8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11).
March 30 - April 1, 2011. Boston, MA.

Diagnosing performance changes by comparing request flows

Raja R. Sambasivan[⋆], Alice X. Zheng[†], Michael De Rosa[‡], Elie Krevat[⋆],
Spencer Whitman[⋆], Michael Stroucken[⋆], William Wang[⋆], Lianghong Xu[⋆], Gregory R. Ganger[⋆]
[⋆]Carnegie Mellon University, [†]Microsoft Research, [‡]Google

Abstract

The causes of performance changes in a distributed system often elude even its developers. This paper develops a new technique for gaining insight into such changes: comparing request flows from two executions (e.g., of two system versions or time periods). Building on end-to-end request-flow tracing within and across components, algorithms are described for identifying and ranking changes in the flow and/or timing of request processing. The implementation of these algorithms in a tool called Spectroscope is evaluated. Six case studies are presented of using Spectroscope to diagnose performance changes in a distributed storage service caused by code changes, configuration modifications, and component degradations, demonstrating the value and efficacy of comparing request flows. Preliminary experiences of using Spectroscope to diagnose performance changes within select Google services are also presented.

1 Introduction

Diagnosing performance problems in distributed systems is difficult. Such problems may have many sources and may be contained in any one or more of the component processes or, more insidiously, may emerge from the interactions among them [21]. A suite of debugging tools is needed to help in identifying and understanding the root causes of the diverse types of performance problems that can arise. In contrast to single-process applications, for which diverse performance debugging tools exist (e.g., DTrace [6], gprof [14], and GDB [12]), too few techniques have been developed for guiding diagnosis of distributed system performance.

Recent research has developed promising new techniques that can help populate the suite. Many build on low-overhead end-to-end tracing (e.g., [4, 7, 9, 11, 31, 34]), which captures the *flow* (i.e., path and timing) of individual requests within and across the components of a distributed system. For example, with such rich information about a system's operation, researchers have developed new techniques for detecting anomalous request flows [4], spotting large-scale departures from performance models [33], and comparing observed behaviour to manually-constructed expectations [26].

This paper develops a new technique for the suite: comparing request flows between two executions to identify why performance has changed between them. Such comparison allows one execution to serve as a model of acceptable performance; highlighting key differences from this model and understanding their performance costs allows for easier diagnosis than when only a single execution is used. Though obtaining an execution of acceptable performance may not be possible in all cases—e.g., when a developer wants to understand why performance has always been poor—there are many cases for which request-flow comparison is useful. For example, it can help diagnose performance changes resulting from modifications made during software development (e.g., during regular regression testing) or from upgrades to components of a deployed system. Also, it can help when diagnosing changes over time in a deployed system, which may result from component degradations, resource leakage, or workload changes.

Our analysis of bug tracking data for a distributed storage service indicates that more than half of the reported performance problems would benefit from guidance provided by comparing request flows. Talks with Google engineers [3] and experiences using request-flow comparison to diagnose Google services affirm its utility.

The utility of comparing request flows relies on the observation that performance changes often manifest as changes in how requests are serviced. When comparing two executions, which we refer to as the *non-problem period* (before the change) and the *problem period* (after the change), there will usually be some changes in the observed request flows. We refer to new request flows in the problem period as *mutations* and to the request flows corresponding to how they were serviced in the non-problem period as *precursors*. Identifying mutations and comparing them to their precursors helps localize sources of change and gives insight into their effects.

This paper describes algorithms for effectively comparing request flows across periods, including for identifying mutations, ranking them based on their contribution to the overall performance change, identifying their most likely precursors, highlighting the most prominent divergences, and identifying low-level parameter differences that most strongly correlate to each.

We categorize mutations into two types: *Response-time mutations* correspond to requests that have increased only in cost between the periods; their precursors are requests that exhibit the same structure, but whose response time is different. *Structural mutations* correspond to requests that take different paths through the system in the problem period. Identifying their precursors requires analysis of all request flows with differing frequencies in the two periods.

Figure 1 illustrates a (mocked up) example of two mutations and their precursors. Ranking and highlighting divergences involves using statistical tests and comparison of mutations and associated precursors.

We have implemented request-flow comparison in a toolset called Spectroscope and used it to diagnose performance problems observed in Ursa Minor [1], a distributed storage service. By describing five real problems and one synthetic one, we illustrate the utility of comparing request flows and show that our algorithms enable effective use of this technique. To understand challenges associated with scaling request-flow comparison to very large distributed systems, this paper also describes preliminary experiences using it to diagnose performance changes within distributed services at Google.

2 End-to-end request-flow tracing

Request-flow comparison builds on end-to-end tracing, an invaluable information source that captures a distributed system's performance and control flow in detail. Such tracing works by capturing *activity records* at each of various trace points within the distributed system's software, with each record identifying the specific tracepoint name, the current time, and other contextual information. Most implementations associate activity records with individual requests by propagating a per-request identifier, which is stored within the record. Activity records can be stitched together, either offline or online, to yield request-flow graphs, which show the control flow of individual requests. Several efforts, including Magpie [4], Whodunit [7], Pinpoint [9], X-Trace [10, 11], Google's Dapper [31], and Stardust [34] have independently implemented such tracing and shown that it can be used continuously with low overhead, especially when request sampling is supported [10, 28, 31]. For example, Stardust [34], Ursa Minor's end-to-end tracing mechanism, adds 1% or less overhead when used with key benchmarks, such as SpecSFS [30].

End-to-end tracing implementations differ in two key respects: whether instrumentation is added automatically or manually and whether the request flows can disambiguate sequential and parallel activity. With regard to the latter, Magpie [4] and recent versions of both Stardust [34] and X-Trace [10] explicitly account for concurrency by embedding information about thread synchronization in their traces (see Figure 2). These implementations are a natural fit for request-flow comparison, as they can disambiguate true structural differences from false ones caused by alternate interleavings of concurrent activity. Whodunit [7], Pinpoint [9], and Dapper [31] do not account for parallelism.

End-to-end tracing in distributed systems is past the research stage. For example, it is used in production Google datacenters [31] and in some production three-tier systems [4]. Research continues, however, on how to best exploit the information provided by such tracing.

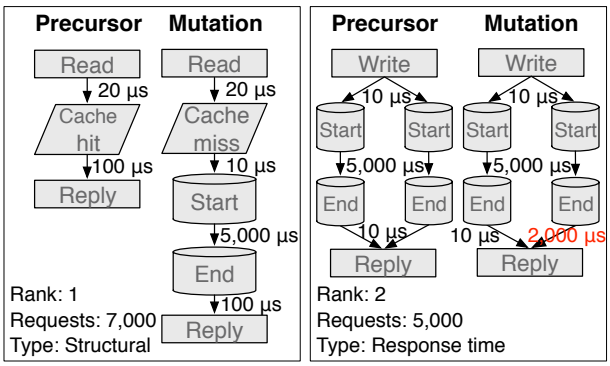

Figure 1: Example output from comparing request flows. The two mutations shown are ranked by their effect on the change in performance. The item ranked first is a structural mutation and the item ranked second is a response-time mutation. Due to space constraints, mocked-up graphs are shown in which nodes represent the type of component accessed.

3 Behavioural changes vs. anomalies

Our technique of comparing request flows between two periods identifies distribution changes in request-flow behaviour and ranks them according to their contribution to the observed performance difference. Conversely, anomaly detection techniques, as implemented by Magpie [4] and Pinpoint [9], mine a single period's request flows to identify rare ones that differ greatly from others. In contrast to request-flow comparison, which attempts to identify the most important differences between two sets, anomaly detection attempts to identify rare elements within a single set.

Request-flow comparison and anomaly detection serve distinct purposes, yet both are useful. For example, performance problems caused by changes in the components used (e.g., see Section 8.2), or by common requests whose response times have increased slightly, can be easily diagnosed by comparing request flows, whereas many anomaly detection techniques will be unable to provide guidance. In the former case, guidance will be difficult because the changed behaviour is common during the problem period; in the latter, because the per-request change is not extreme enough.

4 Spectroscope

To illustrate the utility of comparing request flows, this technique was implemented in a tool called Spectroscope and used to diagnose performance problems seen in Ursa Minor [1] and in certain Google services. This section provides an overview of Spectroscope, and the next describes its algorithms. Section 4.1 describes how *cate-*

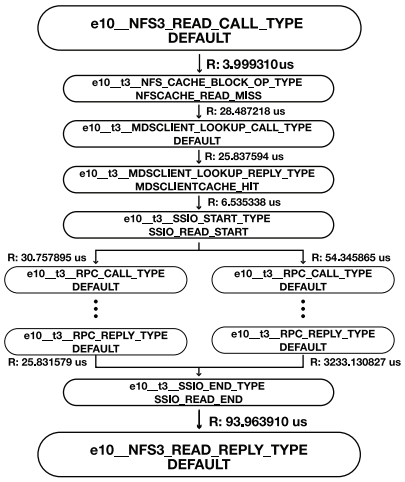

Figure 2: Example request-flow graph. The graph shows a striped READ in the Ursa Minor distributed storage system. Nodes represent trace points and edges are labeled with the time between successive events. Parallel substructures show concurrent threads of activity. Node labels are constructed by concatenating the machine name (e.g., e10), component name (e.g., NFS3), trace-point name (e.g., READ_CALL_TYPE), and an optional semantic label (e.g., NFSCACHE_READ_MISS). Due to space constraints, trace points executed on other components as a result of the NFS server's RPC calls are not shown.

gories, the basic building block on which Spectroscope operates, are constructed. Section 4.2 describes Spectroscope's support for comparing request flows.

4.1 Categorizing request flows

Even small distributed systems can service hundreds to thousands of requests per second, so comparing all of them individually is not feasible. Instead, exploiting a general expectation that requests that take the same path should incur similar costs, Spectroscope groups identically-structured requests into unique *categories* and uses them as the basic unit for comparing request flows. For example, requests whose structures are identical because they hit in a NFS server's data and metadata cache will be grouped into the same category, whereas requests that miss in both will be grouped in a different one. Two requests are deemed structurally identical if their string representations, as determined by a depth-first traversal, are identical. 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 them in alphabetical order, as determined by their root node names) and by the fact that parallelism is limited in most request flows we have observed.

For each category, Spectroscope identifies aggregate statistics, including request count, average response time, and variance. To identify where time is spent, it also computes average edge latencies and corresponding variances. Spectroscope displays categories in either a graph view, with statistical information overlaid, or within train-schedule visualizations [37] (also known as swim lanes), which more directly show the constituent requests' pattern of activity.

Spectroscope uses selection criteria to limit the number of categories developers must examine. For example, when comparing request flows, statistical tests and a ranking scheme are used. The number of categories could be further reduced by using unsupervised clustering algorithms, such as those used in Magpie [4], to bin similar but not necessarily identical requests into the same category. Initial versions of Spectroscope used off-the-shelf clustering algorithms [29], but we found the groups they created too coarse-grained and unpredictable. Often, they would group mutations and precursors within the same category, masking their existence. For clustering algorithms to be useful, improvements such as distance metrics that better align with developers' notions of request similarity are needed. Without them, use of clustering algorithms will result in categories composed of seemingly dissimilar requests.

4.2 Comparing request flows

Performance changes can result from a variety of factors, such as internal changes to the system that result in performance regressions, unintended side effects of changes to configuration files, or environmental issues. Spectroscope helps diagnose these problems by comparing request flows and identifying the key resulting mutations. Figure 3 shows Spectroscope's workflow.

When comparing request flows, Spectroscope takes as input request-flow graphs from two periods of activity, which we refer to as a *non-problem period* and a *problem period*. It creates categories composed of requests from both periods and uses statistical tests and heuristics to identify which contain structural mutations, response-time mutations, or precursors. Categories containing mutations are presented to the developer in a list ranked by expected contribution to the performance change. Note that the periods do not need to be aligned exactly with the performance change (e.g., at Google we often chose day-long periods based on historic average latencies).

Visualizations of categories that contain mutations are similar to those described previously, except per-period statistical information is shown. The root cause of response-time mutations is localized by showing the edges responsible for the mutation in red. The root cause of structural mutations is localized by providing a ranked list of the candidate precursors, so that the developer can determine how they differ. Figure 1 shows an example.

Spectroscope provides further insight into perfor-

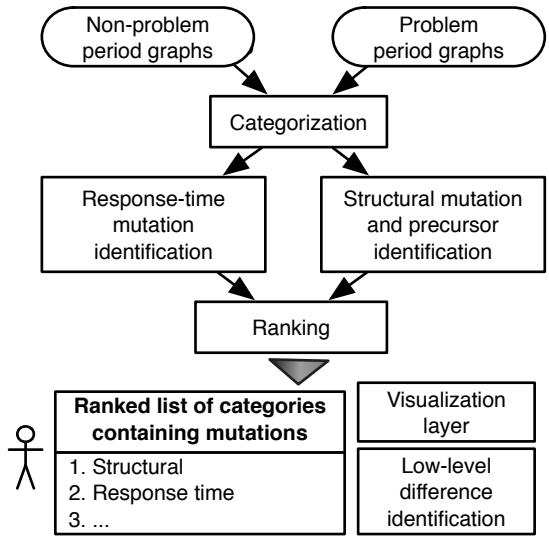

Figure 3: Spectroscope's workflow for comparing request flows. First, Spectroscope groups requests from both periods into categories. Second, it identifies which categories contain mutations or precursors. Third, it ranks mutation categories according to their expected contribution to the performance change. Developers are presented this ranked list. Visualizations of mutations and their precursors can be shown. Also, low-level differences can be identified for them.

mance changes by identifying the low-level parameters (e.g., client parameters or function call parameters) that best differentiate a chosen mutation and its precursors. For example, in Ursa Minor, one performance slowdown, which manifested as many structural mutations, was caused by a change in a parameter sent by the client. For problems like this, highlighting the specific low-level differences can immediately identify the root cause.

Section 5 describes Spectroscope's algorithms and heuristics for identifying mutations, their corresponding precursors, their rank based on their relative influence on the overall performance change, and their most relevant low-level parameter differences. It also describes how these methods overcome key challenges—for example, differentiating true mutations from natural variance in request structure and timings. Identification of response-time mutations and ranking rely on the expectation (reasonable for many distributed systems, including distributed storage) that requests that take the same path through a distributed system will exhibit similar response times and edge latencies. Section 7 describes how high variance in this axis affects Spectroscope's results.

5 Algorithms for comparing request flows

This section describes the key heuristics and algorithms used when comparing request flows. In creating them, we favoured simplicity and those that regulate false positives—perhaps the worst failure mode due to developer effort wasted—whenever possible.

5.1 Identifying response-time mutations

When comparing two periods, there will always be some natural differences in timings. Spectroscope uses the Kolmogorov-Smirnov two-sample, non-parametric hypothesis test [20] to differentiate natural variance from true changes in distribution or behaviour. Statistical hypothesis tests take as input two distributions and output a p-value, which represents uncertainty in the claim that the null hypothesis, that both distributions are the same, is false. Expensive false positives are limited to a preset rate (almost always 5%) by rejecting the null hypothesis only when the p-value is lower than this value. The p-value increases with variance and decreases with the number of samples. A non-parametric test, which does not require knowledge of the underlying distribution, is used because we have observed that response times are not governed by well-known distributions.

The Kolmogorov-Smirnov test is used as follows. For each category, the distributions of response times for the non-problem period and the problem period are extracted and input into the hypothesis test. The category is marked as containing response-time mutations if the test rejects the null hypothesis. By default, categories that contain too few requests to run the test accurately are not marked as containing mutations. To identify the components or interactions responsible for the mutation, Spectroscope extracts the critical path—i.e., the path of the request on which response time depends—and runs the same hypothesis test on the edge latency distributions. Edges for which the null hypothesis is rejected are marked in red in the final output visualization.

5.2 Identifying structural mutations

To identify structural mutations, Spectroscope assumes a similar workload was run in both the non-problem period and the problem period. As such, it is reasonable to expect that an increase in the number of requests that take one path through the distributed system in the problem period should correspond to a decrease in the number of requests that take other paths. Since non-determinism in service order dictates that per-category counts will always vary slightly between periods, a threshold is used to identify categories that contain structural mutations and precursors. Categories that contain `SM_THRESHOLD` more requests from the problem period than from the non-problem period are labeled as containing mutations and those that contain `SM_THRESHOLD` fewer are labeled as containing precursors.

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 they will be given a low rank and so will not mislead the developer in his diagnosis efforts.

If per-category distributions of request counts are available, a statistical test, instead of a threshold, could be used to determine those categories that contain mutations or precursors. This statistical approach would be superior to a threshold-based approach, as it guarantees a set false-positive rate. However, building the distributions necessary would require obtaining many non-problem and problem-period datasets, so we opted for the simpler threshold-based approach instead. Also, our experiences at Google indicate that request structure within large datacenters may change too quickly for such expensive-to-build models to be useful.

5.3 Mapping mutations to precursors

Once the total set of categories that contain structural mutations and precursors has been identified, Spectroscope must iterate through each structural-mutation category to determine the precursor categories that are likely to have donated requests to it. This is accomplished via three heuristics, described below. Figure 4 shows how they are applied.

First, the total list of precursor categories is pruned to eliminate categories with a different root node than those in the structural-mutation category. 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.

Second, remaining precursor categories that have decreased in request count less than the increase in request count of the structural-mutation category are also removed from consideration. This 1:N heuristic reflects the common case that one precursor category is likely to donate requests to N structural-mutation categories. For example, a cache-related problem may result in a portion of requests that used to hit in that cache to miss and hit in the next-level cache. Extra cache pressure at this next-level cache may result in the rest missing in both caches. This heuristic can be optionally disabled.

Third, the remaining precursor categories are ranked according to their likelihood of having donated requests, as determined by the string-edit distance between them and the structural-mutation category. This heuristic reflects the intuition that precursors and structural mutations are likely to resemble each other in structure. The cost of computing the edit distance is O(NM), where N and M are the lengths of the string representations of the categories being compared.

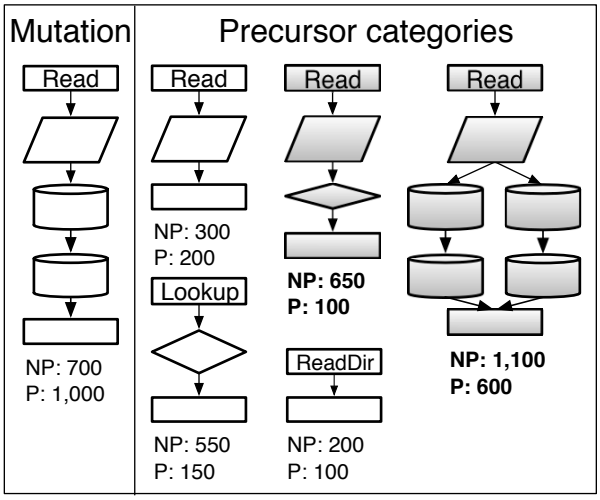

Figure 4: How the precursor categories of a structural-mutation category are identified. One structural-mutation category and five precursor categories are shown, each with their corresponding request counts from the non-problem (NP) and problem (P) periods. For this case, the shaded precursor categories 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.

5.4 Ranking

Ranking of mutations is necessary for two reasons. First, the performance problem might have multiple root causes, each of which causes its own set of mutations. Second, even if there is only one root cause to the problem (e.g., a misconfiguration), many mutations will often still be observed. For both cases, it is useful to identify the mutations that most affect performance in order to focus diagnosis effort where it will yield the most benefit.

Spectroscope ranks categories that contain mutations in descending order by their expected contribution to the performance change. The contribution for a structural-mutation category is calculated as the number of mutations it contains, which is the difference between its problem and non-problem period counts, multiplied by the difference in problem period average response time between it and its precursor categories. If more than one candidate precursor category has been identified, a weighted average of their average response times is used; weights are based on structural similarity to the mutation. The contribution for a response-time-mutation category is calculated as the number of mutations it contains, which is just the non-problem period count, times the change in average response time of that category be-

tween the periods. If a category contains both response-time mutations and structural mutations, it is split into two virtual categories and each is ranked separately.

5.5 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 [5] 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 belonging to the problem or non-problem period. 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 [25], which creates the regression tree. To reduce the 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%. Parameters only need to be extracted the first time the algorithm is run; subsequent iterations can modify the table directly.

5.6 Current limitations

This section describes current limitations with our techniques for comparing request flows.

Diagnosing problems caused by contention: Our techniques assume that performance changes are caused by changes to the system (code changes, configuration changes, etc). Though they will identify mutations caused by contention, they cannot directly attribute them to the responsible process. In some cases our techniques can indirectly help—for example, by showing that many edges within a component are responsible for a response-time mutation, they can help the developer intuit that the problem is due to contention with an external process.

Diagnosing problems when the load differs significantly between periods: In such cases, the load change itself may be the root cause. Though our techniques will identify response-time and structural changes when the load during the problem period is much greater than the non-problem period, the developer must determine whether they are reasonable degradations.

6 Experimental apparatus

Most of the experiments and case studies reported in this paper come from using Spectroscope with a distributed storage service called Ursa Minor. Section 6.1 describes this system. Section 6.2 describes the benchmarks used for Ursa Minor's nightly regression tests, the setting in which many of the case studies were observed.

To understand issues in scaling request-flow comparison to larger systems, we also used Spectroscope to diagnose services within Google. Section 6.3 provides more details. The implementation of Spectroscope for Ursa Minor was written in Perl and MATLAB. It includes a visualization layer built upon Prefuse [16]. The cost of calculating edit distances dominates its runtime, so it is sensitive to the value of `SM_THRESHOLD` used. The implementation for Google was written in C++; its runtime is much lower (on the order of seconds) and its visualization layer uses DOT [15].

6.1 Ursa Minor

Figure 5 illustrates Ursa Minor's architecture. Like most modern scalable distributed storage, Ursa Minor separates metadata services from data services, such that clients can access data on storage nodes without moving it all through metadata servers. An Ursa Minor instance (called a "constellation") consists of potentially many NFS servers (for unmodified clients), storage nodes (SNs), metadata servers (MDSs), and end-to-end-trace servers. To access data, clients must first send a request to a metadata server asking for the appropriate permissions and locations of the data on the storage nodes. Clients can then access the storage nodes directly.

Ursa Minor has been in active development since 2004 and comprises 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].

The components of Ursa Minor are usually run on separate machines within a datacenter. Though Ursa Minor supports an arbitrary number of components, the experi-

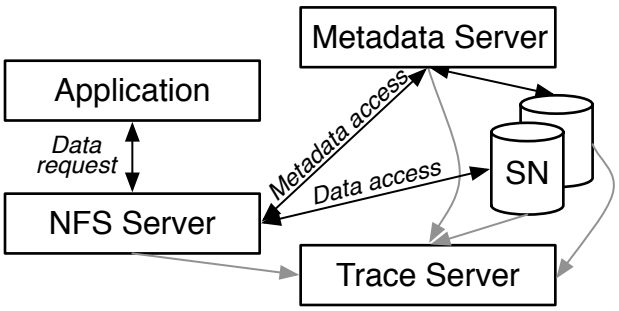

Figure 5: Ursa Minor Architecture. Ursa Minor can be deployed in many configurations, with an arbitrary number of NFS servers, metadata servers, storage nodes (SNs), and trace servers. Here, a simple five-component configuration is shown.

ments and case studies detailed in this paper use a simple five-machine configuration: one NFS server, one metadata server, one trace server, and two storage nodes. One storage node stores data, while the other stores metadata. Not coincidentally, this is the configuration used in the nightly regression tests that uncovered many of the problems described in the case studies.

End-to-end tracing infrastructure via Stardust: Ursa Minor's Stardust tracing infrastructure is much like its peer group, discussed in Section 2. Request sampling is used to capture trace data for a subset of entire requests (10% by default), with a per-request decision made randomly when the request enters the system. Ursa Minor contains approximately 200 trace points, 124 manually inserted as well as automatically generated ones for each RPC send and receive function. In addition to simple trace points, which indicate points reached in the code, explicit split and join trace points are used to identify the start and end of concurrent threads of activity. Low-level parameters are also collected at trace points.

6.2 Benchmarks used with Ursa Minor

Experiments run on Ursa Minor use these benchmarks.

`Linux-build` and `ursa minor-build`: These benchmarks 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. About 145,000 requests are sampled. The average graph size and standard deviation is 12 and 40 nodes. Most graphs are small, but some are very big, so the per-category equivalents are larger: 160 and 500 nodes. `Ursa minor-build` runs for 10 minutes. About 16,000 requests are sampled and the average graph size and standard deviation is 9 and 28 nodes. The per-category equivalents are 96 and 100 nodes.

`Postmark-large`: This synthetic benchmark evaluates the small file performance of storage systems [19]. It utilizes 448 subdirectories, 50,000 transactions, and 200,000 files and runs for 80 minutes. The average graph size and standard deviation is 66 and 65 nodes. The per-category equivalents are 190 and 81 nodes.

`SPEC SFS 97 V3.0 (SFS97)`: This synthetic benchmark is the industry standard for measuring NFS server scalability and performance [30]. 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. The average graph size and standard deviation is 30 and 51 nodes. The per-category equivalents are 206 and 200 nodes.

`IoZone`: This benchmark [23] sequentially writes, rewrites, reads, and re-reads a 5GB file in 20 minutes. The average graph size and standard deviation is 6 nodes. The per-category equivalents are 61 and 82 nodes.

6.3 Dapper & Google services

The Google services for which Spectroscope was applied were instrumented using Dapper, which automatically embeds trace points in Google's RPC framework. Like Stardust, Dapper employs request sampling, but uses a sampling rate of less than 0.1%. Spectroscope was implemented as an extension to Dapper's aggregation pipeline, which groups individual requests into categories and was originally written to support Dapper's pre-existing analysis tools. Categories created by the aggregation pipeline only show compressed call graphs with identical children and siblings merged together.

7 Dealing with high-variance categories

For automated diagnosis tools to be useful, it is important that distributed systems satisfy certain properties about variance. For Spectroscope, categories that exhibit high variance in response times and edge latencies do not satisfy the expectation that "requests that take the same path should incur similar costs" and can affect its ability to identify mutations accurately. Spectroscope's ability to identify response-time mutations is sensitive to variance, whereas for structural mutations only the ranking is affected. Though categories may exhibit high 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 latent performance problems. For example, in early versions of Ursa Minor, several high-variance categories resulted from a poorly written hash table that exhibited slowly increasing lookup times because of a poor hashing scheme.

For response-time mutations, both false negatives and false positives will increase with the number of high-variance categories. False negatives will increase

because high variance will reduce the Kolmogorov-Smirnov test's power to differentiate true behaviour changes from natural variance. False positives, which are much rarer, will increase when it is valid for categories to exhibit similar response times within a single period, but different response times across different ones. The rest of this section concentrates on the false negative case.

To quantify how well categories meet the same path/similar costs expectation within a single period, Figure 6 shows a CDF of the squared coefficient of variation in response time (C^2) for *large categories* induced by `linux-build`, `postmark-large`, and `SFS97` in Ursa Minor. Figure 7 shows the same C^2 CDF for large categories induced by Bigtable [8] running in three Google datacenters over a 1-day period. Each Bigtable instance is shared among the machines in its datacenter and services several workloads. 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 1 and 2 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 benchmarks 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 benchmarks show that part of the observed variance is a result of contention for locks in the metadata server.

The traces collected for Bigtable by Dapper are relatively sparse—often graphs generated for it are composed of only a few nodes, with one node showing the incoming call type (e.g., READ, MUTATE, etc.) and another showing the call type of the resulting GFS [13] request. As such, many dissimilar paths cannot be disambiguated and have been merged together in the observed categories. Even so, 47–69% of all categories exhibit C^2 less than 1. Additional instrumentation, such as those that show the sizes of Bigtable data requests and work done within GFS, would serve to further disambiguate unique paths and considerably reduce C^2.

8 Ursa Minor case studies

Spectroscope is not designed to replace developers; rather it is intended to serve as an important step in the workflow they use to diagnose problems. Sometimes it can help developers identify the root cause immediately, or at least localize the problem to a specific area of the system. In other cases, it can help eliminate the distributed system as the root cause by verifying that its behaviour has not changed, allowing developers to focus their efforts on external factors.

This section presents diagnoses of six performance problems solved by using Spectroscope to compare request flows and analyzes its effectiveness in identifying the root causes. Most of these problems were previously unsolved and diagnosed by the authors without knowledge of the root cause. One problem was observed before Spectroscope was available, so it was re-injected to show how effectively it could have been diagnosed. By introducing a synthetic spin loop of different delays, we also demonstrate Spectroscope's ability to diagnose changes in response time.

8.1 Methodology

Three complementary metrics are provided for evaluating Spectroscope's output.

The percentage of the 10 highest-ranked categories that are relevant: This metric measures the quality of the rankings of the results. It accounts for the fact that developers will naturally investigate the highest-ranked categories first, so it is important for them to be relevant.

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

Request coverage: This metric evaluates quality of the ranked list by identifying the percentage of requests affected by the problem that are identified in it.

Table 3 summarizes Spectroscope's performance using these metrics. Unless otherwise noted, a default value of 50 was used for `SM_THRESHOLD`. We chose this value to yield reasonable runtimes (between 15-30 minutes) when diagnosing problems in larger benchmarks, such as `SFS97` and `postmark-large`. When necessary, it was lowered to further explore the space of possible structural mutations.

8.2 MDS configuration change

After a particular large code check-in, performance of `postmark-large` decayed significantly, from 46tps to 28tps. To diagnose this problem, we 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. Comparing them to their most-likely precursor categories 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. We

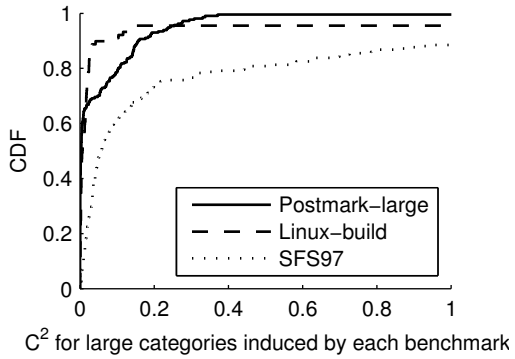

Figure 6: CDF of C^2 for large categories induced by three benchmarks run on Ursa Minor. At least 88% of the categories induced by each benchmark exhibit low variance ($C^2 < 1$). The results for `linux-build` and `SFS` are more heavy-tailed than `postmark-large`, partly due to extra lock contention in the metadata server.

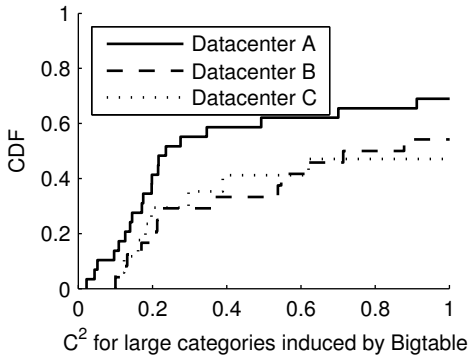

Figure 7: CDF of C^2 for large categories induced by Bigtable instances in three Google datacenters. Dapper's instrumentation of Bigtable is sparse, so many paths cannot be disambiguated and have been merged together in the observed categories, resulting in a higher C^2 than otherwise expected. Even so, 47–69% of categories exhibit low variance.

	Benchmark		
	`Linux-bld`	`Postmark`	`SFS97`
Categories	351	716	1602
Large categories (%)	25.3	29.9	14.7
Requests sampled	145,167	131,113	210,669
In large categories (%)	99.7	99.2	98.9

Table 1: Distribution of requests in the categories induced by three benchmarks run on Ursa Minor. Though many categories are generated, most contain only a small number of requests. *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 2: 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—a small number of categories contain most requests.

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).

We presented this information to the developer of the metadata server, who told us 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 developers 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 various configuration options affect system behaviour.

Quantitative analysis: For the evaluation in Table 3, results in the ranked list were deemed relevant if they included metadata accesses to the data storage node with a most-likely precursor category that included metadata accesses to the metadata storage node.

8.3 Read-modify-writes

This problem was observed and diagnosed before development on Spectroscope began; it was re-injected in Ursa Minor to show how Spectroscope could have helped developers easily diagnose 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. To debug this problem, one of the authors of this paper spent several days manually mining Stardust traces and eventually discovered the

# / Type	Name	Manifestation	Root cause	# of results	Quality of results		
					Top 10 rel. (%)	FPs (%)	Cov. (%)
8.2 / Real	MDS config.	Structural	Config. change	128	100	2	70
8.3 / Real	RMWs	Structural	Env. change	3	100	0	100
8.4 / Real	MDS prefetch. 50	Structural	Internal change	7	29	71	93
8.4 / Real	MDS prefetch. 10			16	70	56	96
8.5 / Real	Create behaviour	Structural	Design problem	11	40	64	N/A
8.6 / Synthetic	100μs delay	Response time	Internal change	17	0	100	0
8.6 / Synthetic	500μs delay			166	100	6	92
8.6 / Synthetic	1ms delay			178	100	7	93
8.7 / Real	Periodic spikes	No change	Env. change	N/A	N/A	N/A	N/A

Table 3: 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.

root cause: 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, Spectroscope was used 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 results in the ranked list were structural-mutation categories that contained RMWs.

We next used Spectroscope to identify the low-level parameter differences between the highest-ranked result and its most-likely precursor category. The output perfectly separated the constituent requests by the `count` parameter, which specifies the amount of data to be read or written by the request. Specifically, requests with `count` parameter values less than or equal to 4KB were classified as belonging to the problem period.

Summary: Diagnosis of this problem demonstrates how comparing request flows can help developers 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, results in the ranked list were deemed relevant if they contained RMWs and their most-likely precursor category did not.

8.4 MDS prefetching

A few years ago, several developers, including one of the authors of this paper, tried to add *server-driven metadata prefetching* to Ursa Minor [17]. This feature was intended to improve performance by prefetching metadata to clients on every mandatory metadata server access, in hopes of minimizing the total number of accesses necessary. However, when implemented, this feature provided no improvement. 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, we compared two runs of `linux-build`, one with prefetching disabled and another with it enabled. `linux-build` was chosen because it is more likely to see performance improvements due to prefetching than the other workloads.

When we ran Spectroscope with `SM_THRESHOLD` set to 50, several categories were identified as containing mutations. The two highest-ranked results immediately piqued our 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, we decreased `SM_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, we added instrumentation around the prefetching function to see whether it caused the database accesses. Altogether, this information provided us with the intuition necessary to determine why server-driven metadata prefetching did not improve performance: the extra time spent in the DB calls by metadata server accesses outweighed the time savings gener-

ated 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, in this case involving developers iteratively modifying SM_THRESHOLD to gain additional insight.

Quantitative analysis: For Table 3, 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_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.

8.5 Create behaviour

During development of Ursa Minor, we noticed that the distribution of request response times for CREATEs in postmark-large increased significantly during the course of the benchmark. To diagnose this performance degradation, we 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, SM_THRESHOLD was set 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 8.

Conversations with the metadata server's developer led us 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, results in the ranked list were deemed relevant if 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.

8.6 Slowdown due to code changes

This synthetic problem was injected 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.

Spectroscope was used 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 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 93%.

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

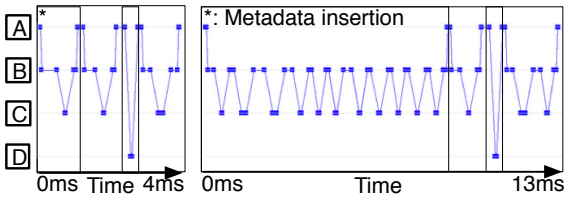

Figure 8: Visualization of create behaviour. Two train-schedule visualizations are shown, the first one a fast early create during postmark-large and the other a slower create issued later in the benchmark. Messages are exchanged between the NFS Server (A), Metadata Server (B), Metadata Storage Node (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.

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

8.7 Periodic spikes

`Ursa minor-build`, which is run as part of the nightly 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%. We 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, we 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. We 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. We probed the issue further by reducing `SM_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, we 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 us the problem was not due to Ursa Minor or processes running on its machines.

We 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 (Figure 9). The visualization shows that during the problem period, response times stay constant but the arrival rate of requests decreases. We currently 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.

9 Experiences at Google

This section describes preliminary experiences using request-flow comparison, as implemented in Spectroscope, to diagnose performance problems within select Google services. Sections 9.1 and 9.2 describe two such experiences. Section 9.3 discusses ongoing challenges in adapting request-flow comparison to large datacenters.

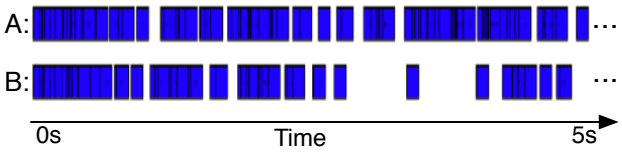

Figure 9: Timeline of inter-arrival times of requests at the NFS Server. A 5s sample of requests, where each rectangle represents the process time of a request, reveals long periods of inactivity due to lack of requests from the client during spiked copy times (B) compared to periods of normal activity (A).

9.1 Inter-cluster performance

A team responsible for an internal service at Google observed that load tests run on their software in two different clusters exhibited significantly different performance, though they expected performance to be similar.

We 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 us to hypothesize that the primary cause of the slowdown was a problem in the cluster in which the slower load test was run. Later, we found out that the Bigtable instance running in the slower cluster was not working properly, confirming our hypothesis. 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.

9.2 Performance change in a large service

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

9.3 Ongoing challenges with scale

Challenges remain in scaling request-flow comparison techniques to large distributed services, such as those within Google. For example, categories generated for well-instrumented large-scale distributed services will be much larger than those observed for the 5-instance version of Ursa Minor. Additionally, they may yield many categories, each populated with too few requests for sta-

tistical rigor. Robust methods are needed to merge categories and visualize them without losing important information about structure, which occurs with Dapper because of its graph compression methods. These methods affected the quality of Spectroscope's results by increasing variance, losing important structural differences between requests, and increasing effort needed to understand individual categories. Our experiences with unsupervised learning algorithms, such as clustering [4, 29], for merging categories indicate they are inadequate. A promising alternative is to use semi-supervised methods, which would allow the grouping algorithm to learn developers' mental models of which categories should be merged. Also, efficient visualization may be possible by only showing the portion of a mutation's structure that differs between it and its precursors.

More generally, request-flow graphs from large services are difficult to understand because such services contain many dependencies, most of which are foreign to their developers. To help, tools such as Spectroscope must strive to identify the semantic meaning of individual categories. For example, they could ask developers to name graph substructures about which they are knowledgeable and combine them into a meaningful meta-name when presenting categories.

10 Related work

A number of techniques have been developed for diagnosing performance problems in distributed systems. Whereas many rely on end-to-end tracing, others attempt to infer request flows from existing data sources, such as message send/receive events [27] or logs [38]. These latter techniques trade accuracy of re-constructed request flows for ease of using existing monitoring mechanisms. Other techniques rely on black-box metrics and are limited to localizing problems to individual machines.

Magpie [4], Pinpoint [9], WAP5 [27], and Xu [38], all identify anomalous requests by finding rare ones that differ greatly from others. In contrast, request-flow comparison identifies the changes in distribution between two periods that most affect performance. Pinpoint also describes other ways to use end-to-end traces, including for statistical regression testing, but does not describe how to use them to compare request flows.

Google has developed several analysis tools for use with Dapper [31]. Most relevant is the Service Inspector, which shows graphs of the unique call paths observed to a chosen function or component, along with the resulting call tree below it, allowing developers to understand the contexts in which the chosen item is used. Because the item must be chosen beforehand, the Service Inspector is not a good fit for problem localization tasks.

Pip [26] compares developer-provided, component-based expectations of structural and timing behaviour to actual behaviour observed in end-to-end traces. Theoretically, Pip can be used to diagnose any type of problem: anomalies, correctness problems, etc. But, it relies on developers to specify expectations, which is a daunting and error-prone task—the developer is faced with balancing effort and generality against the specificity needed to expose particular problems. In addition, Pip's component-centric expectations, as opposed to request-centric ones, complicate problem localization tasks [10]. Nonetheless, in many ways, comparing request flows between executions is akin to Pip, with developer-provided expectations being replaced with the observed non-problem period behaviour. Many of our algorithms, such as for ranking mutations and highlighting the differences, could be used with Pip-style expectations as well.

The Stardust tracing infrastructure on which our implementation builds was originally designed to enable performance models to be induced from observed system performance [32, 34]. Building on that initial work, IRONmodel [33] developed approaches to detecting (and correcting) violations of such models, which can indicate performance problems. In describing IRONmodel, Thereska et al. also proposed that the specific nature of how observed behaviour diverges from the model could guide diagnoses, but they did not develop techniques for doing so or explore the approach in depth.

A number of black-box diagnosis techniques have been devised for systems that do not have the detailed end-to-end tracing on which our approach to comparing request flows relies. For example, Project 5 [2] infers bottlenecks by observing messages passed between components. Comparison of performance metrics exhibited by systems that should be doing the same work can also identify misbehaving nodes [18, 24]. Such techniques can be useful parts of a suite, but are orthogonal to the contributions of this paper.

There are also many single-process diagnosis tools that inform creation of techniques for distributed systems. For example, Delta analysis [36] compares multiple failing and non-failing runs to identify the most significant differences. OptiScope [22] compares the code transformations made by different compilers to help developers identify important differences that affect performance. DARC [35] automatically profiles system calls to identify the greatest sources of latency. Our work builds on some concepts from such single-process techniques.

11 Conclusion

Comparing request flows, as captured by end-to-end traces, is a powerful new technique for diagnosing performance changes between two time periods or system versions. Spectroscope's algorithms for this comparison allow it to accurately identify and rank mutations and identify their precursors, focusing attention on the

most important differences. Experiences with Spectroscope confirm its usefulness and efficacy.

Acknowledgements

We thank our shepherd (Lakshminarayanan Subramanian), the NSDI reviewers, Brian McBarron, Michelle Mazurek, Matthew Wachs, and Ariela Krevat for their insight and feedback. We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NEC Laboratories, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Yahoo! Labs) for their interest, insights, feedback, and support. This research was sponsored in part by a Google research award, NSF grants #CNS-0326453 and #CCF-0621508, by DoE award DE-FC02-06ER25767, and by CyLab under ARO grants DAAD19-02-1-0389 and W911NF-09-1-0273.

References

[1] M. Abd-El-Malek, et al. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies. USENIX Association, 2005. 2, 6

[2] M. K. Aguilera, et al. Performance debugging for distributed systems of black boxes. ACM Symposium on Operating System Principles. ACM, 2003. 13

[3] Anonymous. Personal communication with Google Software Engineers, December 2010. 1

[4] P. Barham, et al. Using Magpie for request extraction and workload modelling. Symposium on Operating Systems Design and Implementation. USENIX Association, 2004. 1, 2, 3, 13

[5] C. M. Bishop. *Pattern recognition and machine learning*, first edition. Springer Science + Business Media, LLC, 2006. 6

[6] B. M. Cantrill, et al. Dynamic instrumentation of production systems. USENIX Annual Technical Conference. USENIX Association, 2004. 1

[7] A. Chanda, et al. Whodunit: Transactional profiling for multi-tier applications. EuroSys. ACM, 2007. 1, 2

[8] F. Chang, et al. Bigtable: a distributed storage system for structured data. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006. 8

[9] M. Y. Chen, et al. Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation. USENIX Association, 2004. 1, 2, 13

[10] R. Fonseca, et al. Experiences with tracing causality in networked services. Internet Network Management Conference on Research on Enterprise Networking. USENIX Association, 2010. 2, 13

[11] R. Fonseca, et al. X-Trace: a pervasive network tracing framework. Symposium on Networked Systems Design and Implementation. USENIX Association, 2007. 1, 2

[12] GDB. http://www.gnu.org/software/gdb/. 1

[13] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003. 8

[14] S. L. Graham, et al. gprof: a call graph execution profiler. ACM SIGPLAN Symposium on Compiler Construction. Published as *SIGPLAN Notices*, **17**(6):120–126, June 1982. 1

[15] Graphviz. http://www.graphviz.org. 6

[16] J. Heer, et al. Prefuse: a toolkit for interactive information visualization. Conference on Human Factors in Computing Systems. ACM, 2005. 6

[17] J. Hendricks, et al. *Improving small file performance in object-based storage*. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006. 10

[18] M. P. Kasick, et al. Black-box problem diagnosis in parallel file systems. Conference on File and Storage Technologies. USENIX Association, 2010. 13

[19] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997. 7

[20] F. J. Massey, Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, **46**(253):66–78, 1951. 4

[21] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. EuroSys. ACM, 2006. 1

[22] T. Moseley, et al. OptiScope: performance accountability for optimizing compilers. International Symposium on Code Generation and Optimization. IEEE/ACM, 2009. 13

[23] W. Norcott and D. Capps. IoZone filesystem benchmark program, 2002. http://www.iozone.org. 7

[24] X. Pan, et al. Ganesha: black-box fault diagnosis for MapReduce systems. Hot Metrics. ACM, 2009. 13

[25] J. R. Quinlan. Bagging, boosting and C4.5. 13th National Conference on Artificial Intelligence. AAAI Press, 1996. 6

[26] P. Reynolds, et al. Pip: Detecting the unexpected in distributed systems. Symposium on Networked Systems Design and Implementation. USENIX Association, 2006. 1, 13

[27] P. Reynolds, et al. WAP5: Black-box Performance Debugging for Wide-Area Systems. International World Wide Web Conference. ACM Press, 2006. 13

[28] R. R. Sambasivan, et al. *Diagnosing performance problems by visualizing and comparing system behaviours*. Technical report 10–103. Carnegie Mellon University, February 2010. 2

[29] R. R. Sambasivan, et al. Categorizing and differencing system behaviours. Workshop on hot topics in autonomic computing (HotAC). USENIX Association, 2007. 3, 13

[30] SPEC SFS97 (2.0). http://www.spec.org/sfs97. 2, 7

[31] B. H. Sigelman, et al. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical report dapper-2010-1. Google, April 2010. 1, 2, 13

[32] E. Thereska, et al. Informed data distribution selection in a self-predicting storage system. International conference on autonomic computing. IEEE, 2006. 13

[33] E. Thereska and G. R. Ganger. IRONModel: robust performance models in the wild. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008. 1, 13

[34] E. Thereska, et al. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2006. 1, 2, 13

[35] A. Traeger, et al. DARC: Dynamic analysis of root causes of latency distributions. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008. 13

[36] J. Tucek, et al. Triage: diagnosing production run failures at the user's site. ACM Symposium on Operating System Principles, 2007. 13

[37] E. R. Tufte. *The visual display of quantitative information*. Graphics Press, Cheshire, Connecticut, 1983. 3

[38] W. Xu, et al. Detecting large-scale system problems by mining console logs. ACM Symposium on Operating System Principles. ACM, 2009. 13