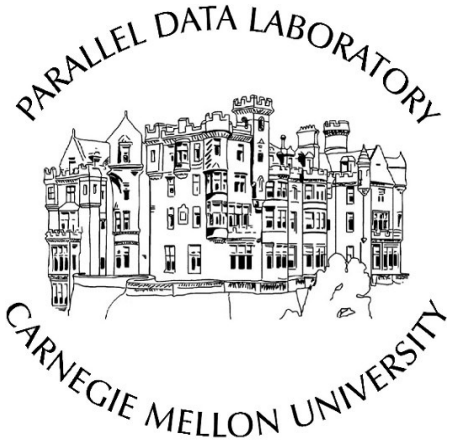


# Diagnosing performance changes by comparing system behaviours

Raja R. Sambasivan*, Alice X. Zheng†,
Elie Krevat*, Spencer Whitman*, Michael Stroucken*,
William Wang*, Lianghong Xu*, Gregory R. Ganger*
**Carnegie Mellon University, †Microsoft Research*

CMU-PDL-10-107
July 2010

**Parallel Data Laboratory**
Carnegie Mellon University
Pittsburgh, PA 15213-3890

## 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 system behaviours 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 described and evaluated. Five case studies are presented of using Spectroscope to diagnose performance changes in a distributed storage system caused by code changes and configuration modifications, demonstrating the value and efficacy of comparing system behaviours.*

**Acknowledgements:** 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, Seagate, 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 grant DAAD19-02-1-0389.

**Keywords:** browsing & visualizing system behaviour, comparing system behaviours, end-to-end tracing, performance debugging, performance problem diagnosis, response-time mutations, request-flow graphs, statistical hypothesis testing, structural mutations, structural performance problems

# 1 Introduction

Diagnosing performance problems in distributed systems is hard. 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 [23]. A suite of debugging tools will most likely be 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 [5], gprof [12], and GDB [11]), 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., [3, 7, 10, 32, 35]) that can capture 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 [3], spotting large-scale departures from performance models [34], and comparing observed behaviour to manually-constructed expectations [27].

This paper develops a new technique for the suite: comparing request flows between two executions. Such comparison can guide understanding of *changes* in performance, as contrasted with determining why a given system's performance has always been poor. For example, comparing request flows can help diagnose performance changes resulting from changes 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, workload change, etc. Our analysis of bug tracking data for a distributed storage system indicates that over half of the reported performance problems would benefit from guidance provided by comparing request flows.

The utility of comparing request flows relies on the observation that performance changes often manifest in 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 give insight into its 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 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 candidate precursors.

This paper also describes our experiences with implementing and using request flow comparison in a toolset called Spectroscope[1] with a distributed storage system called Ursa Minor [1]. In addition to comparing request flows, Spectroscope also includes support for browsing request flows within a single period, in order to find ones that are always slow or design problems.

We have used Spectroscope to diagnoase several recent problems and previously undiagnosed ones

[1] Spectroscopes are used to analyze the composition of astronomical bodies, for example the Ursa Minor constellation.

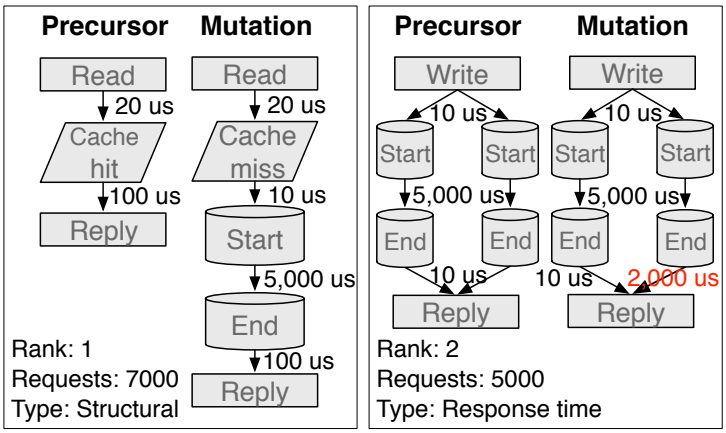


Figure 1: **Example output from comparing system behaviours.** Mutations ranked by their affect on the change in performance. The item ranked first is a structural mutation, whereas the one ranked second is a response-time mutation. For spacing reasons, mocked-up graphs are shown in which nodes represent the type of component accessed.

without prior knowledge of the root cause. Via description of four such problems and one synthetic problem, we illustrate the efficacy of comparing request flows and show that our algorithms enable effective use of this technique. The utility of browsing request flows within a single period is also shown via a description of a previously undiagnosed problem that was solved using this technique.

**Contributions of this paper:** (1) A new technique for diagnosing distributed system performance changes: comparing request flows. (2) Algorithms for identifying mutations and their most likely precursors. (3) A mechanism for ranking mutations by their contribution to the overall performance change. (4) Mechanisms for highlighting divergences between candidates and their precursors and for identifying the low-level parameter differences (i.e., in function calls or those sent by clients) that most strongly correlate with the changed behaviour. (5) Reported experiences of actual use by developers of support for this new technique in diagnosing *real* problems observed in a real distributed system.

## 2 Background and Related Work

This section discusses related work. In particular, comparing request flows builds on end-to-end tracing (2.1) and complements existing performance diagnosis techniques (2.2).

### 2.1 End-to-end request flow tracing

End-to-end tracing offers an invaluable information source, capturing a distributed system's performance and control flow in detail. Such tracing consists of storing *activity records* at each of various trace points in the distributed system's software, with each such record identifying the specific trace point, the current time, and other context information. Several efforts (e.g., [3, 7, 10, 32, 35] have independently implemented such tracing and shown that it can be used continuously with low overhead, especially given appropriate support for sampling [32, 29]—for example, Stardust [35], Ursa Minor's end-to-end tracing mechanism, adds 1% or less overhead when used with key benchmarks, such as SpecSFS [31]. Most of these implementations work by propagating an explicit *breadcrumb*, which allows activity records of trace points reached to be associated with specific requests. Most end-to-end tracing solutions require developers to add trace points at key areas

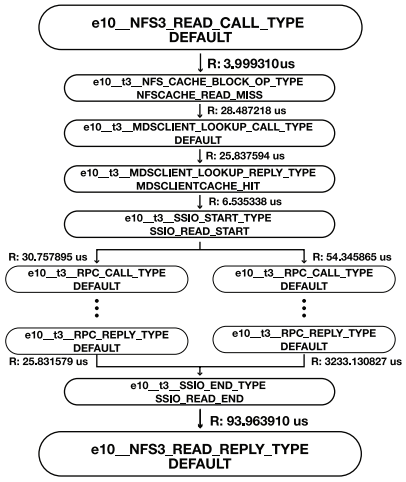


Figure 2: **Example request-flow graph.** The graph shows a striped READ in the Ursa Minor distributed storage system. Nodes represent trace points and edges are labeled with the time between successive events. Node labels are constructed by concatenating the machine name (e.g., `e10`), component name (e.g., `NFS3`), instrumentation 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.

of interest [3, 7, 10, 35], which can be mitigated by embedding trace points in common RPC layers, while a select few automatically capture function names [6]. Either offline or online, activity records can be stitched together (e.g., by using the breadcrumbs) to yield request-flow graphs, which show the detailed flow and timing of individual requests. Figure 2 shows an example request-flow graph generated by Stardust.

End-to-end tracing in distributed systems is past the research stage. For example, it is used in production Google datacenters [32] and in some production three-tier systems [3]. Research continues, however, on how to best exploit the information provided by such tracing.

## 2.2 Performance diagnosis techniques

A number of techniques have been developed, especially recently, for diagnosing performance problems in distributed systems. Many rely on end-to-end tracing for their input data.

Magpie [3] uses unsupervised machine learning algorithms to group individual requests' flows into coarse-grained clusters, based on similarity in structure and timing. Such grouping is particularly useful for detecting anomalies—small sets of requests that have significantly different structure or timing behaviour than others. Pinpoint [8] also attempts to identify anomalies in the set of request flows, using a probabilistic context free grammar instead of clustering. Pinpoint also includes mechanisms for identifying failed components. Spectroscope includes support for Magpie-style clustering, for exploring behaviour in individual executions, but the primary focus of this paper is on algorithms and mechanisms for comparing the request flows of two executions. Such comparison can expose behaviour alterations that affect many requests and, thus, are not anomalous.

Pip [27] 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 manually specify expectations, which is a daunting and error-prone task. In addition, the developer is faced with balancing effort and generality with the specificity needed to expose particular problems. Overly general/high-level

expectations will miss some problems, while overly specific expectations require much labor (e.g., detailing all valid configurations). 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 manual 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 [33, 35]. Building on that initial work, IRONmodel [34] developed approaches to detecting (and correcting) violations of such performance 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, 28] 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, 25]. Whodunit [6] automatically instruments a system to enable distributed profiling. Work has also been done on automatically identifying recurrences of previously diagnosed problems [9, 40]. 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 [37] compares multiple failing and non-failing runs to identify the most significant differences. OSprof [17] and DARC [36] automatically profile system calls and identify the greatest sources of latency. Our work builds on some concepts from such single-process techniques.

# 3 Experimental apparatus

The experiments and case studies reported in this paper come from use of the techniques described with a distributed storage system called Ursa Minor [1]. To provide context for results mentioned during description of the algorithms, as well as the case studies, this section briefly describes Ursa Minor, including the instrumentation and benchmark programs used.

## 3.1 Ursa Minor

Figure 3 illustrates the Ursa Minor 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 (or NFS servers) of Ursa Minor must first send a request to a metadata server asking for the appropriate permissions and location of the data on the storage nodes. Clients are then free to access the storage nodes directly.

Ursa Minor has been in active development since 2004 and is currently comprised of about 230,000 lines of code. Over 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]; space precludes detailed description here.

The components of Ursa Minor are usually run on separate machines within a datacenter. Though Ursa Minor supports an arbitrary number of components, the experiments and case studies detailed in this paper use a simple 5-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

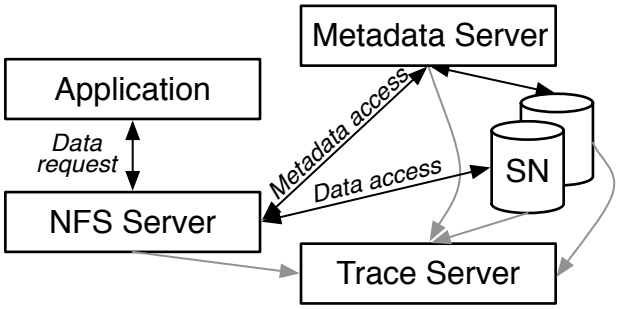


Figure 3: **Ursa Minor Architecture.** Ursa Minor can be deployed in many configurations, with an arbitrary number of NFS servers, metadata servers, storage nodes (SNs), and trace servers. This diagram shows a common 5-component configuration.

studies.

**End-to-end tracing infrastructure via Stardust**: Ursa Minor's Stardust tracing infrastructure is much like its peer group, discussed in Section 2.1. Request-sampling is used to capture trace data for a subset of entire requests (10%, by default), with a per-request decision made when the request enters the system. The system contains approximately 200 trace points, 124 manually inserted and an automatically generated one 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, which is crucial for differentiating structural mutations from alternate interleavings of concurrent activities. Additional data collected at trace points captures low-level parameters and information that allows latent work (e.g., write-backs of dirty cache blocks) to be attributed to the request that introduced it into the system.

### 3.2 Benchmarks used

All data described in the paper come from use of the following subset of benchmarks in our nightly regression testing of Ursa Minor.

`Linux-build` **&** `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, and `Ursa minor-build` runs for 10 minutes.

`Postmark-large`: This synthetic benchmark is designed to evaluate the small file performance of storage systems [19]. Configured to utilize 448 subdirectories, 50,000 transactions, and 200,000 files, this benchmark runs for 80 minutes.

`SPEC SFS 97 V3.0 (SFS97)`: This synthetic benchmark is the industry standard for measuring NFS server scalability and performance [31]. It applies a periodically increasing load of NFS operations to a storage system's NFS server and measures the average response time. SPEC SFS was configured to generate load between 50 and 350 operations/second in increments of 50 ops/second. This benchmark runs for 90 minutes.

`IoZone`: This benchmark [24] sequentially writes, re-writes, reads, and re-reads a 5GB file in 20 minutes.

# 4 Spectroscope

Spectroscope is written as a Perl application that interfaces with MATLAB when performing statistical computations; it also includes a visualization layer written using Prefuse [15] for visualizing output. This section overviews Spectroscope's design, and the next section details its algorithms for comparing system behaviours. Section 4.1 describes how *categories*, the basic building block on which Spectroscope operates, are constructed. Sections 4.2 and 4.3 overview its support for comparing and browsing system behaviours.

## 4.1 Categorizing system behaviour

Even small distributed systems can service 100s to 1000s of requests per second, so browsing or comparing all of them individually is not feasible. Instead, Spectroscope groups requests into *categories* and uses them as the basic unit for browsing and comparing behaviours. In choosing the axis of similarity on which to perform the grouping, Spectroscope exploits the expectation that requests that take the same path through a distributed system should incur similar costs: requests that exhibit the same structure are grouped into a single category. Two requests are deemed to be 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.

For each category, Spectroscope identifies aggregate statistics, including the number of requests it contains, average response time, and variance. To identify where time is spent, Spectroscope 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 [38] (also known as swim lanes), which more directly show the request's pattern of activity.

By default, Spectroscope, unlike Magpie [3], does not use unsupervised clustering algorithms to group requests into categories. Though initial versions of Spectroscope used clustering [30], we have found that the groups they create are too coarse-grained and unpredictable. As a result, they tend to group mutations and precursors within the same category, essentially masking their existence. There is potential for clustering algorithms to be used, but significant challenges exist. Most importantly, distance metrics that better align with developers' notion of request similarity need to be developed. Without it, use of clustering algorithms will result in categories comprised of seemingly dissimilar requests.

## 4.2 Comparing system behaviours

Performance changes can result from a variety of factors, such as internal changes to the system that result in performance regressions, unintended side affects of changes to configuration files, or environmental issues. Spectroscope provides guidance in diagnosing these problems by comparing system behaviours and identifying the key resulting mutations.

When comparing system behaviours, Spectroscope takes as input request-flow graphs from two periods of activity a *non-problem* period and a *problem period*. It creates categories comprised 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 and are ranked by expected contribution to the performance change.

Visualizations of categories that contain mutations are similar to those described above, except now 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 understand how they differ.

Further insight into many performance problems can be provided by explaining the low-level parameters (e.g., client parameters, or function call parameters) that best differentiate a mutation and its precursor.

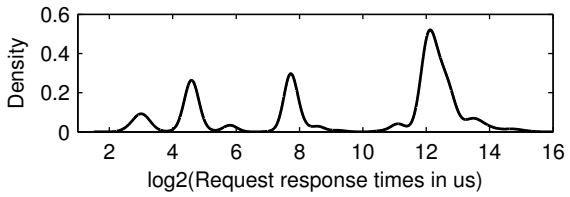


Figure 4: **Density estimate for** `postmark-large`**.** To better differentiate modes of behaviour at smaller response times and increase normality, a $\log_2$ transform is applied to response times before applying the kernel density estimator. The bandwidth parameter is chosen to be optimal for a mixture of normals. The density estimate shown above reveals four main modes. The first three are comprised of metadata requests and whole-block WRITEs serviced exclusively by the NFS Server. The fourth is comprised of metadata requests that require metadata server and storage node accesses.

For example, in Ursa Minor, a performance slowdown, which manifested as many structural mutations, was caused by change in a parameter sent by the client. For problems like this, explaining low-level differences can immediately identify the root cause.

Section 5 describes Spectroscope's algorithms and heuristics for identifying mutations, their corresponding precursors, their relative influence on the overall performance change, and their most relevant low-level parameter differences. It also explains how Spectroscope overcomes key challenges. For example, identifying response-time mutations relies upon the expectation (reasonable for many distributed systems, including distributed storage) that requests taking the same path through a distributed system should exhibit similar response times and edge latencies. High natural variance in timings, therefore, complicates such identification. As another example, accurately matching the many possible precursors to the appropriate ones of many structural mutations is a difficult puzzle.

### 4.3 Browsing system behaviours

To help developers enhance their mental models, Spectroscope allows developers to browse the categories induced by a workload; interesting behaviour can be selected and examined in more depth. By doing so, developers might learn about valid behaviour they did not expect, or might identify problematic behaviour that requires attention—for example, design problems, or constantly slow requests.

When used to browse system behaviour, Spectroscope takes as input request-flow graphs, in DOT [13] format, from a single period of activity. It displays an estimated probability density function (PDF) of individual request response times, constructed using a kernel density estimator [39]. The density estimate provides a rough visualization of system behaviour, allowing users to select and extract ranges which they deem interesting—for example, a spike in the density estimate at high response times. Figure 4 an example for `postmark-large`. Given a range, Spectroscope will output categories comprised of requests that fall in them.

The categories returned can be ranked by any of the aggregate statistics. Those that exhibit high average response times are obvious points for optimizations, whereas high variance indicates contention for a resource.

By default, category visualizations show the entire request structure, including paths of requests within components and inter-component activity. Though such detailed information is useful, it can overwhelm the developer. As such, Spectroscope supports a *zooming* mode that presents categories that are created using filtered versions of request-flow graphs, which only expose the components accessed. Nodes in these graphs represent RPC call and reply events and edge latencies represent time spent within components.

These zoomed-out graphs are both smaller in size than their fully detailed counterparts and induce fewer categories. If the developer notices interesting behaviour in the zoomed-out view, for example a component traversal that takes a long amount of time, he can choose to "zoom in" by exposing the instrumentation within it.

# 5 Algorithms for comparing behaviours

This section describes the key algorithms and heuristics used when comparing system behaviours.

## 5.1 Identifying response-time mutations

Spectroscope uses the Kolmogorov-Smirnov two-sample, non-parametric hypothesis test [22] to automatically identify response-time mutations and localize the change to the specific interactions or components responsible for it. Two-sample hypothesis tests take as input two sample distributions and return a p-value, which ranges between 0 and 1 and represents uncertainty in the claim that the null hypothesis, that both distributions are the same, is false. The p-value increases with variance and decreases with the number of samples. The null hypothesis is rejected when the p-value is below the desired false-positive rate, almost always 5%. A non-parametric test, which does not require knowledge of the underlying distribution, is used because we have observed that response times of requests are not distributed according to well-known distributions. The Kolmogorov-Smirnov test was chosen specifically because it identifies differences in both the shape of a distribution and its median.

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 a response-time mutation if the hypothesis test rejects the null hypothesis. By default, categories that contain too few requests to run the test accurately are not marked as response-time mutations. To identify the components or interactions responsible for the mutation, Spectroscope extracts the critical path [20]—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. Categories that contain `SM_THRESHOLD` more requests from the problem period than from the non-problem period are labeled as containing structural mutations and those that contain `SM_THRESHOLD` less are labeled as containing precursors. The exact choice of the threshold values is not important; it is simply a way to control Spectroscope's runtime. Though use of a low threshold will result in more false positives (mutations not related to the performance change), these will be given a low rank, and will not mislead the developer in his diagnosis efforts. If similar workloads were not run, this heuristic will identify mutations that result due to workload changes in addition to changes internal to the system.

We note that use of a statistical test, as used for response-time mutations, would avoid the need for manually set thresholds. However, we have observed that there is always enough structural variance between system executions for standard categorical hypothesis tests (such as the $\chi^2$ test [39]) to reject the null hypothesis. Structural variance arises in Ursa Minor because, like most distributed systems, the order in

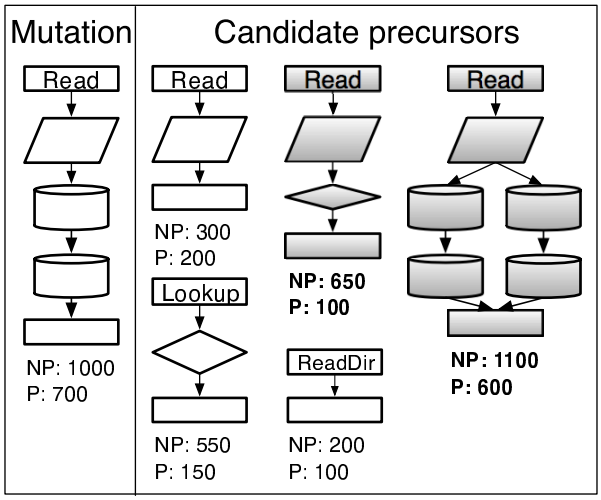


Figure 5: **How the candidate precursor categories of a structural-mutation category are identified.** The shaded precursor categories shown above will be identified as the candidate precursors of the structural-mutation category shown. The precursor categories that contain LOOKUP and READDIR requests will not be identified as candidates because their constituent requests are not READS. The top left-most precursor category contains READS, but it will not be identified as a candidate because the decrease in frequency of its constituent requests between the problem (P) and non-problem (NP) period is less than the increase in frequency of the mutation category.

which concurrent requests can be serviced is non-deterministic, allowing for a request to see more or less of a resource (e.g., cache space) between system executions.

Once the total set of structural-mutation categories and precursor categories are identified, Spectroscope must identify the set of precursor categories that are likely to have contributed requests to a structural-mutation category during the problem period. Several heuristics are used to achieve this and are described below; Figure 5 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 it is safe to assume that requests of different high-level types cannot be precursor/mutation pairs.

Second, a 1:N assumption is made to further prune the remaining precursor categories: precursor categories for which the decrease in request-count is less than the increase in request-count of the structural-mutation category are removed. This reflects the common case that most structural mutations will most often exhibit a common subpath until the point at which they are affected by the problem and many different subpaths after that. For example, a portion of requests that used to hit in cache in the non-problem period may now miss in that cache, but hit in the cache at the next level of the system. The remaining portion might miss at both levels because of the change in resource demand placed on the second-level cache. Since the 1: N assumption may not hold for certain performance problems, the developer can optionally choose to bypass this heuristic when identifying the candidate precursor categories.

Finally, the remaining precursor categories are identified as the candidates and are ranked according to their structural similarity to the mutation, as determined by their normalized string-edit distance. This reflects the heuristic that precursors and structural mutations are likely to resemble each other. Though computation of normalized string-edit distance is relatively expensive ($O(K^2)$) and dominates the cost of

Spectroscope's runtime, its cost can be controlled by choice of SM_THRESHOLD, as edit distances only need to be calculated for categories identified as containing structural mutations and their possible candidate precursor categories.

### 5.3 Ranking

Ranking of mutations is necessary for two reasons. First, the performance problem might have multiple root causes, each of which results in an independent performance problem, and causes its own set of mutations. In this case, it is useful to identify the mutations that most affect performance to focus diagnosis effort where it will yield the most benefit.

Second, even if there is only one root cause to the performance problem (e.g., a misconfiguration), many mutations will still be observed. To help navigate this host of mutations, ranking is useful. In the future, we hope to also infer causality between mutations, which would help Spectroscope better identify the mutations most directly related to the root cause.

Spectroscope ranks both structural-mutation categories and response-time mutation categories in descending order by their expected contribution to the change in performance. The contribution for a structural-mutation category is calculated as the difference in number of problem period and non-problem period requests multiplied by the change in average response time in the problem period between it and its candidate 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 non-problem period requests times the change in average response time between the problem and non-problem period.

It is possible for categories to be labeled as containing both precursors and response-time mutations, or both structural and response-time mutations. To provide guidance in such cases, these categories are split into multiple 'virtual categories' and ranked separately.

### 5.4 Dealing with high-variance categories

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 Spectroscope's ability to identify behaviour changes accurately. Spectroscope's ability to identify response-time mutations is sensitive to variance, whereas only the ranking of structural mutations 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 previous versions of Ursa Minor, several high-variance categories resulted from a poorly written hash table which exhibited slowly increasing lookup times because of a poor hashing scheme.

For response-time mutations, both the number of false negatives and false positives will increase with the number of high-variance categories. The former will increase because high variance will reduce the Kolmogorov-Smirnov test's power to differentiate true behaviour changes from natural variance. The latter, which is much rarer, will increase when it is valid for categories to exhibit similar response times within a single execution of the system or 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, a CDF of the squared coefficient of variation in response time ($C^2$) for *large categories* induced by linux-build, postmark-large, and SFS97 is shown in Figure 6. $C^2$ is a normalized measure of variance and is defined as $(\frac{\sigma}{\mu})^2$. Distributions with $C^2$ less than one are considered to exhibit low variance, whereas those with $C^2$ greater than one exhibit high variance [14]. *Large categories* contain over 10 requests; Table 1 shows that these categories account for only 15%—29% of all categories, but contain more than 99% of all

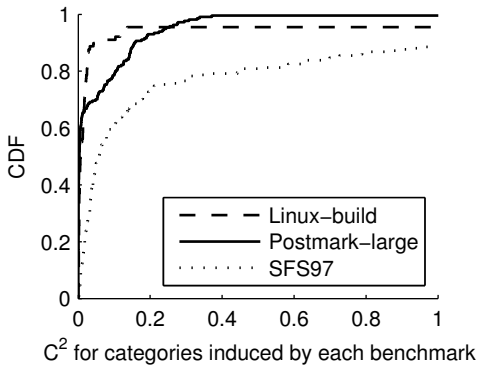


Figure 6: **CDF of $C^2$ for large categories induced by various benchmarks.** The squared coefficient of variation ($C^2$) of response time is plotted for large categories induced by the full request-flow graphs of three benchmarks. At least 88% of the categories induced by each benchmark exhibited low variance. Similar results (not shown) were found for zoomed-out request-flow graphs that include only component traversals, where at least 82% of the categories exhibited low variance.

| | `linux build` | `postmark large` | `SFS97` |
|---|---|---|---|
| # of categories | 351 | 716 | 1602 |
| large categories | 25.3% | 29.9% | 14.7% |
| # of requests sampled | 145,167 | 131,113 | 210,669 |
| requests in large categories | 99.7% | 99.2% | 98.9% |

Table 1: **Distribution of requests in the categories induced by various benchmarks.** This table shows the total number of requests sampled for `linux-build`, `postmark-large`, and `SFS97`, and the number of resulting categories. $C^2$ results for these categories are shown in Figure 6. Though many categories are generated, most contain only a small number of requests. *Large categories*, which contain more than 10 requests, account for between 15–30% of all categories generated, but contain over 99% of all requests.

requests. Categories containing fewer requests are not included because their smaller sample size makes $C^2$ statistic unreliable for them.

For all of the benchmarks, at least 88% of the large categories exhibit low variance. $C^2$ for all the categories generated by `postmark-large` is small. 99% of all 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 all 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 the these benchmarks has revealed that part of the observed variance is a result of contention for locks in the metadata server.

**Variance due to instrumentation granularity**: Unlike Ursa Minor, which has been instrumented extensively, many other systems that use end-to-end tracing are only instrumented at the RPC layer [7, 32, 21]. $C^2$ for categories in these systems will be larger, because the paths taken within components cannot be disambiguated. In the worst case, the intra-category variance might be so large so as to make the statistical tests unusable. This is not the case for Ursa Minor—only `SFS97` changes slightly, from 88% with low

variance to 82% with low variance. Adding RPC parameters that describe the amount of work to be done within a component to the node names of request-flow graphs could help reduce intra-category variance, if this is an issue.

### 5.5 Explaining mutations

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 [4] showing 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 of explaining mutations 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 part of the request structures that are common between requests in both categories. Since low-level parameters are expected to differ once the mutation starts to differ, the template describes only the common structure until the point of the first difference.

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 [26], 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.

## 6 Case Studies

Spectroscope is not designed to replace developers; rather it serves as an important step in the workflow used by developers to diagnose problems. Sometimes Spectroscope can help developers identify the root cause immediately or at least localize the problem to a specific area of the system. In other cases, by verifying that the distributed system's behaviour hasn't changed, it allows developers to focus their debugging efforts on external factors and eliminate the system as the cause.

This section diagnoses several real performance problems with Spectroscope as an important tool and analyzes the effectiveness of Spectroscope's output in identifying the root causes of poor performance. Most of these problems were previously unsolved and diagnosed by the authors without prior 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 with comparisons of system behaviour. By introducing a synthetic spin loop of different delays, we also demonstrate Spectroscope's sensitivity to changes in response time.

### 6.1 Methodology

For cases in which Spectroscope was used to compare system behaviours, three complimentary metrics are provided for evaluating the ranked list of categories it identifies as mutations.

| # / Type | Name | Manifestation | Root cause | # of results | % rel. top 10 | % false pos. | Cov |
|---|---|---|---|---|---|---|---|
| 6.2 / Real | Create behaviour | Odd structure | Design problem | N/A | N/A | N/A | N/A |
| 6.3 / Real | MDS config. | Structural | Config. change | 128 | 100% | 2% | 70% |
| 6.4 / Real | RMWs | Structural | Env. change | 3 | 100% | 0 | 100% |
| 6.5 / Real | MDS prefetch. 50 | Structural | Internal change | 7 | 29% | 71% | 93% |
| 6.5 / Real | MDS prefetch. 10 | Structural | Internal change | 16 | 70% | 56% | 96% |
| 6.6 / Synthetic | 100$\mu$s delay | Response time | Internal change | 17 | 0% | 100% | 0% |
| 6.6 / Synthetic | 500$\mu$s delay | Response time | Internal change | 166 | 100% | 6% | 92% |
| 6.6 / Synthetic | 1ms delay | Response time | Internal change | 178 | 100% | 7% | 93% |
| 6.7 / Real | Periodic spikes | No change | Env. change | N/A | N/A | N/A | N/A |

Table 2: **Overview of the case studies.** This table shows information about each of the 6 problems diagnosed using Spectroscope. For cases where the problem was diagnosed by comparing system behaviours, quantitative metrics for evaluating the quality of the output ranked list are provided.

**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 these categories to be relevant.

**% of false positives**: This metric evaluates the overall quality of the ranked list by identifying the percentage of all results that are *not relevant*. This metric is more meaningful for response-time mutations than structural mutations as the structural false positive rate is dependent on the choice of `SM_THRESHOLD`.

**Coverage**: This metric evaluates quality of the ranked list by identifying the percentage of requests affected.

Table 2 summarizes Spectroscope's performance using these metrics, including the criteria used for each problem to assess the relevance of results. Unless otherwise noted, a default value of 50 was used for `SM_THRESHOLD`. This value was chosen to yield fast runtimes (between 15-20 minutes) when diagnosing problems in larger benchmarks, such as `SFS97` and `postmark-large`. When necessary, it was lowered to further explore the possible space of structural mutations.

## 6.2 Create behaviour

During development of Ursa Minor, we noticed that the distribution of request response times for benchmarks such as `postmark-large` and `SFS97` exhibited a large tail. We used Spectroscope to extract the categories induced by the most expensive requests and noticed that most were CREATES. Analysis of the graph view of these categories revealed two architectural irregularities.

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, requests in the various categories that contained CREATEs differed only in the number of times they executed this loop. Those issued later executed the loop dozens of times, incurring response times on the order of 100s of milliseconds. This inter-component loop can be easily seen if the categories are zoomed out to show only component traversals and plotted in a train schedule, as in Figure 7.

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 is an example of how browsing and visualizing system behaviour can help developers identify system design problems. Though simple counters could have shown that CREATES were

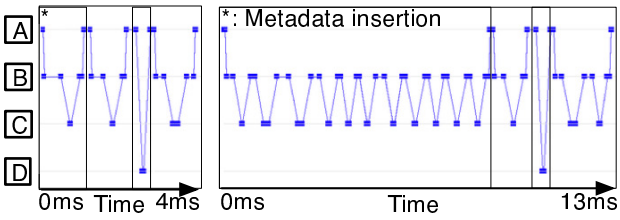


Figure 7: **Visualization of create behaviour.** Two train-schedule visualizations are shown, the first one a fast early create during `postmark-large` and the other a slower create issued later in the benchmark. Messages are exchanged between the NFS Server (A), Metadata Server (B), Metadata Storage 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.

very expensive, they would not have shown that the root cause was excessive metadata server/storage node interactions.

### 6.3 MDS configuration change

After a large code check-in, performance of `postmark-large` decayed significantly, from 46tps to 28tps. To diagnose this problem, we used Spectroscope to compare behaviours 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 candidate precursor categories revealed that the storage node utilized by the metadata server had changed. Before the check-in, the metadata server wrote metadata to only its dedicated storage node. After the check-in, it issued most writes to the data storage node instead. We also used Spectroscope to explain the low-level parameter differences between a few structural-mutation categories and their corresponding candidate-precursor categories. The regression tree identified differences in elements of the encoding scheme (e.g., type of fault tolerance supported, etc).

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 encoding. Without this, Ursa Minor uses its default encoding scheme and sends all writes to the data storage node. The developer was surprised to learn that the elements of the default encoding scheme differed from the ones he had chosen in the configuration file.

**Summary**: Diagnosis of this real problem is an example of how comparing system behaviours can help 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; it can also be used to help end users understand how various configuration options will affect system behaviour.

**Quantitative analysis**: For the evaluation numbers in Table 2, results in the ranked list were deemed relevant if they included metadata accesses to the data storage node with a most-likely candidate precursor category that included metadata accesses to the metadata storage node.

### 6.4 Read-modify writes

This problem was observed and diagnosed before development on Spectroscope began; it has been 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 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 used to track the frequency of RMWs.

To show how comparing system behaviours and explaining mutations could have helped developers quickly identify the root cause, Spectroscope was used to compare behaviours 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 explain the low-level parameter differences between the highest-ranked result and its most-likely candidate-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 less than or equal to 4KB were classified as belonging to the problem period.

**Summary**: Diagnosis of this problem demonstrates how comparing system behaviours can sometimes help developers identify performance problems that arise due to a workload change. It also shows showcases the utility of identifying low-level parameter differences.

**Quantitative analysis**: For Table 2, results in the ranked list were deemed relevant if they contained RMWs and their most-likely precursor category did not.

### 6.5 MDS Prefetching

A few years ago, several developers, including one of the authors of this paper, tried to add *server-driven metadata prefetching* functionality to Ursa Minor [16]. This feature was intended to intelligently prefetch metadata to clients on every metadata access, in hopes of squashing future accesses and thus improving performance. However, after the feature was implemented, no improvement was observed. The developers spent a few weeks 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 the database accesses were caused by it. 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 out-weighed the time savings generated by the increase in cache hits.

**Summary**: This problem demonstrates how comparing system behaviours 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. Diagnosis with Spectroscope was demonstrated to be an interactive experience, requiring developers to iteratively modify SM_THRESHOLD and gain insight from the resulting mutations and assigned ranks.

**Quantitative analysis**: For Table 2, 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 considered false positives.

### 6.6 Slowdown due to code changes

This synthetic problem was injected in Ursa Minor to show how comparing system behaviours can be used to identify slowdowns due to feature additions or regressions, as well as to assess Spectroscope's sensitivity to changes in response time.

Spectroscope was used to compare system behaviours 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 100$\mu$s.

Table 2 shows results from injecting a 100$\mu$s 500$\mu$s, and a 1ms spin loop. 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 results identified, only 6–7% are false positives and 100% of the 10 highest-ranked categories are relevant. The coverage is 92% and 93%.

Variance in response times and the edges in which the delay manifests prevent Spectroscope from properly identifying the affected categories for the 100$\mu$s case. It identifies 11 categories containing affected edges with multiple traversals as response-time mutations, but is unable to assign those edges as responsible for the mutations.

Spectroscope's ability to identify response-time mutations was also evaluated on zoomed-out versions of the graphs, which show only component traversals. The overall results were similar to those presented above, except the higher intra-category variance reduced coverage for the 500$\mu$s and 1ms cases from over 90% to 66%.

Spectroscope could also be used to identify response-time mutations that arise due to interference from external processes. In such cases, Spectroscope would identify many edges as responsible for the performance change, as the external process would likely affect many interactions within the affected component. It would be up to the developer to recognize this as a sign that the problem is most likely the result of an external process.[2]

### 6.7 Periodic spikes

Ursa minor-build, which is run as part of the nightly test suite, often shows a periodic spike in the time required for its copy phase to complete. For example, during one particular instance, copy time increased from 111 seconds to 150 seconds, an increase of 30%. 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 system behaviours between a run in which the spike was observed and another in which it was not.

[2]We have observed at least one case in which such reasoning would be incorrect. In this case, the root cause of the problem was a new macro which performed a large memset on every use.

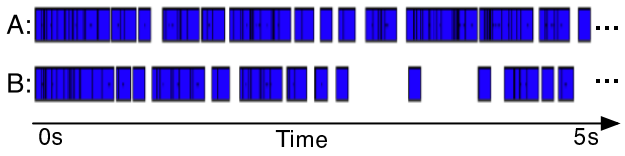


Figure 8: **Timeline of inter-arrival times of requests at the NFS Server.** A 5s sample of requests, where each rectangle represents the process time of a request, reveals long periods of inactivity due to lack of requests from the client during spiked copy times (B) compared to periods of normal activity (A).

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 almost guarantees 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, the null hypothesis was not rejected. All of these observations convinced us that the problem was not due to Ursa Minor or processes running on Ursa Minor's 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 8). The visualization shows that during the problem period, response times stay constant but the arrival rate of requests decrease. We currently suspect the problem to be backup activity initiated from the facilities department from outside of our system.

**Summary**: This problem demonstrates how comparing system behaviours can be used to diagnose performance problems that are not caused by behaviour changes within the system. By alerting developers that nothing within the distributed system has changed, they are free to focus their efforts on other external factors.

# 7 Conclusion

Comparing system behaviours, as captured by end-to-end request flow traces, is a powerful new technique for diagnosing performance changes between two time periods or system versions. Spectroscope's algorithms for such comparing allow it to accurately identify and rank mutations and their corresponding precursors, focusing attention on the most important differences. Experiences with Spectroscope confirm its usefulness and efficacy.

# References

[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 D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM Symposium on Operating System Principles* (Bolton Landing, NY, 19–22 October 2003), pages 74–89. ACM, 2003.

[3] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pages 259–272. USENIX Association, 2004.

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

[5] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. *USENIX Annual Technical Conference* (Boston, MA, 27 June–02 July 2004), pages 15–28. USENIX Association, 2004.

[6] Anupam Chanda, Alan Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. *EuroSys* (Lisbon, Portugal, 21–23 March 2007), pages 17–30. ACM, 2007.

[7] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. *Symposium on Networked Systems Design and Implementation* (San Francisco, CA, 29–31 March 2004), pages 309–322. USENIX Association, 2004.

[8] Mike Y. Chen, Emre Kiciman, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. *International Conference on Dependable Systems and Networks* (Washington, DC, 23–26 June 2002), pages 595–604. IEEE Computer Society, 2002.

[9] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. *ACM Symposium on Operating System Principles* (Brighton, United Kingdom, 23–26 October 2005), pages 105–118. ACM, 2005.

[10] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: a pervasive network tracing framework. *Symposium on Networked Systems Design and Implementation* (Cambridge, MA, 11–13 April 2007). USENIX Association, 2007.

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

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN Symposium on Compiler Construction* (June 1982). Published as *SIGPLAN Notices*, **17**(6):120–126, June 1982.

[13] Graphviz. www.graphviz.com.

[14] Mor Harchol-Balter. 15-857, Fall 2009: Performance modeling class lecture notes, 2009. http://www.cs.cmu.edu/h̃archol/Perfclass/class09fall.html.

[15] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. *Conference on Human Factors in Computing Systems* (Portland, OR, 02–07 April 2005), pages 421–430. ACM, 2005.

[16] James Hendricks, Raja R. Sambasivan, and Shafeeq Sinnamohideen. *Improving small file performance in object-based storage*. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006.

[17] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. *Symposium on Operating Systems Design and Implementation* (Seattle, WA, 06–08 November 2006), pages 89–102. USENIX Association, 2006.

[18] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. *Conference on File and Storage Technologies* (San Jose, CA, 24–26 February 2010). USENIX Association, 2010.

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

[20] James E. Kelley. Critical-path planning and scheduling. *Eastern Joint Computer Conference* (Boston, MA, 01–03 December 1959), pages 160–173. ACM, 1959.

[21] Andrew Konwinski. Technical Presentation at Hadoop Summit: Monitoring Hadoop using X-Trace, March 2008. http://research.yahoo.com/node/2119.

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

[23] Jeffery C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. *EuroSys* (Leuven, Belgium, 18–21 April 2006), pages 293–304. ACM, 2006.

[24] William Norcott and Don Capps. IoZone filesystem benchmark program, 2002. www.iozone.org.

[25] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Ganesha: black-box fault diagnosis for MapReduce systems. *Hot Metrics* (Seattle, WA, 19–19 June 2009). ACM, 2009.

[26] J. R. Quinlan. Bagging, boosting and C4.5. *13th National Conference on Artificial Intelligence* (Portland, Oregon, 4–8 August 1996), pages 725–730. AAAI Press, 1996.

[27] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. *Symposium on Networked Systems Design and Implementation* (San Jose, CA, 08–10 May 2006), pages 115–128. USENIX Association, 2006.

[28] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: Black-box Performance Debugging for Wide-Area Systems. *International World Wide Web Conference* (Edinburgh, Scotland, 23–26 May 2006), pages 347–356. ACM Press, 2006.

[29] Raja R. Sambasivan, Alice X. Zheng, Elie Krevat, Spencer Whitman, and Gregory R. Ganger. *Diagnosing performance problems by visualizing and comparing system behaviours*. Technical report 10–103. Carnegie Mellon University, February 2010.

[30] Raja R. Sambasivan, Alice X. Zheng, Eno Thereska, and Gregory R. Ganger. Categorizing and differencing system behaviours. *Workshop on hot topics in autonomic computing (HotAC)* (Jacksonville, FL, 15 June 2007), pages 9–13. USENIX Association, 2007.

[31] SPECsfs. www.spec.org/sfs.

[32] Benjamin H. Sigelman, Luiz Andre Barroso, Mike 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, April 2010.

[33] 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. *International conference on autonomic computing* (Dublin, Ireland, 12–16 June 2006), pages 187–198, 2006.

[34] Eno Thereska and Gregory R. Ganger. IRONModel: robust performance models in the wild. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, 02–06 June 2008), pages 253–264. ACM, 2008.

[35] 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. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint-Malo, France, 26–30 June 2006), pages 3–14. ACM, 2006.

[36] Avishay Traeger, Ivan Deras, and Erez Zadok. DARC: Dynamic analysis of root causes of latency distributions. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, 02–06 June 2008). ACM, 2008.

[37] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. *ACM Symposium on Operating System Principles* (Stevenson, WA, 14–17 October 2007), pages 131–144, 2007.

[38] Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, Cheshire, Connecticut, 1983.

[39] Larry Wasserman. *All of Statistics*, second edition. Springer Science + Media Inc., March 2004.

[40] 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. *Automated Known Problem Diagnosis with Event Traces* (Leuven, Belgium, 18–21 April 2006), pages 375–388. ACM, 2006.