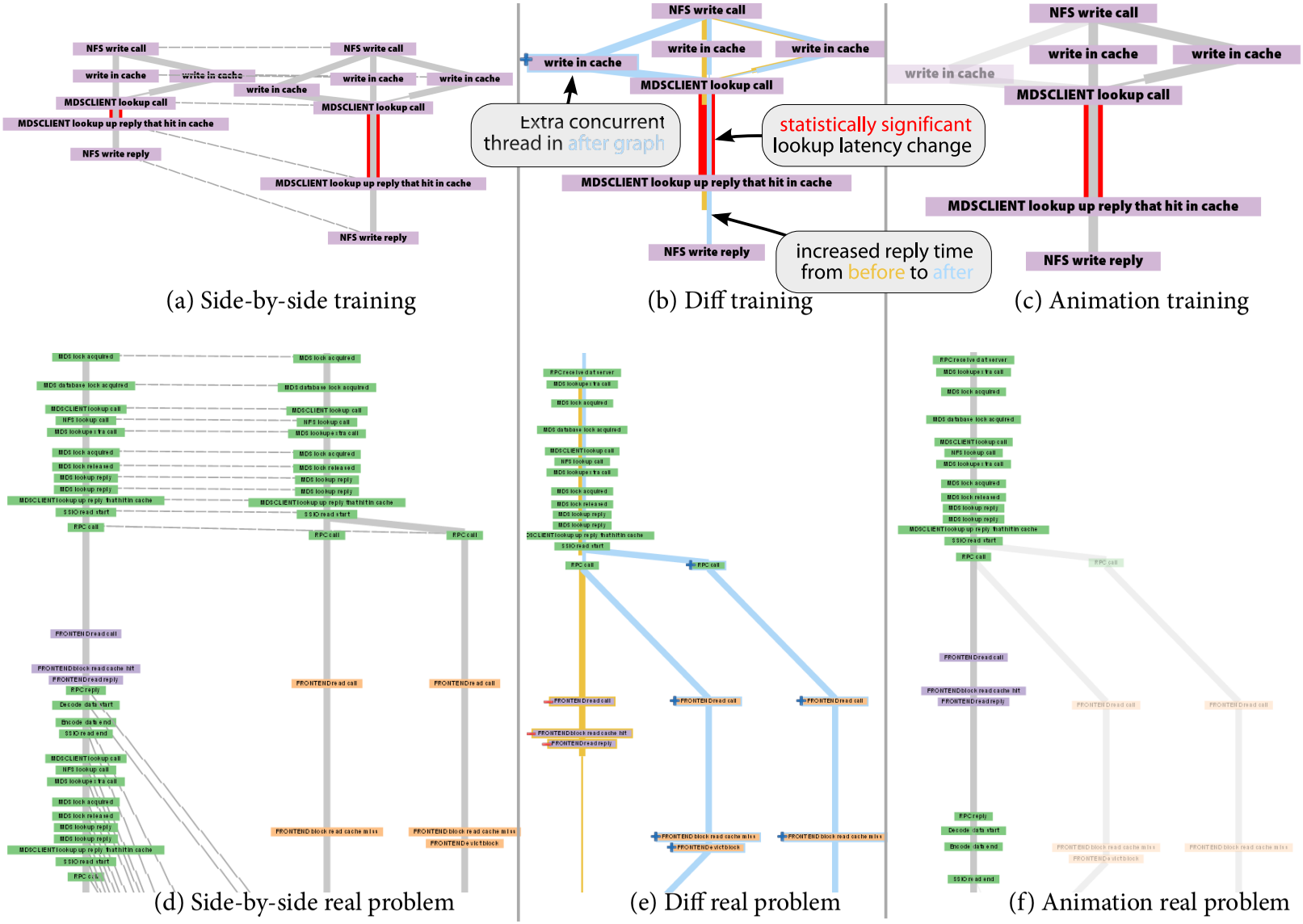


Figure 4.1: Three visualization interfaces. This diagram illustrates the three interfaces to visualizing differences in request-flow graphs used in this study. Figures a, b, and c show the interfaces applied to a mocked-up problem that was used to train participants (slightly modified for clarity on paper). Figures d, e, and f show the interfaces applied to a portion of one of the real-world problems that was presented to participants.

be the same. The interfaces exploit this property to obtain approximate correspondences.

The approximation technique runs in $O(N^2)$ time in the number of nodes. First, it uses a lexically-ordered depth-first traversal to transform both graphs into strings. Next, it keeps track of the insertions, deletions, and mutations made by a string-edit distance transformation of one string into another. Finally, it maps these edits back onto the appropriate interface. Items that were not inserted, deleted, or removed are ones that correspond in both graphs. Despite the limitations of this approach, I have found it to work well in practice. Gao et al. survey a variety of related algorithms [52]; because the problem is hard (in the formal sense), these algorithms are limited in applicability, approximate, or both. Though this approximation technique was implemented independently of Spectroscope's heuristics for identifying corresponding mutations and precursors, both could be combined to reduce overall runtime.

4.2.2 Common features

All three of the interfaces incorporate some common features, tailored specifically for request-flow graphs. All graphs are drawn with a layered layout based on the technique by Sugiyama et al [128]; layouts that modify this underlying approach enjoy widespread use [42].

To navigate the interface, users can pan the graph view by clicking and dragging or by using a vertical scroll bar. In large graphs, this allows for movement in the neighborhood of the current view or rapid traversal across the entire graph. By using the wheel on a mouse, users can zoom in and out, up to a limit. The interfaces employ rubber-banding for both the traversal and zoom features to prevent the graphs from moving off the screen or becoming smaller than the viewing window.

For all of the interfaces, edge lengths are drawn using a sigmoid-based scaling function that allows both large and small edge latencies to be visible on the same graph. Statistically significant edge latency changes are highlighted with a bold red outline. To distinguish wait time from actual latency for threads involved in join operations, the interfaces use thinner lines for the former and thicker lines for the latter (see the "write in cache" to "MDSCLIENT lookup call" edge in Figures 4.1(a-c) for an example).

Each interface also has an annotation mechanism that allows users to add marks and comments to a graph comparison. These annotations are saved as an additional layer on the interface and can be restored for later examination.

4.2.3 Interface Example

To better illustrate how these interfaces show differences, the example of diff shown in Figure 4.1(b) is annotated with the three key differences it is meant to reveal. First, the after graph contains an extra thread of concurrent activity (outlined in blue and marked with a plus sign). Second, there is a statistically significant change in metadata lookup latency (highlighted in red). Third, there is a large latency change between the lookup of metadata and the request's reply that is not identified as statistically significant (perhaps because of high variance). These observations localize the problem to those system components involved in the changes and thus provide starting points for diagnosticians' diagnosis efforts.

4.3 User study overview & methodology

I compared the three approaches via a between-subjects user study, in which I asked participants to complete five assignments using the interfaces. Each assignment asked participants to find key performance-affecting differences for a before/after request-flow graph pair obtained from Ursa Minor [1]. Three of the five assignments used graphs derived from real problems (see Chapter 3.3) observed in the system.

4.3.1 Participants

The interfaces' target users are developers and diagnosticians of the distributed system being diagnosed. As the performance problems used for the user study come from Ursa Minor, I recruited the seven Ursa Minor developers to whom I had access as expert participants. In addition, I recruited six professional distributed-system developers from Google. This chapter refers to the Ursa Minor and Google developers collectively as *professionals*.

Many of the professional participants are intimately familiar with more traditional diagnosis techniques, perhaps biasing their responses to the user-study questions somewhat. To obtain a wider perspective, I also recruited additional participants by advertising in undergraduate and graduate classes on distributed systems and posting fliers on and around Carnegie Mellon University's campus. Potential participants were required to demonstrate (via a pre-screening questionnaire) knowledge of key undergraduate-level distributed systems concepts. Of the 33 volunteers who completed the questionnaire, 29 were deemed eligible; I selected the first 13 to respond as participants. Because all of the selected partici-

pants were graduate students in computer science, electrical and computer engineering, or information networking, this chapter refers to them as *student* participants.

During the user study, each participant was assigned, round-robin, to evaluate one of the three approaches. Table 4.1 lists the participants, their demographic information, and the interface they were assigned. I paid each participant $20 for the approximately 1.5-hour study.

4.3.2 Creating before/after graphs

Each assignment required participants to identify salient differences in a before/after graph pair. To limit the length of the study, I modified the real-problem graph pairs slightly to remove a few differences that were repeated many times. One before/after pair was obtained from Spectroscope's output for the "slowdown due to code changes" case study, which involved artificially injecting a 1ms delay into Ursa Minor to gauge Spectroscope's efficacy in helping diagnose such changes (see Chapter 3.3.5). To further explore the range of differences that could be observed, one assignment used a synthetic before/after graph pair.

ID	Gender	Age	Interface
PS01	M	26	S
PS02	M	33	S
PS03	M	38	S
PS04	M	44	S
PS05	M	30	S
PD06	M	37	D
PD07	M	44	D
PD08	M	37	D
PD09	M	28	D
PA10	M	33	A
PA11	M	26	A
PA12	M	40	A
PA13	M	34	A

(a) Professionals

ID	Gender	Age	Interface
SS01	F	23	S
SS02	M	21	S
SS03	M	28	S
SS04	M	29	S
SD05	M	35	D
SD06	M	22	D
SD07	M	23	D
SD08	M	23	D
SD09	M	25	D
SA10	F	26	A
SA11	M	23	A
SA12	M	22	A
SA13	M	23	A

(b) Students

Table 4.1: Participant demographics. Our 26 participants included 13 professional distributed systems developers and 13 graduate students familiar with distributed systems. The ID encodes the participant group (P=professional, S=student) and the assigned interface (S=side-by-side, D=diff, A=animation). Average ages were 35 (professionals) and 25 (students).

It was created by modifying a real request-flow graph observed in Ursa Minor. Table 4.2 describes the various assignments and their properties.

To make the before/after graphs easier for participants to understand, I changed node labels, which describe events observed during request processing, to more human-readable versions. For example, I changed the label "e10__t3__NFS_CACHE_READ_HIT" to "Read that hit in the NFS server's cache." The original labels were written by Ursa Minor developers and only have meaning to them. Finally, I omitted numbers indicating edge lengths from the graphs to ensure participants used visual properties of the interfaces to find important differences.

4.3.3 User study procedure

The study consisted of four parts: training, guided questions, emulation of real diagnoses, and interface comparison. Participants were encouraged to think aloud throughout the study.

Training

In the training phase, participants were shown an Ursa Minor architecture diagram (similar to the one in Figure 3.1). They were only required to understand that the system consists of five components that can communicate over the network. I also provided a sample request-flow graph and described the meaning of nodes and edges. Finally, I trained each participant on her assigned interface by showing her a sample/before after graph (identical to those shown in Figures 4.1(a-c)) and guiding her through tasks she would have to complete in latter parts of the study. Participants were given ample time to ask questions and were told I would be unable to answer further questions after the training portion.

Finding differences via guided questions

In this phase of the study, I guided participants through the process of identifying differences, asking them to complete five focused tasks for each of three assignments. Rows 1–3 of Table 4.2 describe the graphs used for this part of the study.

Task 1: *Find any edges with statistically significant latency changes.* This task required participants to find all of the graph edges highlighted in red (i.e., those automatically identified by the Spectroscope as having statistically significant changes in latency distribution).

Task 2: *Find any other edges with latency changes worth investigating.* Spectroscope will

Phase	Assignment and type	Ursa Minor problem	Differences	Before/after graph sizes (nodes)
G	1/Injected	Slowdown: code changes (Chapter 3.3.5)	4 statistically sig. 5 other edge latency	122/122
	2/Real	Read-modify-writes (Chapter 3.3.2)	1 structural	3/16
	3/Synth.		4 statistically sig. 2 other edge latency 3 structural	94/128
E	4/Real	MDS configuration change (Chapter 3.3.1)	4 structural	52/77
	5/Real	MDS prefectching (Chapter 3.3.3)	2 structural	82/226

Table 4.2: Before/after graph-pair assignments. Assignments 1–3 were used in the guided questions phase (G); 4 and 5 were used to emulate real diagnoses (E). Three of the five assignments were the output of request-flow comparison for real problems seen in Ursa Minor. Assignments were ordered so as to introduce different types of differences gradually.

not identify all edges worth investigating. For example, edges with large changes in average latency that also exhibit high variance will not be identified. This task required participants to find edges with notable latency changes not highlighted in red.

Task 3: *Find any groups of structural changes*. Participants were asked to identify added or deleted nodes. To reduce effort, I asked them to identify these changes in contiguous groups, rather than noting each changed node individually.

Task 4: *Describe in a sentence or two what the changes you identified in the previous tasks represent*. This task examines whether the interface enables participants to quickly develop an intuition about the problem in question. For example, many of the edge latency changes presented in assignment 1 indicate a slowdown in network communication between machines. Identifying these themes is a crucial step toward understanding the root cause of the problem.

Task 5: *Of the changes you identified in the previous tasks, identify which one most impacts request response time*. The difference that most affects response time is likely the one that should be investigated first when attempting to find the root cause. This task evaluates whether the interface allows participants to quickly identify this key change.

Emulating real diagnoses

In the next phase, participants completed two additional assignments. These assignments were intended to emulate how the interfaces might be used in the wild, when diagnosing a new problem for the first time. For each assignment, the participant was asked to complete tasks 4 and 5 only (as described above). I selected these two tasks because they most closely align with the questions a developer would ask when diagnosing an unknown problem.

After this part of the study, participants were asked to agree or disagree with two statements using a five-point Likert scale: "I am confident my answers are correct" and "This interface was useful for solving these problems." I also asked them to comment on which features of the interface they liked or disliked, and to suggest improvements.

Interface comparison

Finally, to get a more direct sense of what aspects of each approach were useful, I showed participants an alternate interface. To avoid fatiguing participants and training effects, I did not ask them to complete the assignments and tasks again; instead I asked them to briefly consider (using assignments 1 and 3 as examples) whether the tasks would be easier or harder to complete with the second interface, and to explain which features of both approaches they liked or disliked. Because the pilot study suggested animation was most difficult to use, I focused this part of the study on comparing side-by-side and diff.

4.3.4 Scoring criteria

I evaluated participants' responses by comparing them to an "answer key" that I created. Tasks 1–3, which asked for multiple answers, were scored using precision/recall. *Precision* measures the fraction of a participant's answers that were also in the key. *Recall* measures the fraction of all answers in the key identified by the participant. Note that is possible to have high precision and low recall—for example, by identifying only one change out of ten possible ones. For task 3, participants who marked any part of a correct group were given credit.

Tasks 4 and 5 were graded as correct or incorrect. For both, I accepted multiple possible answers. For example, for task 4 ("identify what changes represent"), I accepted an answer as correct if it was close to one of several possibilities, corresponding to different levels of background knowledge. In one assignment, several students identified the changes as representing extra cache misses in the after graph, which I accepted. Some participants

with more experience explicitly identified that the after graph showed a read-modify write, a well-known bane of distributed storage system performance.

I also captured completion times for the five quantitative tasks. For completion times as well as precision/recall, I used the Kruskal-Wallis test to establish differences among all three interfaces, then pairwise Wilcoxon Rank Sum tests (chosen a priori) to separately compare the animation interface to each of side-by-side and diff. I recorded and analyzed participants' comments from each phase as a means to better understand the strengths and weaknesses of each approach.

4.3.5 Limitations

My user study methodology has several limitations. Most importantly, it is difficult to fully evaluate visualization approaches for helping developers diagnose problems without asking them to go through the entire process of debugging a real, complex problem. However, such problems are often unwieldy and can take hours or days to diagnose. As a compromise, I designed the user study tasks to test whether the interfaces enable participants to understand the gist of the problem and identify starting points for diagnosis.

Deficiencies in the interface implementations may skew participants' notions of which approaches work best for various scenarios. I explicitly identify such cases in the evaluation and suggest ways for improving the interfaces so as to avoid them in the future.

I stopped recruiting participants for the study when their qualitative comments converged, leading me to believe I had enough information to identify the useful aspects of each interface. However, the study's small sample size may limit the generalizability of our quantitative results.

Many of the participants were not familiar with statistical significance and, as such, were confused by the wording of some of the tasks (especially tasks 1 and 2). I discuss this in more detail in the Future Work section.

The participants skewed young and male. To some extent this reflects the population of distributed-systems developers and students, but it may limit the generalizability of the results somewhat.

4.4 User study results

No single approach was best for all participants and types of graph differences. For example, side-by-side was preferred by novices, and diff was preferred by advanced users and experts.

Similarly, where side-by-side and diff proved most useful for most types of graph differences, animation proved better than side-by-side and diff for two very common types. When one of the participants (PD06) was asked to pick his preferred interface, he said, "If I had to choose between one and the other without being able to flip, I would be sad." When asked to contrast side-by-side with diff, SS01 said, "This is more clear, but also more confusing." Section 4.4.1 compares the approaches based on participants' quantitative performance on the user study tasks. Sections 4.4.2 to 4.4.4 describe my observations and participants' comments about the various interfaces and, based on this data, distill the approaches best suited for specific graph difference types and usage modes.

4.4.1 Quantitative results

Figure 4.2 shows completion times for each of the three interfaces. Results for individual tasks, aggregated over all assignments, are shown (note that assignments, as shown in Table 4.2, may contain one or multiple types of differences). Participants who used animation took longer to complete all tasks compared to those who used side-by-side or diff, corroborating the results of several previous studies [9, 44, 113]. Median completion times for side-by-side and diff are similar for most tasks. The observed differences between animation and the other interfaces are statistically significant for task 1 ("identify statistically significant changes")[1] and task 4 ("what changes represent").[2] The observed trends are similar when students and professionals are considered separately, except that the differences between animation and the other interfaces are less pronounced for the latter.

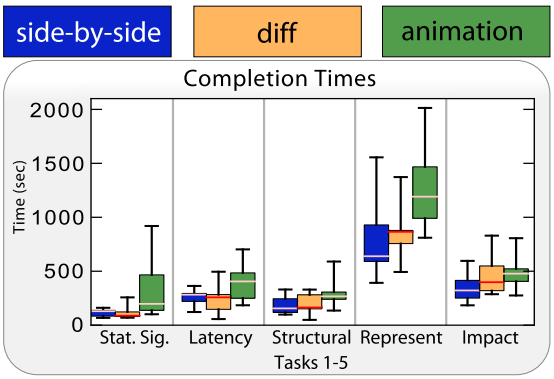
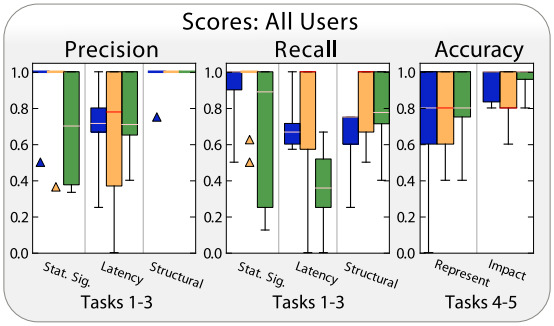


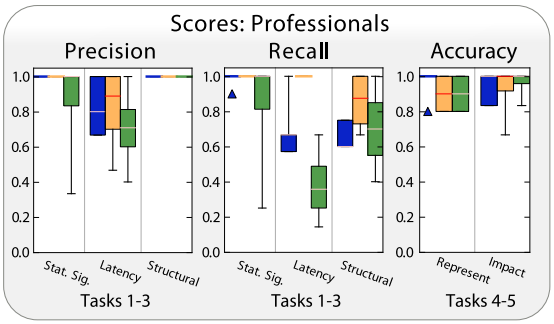
Figure 4.2: Completion times for all participants. The boxplots show completion times for individual tasks, aggregated across all assignments.

Figure 4.3a shows the precision, recall, and accuracy results for each of the three interfaces. The results are not statistically significant, but do contain artifacts worth describing. Overall, both side-by-side and diff fared well, and their median scores for most tasks are

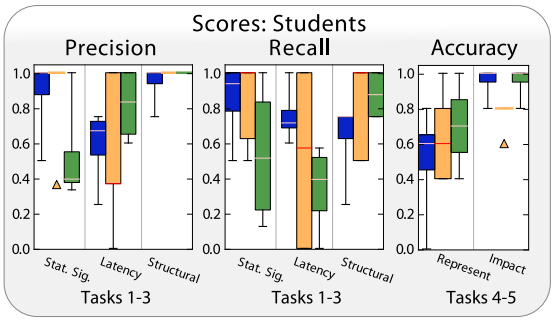
[1] p-value=0.03 (side-by-side vs. anim), p-value=0.02 (diff vs. anim)
[2] p-value=0.003 (side-by-side vs. anim), p-value=0.03 (diff vs. anim)



(a) Precision, recall, and accuracy scores for all participants



(b) Precision, recall, and accuracy scores for professionals



(c) Precision, recall, and accuracy scores for students

Figure 4.3: Precision/recall scores. The boxplots show precision, recall, and accuracy scores for individual tasks, aggregated across all assignments.

similar for precision, recall, and accuracy. Notable exceptions include recall for task 2 ("find other latency changes") and recall for task 3 ("identify structural changes"), for which diff performed better. Overall, both diff and animation exhibit much higher variation in scores than side-by-side. Though animation's median scores are better than or comparable to the other interfaces for tasks 3, 4, and 5, its scores are worse for precision for task 1 and recall for task 2.

Figures 4.3b and 4.3c show the results broken down by participant type. No single interface yielded consistently higher median scores for either group. Though professionals performed equally well with diff and side-by-side for many tasks, their scores with diff are higher for tasks 2 and 3 and higher with side-by-side for task 4. Students' median scores were higher with side-by-side for task 2 and task 5 and higher for recall with diff for task 1 and task 3. Also, students' diff scores exhibit significantly more variation than side-by-side, perhaps because not all of them were familiar with text-based diff tools, which are often used by professionals for source code-revision control. For professionals, animation's median scores are almost never higher than side-by-side. Students had an easier time with animation. For

them, animation's median scores are higher than diff and side-by-side for task 2 (precision), task 4, and task 5. Animation's median score is higher than side-by-side for task 3 (recall).

Figure 4.4 shows likert-scale responses to the questions "I am confident my answers are correct" and "This interface was useful for answering these questions." Diff and side-by-side were tied in the number of participants that strongly agreed or agreed that they were confident in their answers (5 of 9, or 56%). However, where one side-by-side user strongly agreed, no diff users did so. Only one animation user (of eight; 12.5%) was confident in his answers, so it is curious that animation was selected as the most useful interface. I postulate this is because participants found animation more engaging and interactive than the other interfaces, an effect also noted by other studies [44, 113].

4.4.2 Side-by-side

Participants liked the side-by-side interface because it was the most straightforward of the three interfaces. It showed the true response times (i.e., overall latencies) of both graphs, enabling participants to quickly get a sense of how much performance had changed. Correspondence lines clearly showed matching nodes in each graph. Also, this interface allowed independent analyses of both graphs, which the professional participants said was important. Comparing diff to side-by-side, PD08 said "it's [side-by-side] a lot easier to tell what the overall latency is for each operation. . . . [the nodes are] all put together without any gaps in the middle." SD09 said, "With [side-by-side], I can more easily see this is

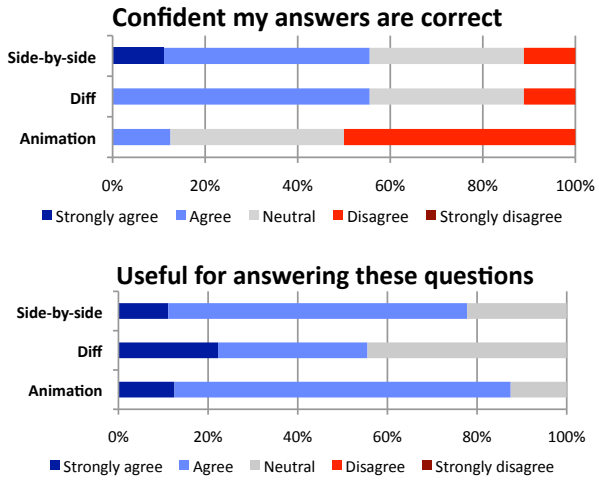


Figure 4.4: Likert responses, by condition. Each participant was asked to respond to the statements "I am confident my answers are correct" and "The interface was useful for answering these questions."

happening here before and after. Without the dashed lines, you can't see which event in the previous trace corresponds to the after trace." These sentiments were echoed by many other participants (e.g., SD06, SD07, PD07).

The side-by-side interface suffers from two key drawbacks. First, it is difficult to identify differences when before/after graphs differ significantly because corresponding ones become further apart. PS01 complained that "the points that should be lining up are getting farther and farther away, so it's getting more difficult to compare the two." PD06 complained that it was more difficult to match up large changes since the other one could be off the screen. Similar complaints were voiced by other participants (e.g., PS02, SS02, PS04). Adding the ability to pin one graph relative to another to my side-by-side implementation would limit vertical distance between differences. However, horizontal distance, which increases with the number of concurrent threads in each request, would remain.

Second, when nodes are very close to another, correspondence lines became too cluttered and difficult to use. This led to complaints from several participants (e.g., PS03, SS01, SS03, PA13). To cope, SS03 and PS05 gave up trying to identify corresponding nodes between the graphs and instead identified structural differences by determining if the number of correspondence lines on the screen matched the number of nodes visible in both the before and after graph. Modifying the side-by-side interface to draw correspondence lines only at the start of a contiguous run of corresponding nodes would help reduce clutter, but would complicate edge latency comparisons.

Based on participants' comments above and my observations, Table 4.3 shows the use cases for which I believe side-by-side is the best of the three approaches. As shown in Table 4.3, side-by-side's simple approach works best for aiding comprehension. However, due to potential for horizontal skew and clutter, it is not the best approach for identifying any type of difference.

4.4.3 Diff

Participants' comments about the diff interface were polarized. Professionals and more advanced students preferred diff's compactness, whereas others were less decisive. For example, PS03 claimed diff's compact representation made it easier for him to draw deductions. Indeed, unlike side-by-side, diff always shows differences right next to each other, making it easier to find differences when before and after graphs have diverged significantly. Also, by placing differences right next to each other, diff allows for easier identification of

smaller structural and edge latency changes. In contrast, SS04 said, "[Side-by-side] may be more helpful than [diff], because this is not so obvious, especially for structural changes."

Though participants rarely made explicit comments about finding diff difficult to use, I found that it encouraged incorrect mental models in student participants. For example, SD08 and SD09 confused structural differences within a single thread of activity with a change in the amount of concurrency. It is easy to see why participants might confuse the two, as both are represented by fan-outs in the graph.

I believe that participants' comments about diff vary greatly because its compact representation requires more knowledge about software development and distributed systems than that required by the more straightforward side-by-side interface. Additionally, many of the professionals are familiar with diff tools for text, which would help them understand my graph-based diff technique more easily.

Since it places differences close together, Table 4.3 lists diff as the best approach for showing edge latency changes. However, because it encourages poor mental models for many structural differences, it is not the best approach for showing concurrency changes and intra-thread event changes.

4.4.4 Animation

My user study participants often struggled when using the animation interface. With this interface, all differences between the two graphs appear and disappear at the same time. This combined movement confused participants when multiple types of changes were present in the same thread of activity, an effect also noticed by Ghani et al. [54]. In such cases, edge latency changes would cause existing nodes to move down and, as they were doing so, trample over nodes that were fading in or out due to structural changes. PA11 complained, "Portions of graphs where calls are not being made in the after trace are fading away while other nodes move on top of it and then above it . . . it is confusing." These sentiments were echoed by many other participants (e.g., SA11, PA12, SA10, PA13).

The combined movement of nodes and edges also prevented participants from establishing static reference points for gauging the impact of a given difference. SA10 told us: "I want to. . . pick one node and switch it between before and after. [But the same node] in before/after is in a different location completely." SA12 said he did not like the animation interface because of the lack of consistent reference points. "If I want to measure the size of an edge, if it was in the same position as before. . . then it'd be easy to see change in position

or length." Staged animation, in which individual differences are animated in one at a time, could reduce the trampling effect mentioned above and allow users to establish reference points [65]. However, significant research is needed to understand how to effectively stage animations for graphs that exhibit both structural and edge length changes. Many graph animation visualizations do not use staging and only recent work has begun to explore where such basic approaches fail [54].

Another negative aspect of animation (staged or not) is that it suggests false intermediate states. As a result, SA13 interpreted the interface's animation sequence as a timeline of changes and listed this as a feature he really liked. PA13 told me I should present a toggle instead of a slider so as to clarify that there are only two states.

Despite the above drawbacks, animation excels at showing structural differences—i.e., a change in concurrency or change in intra-thread activity—in graphs without nearby edge latency changes. Such changes do not create the trampling effect stated above. Instead, when animated, distinct portions of the graph appear and disappear, allowing users to identify changes easily. For one such assignment, both PA11 and PA12 told me the structural difference was very clear with animation. Other studies have also noted that animation's effectiveness increases with separation of changed items and simultaneous appearance/disappearance (e.g., [9, 54]).

Due to the blinking effect it creates, Table 4.3 lists animation as the best approach

	Comprehension		**Difference identification**			
	Shows overall latency change	Supports indep. analyses	Conc. change	Intra-thread event change	Edge latency change	Intra-thread mixed
Side	✓	✓				
Diff		×			✓	
Anim		✓	✓	✓		×

Table 4.3: Most useful approaches for aiding overall comprehension and helping identify the various types of graph differences contained in the user study assignments. These results are based on my qualitative observations and participants' comments. A ✓ indicates the best approach for a particular category, whereas a × indicates an approach poorly suited to a particular category. Side-by-side is best for aiding overall comprehension because of its straightforward presentation. Diff is best for showing edge latency changes because it places such changes right next to one another. Animation is best for showing structural differences due to extra concurrency and event changes within a single thread due to the blinking effect it creates. Due to their various drawbacks, no single approach is best for showing mixed differences within a single thread of activity.

for showing differences due to concurrency changes and intra-thread event changes. The potential for trampling means it is the worst of the three for showing both latency and structural differences within a single thread of activity.

4.5 Future work

Comparing these three interfaces has yielded insights about which approaches are useful for different circumstances. Performing this study also produced a number of directions for improving each of the interfaces. Here I highlight a few that participants found important and that are complex enough to warrant further research.

Addressing skew, horizontal layout, and trampling: Many users struggled with the increasing skew in the side-by-side layout, as well as the inability to quickly trace a correspondence from one graph to another (e.g., PS02, SA10, and PS05). The animation interface, which produces a trampling effect as all changes are animated together, also made it difficult to focus on individual differences. A future interface might anchor the comparison in multiple or user-selectable locations to mitigate this problem. However, there are subtleties involved in choosing and using anchor points.

One option, as requested by most participants (e.g., PA12 and PD08), is to anchor the comparison at a user-selectable location. Another is to re-center the graph as users scroll through it. However, both techniques distort the notion that time flows downward, and neither would reduce horizontal distance or clutter. Approaches that restructure the comparison to minimize the horizontal distance between corresponding nodes are an interesting opportunity.

For the animation technique, anchoring in multiple locations could be achieved by staging changes. Questions of ordering immediately arise: structural changes might be presented before, after, or between latency changes. The choice is non-obvious. For example, it is not clear whether to animate structural and latency changes together when the structural change *causes* the latency change or even how to algorithmically determine such cases.

Exploring semantic zooming: Participants found the larger graphs used for this study, which were about 200 nodes in size, unwieldy and difficult to understand. Though the straightforward zooming approach used by my interfaces allowed entire graphs to be seen, zooming out resulted in nodes, edges, and correspondence lines that were too small to be interpreted. As a result, participants needed to scroll through graphs while zoomed in, making it difficult for them to obtain a high-level understanding of the problem. Several

participants complained about this issue (e.g., SD05, SA10). Further work is needed to investigate ways to semantically zoom the graphs being compared so as to preserve interpretability. One option is to always coalesce portions of the comparison that are the same in both graphs and progressively coalesce portions that contain differences.

Exploiting annotation: Annotation was used in this study to record answers. But, it also has the potential to be a valuable tool for collaborative debugging, especially in large distributed systems with components developed by independent teams. In such systems, it is likely that no single person will have the end-to-end knowledge required to diagnose a complex problem that affects several components. For this reason, several professional participants from Google listed the interfaces' annotation mechanism as a strength of the interfaces (e.g., PA13, PS04, and PD08). PS04 said "[I] really like the way you added the annotation.…So other people who are later looking at it can get the benefit of your analysis." Supporting cooperative diagnosis work with an annotation interface, such as used that used in Rietveld [112] for code reviews, is an interesting avenue of future work.

Matching automation to users' expectations: Like many other diagnosis tools [72, 88, 98], Spectroscope uses statistical tests to automatically identify differences, since they limit expensive false positives. However, many of the user study participants did not have a strong background in statistics and so mistook "statistically significant" to mean "large changes in latency." They did not know that variance affects whether an edge latency change is deemed statistically significant. This generated confusion and accounted for lower than expected scores for some tasks. For example, some participants (usually students) failed to differentiate between task 1 and task 2, and a few students and professionals refused to mark a change as having the most impact unless it was highlighted in red (as statistically significant). Trying to account for why one particularly large latency change was not highlighted, SA10 said, "I don't know what you mean by statistically significant. Maybe it's significant to me." These concerns were echoed by almost all of the participants, and demonstrate that automation must match users' mental models and that diagnosis tools must limit both false positives *and* false negatives. Statistics and machine learning techniques can provide powerful automation tools, but to take full advantage of this power—which becomes increasingly important as distributed systems become more complex—diagnosticians must have the right expectations about how they work. Both better techniques and more advanced training may be needed to achieve this goal.

4.6 Summary

For tools that automate aspects of problem diagnosis to be useful, they must present their results in a manner developers find clear and intuitive. This chapter investigates improved visualizations for Spectroscope, one particular automated problem localization tool. It contrasts three approaches for presenting Spectroscope's results through a 26-participant user study, and find thats each approach has unique strengths for different usage modes, graph difference types, and users. Moving forward, I expect that these strengths and the research directions inspired by their weaknesses will inform the presentation of future request-flow comparison tools and other diagnosis tools.

Chapter 5

The importance of predictability

Though Spectroscope has proven useful for automatically localizing performance changes, it has been unable to reach its full potential because most distributed systems do not satisfy a key property needed for automation—predictability (i.e., low variance in key metrics). This chapter highlights this important stumbling block toward increased automation. Section 5.1 describes how high variance affects predictability and discusses potential solutions for lowering variance. Section 5.2 describes how existing automated diagnosis tools are affected by high variance. Section 5.3 describes a nomenclature for helping system builders disambiguate wanted and unwanted variance sources in their distributed systems, and Section 5.4 describes a tool for helping system builders find such variance sources. Section 5.5 discusses open questions, and Section 5.6 concludes.

5.1 How to improve distributed system predictability?

The predictability of a distributed system is affected by variance in the metrics used to make determinations about it. As variance increases, predictability decreases, and it becomes harder for automation tools to make confident, or worse, correct determinations. Though variance cannot be eliminated completely due to fundamental non-determinism (e.g., actions of a remote system and bit errors), it can be reduced, improving predictability.

To aid automation, system builders could be encouraged to always minimize variance in key metrics. This policy dovetails nicely with areas in which predictability is paramount. For example, in the early 1990s, the US Postal Service slowed down mail delivery because they decided consistency of delivery times was more important than raw speed. When asked why this tradeoff was made, the postmaster general responded: "I began to hear complaints

from mailers and customers about inconsistent first-class mail delivery. . . . We learned how important consistent, reliable delivery is to our customers" [24]. In scientific computing, inter-node variance can drastically limit performance due to frequent synchronization barriers. In real-time systems, it is more important for programs to meet each deadline than run faster on average. Google has recently identified low response-time variance as crucial to achieving high performance in warehouse-scale computing [16].

Of course, in many cases, variance is a side effect of desirable performance enhancements. Caches, a mainstay of most distributed systems, intentionally trade variance for performance. Many scheduling algorithms do the same. Also, reducing variance blindly may lead to synchronized "bad states," which may result in failures or drastic performance problems. For example, Patil et al. describe an emergent property in which load-balancing in GIGA+, a distributed directory service, leads to a large performance dropoff as compute-intensive hash bucket splits on various nodes naturally become synchronized [105].

Some variance is intrinsic to distributed systems and cannot be reduced without wholesale architectural changes. For example, identical components, such as disks from the same vendor, can differ significantly in performance due to fault-masking techniques and manufacturing effects [12, 84]. Also, it may be difficult to design complex systems to exhibit low variance because it is hard to predict their precise operating conditions [62, 94].

In practice, there is no easy answer in deciding how to address variance to aid automation. There is, however, a wrong answer—ignoring it, as is too often being done today. Instead, for the highly touted goal of automation to become a reality, system builders must treat variance as a *first-class metric*. They should strive to localize the sources of variance in their systems and make conscious decisions about which ones should be reduced. Such explicit decisions about variance properties will result in more robust systems and will vastly improve the utility of automation tools that rely on low variance to work.

5.2 Diagnosis tools & variance

Tools that automate aspects of performance diagnosis each assume a unique model of system behaviour and use deviations from it to predict diagnoses. Most do not identify the root cause directly, but rather automatically localize the source of the problem from any of the numerous components in the system to just the specific components or functions responsible. Different tools exhibit different failure modes when variance is high, depending on the underlying techniques they use.

Tools that rely on thresholds make predictions when important metrics chosen by experts exceed pre-determined values. Their failure mode is the most unpredictable, as they will return more false positives (inaccurate diagnoses) or false negatives (diagnoses not made when they should have been), depending on the value of the threshold. A low threshold will result in more false positives, whereas increasing it to accommodate the high variance will mask problems, resulting in more false negatives. My conversations with Facebook and Google engineers indicate that false positives are perhaps the worst failure mode, due to the amount of developer effort wasted [107]. However, my experiences running the visualization user study presented in Chapter 4 indicate that false negatives may be just as nefarious because they decay developers' confidence in the tool's ability.

To avoid false positives, some tools use statistical techniques to avoid predicting when the expected false positive rate exceeds a pre-set one (e.g., 5%). The cost of high variance for them is an increase in false negatives. Some statistical tools use adaptive techniques to increase their confidence before making predictions—e.g., by collecting more data samples. The cost of high variance for them is increased storage/processing cost and an increase in time required before predictions can be made.

Many tools use machine learning to automatically learn the model (e.g., metrics and values) that best predicts performance. The false positive rate and false negative rate are controlled by selecting the model that best trades generality (which usually results in more false negatives) with specificity (which results in either more false positives or false negatives).

5.2.1 How real tools are affected by variance

Real tools use a combination of the techniques described above to make predictions. This section lists four such tools and how they are affected by variance. Table 5.1 provides a summary and lists additional tools.

Magpie [15]: This tool uses an unsupervised machine learning algorithm (clustering) to identify anomalous requests in a distributed system. Requests are grouped together based on similarity in request structure, performance metrics, and resource usage. Magpie expects that most requests will fall into one of several "main" clusters of behaviour, so it identifies small ones as anomalies. A threshold is used to decide whether to place a request in the cluster deemed most similar to it or whether to create a new one. High variance in the values of the features used and use of a low threshold will yield many small clusters,

Tool	FPs / FNs	Tool	FPs / FNs
Magpie [15]	⇑ / ⇑	DARC [144]	- / ⇑
Spectroscope [117]	⇑ / ⇑	Distalyzer [98]	- / ⇑
Peer comparison. [75, 76, 104]	⇑ / ⇑	Pinpoint [29]	- / ⇑
NetMedic [72]	- / ⇑	Shen [122]	- / ⇑
Oliner et al. [103]	- / ⇑	Sherlock [14]	- / ⇑

Table 5.1: How the predictions made by automated performance diagnosis tools are affected by high variance. As a result of high variance, diagnosis tools will yield more false positives (FPs), false negatives (FNs), or both, depending on the techniques they use to make predictions. Note that the choice of failure mode attributed to each tool was made conservatively.

resulting in an increase in false positives. Increasing the threshold will result in more false negatives.

Spectroscope [117]: This tool uses a combination of statistical techniques and thresholds to identify the changes in request processing most responsible for an observed performance change. It relies on the expectation that requests with identical structures (i.e., those that visit the same components and show the same amount of parallelism) should incur similar performance costs and that the request structures observed should be similar across executions of the same workload. High variance in these metrics will increase the number of false positives and false negatives. Experiments run on Bigtable [28] in three different Google datacenters show that 47–69% of all unique call graphs observed satisfy the similar structures expectation, leaving much room for improvement [117]. Those requests that do not satisfy it suffer from a lack of enough instrumentation to tease out truly unique structures and high contention with co-located processes.

Peer comparison [75, 76, 104]: These diagnosis tools are intended to be used on tightly coupled distributed systems, such as Hadoop [8] and PVFS [137]. They rely on the expectation that every machine in a given cluster should exhibit the same behaviour. As such, they indict a machine as exhibiting a problem if its performance metrics differ significantly from others. Thresholds are used to determine the degree of difference tolerated. High variance in metric distributions between machines will result in more false positives, or false negatives, depending on the threshold chosen. Recent trends that result in increased performance variance from same-batch, same-model devices [84] negatively affect this peer-comparison expectation.

Tool described by Oliner et al. [103]: This tool identifies correlations in anomalies

across components of a distributed system. To do so, it first calculates an *anomaly score* for discrete time intervals by comparing the distribution of some signal—e.g., average latency—during the interval to the overall distribution. The strength of this calculation is dependent on low variance in the signal. High variance will yield lower scores, resulting in more false negatives. The authors themselves state this fact: "The [anomaly] signal should usually take values close to the mean of its distribution—this is an obvious consequence of its intended semantics" [103].

5.3 The three I's of variance

Variance in distributed systems is an important metric that directly affects potential for automated diagnosis. To reduce it, two complementary courses of action are necessary. During the design phase, system builders should make conscious decisions about which areas of the distributed system should be more predictable (exhibit low variance with regard to important metrics). Since the complexity of distributed systems makes it unlikely they will be able to identify all of the sources of variance during design [12, 62, 94], they must also work to identify sources of variance during development and testing. To help with the latter, this section describes a nomenclature for variance sources that can help system builders reason about them and understand for which ones' variance should be reduced.

Intentional variance sources: These are a result of a conscious tradeoff made by system builders. For example, such variance may emanate from a scheduling algorithm that lowers mean response time at the expense of variance. Alternatively, it may result from explicit anti-correlation added to a component to prevent it from entering synchronized, stuck states (e.g., Microreboots [106]). Labeling a source as intentional indicates the system builder will not try to reduce its variance.

Inadvertent variance sources: These are often the result of poorly designed or implemented code; as such, their variance should be reduced or eliminated. For example, such variance sources may include functions that exhibit extraordinarily varied response times because they contain many different control paths (spaghetti code). In a recent paper [75], Kasick et al. describe how such high variance functions were problematic for an automated diagnosis tool developed for PVFS [137]. Such variance can also emanate from unforeseen interactions between components, or may be the result of extreme contention for a software resource (e.g., locks). The latter suggests that certain contention-related performance problems can be diagnosed directly by localizing variance. In fact, while developing Spec-

troscope [117], I found that high variance was often a good predictor of such problems (see Chapter 3.2).

Intrinsic variance sources: These sources are often a result of fundamental properties of the distributed system or datacenter—for example, the hardware in use. Short of architectural changes, their variance cannot be reduced. Examples include non-flat topologies within a datacenter or disks that exhibit high variance in bandwidth between their inner and outer zones [12].

Variance from intentional and intrinsic sources may be a given, so the quality of predictions made by automation tools in these areas will suffer. However, it is important to guarantee their variance does not impact predictions made for other areas of the system. This may be the case if the data granularity used by an automation tool to make predictions is not high enough to distinguish between a high-variance source and surrounding areas. For example, problems in the software stack of a component may go unnoticed if an automation tool does not distinguish it from a high-variance disk. To avoid such scenarios, system builders should help automation tools account for high-variance sources directly—for example, by adding markers around them that are used by automation tools to increase their data granularity.

5.4 VarianceFinder

To illustrate a variance-oriented mindset, this section proposes one potential mechanism, called *VarianceFinder*, for helping system builders identify the main sources of variance in their systems during development and testing. The relatively simple design outlined here focuses on reducing variance in response times for distributed storage systems such as Ursa Minor [1], Bigtable [28], and GFS [55]. However, I believe this basic approach could be extended to include other performance metrics and systems.

VarianceFinder utilizes request-flow graphs obtained from end-to-end tracing (see Chapter 2.2) and follows a two-tiered approach. First, it shows the variance associated with aspects of the system's overall functionality that should exhibit similar performance (Section 5.4.1). Second, it allows system builders to select functionality with high variance and identifies the components, functions, or RPCs responsible, allowing them to take appropriate action (Section 5.4.2). I believe this tiered approach will allow system builders to expend effort where it is most needed.

5.4.1 Id'ing functionality & first-tier output

To identify functionality that should exhibit similar performance, VarianceFinder employs the expectation that requests with identical structures should incur similar performance costs. Like Spectroscope, VarianceFinder groups request-flow graphs that exhibit the same structure—i.e., those that represent identical activities and execute the same trace points—into categories and calculates average response times, variances, and squared coefficients of variation (C^2) for each. C^2, which is measured as $(\frac{\sigma}{\mu})^2$, is a normalized measure of variance. It captures the intuition that categories whose standard deviation is much greater than the mean are worse offenders than those whose standard deviation is less than or close to the mean. In practice, categories with $C^2 > 1$ are said to have high variance around the mean, whereas those with $C^2 < 1$ exhibit low variance around the mean.

The first-tier output from VarianceFinder consists of the list of categories ranked by C^2 value. System builders can click through highly-ranked categories to see a graph view of the request structure, allowing them to determine whether it is important. For example, a highly-ranked category that contains READ requests likely will be deemed important, whereas one that contains rare requests for the names of mounted volumes likely will not.

5.4.2 Second-tier output & resulting actions

Once the system builder has selected an important highly-ranked category, he can use VarianceFinder to localize its main sources of variance. This is done by highlighting the highest-variance edges along the critical path of the category's requests. Figure 5.1 illustrates the overall process. In some cases, an edge may exhibit high variance, because of another edge—for example, an edge spanning a queue might display high variance because the component to which it sends data also does so. To help system builders understand these dependencies, clicking on a highlighted edge will reveal other edges that have non-zero covariance with it.

Knowing the edges responsible for the high variance allows the system builder to investigate the relevant areas of the system. Variance from sources that he deems inadvertent should be reduced or eliminated. Alternatively, he might decide that variance from certain sources should not or cannot be reduced because they are intentional or intrinsic. In such cases, he should add tight instrumentation points around the source to serve as markers. Automation tools that use these markers to increase their data granularity—especially those, like Spectroscope [117], that use end-to-end traces directly [15, 29, 117, 125]—will be able to

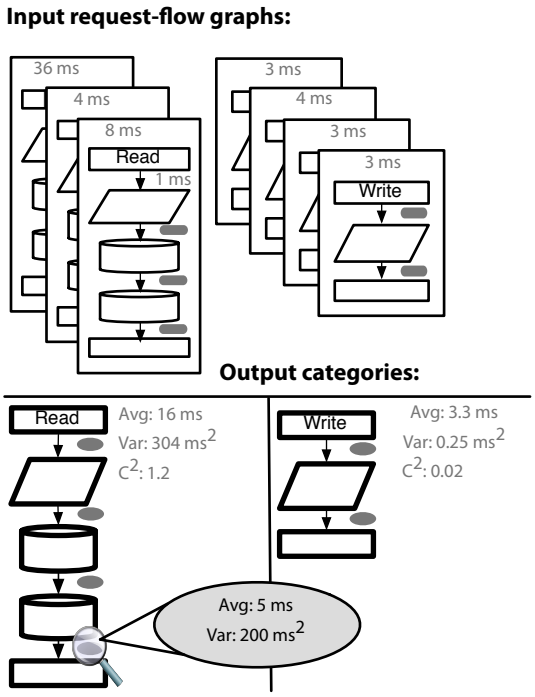


Figure 5.1: Example of how a VarianceFinder implementation might categorize requests to identify functionality with high variance. VarianceFinder assumes that requests with identical structures should incur similar costs. It groups request-flow graphs that exhibit the same structure into *categories* and calculates statistical metrics for them. Categories are ranked by the squared coefficient of variation (C^2) and high-variance edges along their critical path are automatically highlighted (as indicated by the magnifying glass).

make better predictions about areas surrounding the high-variance source.

Adding instrumentation can also help reveal previously unknown interesting behaviour. The system builder might decide that an edge exhibits high variance because it encompasses too large of an area of the system, merging many dissimilar behaviours (e.g., cache hits and cache misses). In such cases, extra instrumentation should be added to disambiguate them.

5.5 Discussion

This chapter argues that variance in key performance metrics needs to be addressed explicitly during design and implementation of distributed systems, if automated diagnosis is to become a reality. But, much additional research is needed to understand how much variance can and should be reduced, the difficulty of doing so, and the resulting reduction in management effort.

To answer the above questions, it is important that we work to identify the breakdown of intentional, inadvertent, and intrinsic variance sources in distributed systems and datacenters. To understand if the effort required to reduce variance is worthwhile, the benefits of better automation must be quantified by how real people utilize and react to automation tools, not via simulated experiments or fault injection. If this tradeoff falls strongly in favour of automation, and intrinsic variance sources are the largest contributers, architectural changes to datacenter and hardware design may be necessary. For example, system builders may need to increase the rate at which they adopt and develop strong performance isolation [63] or insulation [148] techniques. Also, hardware manufacturers, such as disk drive vendors, may need to incorporate performance variance as a first-class metric and strive to minimize it.

Similarly, if (currently) intentional sources are the largest contributors, system builders may need to re-visit key design tradeoffs. For example, they may need to consider using datacenter schedulers that emphasize predictability and low variance in job completion times [45] instead of ones that dynamically maximize resource utilization at the cost of predictability and low variance [56].

Finally, automated performance diagnosis is just one of many reasons why low variance is important in distributed systems and datacenters. For example, strong service-level agreements are difficult to support without expectations of low variance. As such, many of the arguments posed in this chapter are applicable in a much broader sense.

5.6 Conclusion

Though automation in large distributed systems is a desirable goal, it cannot be achieved when variance is high. This chapter presents a framework for understanding and reducing variance in performance metrics so as to improve the quality of automated performance diagnosis tools. I imagine that there are many other tools and design patterns for reducing variance and enhancing predictability. In the interim, those building automation tools must consider whether the underlying system is predictable enough for their tools to be effective.

Chapter 6

Related work on performance diagnosis

There has been a proliferation of research in the past several years on how to diagnose distributed systems problems. Due to the many different types of systems (e.g., distributed storage, dynamic hash tables, those for high-performance computing, those for cloud computing), available data sources (e.g., end-to-end traces, logs, performance counters), and potential problems (e.g., performance anomalies, behavioural changes, correctness issues), it is likely that there is no single "magic bullet" solution. Rather, the likely outcome of these research efforts is a toolbox of diagnosis techniques, each serving their own unique and useful purpose. This chapter samples the constituents of this toolbox for performance diagnosis.

Tools for performance diagnosis can be categorized on many axes. For example, they could be categorized according to the type of instrumentation they use (see Oliner's dissertation [101]) or the techniques they use to diagnose relevant problems (e.g., statistics, machine learning, mathematical modeling). In this related work chapter, tools are categorized according to their intended *functionality*. Viewed through this lens, most diagnosis tools are designed to help diagnosticians with a specific phase of the diagnosis workflow: problem localization, root-cause identification, or problem rectification (see Figure 1.1). Most research has focused on tools for helping diagnosticians localize the source of a new problem; they are described in Section 6.1. Some tools are designed to help identify the root cause directly and are described in Section 6.2. Tools in this category are limited to suggesting previously observed root causes, but doing so allows diagnosticians to skip the problem localization step. A few tools attempt to automatically fix an observed problem by applying simple actions, such as rebooting or re-imaging an offending machine; examples

are described in Section 6.3.

Recent research has also focused on tools for helping optimize distributed system performance when no "problem" has been explicitly identified—for example, tools that help administrators with decisions about configuration choices. Such tools are described in Section 6.4. A significant amount of research has also focused on diagnosis tools for single-process systems; examples of such tools are described in Section 6.5.

6.1 Problem-localization tools

Tools that localize performance problems can be divided into six main categories: tools that automatically identify anomalies, tools that automatically identify behavioural changes (i.e., steady-state changes), tools that automatically identify dissenting nodes in tightly coupled distributed systems (e.g., PVFS [137]), tools that aid in distributed profiling, tools that exhaustively explore potential behaviours to find potential problems, and tools that help localize problems by visualizing trends amongst important metrics. Tables 6.1, 6.2, 6.3, 6.4, 6.5, and 6.6 summarize these tools in terms of their functionality, the *comparative approach* they use to identify problems, and their building blocks (i.e., the fundamental techniques and data source they use). End-to-end traces are the most detailed and comprehensive data source, but require distributed systems to be modified to generate them. To work with unmodified systems, many tools use more commonly available data sources, such as logs. However, in practice, the potential of these tools is often limited because the data sources they use are not specifically designed to support their intended use cases. To help, some research efforts focus on mechanisms to automatically retrofit existing distributed systems with the instrumentation necessary to support specific diagnosis tasks [154].

Many tools that localize performance problems work by comparing expected distributed system behaviour to poorly-performing runtime behaviour. Expected behaviour can be expressed in a variety of ways, including via:

Current runtime behaviour: When determining whether a problem should be localized to a given distributed system component or request, expected behaviour can be expressed in terms of that of other nodes, components, or requests observed during the same execution. This comparison approach is most commonly used for anomaly detection or for detecting dissenters in tightly coupled distributed systems in which all nodes are expected to perform similarly.

Previous runtime behaviour: In systems in which all nodes are not expected to perform

similarly, behaviour from previous time periods during which performance was acceptable can be stored and used as expected behaviour. Such tools may use machine learning to learn a model from previous behaviour or may use previous behaviour directly.

Look-back window: This approach is similar to the previous one, except instead of using a distinct specific period or execution as expected behaviour, a look-back window of the previous N minutes is used. Look-back windows are most commonly used by tools that operate on time-series data built from performance counters or other key metrics.

Mathematical models: Some tools use models derived from first principles, such as queuing theory, as expected behaviour. Compared to using runtime behaviour directly, as in the approaches described above, mathematical models can more completely and correctly express desired behaviour. For example, expected behaviour derived from a previous execution may itself contain latent performance problems and is limited in scope to what was observed. Despite these advantages, many tools do not use mathematical models because they are very difficult to create and scale [132].

Expectations: Another approach involves asking diagnosticians to explicitly specify expected behaviour, for example via a custom language [110]. Manually written expectations are difficult to specify for large systems that exhibit many complex behaviours.

Diagnostician's intuition: In some cases, especially for distributed profiling and visualization, expected behaviour is not *explicitly* specified; rather, these tools rely on the diagnostician's intuition to judge where the problem lies. For example, a diagnostician using a profiling tool, such as Whodunit [27], must himself decide whether a large latency observed for a function is acceptable, whereas this decision can be made automatically by tools that use explicit models.

The rest of this section describes various localization tools.

6.1.1 Anomaly detection

Table 6.1 summarizes tools intended to help diagnosticians localize problems that manifest as anomalies—components, requests, or other behaviours that show up in the tail of some important performance metric's distribution. More generally, anomalies are rare and extremely different from what has been observed before. Mapgie [15] uses end-to-end traces and unsupervised machine learning to identify request flows that exhibit very different structures or resource utilizations compared to other requests. Xu et al. [152] attempt to replicate Magpie's functionality for systems without native end-to-end tracing support by

Tool	Function	Comparison method	Data source
Magpie [15]	Find anomalous flows	Current runtime	E-e traces
Xu et al. [152]	Find anomalous paths	”	Logs
Oliner et al. [103]	Show anomaly propagation	Look-back window	Counters
PAL [99]	”	”	”

Table 6.1: Localization tools that identify anomalies.

mining logs to identify causally-related activity. The resulting request-path graphs are not as expressive or comprehensive as the request-flow graphs created by Magpie's tracing infrastructure, limiting the types of anomalies Xu et al.'s approach can identify. For example, request paths cannot be used to identify anomalous requests that exhibit excessive or too little parallelism. Xu et al. also describe a way to compare ratios of observed system states across time windows of execution to identify anomalous windows. System states are extracted from log entries.

Performance anomalies often originate on particular components or nodes of a system, but, due to inter-component influences, can quickly propagate to other components or nodes. This makes it difficult to localize the problem to the true culprit. For example, an upstream component that sends requests slower than usual to a downstream one will cause the throughput of both to appear slower, even though the upstream component is the true culprit. Conversely, slow response times from a downstream component will quickly increase queuing times on an upstream one. Oliner et al. [103] infer such inter-component influences using time series data constructed from a chosen performance counter (e.g., running average latency). Their approach involves creating per-component anomaly signals describing how abnormal a current time series window is to past windows and then correlating time-shifted versions of these signals with each other to determine influences. PAL [99] identifies influences by identifying per-component change points in time series data and then ordering ones that occur on different components by timestamp.

Note that end-to-end traces could be used by by both tools above to lower false positives (i.e., influences identified between components that never directly or indirectly communicate with each other). However, end-to-end traces are not sufficient to identify influences by themselves because most do not capture this form of causality.

6.1.2 Behavioural-change detection

Table 6.2 shows tools intended to help diagnosticians localize problems that manifest as steady-state changes in the distribution of important performance metrics (e.g., response times or request structures). Unlike anomalies, which affect only a distribution's tail, behavioural changes are not as rare (i.e., they affect the 50^{th} or 75^{th} percentile) and are not necessarily as extreme. They are often identified and localized using statistical methods, such as hypothesis testing, thresholds on key metrics, and machine learning.

Spectroscope [117], the localization tool discussed in this dissertation, uses end-to-end traces, statistical hypothesis tests, and thresholds to help identify the most performance-affecting changes in the timing and structure of observed request flows between a period of acceptable performance and one of poor performance. Doing so allows diagnosticians to use the changed areas as starting points in their diagnosis efforts. Spectroscope also uses machine learning to further localize the problem to the low-level parameters (e.g., function parameters, configuration values or client-sent parameters) that best distinguish changed and original flows. Distalyzer [98] uses similar techniques to identify performance-affecting differences between two periods of log data collected from individual machines. Since it uses logs instead of end-to-end traces, Distalyzer is restricted to showing per-machine differences instead of that in the end-to-end workflow of servicing client requests. Portions of Pinpoint [29] uses statistical tests to determine performance regressions in request flows.

IRONModel [140], uses runtime behaviour derived from end-to-end traces to evolve mathematics models of expected behaviour built from queuing theory. As such, it differs from many of the tools described above, which use a period of runtime data as expected behaviour. Observing how the model is evolved shows how expected behaviour differs from

Tool	Function	Comparison method	Data source
Spectroscope [117]	Find perf. affecting changes in flows	Previous runtime	E-e traces
Distalyzer [98]	Find perf. affecting changes in logs	”	Logs
Pinpoint [29]	Find latency changes in flows	”	E-e traces
IronModel [140]	Find model/runtime deviations	Mathematical model	”
Pip [110]	Find expectation/runtime deviations	Expectations	”
NetMedic [72]	Find influencing components	Current runtime	Counters
Sherlock [14]	”	”	& traceroute
Giza [88]	Find problematic node & influencers	”	Counters

Table 6.2: Localization tools that identify behavioural changes.

actual behaviour and can sometimes serve to localize performance problems. Pip [110] requires developers to write expectations of expected behaviour and compares them to end-to-end traces showing runtime behaviour. Due to the effort required to manually write expectations, Pip is best used to debug small, yet complex, areas of a system's codebase. To ameliorate the effort necessary, Pip can generate expectations automatically, but these are often too restrictive and must be manually examined and refined. Though detecting behavioural differences is perhaps Pip's best application, the flexibility of its expectation language allows it to be used for many purposes, including anomaly detection and correctness debugging.

Similar to the anomaly propagation case, some tools are designed to identify what other distributed system components or events might have influenced a behavioural change observed in a problematic component. NetMedic [72] identifies the most likely path of influence using a dependency diagram specified by diagnosticians and past inter-node performance correlations. Sherlock [14] models a distributed system's components as a state vector in which each element is a tuple that indicates the probability a given component is up, down, or problematic. It uses a probabilistic dependency diagram to gauge how a particular assignment of states might influence that of a known problematic component. The assignment that yields the best match to the observed component's behaviour is identified and problematic/non-working components in this best assignment are selected as those influencing the observed component's problematic behaviour.

The dependency diagrams used by NetMedic can be more granular, but those used by Sherlock are automatically inferred via network messages; end-to-end traces could be used instead in systems with support for them. Unlike the anomaly propagation tools discussed in the previous section, NetMedic and Sherlock do not account for influences in the direction opposite normal component-wise dependencies (e.g., a downstream component affecting an upstream one).

Giza [88] uses statistical tests that exploit the hierarchical structure of IPTV networks to identify nodes with a larger than expected number of problematic events. Other event streams observed on such nodes that correlate highly with the problematic event stream are identified and an influence graph is created to identify how these streams depend on each other.

6.1.3 Dissenter detection

Table 6.3 shows tools designed to localize performance problems by identifying nodes whose current performance differs significantly than the majority. They are designed to be used in tightly-coupled distributed systems, such as PVFS [137], Lustre [136], and HDFS [134], in which every node is expected to exhibit identical (or very similar) behaviour. For example, in PVFS, every node is included in the stripe set of every file, so all reads should induce the same amount of work on each node. In HDFS, data is replicated on datanodes randomly, so all of them should exhibit similar performance over a large enough time period (assuming they use similar hardware and software).

For PVFS and Lustre, Kasick et al. [74, 76] describe a tool that localizes problems by comparing distributions of expert-selected performance counter values across nodes; problematic nodes are identified as those that exhibit performance counter distributions that differ significantly from more than half their peers. This peer comparison approach is also used by Ganesha [104] to diagnose problems in Hadoop [8] and HDFS, except with modifications to better allow inter-node performance to vary over small time scales. Both of these previous tools operate on raw counter values, so they will not work properly with nodes that use different hardware. PeerWatch [73] addresses this limitation by using statistical techniques to extract correlations between time series descriptions of counters on different nodes and performing comparisons on the correlations instead.

6.1.4 Exploring & finding problematic behaviours

Table 6.4 shows tools designed to identify and localize performance problems by exhaustively exploring possible distributed system behaviours; they are useful for identifying latent problems that have not yet been observed during regular use. MacePC [81] identifies latent performance problems in event-driven systems built using the MACE programming language [82]. To do so, it first learns latency distributions of event processing on various

Tool	Function	Comparison method	Data source
Kasick et al. [74, 76]	Find dissenting nodes	Current runtime	Counters
Gensesha [104]	”	Current runtime	Counters
PeerWatch [73]	Find dissenting VMs	Current runtime	Counters

Table 6.3: Localization tools that identify dissenters in tightly coupled systems.

Tool	Function	Comparison method	Data source
MacePC [81]	Find latent perf. bugs	Previous runtime	State-changing events
HangWiz [151]	Find soft hang bugs	Static analysis	Source code

Table 6.4: Localization tools that explore potential system behaviours to find problems.

nodes during real executions of the system. It then explores previously unseen system executions in a simulator, in which event completion times are determined from the learned distributions. Simulations that exhibit poor performance are further examined to identify the specific events responsible. HangWiz [151] uses static analysis to identify source code locations that may trigger responsiveness problems (e.g., those problems that result in the Mac OS X beach ball appearing temporarily).

6.1.5 Distributed profiling & debugging

Table 6.5 shows tools intended to help profile distributed system performance. Whodunit [27] uses end-to-end traces to generate calling context trees [5], in which each node represents a function or component and is annotated with the percentage of total CPU time spent within it. Calling context trees differ from regular call graphs in that every unique path from root to leaf is guaranteed to be a real path taken by at least one observed request. Dapper [125], ETE [66], and NetLogger [142] can also help diagnosticians profile individual requests by showing them in a Gantt chart.

6.1.6 Visualization

Several diagnosis tools help diagnosticians localize performance problems by visualizing important metrics so as to show trends amongst them; Table 6.6 shows some of these tools. Maya [22] visualizes a social graph of component dependencies, culled from analysis of a

Tool	Function	Comparison method	Data source
Whodunit [27]	Profile workload perf.	Diagnostician's intuition	E-e traces
Dapper [125]	Profile individual requests	”	”
ETE [66]	”	”	”
NetLogger [142]	”	”	”

Table 6.5: Localization tools that profile distributed system performance.

Tool	Function	Comparison method	Data source
Maya [22]	Visualize dependencies & metrics	Diagnostician's intuition	Counters
Artemis [37]	Visualize metric correlations	”	”
Otus [109]	”	”	”
Tan et al. [131]	Visualize map-reduce flows	”	Logs

Table 6.6: Localization tools that visualize important metrics.

common middleware framework. Deployed at Amazon, diagnosticians can zoom into any component to view performance metrics and understand how problems propagate across nodes. Artemis [37] and Otus [109] are log collection and analysis frameworks that allow diagnosticians to plot and compare arbitrary performance metrics collected from different nodes. For example, Otus can be used to show memory usage across all Hadoop nodes, memory used by each mapper running on a single node, or memory used by a particular job on every node. Tan et al. [131] describe a tool for Hadoop that mines Hadoop logs to extract very low granularity flows that show causality between individual mappers, reducers, and HDFS. These flows can be visualized in swimlanes to identify outliers and can be sliced in various ways to show other types of anomalous behaviour.

Though useful, tools that rely on visualization alone are likely to be less useful than those that automatically localize problems *and* use strong visualizations to present their results effectively. NetClinic [86], the visualization layer for NetMedic [72], allows diagnosticians to browse NetMedic's results and explore other alternatives when the automated localization results are incorrect. Spectroscope includes a visualization layer [116] that helps diagnosticians identify relevant differences between the problematic and corresponding non-problematic requests identified by its algorithms [117].

6.2 Root-cause identification tools

Unlike problem localization tools, which only reduce the amount of effect necessary to identify the root cause of a problem, tools in this category attempt to identify the root cause automatically. This extra power comes at the cost of generality: most root-cause identification tools can only identify the root cause if the current problem matches or is similar to a previously diagnosed one. The tools presented in this section mainly differ based on how they construct *signatures* for representing unique problems and the machine learning techniques they to detect recurrences.

SLIC [35] constructs signatures of past problems using bit vectors in which the i^{th} element indicates whether some observed performance counter's running average value was more likely to come from a period of service-level objective violation or compliance. Probabilities are learned from past behaviour using a tree-augmented Bayesian network [34]. Since the same problem may manifest as many similar, yet distinct signatures, unsupervised clustering is used to group similar signatures and a new problem is identified as a recurrence of the cluster to which it is assigned. Bodik et al. [23] improve upon SLIC's technique by using more robust data (e.g., quantiles instead of averages) and more effective machine learning techniques. Finally, instead of using performance counters, Yuan et al. [153] use signatures constructed from n-grams of system-call invocations observed during a problem occurrence. Support vector machines, each trained to detect whether a new signature is representative of a previously observed problem, are used to detect recurrences.

6.3 Problem-rectification tools

A small number of tools explore how to automatically take corrective actions to fix an observed problem. For example, Autopilot [59, 70] uses data from past automated actions to compute a survivor function describing how long a failed machine or device will last given a certain recovery action is performed. Given a desired survival time, it picks the best action and performs it. Only simple recovery actions, such as rebooting or re-imaging are considered. Netprints [2] crowdsources problem instances and network configuration data and uses both to create regression trees that identify potential fixes for a given problem. If possible, these fixes are applied automatically.

6.4 Performance-optimization tools

Many of the tools described in the previous section are meant to be used to fix an observed problem. Some tools help developers understand how to improve performance even when no overt problem has been identified. "What-if analysis" [132, 138, 139, 141] allows developers to explore the effect on performance of configuration changes (e.g., "What would the response time be if I upgraded the hard disks on one server?"). Where many of these tools use mathematical models derived from queuing theory [138, 141], WISE [132], designed to answer what-if questions about large-scale CDNs, eschews models derived from first principles for easier to specify ones derived from machine learning. Another class of tools

allows users to explore configuration changes in sandboxed environments [156] or via speculative execution that can be rolled back [13]. Note that the profiling tools, discussed in Section 6.1.5, could also be included in this category.

6.5 Single-process tools

There are also many single-process diagnosis tools that inform diagnosis techniques for distributed systems. For example, conceptually similar to Spectroscope [117], OptiScope [96] compares the code transformations made by different compilers to help developers identify important performance-affecting differences. Delta analysis [145] compares multiple failing and non-failing runs to identify differences responsible for failures. The tool described by Shen et al. [122] compares two executions of Linux workloads using statistical techniques to identify unexpected performance changes. In another paper, Shen et al. [123] describe a tool that identifies performance problems in I/O subsystems by comparing an analytical model of the subsystem to observed runtime behaviour. DARC [144] creates histograms of Linux system call latencies and automatically profiles selected peaks to identify the most dominant latency contributors.

Chapter 7

Systemizing end-to-end tracing knowledge

This dissertation focuses primarily on request-flow comparison, a diagnosis technique that uses end-to-end traces as its data source. However, in the process of developing and evaluating this technique, I had to design, engineer, and maintain two of the most well-known tracing infrastructures (i.e., Stardust [117, 141] and the version of X-Trace [47] I used for my HDFS explorations [150]). I also worked on Google's end-to-end tracing infrastructure, Dapper [125], in the context of building Spectroscope [117] and other tools as extensions to it. These experiences have given me considerable insight into end-to-end tracing.

Most notably, I have learned that end-to-end tracing is not a one-size-fits-all solution. For example, when developing Spectroscope, my collaborators and I initially thought that the original version of Stardust [141], which had been designed for resource attribution (i.e., to charge work done deep in the distributed system to the client responsible for submitting it), would also be useful for diagnosis. However, I quickly found that the traces it yielded were of little value for most diagnosis tasks, leading me to create a revised version of Stardust useful specifically for diagnosis [117]. More generally, I have found that key design axes dictate a tracing infrastructure's utility for each of the various use cases generally attributed to end-to-end tracing. Since efficiency concerns make it impractical to support all use cases, designers of a tracing infrastructure must be careful to choose options for these axes that are best suited to the infrastructure's intended purposes. Otherwise, it will fail to satisfy expectations.

Unfortunately, despite the strong interest in end-to-end tracing from both the research community [3, 15, 27, 29, 47, 48, 66, 78, 110, 111, 117, 130, 131, 141, 142, 152] and industry [36,

125, 143, 147], there exists little information to help designers make informed choices about end-to-end tracing's design axes. Since these axes are not well understand, some well-known tracing infrastructures have indeed failed to live up to expectations.

This dissertation chapter seeks to help. Based on my experiences and the previous ten years of research on end-to-end tracing, it distills the key design axes and explains trade-offs associated with the options for each. Beyond describing the degrees of freedom, it suggests specific design points for each of several key tracing use cases and identifies which previous tracing infrastructures do and do not match up. Overall, I believe this chapter represents the first attempt to bring together a coherent view of the range of end-to-end tracing implementations and provide system designers a roadmap for integrating it.

The remainder of this chapter is organized as follows. Section 7.1 discusses use cases for and the basic anatomy of end-to-end tracing. Sections 7.2 to 7.5 describe key design axes and tradeoffs for them. These axes include the sampling strategy, which causal relationships are captured, how causal relationships are tracked, and how end-to-end traces are visualized. Section 7.6 applies these insights to suggest specific design choices for each of several use cases. It also compares my suggestions to that of existing tracing infrastructures to show where prior infrastructures fall short. Section 7.7 discusses some challenges and opportunities that remain in realizing the full potential of end-to-end tracing, and Section 7.8 concludes.

7.1 Background

This section describes relevant background about end-to-end tracing. Section 7.1.1 describes its key use cases from the literature. Section 7.1.2 lists the three commonly used approaches to end-to-end tracing, and Section 7.1.3 describes the architecture of the approach advocated in this paper.

7.1.1 Use cases

Table 7.1 summarizes the key use cases of end-to-end tracing. They are described below.

Anomaly detection: This diagnosis-related use case involves detecting rare flows that incur significant latency and/or have extremely different structures than other flows and understanding why they occur. Anomalies may be related to correctness (e.g., timeouts or component failures) or performance (e.g., slow requests that fall in the 99.9th percentile of observed response times). Magpie [15] identifies both types by finding requests that

Intended use	Implementations		
Anomaly detection	Magpie [15]	Pinpoint [29]	
Diagnosing steady-state problems	Dapper [125] Pip [110]	Pinpoint [29] Stardust‡ [117]	X-Trace [48] X-Trace‡ [47]
Distributed profiling	ETE [66]	Dapper [125]	Whodunit [27]
Resource attribution	Stardust [141]	Quanto [46]	
Workload modeling	Magpie [15]	Stardust [141]	

Table 7.1: Main uses of end-to-end tracing. This table lists the key use cases for end-to-end tracing and corresponding implementations. Some implementations appear for multiple use cases. The revised versions of Stardust and X-Trace are denoted by *Stardust‡* and *X-Trace‡*.

are anomalous in request structure and resource usage. Pinpoint's [29] anomaly detection component focuses on correctness problems and so identifies requests with anomalous structures only.

Diagnosing steady-state problems (i.e., behaviourial changes): This is another diagnosis-related use case and involves identifying and debugging problems that manifest in many requests and so are not anomalies. Such problems affect the 50th or 75th percentile of some important metric, not the 99th. They are generally not correctness related, but related to performance—for example, a configuration change that modifies the workflow of a set of requests and increases their response times. Pip [110], the latest version of Stardust (Stardust‡) [117], both versions of X-Trace [47, 48], Dapper [125], and parts of Pinpoint [29] all help diagnose steady-state problems that manifest in the structure of requests or their timing.

Distributed profiling: The goal of distributed profiling is to identify slow components or functions. Since the time a function takes to execute may differ based on how it is invoked, profilers often maintain separate bins for every unique calling stack. Whodunit [27] is explicitly designed for this purpose and can be used to profile entire workloads. Dapper [125] and ETE [66] show visualizations designed to help profile individual requests.

Resource attribution: This use case is designed to answer questions of the form "Who should be charged for this piece of work executed deep in the stack of my distributed system's components?" It involves tying work done at an arbitrary component of the distributed system to the client or request that originally submitted it. The original version of Stardust [141] and Quanto [46] are designed for resource attribution; the former answers

"what-if" questions (e.g., "What would happen to the performance of workload A if I replaced the CPU on a certain distributed system component with a faster one?") and the latter tracks energy usage in distributed embedded systems. Note that resource attribution-based tracing can be especially useful for accounting and billing purposes, especially in distributed services shared by many clients, such as HDFS or Amazon's EC2 [149].

Workload modeling: End-to-end tracing can also be used to model workloads. For example, Magpie [15] clusters its traces to identify those that are representative of the entire workload. Stardust [141] can be used to create queuing models that served to answer its "what-if" analyses.

7.1.2 Approaches to end-to-end tracing

Most end-to-end tracing infrastructures use one of three approaches to identify causally-related flows: metadata propagation, schemas, or black-box inference. This paper focuses on design decisions for tracing infrastructures that use the first, as they are more scalable and produce more accurate traces than those that use the other two. However, many of my analyses are also applicable to the other approaches.

Metadata propagation: Like security, end-to-end tracing works best when it is designed as part of the distributed system. As such, many implementations are designed for use with white-box systems, for which the distributed system's components can be modified to propagate metadata (e.g., an ID) delineating causally-related flows [27, 29, 46, 47, 48, 110, 117, 125, 141]. All of them identify causality between individual functions or trace points, which resemble log messages and record the fact that a flow reached a particular point in the system. However, Dapper [125] chooses to emphasize causality between individual RPCs. To keep runtime overhead (e.g., slowdown in response time and throughput) to a minimum so that tracing can be "always on," most tracing infrastructures in this category use sampling to collect only a small number of trace points or flows.

Schema-based: A few implementations, such as ETE [66] and Magpie [15] do not propagate metadata, but rather require developers to write *temporal join-schemas* that establish causal relationships among variables exposed in individual log messages. Developers must modify the distributed system to expose appropriate variables. A temporal join defines a valid time interval during which trace-point records that store specific identical variable values should be considered causally-related. For example, in most systems, a worker thread pulling an item off of a queue defines the start of a valid time interval during which all

log messages that store the same thread ID should be considered causally related. Schema-based approaches are not compatible with sampling, since they delay determining what is causally related until after all logs are collected. Therefore, they are less scalable than metadata-propagation approaches.

Black-box inference: Several implementations try to create traces for black-box systems—i.e., systems for which source code cannot or will not be modified [3, 21, 78, 83, 111, 130, 131, 152]. Some try to infer causality by correlating variables exposed in pre-existing log statements [21, 78, 131, 152]. Others use simplifying assumptions. For example, Tak et al. [130] assume synchronous behaviour between components and, within components, that a single worker thread will be responsible for synchronously performing all work for a given request, including sending/receiving sub-requests and responding to the original client. Aguilera et al. [3] and Reynolds et al. [111] observe only send/receive events between components and use models of expected delay between causally-related events to infer causality. Though the promise of obtaining end-to-end traces without software modification is appealing, these approaches cannot properly attribute causality in the face of asynchronous behaviour (e.g., caching, event-driven systems), concurrency, aggregation, or code-specific design patterns (e.g., 2-of-3 storage encodings), all of which are common in distributed systems.

7.1.3 Anatomy of end-to-end tracing

Figure 7.1 shows the anatomy of most end-to-end tracing infrastructures that use metadata propagation. Two conceptual design choices dictate its functionality and the design of its physical components: how to decide what flows or trace points to sample and what observed causal relationships to preserve. Section 7.2 describes the tradeoffs between various sampling techniques, and Section 7.3 describes what causal relationships should be preserved for various end-to-end tracing use cases. The sampling technique used affects the types of causal relationships that can be preserved with low overhead, so care must be taken to choose compatible ones.

The software components of the tracing infrastructure work to implement the chosen sampling technique, preserve desired causal relationships, optionally store this information, and create traces. They include individual trace points, the causal-tracking mechanism, the storage component, trace construction code, and the presentation layer.

Individual trace points record locations in the codebase accessed by individual flows. They are often embedded in commonly used libraries (e.g., RPC libraries) and manu-

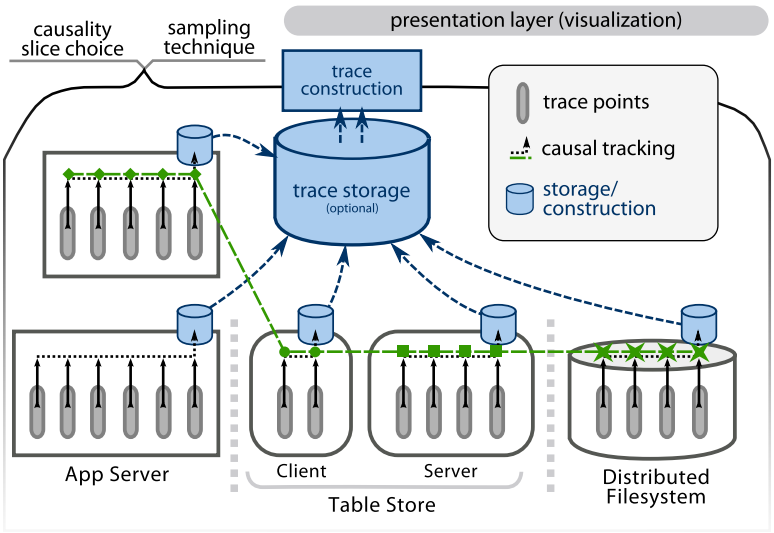


Figure 7.1: Anatomy of end-to-end tracing. The elements of a typical metadata-propagation-based tracing infrastructure are shown.

ally added by developers in areas of the distributed system's software they deem important [47, 117, 125]. Alternatively, binary re-writing or interposition techniques can be used to automatically add them at function boundaries [27, 43]. Design decisions for where to add trace points are similar to those for logging and are not discussed in detail here.

The causal-tracking mechanism propagates metadata with flows to identify causally-related activity. It is critical to end-to-end tracing and key design decisions for it are described in Section 7.4.

The storage component stores records of sampled trace points and its associated metadata. Such trace-point records may also include additional useful information, such as the current call stack. The trace construction code joins trace-point records that have related metadata to construct traces of causally-related activity. These components are optional—trace points need not be stored if the desired analysis can be performed online and trace construction is not necessary if the analyses do not need full traces. For example, for some analyses, it is sufficient to propagate important data with causally-related activity and read it at executed trace points.

Several good engineering choices, as implemented by Dapper [125], can minimize the performance impact of storing trace points. First, on individual components, sampled trace points should be logged asynchronously (i.e., off the critical path of the distributed system). For example, this can be done by copying them to a in-memory circular buffer

(or discarding them if the buffer is full) and using a separate thread to write trace points from this buffer to local disk or to a table store. A map-reduce job can then be used to construct traces. Both Stardust [141] and Dapper [125] suggest storing traces for two weeks for post-hoc analyses before discarding them.

The final aspect of an end-to-end tracing infrastructure is the presentation layer. It is is responsible for showing constructed traces to users and is important for diagnosis-related tasks. Various ways to visualize traces and tradeoffs between them are discussed in Section 7.5.

7.2 Sampling techniques

Sampling determines what trace points are collected by the tracing infrastructure. It is the most important technique used by end-to-end tracing infrastructures to limit runtime and storage overhead [27, 47, 117, 125, 141]. For example, even though Dapper writes trace points records to stable storage asynchronously (i.e., off the critical path of the distributed system), it still imposes a 1.5% throughput and 16% response time overhead when capturing all trace points executed by a web search workload [125]. When using sampling to capture just 0.01% of all trace points, the slowdown in response times is reduced to 0.20% and in throughput to 0.06% [125]. Even when trace points need not be stored because the required analyses can be performed online, sampling is useful to limit the sizes of analysis-specific data structures [27].

There are three commonly-used options with regards to deciding what trace points to sample. The first two are necessary if traces showing causally-related activity are to be constructed.

Head-based coherent sampling: In order to construct a trace of causally-related activity, all trace points executed on its behalf must be sampled. Coherent sampling satisfies this requirement by sampling all or none of the trace points executed on behalf of a flow. With *head-based* sampling, the decision to collect a flow is made at its start (e.g., when a request enters the system) and metadata is propagated along with the the flow indicating whether to collect its trace points. This method of coherent sampling is used by several implementations [47, 117, 125]. Using this method to preserve certain causality slices will result in high overheads (see Section 7.3).

Tail-based coherent sampling: This method is similar to the previous one, except that the sampling decision is made at the end of causally-related flows, instead of at their start.

Delaying the sampling decision allows for more intelligent sampling—for example, the tracing infrastructure can examine a flow's properties (e.g., response time) and choose only to collect anomalous ones. But, trace point records for every flow must be cached somewhere until the sampling decision is made for it. Since many requests can execute concurrently and because each request can execute many trace points, such temporary collection is not always feasible. Whodunit [27], the only tracing implementation that uses tail-based sampling, caches trace points by propagating them as metadata with individual flows.

Unitary sampling: With this method, the sampling decision is made at the level of individual trace points. No attempt is made at coherence (i.e., capturing all trace points associated with a given flow), so traces cannot be constructed using this approach. This method is best used when coherent sampling is unnecessary or infeasible. For example, for some use cases, the information needed for analysis can be propagated with casual flows and retrieved at individual trace points without having to having to construct traces.

In addition to deciding how to sample trace points, developers must decide how many of them to sample. Many infrastructures choose to randomly sample a small, set percentage—often between 0.01% and 10%—of trace points or causal flows [27, 47, 117, 125, 141]. However, this approach will capture only a few trace points for small workloads, limiting its use for them. Using per-workload sampling percentages can help, but requires knowing workload sizes a priori. A more robust solution, proposed by Sigelman at al. [125], is an adaptive scheme, in which the tracing infrastructure aims to always capture a set rate of trace points or causal flows (e.g., 500 trace points/second or 100 causal flows/second) and dynamically adjusts the sampling percentage to accomplish this set goal. Though promising, care must be taken to avoid biased results when the captured data is used for statistical purposes. For distributed services built on top of shared services, the adaptive sampling rate should be based on the tracing overhead the lowest-tier shared service can support (e.g., Bigtable [28]) and proportionately propagated backward to top-tier services.

7.3 Causal relationship preservation

Since preserving causality is the ultimate goal of end-to-end tracing, the ultimate tracing infrastructure would preserve all true or *necessary* causal relationships. It would preserve the workflow of servicing individual requests and background activities, read after write accesses to memory, caches, files, and registers, provenance of stored data, inter-request

causal relationships due to resource contention (e.g., for caches) or built-up state, and so on. Lamport's happens-before relationship ($\rightarrow$) [85], states that if a and b are events and $a \rightarrow b$, then a may have influenced b. Thus, b might be causally dependent on a, and capturing all such relationships allows one to create a *happens-before graph*, showing all possible channels of influence.

Unfortunately, the general happens-before graph is not sufficient for two reasons. First, it may be infeasible to capture all possible relations—even the most efficient software taint tracking mechanisms yield a 2x to 8x slowdown [79]. It may also be impossible to know all possible channels of influence [30]. Second, it is too indiscriminate: in most cases, we are interested in a stronger, 'is necessary for', or 'cannot happen without' relationship, which is a subset of the full happens-before relationship. To help alleviate the first issue, tracing infrastructures often ask developers to explicitly instrument parts of the distributed system they know to be important. To account for the second, most tracing infrastructures use knowledge about the system to remove spurious edges in the happens-before graph. For example, by assuming a memory protection model the system may exclude edges between events in different processes, or even between different events in a single-threaded event-based system. By removing selected edges from this graph, one creates *slices* of the full relation that can be more useful for specific purposes. A tracing infrastructure chooses slices that are most useful for how its output will be used and works to preserve them. This section describes slices that have proven useful for various use cases.

When choosing slices, developers must first identify one that defines the workflow of a request as it is being serviced by a distributed system. Though any request will induce a certain set of activities that must be performed eventually, latent ones need not be considered part of its flow, but rather part of the request that forces that work to be executed. This observation forms the basis for two potential *intra-request* slices—submitter and trigger preserving—that preserve different information and are useful for different use cases. Section 7.3.1 and Section 7.3.2 describe the tradeoffs involved in preserving these slices in more detail. Section 7.3.3 lists the advantages of preserving both submitter causality and trigger causality. Section 7.3.4 discusses the benefits of delineating concurrent behaviour from sequential behaviour and preserving forks and joins in individual traces. Table 7.2 shows intra-request slices most useful for the key uses of end-to-end tracing.

Developers may also want to preserve important dependencies between individual requests. Two such *inter-request* slices are discussed in Section 7.3.5.

Intended use	Slice	Preserve forks/joins/concurrency?
Anomaly detection	Trigger	Y
Diagnosing steady-state problems	”	”
Distributed profiling	Either	N
Resource attribution	Submitter	”
Workload modeling	Depends	Depends

Table 7.2: Suggested intra-flow slices to preserve for various intended uses. Since the same necessary work is simply attributed differently for both trigger and submitter preserving slices, either can be used for profiling. The causality choice for workload modeling depends on what aspects of the workload are being modeled.

7.3.1 The submitter-preserving slice

Preserving this slice means that individual end-to-end traces will show causality between the original submitter of a request and work done to process it through every component of the system. It is most useful for resource attribution and perhaps workload modeling, since these usage modes require that end-to-end traces tie the work done at a component several levels deep in the system to the client, workload, or request responsible for originally submitting it. Quanto [46], Whodunit [27], and the original version of Stardust [141] preserve this slice of causality. The two left-most diagrams in Figure 7.2 show submitter-preserving traces for two WRITE requests in a distributed storage system. Request one writes data to the system's cache and immediately replies. Sometime later, request two enters the system and must evict request one's data to place its data in the cache. To preserve submitter causality, the tracing infrastructure attributes the work done for the eviction to request one, not request two. Request two's trace only shows the latency of the eviction.

Submitter causality cannot be preserved with low overhead when head-based coherent sampling is used. To understand why, consider that preserving this causality slice means latent work must be attributed to the original submitter. So, when latent work is aggregated by another request or background activity, trace points executed by the aggregator must be captured if *any one* of the aggregated set was inserted into the system by a sampled request. In many systems, this process will resemble a funnel and will result in capturing almost all trace points deep in the system. For example, if head-based sampling is used to sample

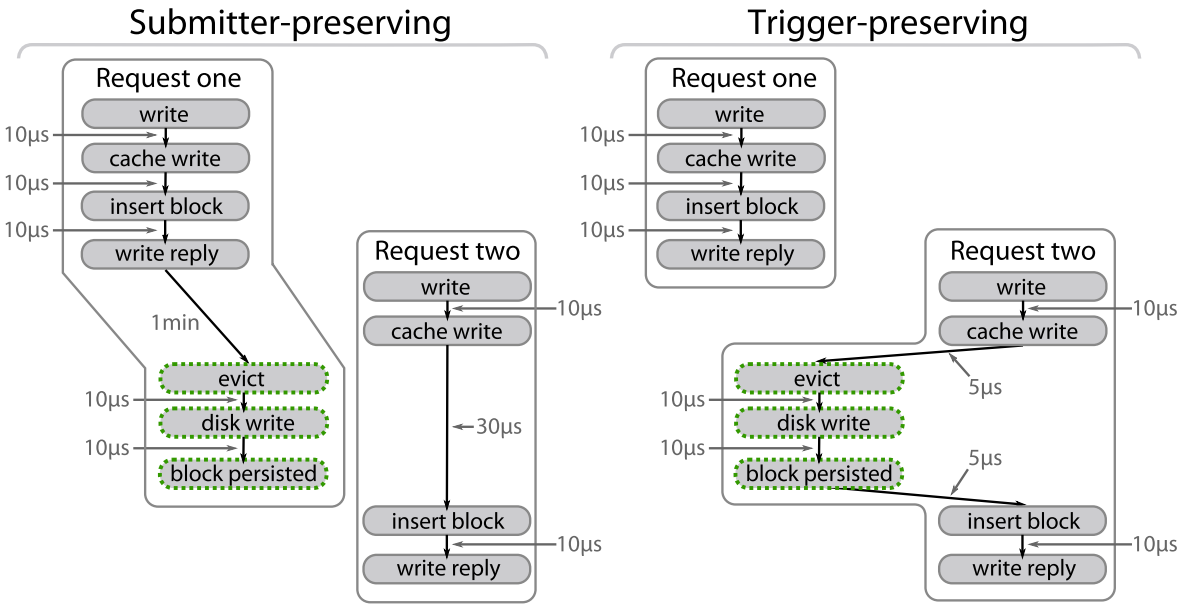


Figure 7.2: Traces for two storage system WRITE requests when preserving different slices of causality. Request one places its data in a write-back cache and returns immediately to the client. Sometime later, request two enters the system and must perform an on-demand eviction of request one's data to place its data in the cache. This latent work (highlighted in dotted green) may be attributed to request one (if submitter causality is preserved) or request two (if trigger causality is preserved). The one minute latency for the leftmost trace is an artifact of the fact that the traces show latencies between trace-point executions. It would not appear if they showed latencies of function call executions instead, as is the case for Whodunit [27].

trace points for only 0.1% of requests, the probability of logging an individual trace point will also be 0.1%. However, after aggregating 32 items, this probability will increase to 3.2% and after two such levels of aggregation, the trace-point sampling percentage will increase to 65%. Tail-based coherent sampling avoids this problem, as the sampling decision is not made until a flow's end. But, it is not often used because it requires temporarily caching all trace points. An alternative is to use unitary sampling. Since traces cannot be constructed with this approach, information needed for analysis (e.g., the submitter or client ID) should be embedded directly in each trace point.

7.3.2 The trigger-preserving slice

In addition to requiring a large percentage of trace points to be logged due to aggregation events, the submitter-preserving trace for request one shown in Figure 7.2 is unintuitive and hard to understand when visualized because it attributes work done to the request

after the client reply has been sent. Also, latent work attributed to this request (i.e., trace points executed after the reply is sent) is performed in the critical path of request two. In contrast, trigger causality guarantees that a trace of a request will show all work that must be performed before a client response can be sent, including another client's latent work if it is in its critical path. The right two traces in Figure 7.2 show the same two requests as in the submitter-preserving example, with trigger causality preserved instead. Since these traces are easier to understand when visualized (they always end with a client reply) and always show all work done on requests' critical paths, trigger causality should be preserved for diagnosis tasks, which often involve answering questions of the form "Why is this request so slow?" Indeed, trigger causality is explicitly preserved by the revised version of Stardust [117]. Since it takes less effort to preserve than submitter causality, many other tracing implementations implicitly preserve this slice of causality [15, 29, 47, 110, 125]. Trigger causality can be used with head-based coherent sampling without inflating the trace-point sampling percentage, since only the sampling decision made for the request that forces latent work to be executed determines whether trace points after aggregation points are sampled.

7.3.3 Is anything gained by preserving both?

The slices suggested above are the most important ones that should be preserved for various use cases, not the only ones that should be preserved. Indeed, preserving both submitter causality and trigger causality will enable a deeper understanding of the distributed system than what is possible by preserving only one of them. For example, for diagnosis, preserving submitter causality in addition to trigger causality will allow the tracing infrastructure to answer questions such as "Who was responsible for evicting my client's cached data?" or, more generally, "Which clients tend to interfere with each other most?"

7.3.4 Preserving concurrency, forks, and joins

For both submitter-preserving causality and trigger-preserving causality, delineating concurrent activity from sequential activity and preserving forks and joins is optional. Doing so often requires more effort on the part of developers (e.g., finding forks and joins in the system and instrumenting them) and additional metadata, but enables developers to view actual request structure, and diagnose problems that arise as a result of excessive parallelism, not enough parallelism, or excessive waiting at synchronization points. It also allows

for easier automated critical-path identification. As such, diagnosis tasks benefit greatly from preserving these causal relationships. The revised versions of both Stardust [117] and X-Trace [47] explicitly delineate concurrency and explicitly identify forks and joins. For other use cases, such as profiling and resource attribution, delineating concurrency is not necessary, and most profilers (including Whodunit [27] and `gprof` [58]) do not do so.

7.3.5 Preserving inter-request slices

In addition to relationships within a request, many types of causal relationships may exist between requests. This section describes the two most common ones.

The contention-preserving slice: Requests may compete with each other for resources, such as access to a shared variable. Preserving causality between requests holding a resource lock and those waiting for it can help explain unexpected performance slowdowns or timeouts. Only Whodunit [27] preserves this slice.

The read after write-preserving slice: Requests that read data (e.g., from a cache or file) written by others may be causally affected by the contents. For example, a request that performs work dictated by the contents of a file—e.g., a map-reduce job [40]—may depend on that file's original writer. Preserving read-after-write dependencies can help explain such requests' behaviour.

7.4 Causal tracking

All end-to-end tracing infrastructures must employ a mechanism to track the slices of intra-request and inter-request causality most relevant to their intended use cases. To avoid capturing superfluous relationships, tracing infrastructures "thread" metadata along with individual flows and establish happens-before relationships only to items with the same (or related) metadata [27, 29, 47, 48, 110, 117, 125, 141, 142]. Section 7.4.1 describes different options for what to propagate as metadata and tradeoffs between them. In general, metadata can be propagated by storing them in thread-local variables when a single thread is performing causally-related work and encoding logic to propagate metadata across boundaries (e.g., across threads or components) in commonly used libraries.

Though any of the approaches discussed below can preserve concurrency by establishing happens-before relationships, additional instrumentation is needed to capture forks and joins, which are needed to properly order concurrency-related causal relationships. For example, if joins are not preserved, a trace point may erroneously be deemed causally

dependent on the last thread to synchronize, rather than on multiple concurrent activities. Section 7.4.2 further discusses how to preserve forks and joins.

7.4.1 What to propagate as metadata?

Most tracing infrastructures propagate one of the options listed below. They differ in size, whether they are fixed or variable width, and how they establish happens-before relationships to order causally-related activity and distinguish concurrent behaviour from sequential behaviour. Perhaps surprisingly, all of them are equivalent in functionality, except for the case of event-based systems, for which mutable breadcrumbs are needed.

An immutable, fixed-width trace ID (TID): The trace ID identifies causally-related work and could be a unique value chosen at the start of such activity (e.g., when a request is received or at the start of background activity). Tracing implementations that use this method must rely on visible clues to establish happens-before relationships between trace points. For example, since network messages must always be sent by a client before being received by a server, tracing infrastructures that do not rely on synchronized clocks might establish happens-before relationships between client and server work using network send and receive trace points on both machines. Similarly, to order causally-related activity within a single thread, they must rely on an external clock. To identify concurrent work within components, they might establish happens-before relationship via thread IDs. Pip [110], Pinpoint [29], and Quanto [46] use an immutable fixed-width trace ID as metadata.

An immutable fixed-width trace ID and a mutable, fixed-width breadcrumb (TID & BC): Like the previous method, the trace ID allows all causally-related activity to be identified and quickly extracted. Instead of relying on external clues or trace points, mutable breadcrumbs are used to encode happens-before relationships. For example, to show that a given trace point occurs after another, the latter will be assigned a new breadcrumb and a happens-before relationship will be created between it and the previous trace point's breadcrumb. Similarly, when sending an RPC, the tracing infrastructure will assign the server a different breadcrumb than the client and establish "client breadcrumb happens before server breadcrumb." In contrast to the previous approach, this metadata-propagation approach relies less on externally observable clues. For example, if thread IDs are not visible, the previous approach might not be able to identify concurrent activity within a component. Also, mutable breadcrumbs are needed to identify different causally-related flows flows in event-based systems, in which a single thread performs all work. Both versions of X-

Trace [47, 48] use this metadata-propagation method. Dapper [125] and both versions of Stardust [117, 141] use a hybrid approach that combines the previous approach and this one. Stardust uses the same breadcrumb for all causally-related activity within a single thread, whereas Dapper does the same for all work done on behalf of a request at the client and server.

An immutable fixed-width trace ID and a mutable, variable-width logical clock (TID & var clock): Both metadata-propagation methods described above are brittle. The trace ID (TID) approach relies on clues to establish happens-before relationships and properly order trace points. The breadcrumb approach (TID & BC) will be unable to properly order a given set of trace points if the happens-before relationship relating one breadcrumb to the other is lost. One option to avoid such brittleness is to carry vector clocks instead of breadcrumbs, however, doing so would require a metadata field as wide as the number of threads in the entire distributed system. Instead, interval-tree clocks [4] should be used. They are similar to vector clocks, but only require variable width proportional to the current number of threads involved in servicing a request. To my knowledge, no existing tracing infrastructure uses this method of metadata propagation.

The trace points themselves: A distinct class of options involves carrying trace point records along with requests as metadata. Since traces are available immediately after a flow has finished executing, analyses that require traces can be performed online. For example, critical paths can be identified and the system's configuration changed to lessen their impact. Whodunit [27] is the only tracing infrastructure that uses trace points (function names) as metadata. Since Whodunit is designed for profiling, it is not concerned with preserving concurrency, and so maintains trace-point names as a simple vector. Heuristics are used to reduce the number of propagated trace points, but at the cost of trace fidelity.

7.4.2 How to preserve forks and joins

For all of the metadata-propagation approaches discussed above, trace points can be added to preserve forks and joins. For the single trace ID approach (TID), such trace points must include clues that uniquely identify the activity being forked or waited on—for example, thread IDs. For the breadcrumb approach (TID & BC), such trace points should include breadcrumbs created by establishing one-to-many or many-to-one happens-before relationships. Note that Dapper's hybrid metadata-propagation approach, which keeps breadcrumbs the same across network activity, requires that it use the single trace ID

method for preserving forks and joins.

An alternate approach, explored by Mann et al. [89] involves comparing large volumes of traces to automatically determine how many concurrent threads a join depends on. This approach reduces the developer burden of explicit instrumentation, but is not yet used by any of the tracing infrastructures discussed in this paper.

7.5 Trace visualization

Good visualizations are important for use cases such as diagnosis and profiling. Effective visualizations will amplify developers' efforts whereas ineffective ones will hinder their efforts and convince them to use other tools and techniques [86, 116]. Indeed, Oliner et al. identify visualization as one of the key future challenges in diagnosis research [102]. This section highlights common approaches to visualizing end-to-end traces. The choices between them depend on the visualization's intended use, previous design choices, and whether precision (i.e., the ability to show forks, joins, and concurrency) is preferred over volume of data shown. Table 7.3 summarizes the tradeoffs between the various visualizations. Figure 7.3 shows how some of the visualizations would differ in showing requests. Instead of visualizing traces, Pip [110] uses an expectation language to describe traces textually. Formal user studies are required to compare the relative benefits of visualizations and expectations and I make no attempt to do so here.

Gantt charts: These visualizations are most often used to show individual traces. The Y-axis shows the overall request and resulting sub-requests issued by the distributed system and the X-axis shows relative time. The relative start time and latency (measured in wall-

	Precision			Multiple flows?	
	Forks	**Joins**	**Concurrency**	**Same**	**Different**
Gantt charts	I	I	I	N	N
Flow graphs	Y	Y	Y	Y	N
Focus graphs	N	N	N	Y	N
CCTs	N	N	N	Y	Y

Table 7.3: Tradeoffs between trace visualizations. Different visualizations differ in precision—i.e., if they can show forks, joins and concurrency ("Y"), or if it must be inferred ("I"). They also differ in their ability to show multiple flows, and whether those multiple flows can be different (e.g., multiple requests with different workflows through the distributed system). To our knowledge, these visualizations have been used to show traces that contain up to a few hundred trace points.

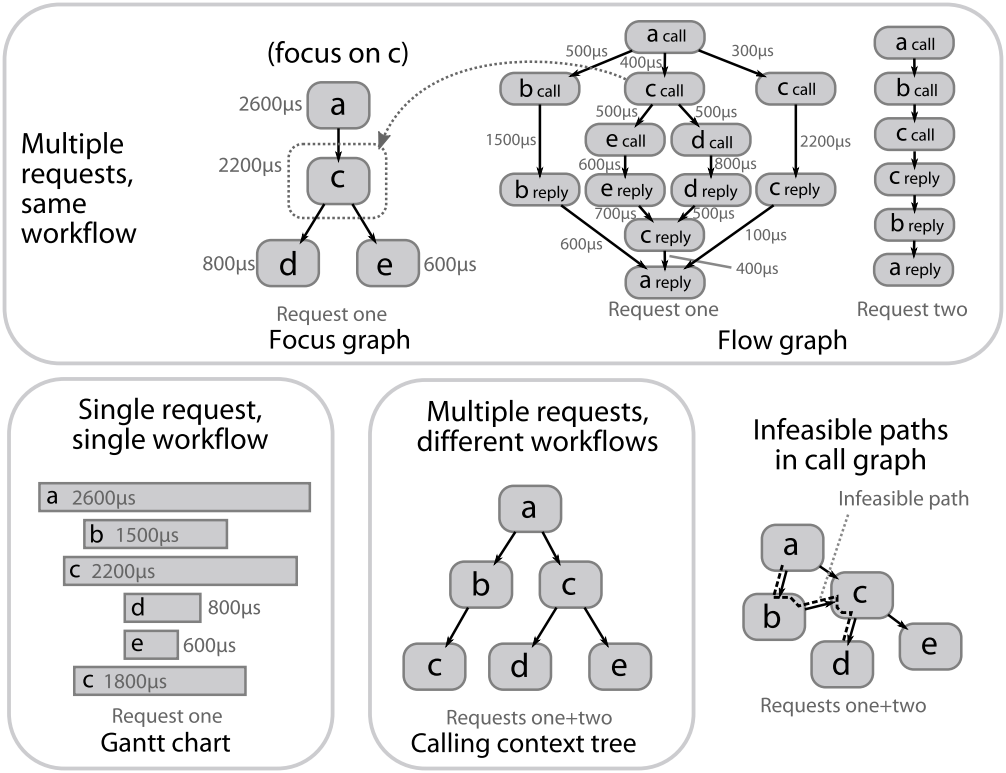


Figure 7.3: Comparison of various approaches for visualizing traces. Gantt charts are often used to visualize individual requests. Flow graphs allow multiple requests with identical workflows to be visualized at the same time while showing forks, joins, and concurrency. However, they must show requests with different workflows separately (as shown by requests one and two). CCTs trade precision for the ability to visualize multiple requests with different workflows (e.g., an entire workload). Call graphs can also show multiple workflows, but may show infeasible paths that did not occur in an actual execution. For example, see the $a \rightarrow b \rightarrow c \rightarrow d$ path in the call graph shown, which does not appear in either request one or two.

clock time) of items shown on the Y-axis are encoded by horizontal bars. Concurrency can easily be inferred by visually identifying bars that overlap in X-axis values. Forks and joins must also be identified visually, but it is harder to do so. Both ETE [66] and Dapper [125] use Gantt charts to visualize individual traces. In addition to showing latencies of the overall request and sub requests, Dapper also identifies network time by subtracting time spent at the server from the observed latency of the request or sub-request.

Flow graphs (also called request-flow graphs): These directed-acyclic graphs faithfully show requests' workflows as they are executed by the various components of a distributed system. They are often used to visualize and show aggregate information about multiple requests that have identical workflows. Since such requests are often expected to perform

similarly, flow graphs are a good way to preserve precision, while still showing multiple requests. Fan-outs in the graph represent the start of concurrent activity (forks), events on different branches are concurrent, and fan-ins represent synchronization points (joins). The revised version of Stardust [117] and the revised version of X-Trace [47] visualize traces via flow graphs. Note that for flow graphs to be used, the underlying causal tracking mechanism must preserve concurrency, forks, and joins. If these properties are not preserved, CCTs could be used instead.

Call graphs and focus graphs: These visualizations are also often used to show multiple traces, but do not preserve concurrency, forks, or joins, and so are not precise. Call graphs use fan-outs to show functions accessed by a parent function. Focus graphs show the call stack to a chosen component or function, called the "focus node," and the call graph that results from treating the focus node as its root. In general, focus graphs are best used for diagnosis tasks for which developers already knows which functions or components are problematic. Dapper [125] uses focus graphs to show multiple request with identical workflows, but owing to its RPC-oriented nature, nodes do not represent components or functions, but rather all work done to execute an RPC at the client and server. Note that when used to visualize multiple requests with different workflows, call graphs can show infeasible paths [5]. This is demonstrated by the $a \rightarrow b \rightarrow c \rightarrow d$ path for the call graph shown in Figure 7.3.

Calling Context Trees (CCTs) [5]: These visualizations are best used to show multiple traces with different workloads, as they guarantee that every path from root to leaf is a valid path through the distributed system. To do so in a compact way, they use fan-outs to show function invocations, not forks, and, as such, are not precise. CCTs can be constructed in amortized constant time and are best used for tasks for which a high-level summary of system behaviour is desired (e.g., profiling). Whodunit [27] uses CCTs to show profiling information for workloads.

7.6 Putting it all together

Based on the tradeoffs described in previous sections and my own experiences, this section identifies good design choices for the key uses of end-to-end tracing. I also show previous implementations' choices and contrast them to my suggestions.

7.6.1 Suggested choices

The italicized rows of Table 7.4 show suggested design choices for key use cases of end-to-end tracing. To preserve concurrent behaviour, I suggest the hybrid fixed-width immutable thread ID and fixed-width mutable breadcrumb (TID & BC) approach used by Stardust [117, 141], because it offers the best tradeoffs among the various approaches. Specifically, it is of constant size, does not rely much on external clues (except for a timestamp to order intra-component trace points), and is less brittle than the straightforward TID & BC approach (since it only switches breadcrumbs between components). However, the immutable, fixed-width thread ID (TID) approach is also a good choice if the needed external clues will always be available. I also suggest developers should consider using variable-width approaches if feasible. For use cases that require coherent sampling, I conservatively suggest the head-based version when it is sufficient, but tail-based based coherent sampling should also be considered since it subsumes the former and allows for a wider range of uses. The rest of this section explains design choices for the various use cases.

Anomaly detection: This use case involves identifying and recording rare flows that are extremely different from others so that developers can analyze them. As such, tail-based coherent sampling should be used so that traces can be constructed and so that the tracing infrastructure can gauge whether a flow is anomalous before deciding whether or not to sample it. Either trigger causality or submitter causality can be preserved with low overhead, but the former should be preferred since it shows critical paths and because they are easier to understand when visualized. To identify anomalies that result due to excessive parallelism, too little parallelism, or excessive waiting for one of many concurrent operations to finish, implementations should preserve forks, joins, and concurrent behaviour. Flow graphs are best for visualizing anomalies because they are precise and because anomaly detection will, by definition, not generate many results. Gantt charts can also be used to visualize individual flows.

Diagnosing steady-state problems: This use case involves diagnosing performance and correctness problems that can be observed in many requests. Design choices for it are similar to anomaly detection, except that head-based sampling can be used, since even with low sampling rates it is unlikely that problems will go unnoticed.

Distributed profiling: This use case involves sampling function (or inter-trace point) latencies. The inter- and intra-component call stacks to a function must be preserved so that sampled times can be grouped together based on context, but complete traces need not be

Use	Name	Design axes				
		Sampling	**Causality slices**	**Forks/ joins/conc.**	**Metadata**	**Visualization**
Anomaly detection	*Suggested*	*Coherent (T)*	*Trigger*	*Yes*	*TID & BC (H)*	*Flow graphs*
	Magpie [15]	No	”	”	None	Gantt charts (V)
	Pinpoint [29]	”	”	No	TID	Paths
Diagnosing steady-state problems	*Suggested*	*Coherent (H)*	*Trigger*	*Yes*	*TID & BC (H)*	*Flow graphs*
	Stardust‡ [117]	”	”	”	”	”
	X-Trace‡ [47]	”	”	”	TID & BC	”
	Dapper [125]	”	”	Conc. only	TID & BC (H)	Gantt charts & focus graphs
	Pip [110]	No	”	Yes	TID	Expectations
	X-Trace [48]	”	Trigger & TCP layers	No	TID & BC	Paths & network layers
	Pinpoint [29]	”	Trigger	”	TID	Paths
Distributed profiling	*Suggested*	*Coherent (T)*	*Either*	*No*	*Trace points*	*CCTs*
	Whodunit [27]	”	Submitter	”	”	”
	ETE [66]	No	Trigger	”	None	Gantt charts
	Dapper [125]	Coherent (H)	”	conc. only	TID & BC (H)	Gantt charts & focus graphs
Resource attribution	*Suggested*	*Unitary*	*Submitter*	*No*	*TID*	*None*
	Stardust [141]	”	”	Forks/conc.	TID & BC (H)	Call graphs
	Quanto [46]	No	”	No	TID	None
Workload modeling	*Suggested*	*Depends*	*Depends*	*Depends*	*Depends*	*Flow graphs or CCTs*
	Magpie [15]	No	Trigger	Yes	None	Join-based charts
	Stardust [141]	Unitary	Submitter	Forks/conc.	TID & BC (H)	Call graphs

Table 7.4: Suggested design choices for various use cases and choices made by existing tracing implementations. Suggested choices are shown in italics. Existing implementations' design choices are ordered according to similarity with the suggested choices. Magpie and ETE do not propagate metadata, but rather use schemas to determine causal relationships. The revised versions of Stardust and X-Trace are denoted by *Stardust‡* and *X-Trace‡*. Hybrid versions of the TID & BC approach are denoted by *TID & BC (H)*. Coherent head-based sampling is referred to as *coherent (H)* and coherent tail-based sampling is referred to as *coherent (T)*. *(V)* indicates a variant of the stated item is used

constructed. These requirements align well with tail-based coherent sampling (where "tail" is defined to be the current function being executed) and carrying trace points as metadata. Together, these options will allow profiles to be collected online. Call stacks do not need to preserve forks, joins, or concurrency. CCTs are best for visualizing distributed profiles, since they can show entire workloads and infeasible paths do not appear.

Resource attribution: This use case involves attributing work done at arbitrary levels of the system to the original submitter, so submitter causality must be preserved. Unitary sampling is sufficient, since submitter IDs can be propagated with requests and stored with trace points. It is not important to preserve forks, joins, or concurrency, so the TID approach to causal tracking is sufficient. Since traces need not be constructed, trace visualization is not necessary.

Workload modeling: The design decisions for this use case depend on what properties of the workload are being modeled. For example, when used to model workloads, Magpie [15] aims to identify a set of flows and associated resource usages that are representative of an entire workload. As such, it is useful for Magpie to preserve forks, joins, and concurrent behaviour. If traces for this use case are to be visualized, flow graphs or CCTs should be used since they allow for visualizing multiple traces at one time.

7.6.2 Existing tracing implementations' choices

Table 7.4 also shows existing tracing implementations and the choices they make. Existing implementations are qualitatively ordered by similarity in design choices to my suggested ones. In many cases, the design choices made by existing tracing implementations are similar to my suggestions. However, in cases where differences exist, existing implementations often fall short of their intended range of uses or achieve needed functionality via inefficient means.

For anomaly detection, I suggest tail-based sampling, but both Magpie [15] and Pinpoint [29] do not use any sampling techniques whatsoever. Collecting and storing trace points for every request guarantees that both implementations will not miss capturing any rare events (anomalies), but also means they cannot scale to handle large workloads. Magpie cannot use sampling, because it does not propagate metadata. Pinpoint is concerned mainly with correctness anomalies, and so does not bother to preserve concurrency, forks, or joins.

For diagnosing steady-state problems, the design choices made by the revised version of Stardust [117] and the revised version of X-Trace [47] are identical to my suggested choices.

I originally tried to use the original version of Stardust, which was designed for resource accounting, for diagnosis, but found it insufficient, motivating the need for the revised version. X-Trace was originally designed to help with diagnosis tasks by showing causality within and across network layers, but over time its design was evolved to reflect my current choices because they proved to be more useful.

Pip [110] differs from many other tracing infrastructures in that it uses an expectation language to show traces. Pip's expectation language describes how other components interact with a component of interest and so is similar in functionality to Dapper's component-based graphs. Both are best used when developers already have a component-of-interest in mind, not for problem localization tasks.

For the most part, for distributed profiling and resource attribution, existing infrastructures either meet or exceed my suggestions.

7.7 Challenges & opportunities

Though end-to-end tracing has proven useful, many important challenges remain before it can reach its full potential. Most are a result of the complexity and volume of traces generated by today's large-scale distributed systems. This section summarizes them.

As instrumented systems scale both in size and workload, tracing infrastructures must accommodate larger, more complex, traces at higher throughput, while maintaining relevance of tracing data. Though head-based sampling meets the first two criteria of this key challenge, it does not guarantee trace relevance. For example, it complicates diagnostics on specific traces and will not capture rare bugs (i.e., anomalies). Conversely, tail-based sampling, in which trace points are cached until requests complete, meets the relevance criteria, but not the first two.

An in-between approach, in which all trace points for requests are discarded *as soon as* the request is deemed uninteresting, seems a likely solution, but important research into finding the trace attributes that best determine when a trace can be discarded is needed before this approach can be adopted. An alternate approach may be to collect low-resolution traces in the common case and to increase resolution only when a given trace is deemed interesting. However, this approach also requires answering similar research questions as that required for the in-between approach.

Another challenge, which end-to-end tracing shares with logging, involves trace interpretability. In many cases, the developers responsible for instrumenting a distributed system

are not the same as those tasked with using the resulting traces. This leads to confusion because of differences in context and expertise. For example, in a recent user study, Sambasivan et al. had to manually translate the trace-point names within end-to-end traces from developer-created ones to ones more readily understood by general distributed systems experts [116]. To help, key research must be conducted on how to define good instrumentation practices, how to incentivize good instrumentation, and how to educate users on how to interpret instrumented traces or logs. Research into automatic instrumentation and on the fly re-instrumentation (e.g., as in DTrace [26]) can also help reduce instrumentation burden and help interpretability.

A third important challenge lies in the integration of different end-to-end tracing infrastructures. Today's distributed services are composed of many independently-developed parts, perhaps instrumented with different tracing infrastructures (e.g., Dapper [125], Stardust [117, 141], Tracelytics [143], X-Trace [47, 48], or Zipkin [147]). Unless they are modified to be interoperable, we miss the opportunity to obtain true end-to-end traces of composed services. The provenance community has moved forward in this direction by creating the Open Provenance Model [32], which deserves careful examination.

7.8 Conclusion

End-to-end tracing can be implemented in many ways, and the choices made dictate the utility of the resulting traces for different development and management tasks. Based on my experiences developing tracing infrastructures and the past ten years of research on the topic, this chapter provides guidance to designers of such infrastructures and identifies open questions for researchers.

Chapter 8

Conclusion

Performance diagnosis in distributed systems is extremely challenging, because the problem could be contained in any of the system's numerous components or dependencies or may be an artifact of interactions among them. As distributed services continue to grow in complexity and in the number of other distributed services upon which they depend, performance diagnosis will only become more challenging. This dissertation explores the use of a novel technique, called *request-flow comparison* [117], for helping diagnose problems in distributed systems. Request-flow comparison uses end-to-end traces, which show the entire workflow of individual requests through the distributed service and its dependencies, to automatically localize the problem from the many possible culprits to just a few potential ones. The results presented in this dissertation show the usefulness of request-flow comparison and the efficacy of the algorithms used to implement it.

8.1 Contributions

The key contributions of this dissertation are the request-flow comparison workflow and the case studies demonstrating its usefulness in helping diagnose real, previously unsolved distributed systems problems. To enable these contributions, this dissertation addresses all aspects of building a request-flow comparison diagnosis tool. For example, it describes algorithms and heuristics for implementing request-flow comparison and where these algorithms fall short. Also, via a 26-person user study involving real distributed systems developers, it identifies promising approaches for visualizing request-flow comparison.

Another important contribution is the observation that distributed systems' potential for automation will always be limited unless they are built to be more predictable. To demon-

strate the validity of this observation, this dissertation shows how existing automation tools are affected by unpredictability, or high variance in key performance metrics. It suggests that to make systems more predictable, system builders must incorporate performance variance as an important metric and strive to minimize it where appropriate. To help system builders lower performance variance, it conceptually describes a tool for identifying sources of variance in distributed systems and a nomenclature for helping system builders understand what to do about different variance sources.

The final major contribution of this dissertation is a design study of end-to-end tracing. It aims to guide designers of tracing infrastructures by distilling key design axes that dictate a tracing infrastructure's utility for different use cases, such as resource attribution and diagnosis. Existing literature [3, 15, 27, 29, 47, 48, 66, 78, 110, 111, 117, 125, 130, 131, 141, 142, 152] treats end-to-end tracing as a one-size-fits-all solution and so does not identify these axes. Since the axes are not well understood, many existing tracing infrastructures do not effectively support some desired use cases.

8.2 Thoughts on future work

I believe my work on request-flow comparison has helped identify a wealth of future work opportunities, some of which were described in previous chapters. This section highlights the most important ones.

8.2.1 Generalizing request-flow comparison to more systems

This dissertation describes algorithms and heuristics for implementing request-flow comparison's workflow that proved effective for Ursa Minor and select Google services. It also described modifications needed for these algorithms to be useful in HDFS [134]. However, due to the wide variety of request-based distributed systems in existence, different (or modified) request-flow comparison algorithms may be needed for different distributed systems and problem types.

Before developing request-flow comparison algorithms for a candidate distributed system, developers must first ensure that end-to-end traces showing request workflows can be obtained for it. Developers must carefully consider what constitutes the workflow of an "individual request" (i.e., a fundamental unit of work) and guarantee that individual traces show this information. Doing so may be easy for some systems (e.g., storage systems) and difficult for others (e.g., batch-processing systems). Chapter 7.3 provides guidance on

what causal relationships to preserve when defining the workflow of individual requests for diagnosis tasks. Most notably, for request-flow comparison to be effective, request workflows must show work done on the critical path of the request. Note that if end-to-end traces will be obtained via a metadata-propagation-based tracing infrastructure, the design choices suggested in Chapter 7.6.1 for diagnosing steady-state problems may be of value in choosing how to define request workflows, how to avoid high tracing overheads, and how to present trace data.

When creating a request-flow-comparison tool for a candidate system, developers must carefully consider which algorithms to use to implement request-flow comparison's categorization step. This workflow step is responsible for grouping requests that are expected to perform similarly into the same category. I believe Spectroscope's expectation that requests with similar or identical structures should perform similarly is fundamental and valid in many systems. However, additional or looser constraints may be needed for other systems. For example, Chapter 3.5.3 demonstrated that a straightforward interpretation of the same structures/similar cost expectation is too strict for HDFS, as using it results in too many categories with too few requests in each to identify mutations. The chapter suggested ways to loosen the expectation by removing certain structural details while still guaranteeing that categories will contain requests that are expected to perform similarly. For example, for homogeneous, dedicated-machine environments, in which all machines are expected to perform similarly, it suggested removing machine names from node labels. For heterogeneous or shared-machine environments, it suggested grouping individual machines into equivalence classes of similarly-performing ones and using equivalence class names instead of machine names in node labels.

In practice, it may be difficult to know up-front which properties of requests dictate performance. To help, I believe additional research is needed to investigate *semi-supervised* methods for categorization. With such methods, the request-flow-comparison tool would use basic expectations (such as the same structures/similar performance one) and observed performance to present an initial categorization of requests. Diagnosticians could then use a visual interface to specify which groupings align with their intuition of what requests are expected to perform similarly. The comparison tool could then use diagnosticians' feedback to, over time, learn what request properties dictate performance and improve the results of its categorization step. Note that differences between automatically-learned groupings and diagnosticians' intuition may themselves indicate important performance problems.

I believe the algorithms used by Spectroscope to identify response-time mutations and

structural mutations are be applicable to a broad range of distributed systems. However, there is room for improvement. For response-time mutations, research is needed to identify whether alternate algorithms exist that are more robust to high variance caused by resource contention. Opportunities also exist to further reduce diagnostician burden when comparing structural mutations to their precursors—for example, by automating the identification of changed substructures between mutations and precursors. It remains to be seen whether algorithms, such as graph kernels [124], frequent-graph mining [71], or advanced structural anomaly detection [41] can be used to enable such additional automation.

In addition to determining what algorithms will work best when implementing request-flow comparison's basic workflow for a new distributed system, developers should also consider whether additional workflow steps would be useful. For example, additional workflow steps that directly identify contending requests or processes would be a valuable addition for distributed services that operate in shared-machine environments. However, additional research is needed to identify how best to identify competing activities and whether modifications to the underlying tracing infrastructure are necessary to do so.

In the long term, it is important to consider whether request-flow comparison (or other existing diagnosis techniques) can be extended to work in newly emerging *adaptive cloud environments* [68]. To increase resource utilization, these environments use schedulers that dynamically move workloads and change resource allocations. For example, to satisfy global constraints, they may move a running application to a different set of machines with very different hardware. If only a few applications are running, they may bless the running ones with very high resource allocations, only to steal back these resources later. Unfortunately, such dynamism reduces the effectiveness of most existing diagnosis tools, because they are not built to expect it and because increasing resource utilization often also increases performance variance. I believe that a first step toward extending existing diagnosis techniques to adaptive clouds will involve making them privy to datacenter scheduler decisions, so that they can determine whether or not an observed performance change is the expected result of a scheduler decision [119].

8.2.2 Improving request-flow comparison's presentation layer

Though the systems research community has created many problem localization techniques [14, 15, 29, 72, 76, 98, 103, 117, 122, 144], it has performed very little research on how to effectively present their results. This is unfortunate because the effectiveness of such

tools is limited by how easily diagnosticians can understand their results. The user study I ran comparing three approaches for visualizing Spectroscope's results (see Chapter 4) is but a first step. Participants' comments during this study, and my own experiences with Spectroscope, have revealed many insights that can guide future visualization work.

Most important is the insight that, by themselves, comparisons of full request-flow graphs are not sufficient for helping diagnosticians' interpret request-flow comparison's results. Even though the graphs presented to the user study participants often contained less than 200 nodes, participants often complained about their size. Worse, in complex distributed services—for example, ones built upon other distributed services—it is likely that no single person will have complete knowledge of the entire system. As such, diagnosticians of such systems will not possess the expertise to determine whether a performance change observed deep in the bowels of the systems represents valid behaviour or an an unexpected performance problem.

Semantic zooming, annotation mechanisms (as suggested by the user study participants), and graph collapsing (as discussed in Chapter 3.5.2) can somewhat help. But, I do not believe they are sufficient. Ideally, diagnosis tools should be able to identify the meaning of an observed problem automatically. For example, request-flow-comparison tools, like Spectroscope, could learn the textual meaning of individual graphs, enabling them to output both observed mutations and what they represent. One promising method for enabling such learning is crowdsourcing. As a game, developers of individual distributed systems components could be presented with graph substructures relevant only to their component and asked to name them. When presenting mutations and precursors, the diagnosis tool could display their meanings by combining the names given to their constituent substructures.

8.2.3 Building more predictable systems

Without increased predictability, automation techniques, such as request-flow comparison, will never reach their full potential. But, it is not clear how best to increase predictability for most distributed systems. Research is needed to identify good techniques for localizing inadvertent variance sources. Research to understand the dominant sources of variance in distributed systems would also help.

An alternate approach, suggested by Dean et al. [39], is to treat unpredictability as a given and build systems that are naturally resilient to it. I believe this approach and that of

reducing variance merit comparison, especially with regards to their implications toward automation and future diagnosis techniques.

8.2.4 Improving end-to-end tracing

Despite end-to-end tracing's usefulness, many organizations are still wary of it because of the enormous storage overhead it can impose when head-based sampling is used. Even for Ursa Minor [1], traces collected for its regression tests were a few gigabytes in size. For large organizations, such as Google, storage overhead could balloon to petabytes even with minimal sampling. Though tail-based sampling reduces storage overhead, it increases memory cost. A very pertinent area of research involves creating tracing techniques that can capture interesting traces while minimizing storage-related overhead.

Bibliography

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John Strunk, Eno Thereska, Matthew Wachs, and Jay Wylie. Ursa minor: versatile cluster-based storage. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005. Cited on pages 2, 4, 7, 11, 12, 16, 25, 26, 48, 54, 74, and 120.

[2] Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddarth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. NetPrints: diagnosing home network misconfigurations using shared knowledge. In *NSDI'09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009. Cited on page 88.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003. Cited on pages 3, 10, 91, 95, and 116.

[4] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *OPODIS'08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, 2008. Cited on page 105.

[5] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97: Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997. Cited on pages 86 and 108.

[6] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems*

Design and Implementation, 2010. Cited on page 16.

[7] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *STOC'09: Proceedings of the 41st Annual ACM Symposium on Theory of computing*, 2009. Cited on page 21.

[8] Apache hadoop. http://hadoop.apache.org. Cited on pages 15, 39, 40, 50, 72, and 85.

[9] Daniel Archambault, Helen C. Purchase, and Bruno Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):539–552, April 2011. Cited on pages 48, 60, and 65.

[10] Daniel Archambault, Helen C. Purchase, and Bruno Pinaud. Difference map readability for dynamic graphs. In *GD'10: Proceedings of the 18th International Conference on Graph Drawing*, 2011. Cited on pages 48 and 49.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4), April 2010. Cited on page 1.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *HotOS'01: Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, 2001. Cited on pages 70, 73, and 74.

[13] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. Cited on page 89.

[14] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM'07: Proceedings of the 2007 ACM SIGCOMM Conference on Data Communications*, 2007. Cited on pages 47, 72, 83, 84, and 118.

[15] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. Cited

on pages 3, 6, 10, 11, 17, 47, 71, 72, 75, 81, 82, 91, 92, 93, 94, 102, 110, 111, 116, and 118.

[16] Luiz A. Barroso. Warehouse-scale computing: entering the teenage decade, August 2011. Cited on pages 8 and 70.

[17] Fabian Beck and Stephan Diehl. Visual comparison of software architectures. In *SOFTVIS'10: Proceedings of the 5th International Symposium on Software Visualization*, 2010. Cited on page 49.

[18] Nicolas Garcia Belmonte. The Javascript Infovis Toolkit. http://www.thejit.org. Cited on page 50.

[19] Hal Berghel and David Roach. An extension of Ukkonen's enhanced dynamic programming ASM algorithm. *Transactions on Information Systems (TOIS*, 14(1), January 1996. Cited on page 21.

[20] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer Science + Business Media, LLC, first edition, 2006. Cited on page 23.

[21] Ledion Bitincka, Archana Ganapathi, Stephen Sorkin, and Steve Zhang. Optimizing data analysis with a semi-structured time series database. In *SLAML'10: Proceedings of the 2010 ACM Workshop on Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, 2010. Cited on pages 10 and 95.

[22] Peter Bodik, Armando Fox, Michael Jordan, and David Patterson. Advanced tools for operators at Amazon.com. In *HotAC'06: Proceedings of the 1st Workshop on Hot Topics in Autonomic Computing*, 2006. Cited on pages 86 and 87.

[23] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys'10: Proceedings of the 5th ACM SIGOPS European Conference on Computer Systems*, 2010. Cited on page 88.

[24] James Bovard. Slower is better: The new postal service. *Cato Policy Analysis*, February 1991. Cited on page 70.

[25] Jason D. Campbell, Arun B. Ganesan, Ben Gotow, Soila P. Kavulya, James Mulholland, Priya Narasimhan, Sriram Ramasubramanian, Mark Shuster, and Jiaqi Tan. Understanding and improving the diagnostic workflow of MapReduce users. In *CHIMIT'11: Proceedings of the 5th ACM Symposium on Computer Human Interaction for Management of Information Technology*, 2011. Cited on page 50.

[26] Bryan M. Cantrill and Michael W. Shapiro. Dynamic instrumentation of production systems. In *ATC'04: Proceedings of the 2004 USENIX Annual Technical Conference*, 2004. Cited on pages 3 and 113.

[27] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. In *EuroSys'07: Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*, 2007. Cited on pages 3, 6, 10, 26, 81, 86, 91, 93, 94, 96, 97, 98, 100, 101, 103, 105, 108, 110, and 116.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI'06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. Cited on pages 1, 25, 28, 29, 72, 74, and 98.

[29] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, David Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *NSDI'04: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, 2004. Cited on pages 3, 6, 10, 26, 47, 72, 75, 83, 91, 93, 94, 102, 103, 104, 110, 111, 116, and 118.

[30] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP'93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993. Cited on page 99.

[31] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? Application change? or Workload change? towards automated detection of application performance anomaly and change. In *DSN'08: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2008. Cited on page 7.

[32] Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011. Cited on page 113.

[33] Cloudera Htrace. http://github.com/cloudera/htrace. Cited on page 4.

[34] Ira Cohen, Moises Goldszmidt, and Terence Kelly. Correlating instrumentation

data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. Cited on page 88.

[35] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP'05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005. Cited on page 88.

[36] Compuware dynaTrace PurePath. http://www.compuware.com. Cited on pages 4 and 91.

[37] Gabriela F. Creţu-Ciocârlie, Mihai Budiu, and Moises Goldszmidt. Hunting for problems with Artemis. In *WASL'08: Proceedings of the 1st USENIX conference on Analysis of System Logs*, 2008. Cited on page 87.

[38] L. Dailey Paulson. Computer system, heal thyself. *Computer*, 35(8):20–22, 2002. Cited on page 1.

[39] Jeffrey Dean and Luiz A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013. Cited on page 119.

[40] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. Cited on pages 39 and 103.

[41] William Eberle and Lawrence B. Holder. Discovering structural anomalies in graph-based data. In *ICDMW'07: Proceedings of the 7th IEEE International Conference on Data Mining Workshops*, 2007. Cited on page 118.

[42] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In *Graph Drawing Software*. Springer-Verlag, 2003. Cited on page 53.

[43] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: extensible distributed tracing from kernels to clusters. In *SOSP'11: Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011. Cited on page 96.

[44] Michael Farrugia and Aaron Quigley. Effective temporal graph layout: a comparative study of animation versus static display methods. *Information Visualization*, 10(1):47–64, January 2011. Cited on pages 48, 49, 60, and 62.

[45] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo

Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys'12: Proceedings of the 7th ACM SIGOPS European Conference on Computer Systems*, 2012. Cited on page 77.

[46] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: tracking energy in networked embedded systems. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008. Cited on pages 10, 26, 93, 94, 100, 104, and 110.

[47] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 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*, 2010. Cited on pages 3, 6, 10, 26, 39, 91, 93, 94, 96, 97, 98, 102, 103, 105, 108, 110, 111, 113, and 116.

[48] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: a pervasive network tracing framework. In *NSDI'07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007. Cited on pages 3, 6, 10, 26, 91, 93, 94, 103, 105, 110, 113, and 116.

[49] Foundations for future clouds. Intel Labs, August 2011. http://www.istc-cc.cmu.edu/publications/papers/2011/ISTC-Cloud-Whitepaper.pdf. Cited on page 1.

[50] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* Storage: brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, September 2003. Cited on page 1.

[51] Emden R. Gansner, Eleftherios E. Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993. Cited on page 22.

[52] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, January 2009. Cited on page 53.

[53] Gapminder. http://www.gapminder.org. Cited on page 49.

[54] Sohaib Ghani, Niklas Elmqvist, and Ji S. Yi. Perception of animated node-link diagrams for dynamic graphs. *Computer Graphics Forum*, 31(3), June 2012. Cited on pages 64 and 65.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system.

In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003. Cited on pages 1, 13, 39, and 74.

[56] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI'11: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011. Cited on page 77.

[57] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *ASPLOS'98: Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998. Cited on pages 25 and 39.

[58] GNU gprof. http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html. Cited on page 103.

[59] Moises Goldszmidt, Mihai Budiu, Yue Zhang, and Michael Pechuk. Toward automatic policy refinement in repair services for large distributed systems. *ACM SIGOPS Operating Systems Review*, 44(2):47–51, April 2010. Cited on page 88.

[60] John Alexis Guerra Gomez, Audra Buck-Coleman, Catherine Plaisant, and Ben Shneiderman. Interactive visualizations for comparing two trees with structure and node value changes. Technical Report HCIL-2012-04, University of Maryland, 2012. Cited on page 49.

[61] Graphviz. http://www.graphviz.org. Cited on pages 22 and 47.

[62] Steven D. Gribble. Robustness in complex systems. In *HotOS'01: Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, 2001. Cited on pages 70 and 73.

[63] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: handling throughput variability for hypervisor IO scheduling. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. Cited on pages 29 and 77.

[64] Mountaz Hascoët and Pierre Dragicevic. Visual comparison of document collections using multi-layered graphs. Technical report, Laboratoire d'Informatique, de Robotique et de Microèlectronique de Montpellier (LIRMM), 2011. Cited on page 49.

[65] Jeffrey Heer and George G. Robertson. Animated Transitions in Statistical Data Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1240–1247, 2007. Cited on page 65.

[66] Joseph L. Hellerstein, Mark M. Maccabe, W. Nathaniel Mills III, and John J. Turek. ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems. In *ICDCS'99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999. Cited on pages 3, 11, 86, 91, 93, 94, 107, 110, and 116.

[67] James Hendricks, Raja R. Sambasivan, Shafeeq Sinnamohideen, and Gregory R. Ganger. Improving small file performance in object-based storage. Technical Report CMU-PDL-06-104, Carnegie Mellon University, 2006. Cited on page 34.

[68] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI'11: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011. Cited on page 118.

[69] Alfred Inselberg. *Parallel coordinates: Visual multidimensional geometry and its applications*. Springer-Verlag, 2009. Cited on page 50.

[70] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, April 2007. Cited on page 88.

[71] Ruoming Jin, Chao Wang, Dmitrii Polshakov, Srinivasan Parthasarathy, and Gagan Agrawal. Discovering frequent topological structures from graph datasets. In *KDD'05: Proceedings of the 11th ACM International Conference on Knowledge Discovery in Data Mining*, 2005. Cited on page 118.

[72] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM'09: Proceedings of the 2009 ACM SIGCOMM Conference on Data Communications*, 2009. Cited on pages 47, 50, 67, 72, 83, 84, 87, and 118.

[73] Hui Kang, Haifeng Chen, and Guofei Jiang. PeerWatch: a fault detection and diagnosis tool for virtualized consolidation systems. In *ICAC'10: Proceedings of the 7th IEEE International Conference on Autonomic Computing*, 2010. Cited on page 85.

[74] Michael P. Kasick, Keith A. Bare, Eurene E. Marinelli III, Jiaqi Tan, Rajeev Gandhi,

and Priya Narasimhan. System-call based problem diagnosis for PVFS. In *HotDep'09: Proceedings of the 5th USENIX Workshop on Hot Topics in System Dependability*, 2009. Cited on page 85.

[75] Michael P. Kasick, Rajeev Gandhi, and Priya Narasimhan. Behavior-based problem localization for parallel file systems. In *HotDep'10: Proceedings of the 6th USENIX Workshop on Hot Topics in System Dependability*, 2010. Cited on pages 44, 72, and 73.

[76] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *FAST'10: Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010. Cited on pages 44, 47, 72, 85, and 118.

[77] Jeffrey Katcher. Postmark: a new file system benchmark. Technical Report TR3022, Network Applicance, 1997. Cited on page 27.

[78] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matt Hultunen, Rajeev Gandhi, and Priya Narasimhan. Draco: statistical diagnosis of chronic problems in distributed systems. In *DSN'12: Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012. Cited on pages 3, 10, 91, 95, and 116.

[79] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *VEE'12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012. Cited on page 99.

[80] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. Cited on page 1.

[81] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *FSE'10: Proceedings of the 18th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2010. Cited on pages 85 and 86.

[82] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007. Cited on page 85.

[83] Eric Koskinen and John Jannotti. Borderpatrol: isolating events for black-box tracing. In *EuroSys'08: Proceedings of the 3rd ACM SIGOPS European Conference on Computer*

Systems, 2008. Cited on pages 10 and 95.

[84] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks are like snowflakes: no two are alike. In *HotOS'11: Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems*, 2011. Cited on pages 70 and 72.

[85] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978. Cited on pages 11 and 99.

[86] Zhicheng Liu, Bongshin Lee, Srikanth Kandula, and Ratul Mahajan. NetClinic: interactive visualization to enhance automated fault diagnosis in enterprise networks. In *VAST'10: Proceedings 2010 IEEE Conference on Visual Analytics Science and Technology*, 2010. Cited on pages 47, 50, 87, and 106.

[87] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and merging files with GNU diff and patch*. Network Theory Ltd, 2002. Cited on page 51.

[88] Ajay A. Mahimkar, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Qi Zhao. Towards automated performance diagnosis in a large IPTV network. In *SIGCOMM'09: Proceedings of the 2009 ACM SIGCOMM Conference on Data Communications*, 2009. Cited on pages 47, 67, 83, and 84.

[89] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-dar. Modeling the parallel execution of black-box services. In *HotCloud'11: Proceedings of the 2011 USENIX Workshop on Hot Topics in Cloud Computing*, 2011. Cited on page 106.

[90] Florian Mansmann, Fabian Fischer, Daniel A. Keim, and Stephen C. North. Visual support for analyzing network traffic and intrusion detection events using TreeMap and graph representations. In *CHIMIT'09: Proceedings of the 3rd ACM Symposium on Computer Human Interaction for Management of Information Technology*, 2009. Cited on pages 47 and 50.

[91] Frank J. Massey, Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):66–78, 1951. Cited on page 17.

[92] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004. Cited on pages 3 and 50.

[93] Alan G. Melville, Martin Graham, and Jessie B. Kennedy. Combined vs. separate views in matrix-based graph analysis and aomparison. In *IV'11: Proceedings of the*

15th International Conference on Information Visualisation, 2011. Cited on page 49.

[94] Jeffrey Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys'06: Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, 2006. Cited on pages 70 and 73.

[95] Hossein Momeni, Omid Kashefi, and Mohsen Sharifi. How to realize self-healing operating systems? In *ICTTA'08: Proceedings of the 3rd International Conference on Information and Communication Technologies*, 2008. Cited on page 1.

[96] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. Optiscope: performance accountability for optimizing compilers. In *CGO'09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009. Cited on page 89.

[97] Tamara Munzner, François Guimbretière, Serdar Tasiran, Li Zhang, and Yunhong Zhou. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. In *SIGGRAPH'03: Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques*, 2003. Cited on page 49.

[98] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI'12: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012. Cited on pages 47, 67, 72, 83, and 118.

[99] Hiep Nguyen, Yongmin Tan, and Xiaohui Gu. PAL: propogation-aware anomaly localization for cloud hosted distributed applications. In *SLAML'11: Proceedings of the 2011 ACM Workshop on Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, 2011. Cited on pages 7 and 82.

[100] William Norcott and Don Capps. Iozone filesystem benchmark program, 2002. http://www.iozone.org. Cited on page 27.

[101] Adam J. Oliner. *Using Influence to Understand Complex Systems*. PhD thesis, Stanford University, 2011. Cited on page 79.

[102] Adam J. Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2), February 2012. Cited on pages 50 and 106.

[103] Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. Using correlated surprise

to infer shared influence. In *DSN'10: Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010. Cited on pages 47, 72, 73, 82, and 118.

[104] Xinghao Pan, Jiaqi Tan, Soila P. Kavulya, Rajeev Gandhi, and Priya Narasimhan. Ganesha: black-box diagnosis of MapReduce systems. *SIGMETRICS Performance Evaluation Review*, 37(3), 2010. Cited on pages 72 and 85.

[105] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: file system directories with millions of files. In *FAST'11: Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011. Cited on page 70.

[106] David Patterson, Arron Brown, Pete Broadwell, George Candea, Mike Y. Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathon Traupman, and Noah Treuhaft. Recovery oriented computing (ROC): motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, University of California, Berkeley, 2002. Cited on page 73.

[107] Personal communications with Facebook and Google engineers. Cited on pages 1, 16, and 71.

[108] John R. Quinlan. Bagging, boosting, and C4.5. In *ICAI'96: Proceedings of the 13th National Conference on Artificial Intelligence*, 1996. Cited on page 23.

[109] Kai Ren, Julio López, and Garth Gibson. Otus: resource attribution in data-intensive clusters. In *MapReduce'11: Proceedings of the 2nd International Workshop on MapReduce and its Applications*, 2011. Cited on page 87.

[110] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin M. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006. Cited on pages 3, 6, 10, 26, 81, 83, 84, 91, 93, 94, 102, 103, 104, 106, 110, 112, and 116.

[111] Patrick Reynolds, Janet Wiener, Jeffrey Mogul, Marcos K. Aguilera, and Amin M. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW'06: Proceedings of the 15th International World Wide Web Conference*, 2006. Cited on pages 3, 10, 91, 95, and 116.

[112] Rietveld code review system. http://code.google.com/p/rietveld/. Cited

on page 67.

[113] G. Robertson, R. Fernandez, D. Fisher, B. Lee, and J. Stasko. Effectiveness of animation in trend visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1325–1332, November 2008. Cited on pages 48, 49, 60, and 62.

[114] George Robertson, Kim Cameron, Mary Czerwinski, and Daniel Robbins. Polyarchy visualization: visualizing multiple intersecting hierarchies. In *CHI'02: Proceedings of the 2002 ACM SIGCHI Conference on Human Factors in Computing Systems*, 2002. Cited on page 49.

[115] Raja R. Sambasivan and Gregory R. Ganger. Automated diagnosis without predictability is a recipe for failure. In *HotCloud'12: Proceedings of the 2012 USENIX Workshop on Hot Topics in Cloud Computing*, 2012. Cited on page 8.

[116] Raja R. Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. Technical Report CMU-PDL-13-104, Carnegie Mellon University, 2013. Cited on pages 4, 8, 87, 106, and 113.

[117] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011. Cited on pages 2, 3, 4, 6, 8, 10, 26, 42, 47, 72, 74, 75, 83, 87, 89, 91, 93, 94, 96, 97, 98, 102, 103, 105, 108, 109, 110, 111, 113, 115, 116, and 118.

[118] Raja R. Sambasivan, Alice X. Zheng, Eno Thereska, and Gregory R. Ganger. Categorizing and differencing system behaviours. In *HotAC'07: Proceedings of the 2nd Workshop on Hot Topics in Autonomic Computing*, 2007. Cited on pages 8, 17, and 42.

[119] Ilari Shafer, Snorri Gylfason, and Gregory R. Ganger. vQuery: a platform for connecting configuration and performance. *VMWare Technical Journal*, 1:1–8, December 2012. Cited on pages 15 and 118.

[120] Michael W. Shapiro. Self-healing in modern operating systems. *ACM Queue*, 2(9), December 2004. Cited on page 1.

[121] Hossam Sharara, Awalin Sopan, Galileo Namata, Lise Getoor, and Lisa Singh. G-PARE: a visual analytic tool for comparative analysis of uncertain graphs. In *VAST'11: Proceedings 2011 IEEE Conference on Visual Analytics Science and Technology*, 2011.

Cited on page 49.

[122] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. In *SIGMETRICS'09: Proceedings of the 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2009. Cited on pages 47, 72, 89, and 118.

[123] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O system performance debugging using model-driven anomaly characterization. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005. Cited on page 89.

[124] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman graph kernels. *The Journal of Machine Learning Research*, 12, feb 2011. Cited on page 118.

[125] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, 2010. Cited on pages 3, 4, 5, 6, 10, 16, 26, 75, 86, 91, 92, 93, 94, 96, 97, 98, 102, 103, 105, 107, 108, 110, 113, and 116.

[126] SPEC SFS97 (2.0). http://www.spec.org/sfs97. Cited on page 27.

[127] Roy Sterritt and Maurice G. Hinchey. Why computer-based systems should be autonomic. In *ECBS'05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, 2005. Cited on page 1.

[128] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981. Cited on page 53.

[129] Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region, April 2011. http://aws.amazon.com/message/65648. Cited on page 1.

[130] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vPath: precise discovery of request processing paths from black-box observations of thread and network activities. In *ATC'09: Proceedings of the 2009 USENIX Annual Technical Conference*, 2009. Cited on pages 3, 11, 91, 95, and 116.

[131] Jiaqi Tan, Soila P. Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, log-based causal tracing for performance debugging of MapReduce systems. In *ICDCS'10:*

Proceedings of the 30th IEEE International Conference on Distributed Computing, 2010. Cited on pages 3, 11, 50, 87, 91, 95, and 116.

[132] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with WISE. In *SIGCOMM'08: Proceedings of the 2008 ACM SIGCOMM Conference on Data Communications*, 2008. Cited on pages 13, 81, and 88.

[133] The 10 worst cloud outages (and what we can learn from them), June 2011. http://www.infoworld.com/d/cloud-computing/the-10-worst-cloud-outages-and-what-we-can-learn-them-902. Cited on page 1.

[134] The Apache Hadooop File System. http://hadoop.apache.org/hdfs/. Cited on pages 3, 4, 8, 16, 24, 25, 39, 85, and 116.

[135] The HDFS JIRA. https://issues.apache.org/jira/browse/HDFS. Cited on page 3.

[136] The Lustre file system. http://wiki.lustre.org. Cited on page 85.

[137] The Parallel Virtual File System. http://www.pvfs.org. Cited on pages 72, 73, 80, and 85.

[138] Eno Thereska, Michael Abd-El-Malek, Jay J. Wylie, Dushyanth Narayanan, and Gregory R. Ganger. Informed data distribution selection in a self-predicting storage system. In *ICAC'06: Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, 2006. Cited on page 88.

[139] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical performance models for complex, popular applications. In *SIGMETRICS'10: Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2010. Cited on page 88.

[140] Eno Thereska and Gregory R. Ganger. IRONModel: robust performance models in the wild. In *SIGMETRICS'08: Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2008. Cited on page 83.

[141] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS'06: Proceedings of the 2006 ACM SIGMETRICS*

International Conference on Measurement and Modeling of Computer Systems, 2006. Cited on pages 3, 6, 10, 12, 26, 28, 88, 91, 93, 94, 97, 98, 100, 103, 105, 109, 110, 113, and 116.

[142] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *HPDC'98: Proceedings of the 7th International Symposium on High Performance Distributed Computing*, 1998. Cited on pages 3, 86, 91, 103, and 116.

[143] Tracelytics. http://www.tracelytics.com. Cited on pages 4, 92, and 113.

[144] Avishay Traeger, Ivan Deras, and Erez Zadok. DARC: dynamic analysis of root causes of latency distributions. In *SIGMETRICS'08: Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2008. Cited on pages 47, 72, 89, and 118.

[145] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP'07: Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007. Cited on page 89.

[146] Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, Cheshire, Connecticut, 1983. Cited on page 23.

[147] Twitter Zipkin. https://github.com/twitter/zipkin. Cited on pages 4, 92, and 113.

[148] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *FAST'07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, 2007. Cited on pages 29 and 77.

[149] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, and Gregory R. Ganger. Exertion-based billing for cloud storage access. In *HotCloud'11: Proceedings of the 2011 USENIX Workshop on Hot Topics in Cloud Computing*, 2011. Cited on page 94.

[150] William Wang. End-to-end tracing in HDFS. Master's thesis, Carnegie Mellon University, 2011. Cited on pages 8, 40, and 91.

[151] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. Hang analysis: fighting responsiveness bugs. In *EuroSys'08: Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems*, 2008. Cited on

page 86.

[152] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting large-scale system problems by mining console logs. In *SOSP'09: Proceedings of 22nd ACM Symposium on Operating Systems Principles*, 2009. Cited on pages 3, 11, 81, 82, 91, 95, and 116.

[153] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *EuroSys'06: Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, 2006. Cited on page 88.

[154] Ding Yuan, Jin Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS'11: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011. Cited on page 80.

[155] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008. Cited on page 16.

[156] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: experiment-based management of virtualized data centers. In *ATC'09: Proceedings of the 2009 USENIX Annual Technical Conference*, 2009. Cited on page 89.