Diagnosing performance problems by visualizing and comparing system behaviours

Raja R. Sambasivan[⋆], Alice X. Zheng[†],
Elie Krevat[⋆], Spencer Whitman[⋆], Gregory R. Ganger[⋆]
[⋆]*Carnegie Mellon University,* [†]*Microsoft Research*

CMU-PDL-10-103
February 2010

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

Abstract

Spectroscope is a new toolset aimed at assisting developers with the long-standing challenge of performance debugging in distributed systems. To do so, it mines end-to-end traces of request processing within and across components. Using Spectroscope, developers can visualize and compare system behaviours between two periods or system versions, identifying and ranking various changes in the flow or timing of request processing. Examples of how Spectroscope has been used to diagnose real performance problems seen in a distributed storage system are presented, and Spectroscope's primary assumptions and algorithms are evaluated.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, Data Domain, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NEC Laboratories, Network Appliance, Oracle, Seagate, Sun, Symantec, VMWare, and Yahoo! Labs) for their interest, insights, feedback, and support. This research was sponsored in part by NSF grants #CNS-0326453 and #CCF-0621508, by a Google research award, by DoE award DE-FC02-06ER25767, and by CyLab under ARO grant DAAD19-02-1-0389.

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

1 Introduction

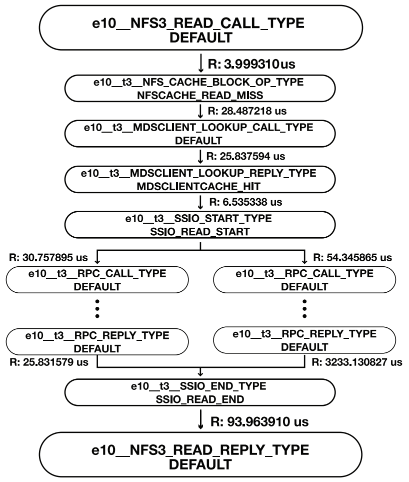


Figure 1: **Example request-flow graph.** The graph shows a striped READ in Ursa Minor [1]. Nodes represent trace points and edges are labeled with the time between successive events. Node labels are constructed by concatenating the machine name (e.g., `e10`), component name (e.g., `NFS3`), instrumentation point name (e.g., `READ_CALL_TYPE`), and an optional semantic label (e.g., `NFSCACHE_READ_MISS`).

Diagnosing the root cause of performance problems in a distributed system is hard. Doing so is particularly difficult when the problem relates to the interactions between components [23] rather than being isolated to a single component (which can enable use of single-component tools, like DTrace [5]). Unfortunately, little assistance is available for performance debugging in distributed systems—in most, there is even a lack of raw information with which to start.

Recent work has offered a solution to the dearth of information: end-to-end traces of activity in the distributed system [3, 7, 9, 10, 31]. These traces capture the path (e.g., the sequence of functions executed) and timing of each request within and across the components of the system and, as such, represent the entire behaviour of the system in response to a workload. See Figure 1 for an example. Research has shown that such tracing introduces little overhead.

The richness of such traces creates a “haystack” within which insights (”needles”) into performance problems exist. The challenge is to find them. We need tools that help developers obtain improved understanding of their systems and focus their attention on the parts of the system most likely causing the problem.

This paper describes Spectroscope and experiences using it. Spectroscope analyzes end-to-end traces to help developers understand and diagnose performance problems in a distributed storage system called Ursa Minor [1]. Specifically, Spectroscope helps developers build intuition about the system by visualizing its behaviour in response to a workload. In addition, it helps diagnose the root cause of changes in performance by comparing the behaviour of the system before and after the change.

Visualizing system behaviour: To help with building intuition, Spectroscope displays *request-flow graphs*, which are paths taken through the system by individual requests. Since even relatively small benchmarks involve many thousands of requests, Spectroscope groups identical paths into categories and allows them to be ranked according to a metric of interest (e.g., average response time). A given visualization shows the structure of requests assigned to a category, identifying trace points reached, parallel paths, and synchronization points. Spectroscope also shows statistical information about each category, such as the

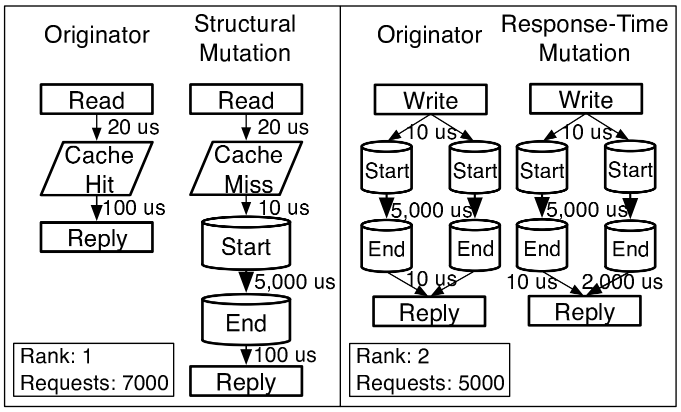


Figure 2: **Spectroscope's output when comparing system behaviours.** Structural mutations and response-time mutations are ranked by their affect on the change in performance. The item ranked first is a structural mutation, whereas the one ranked second is a response-time mutation. For spacing reasons, mocked-up graphs are shown in which nodes represent the type of component accessed.

average response time, frequency, variance in response time, and average latency between instrumentation points executed. Simply viewing this information can often help developers identify architectural problems or problems induced by excessively slow interactions. For example, the developer might notice a set of interactions that can be processed in parallel, instead of sequentially.

Comparing system behaviours: A primary focus of Spectroscope, however, is assisting with diagnosis of changes in performance. Many real-world scenarios require such diagnosis, including violations of previously-met service-level objectives and performance regressions due to code upgrades. Spectroscope builds on the fact that many performance changes manifest in the system as mutations in the way requests are processed. It takes as input requests-flow graphs from two system executions (or periods) and identifies new behaviour observed only after the performance change. To aid diagnosis, Spectroscope ranks these *mutations* by their expected contribution to the performance change and, for each, identifies their candidate originators—what the mutation might have looked like before the change.

Mutations can occur in two flavors: response-time and structural. *Response-time* mutations are requests that have changed in the amount of time needed for them to be serviced. For example, an "upgraded" data structure may be slower to access than in the previous system version. In addition to identifying response-time mutations, Spectroscope attempts to identify the specific interactions along the request's path that now take longer. Figure 2 illustrates an example, on the right.

Structural mutations are requests that have changed in the path they take through the distributed system, such as a request that used to hit in a cache, but now misses and must be serviced by a remote component. Comparing a structural mutation to its originator exposes where the request flows diverge, which often localizes the source of the problem. On the left, Figure 2 illustrates a simplified structural mutation, which would guide a developer to examine the particular cache in question.

It is often possible to "explain" an observed mutation—that is, identify why it occurred—by identifying the differences in low-level parameters (e.g., function call or RPC parameters) seen by the mutation and its originator. Knowledge of such differences can sometimes immediately reveal the root cause, or can further aid diagnosis. For example, a change in performance that yields response-time mutations may be caused because extra, unnecessary, data is now being sent in RPC calls. In this case, in addition to identifying the resulting response-time mutations, the problem can be further localized by identifying that the RPC message lengths have increased. Given a category containing structural mutations and a category containing its

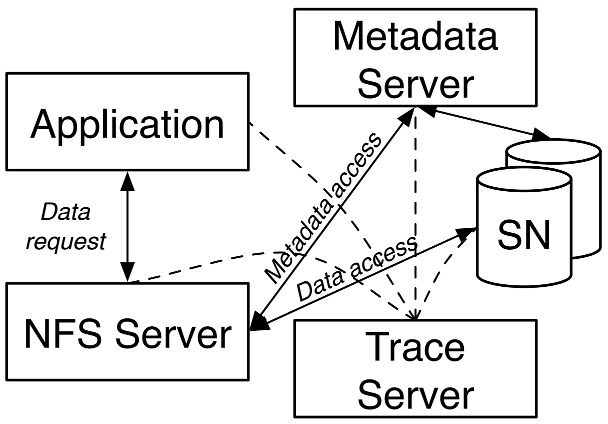


Figure 3: **Ursa Minor Architecture.** Ursa Minor can be deployed in many configurations, with an arbitrary number of NFS servers, metadata servers, storage nodes (SNs), and trace servers. This diagram shows a common 5-component configuration.

corresponding originators, Spectroscope can generate the set of low-level parameters that best differentiate them.

The rest of this paper is organized as follows. Section 2 describes several example problems encountered with Ursa Minor, illustrating a range amenable to diagnosis by comparing system behaviours. Section 3 describes the end-to-end tracing infrastructure on which Spectroscope is built. Section 4 describes the design and implementation of the Spectroscope analysis engine. Section 5 evaluates Spectroscope's key underlying assumptions and algorithms. Section 6 describes related work and Section 7 concludes.

2 Problems seen in Ursa Minor

In developing Ursa Minor, we encountered many performance problems that were very difficult to diagnose without better tools. This section describes some of these problems and how Spectroscope helped the authors diagnose them. It is worth noting that root causes for all of the problems described in this section were not known a priori; rather, they were identified by the authors while iteratively developing and using Spectroscope.

Section 2.1 describes necessary details about Ursa Minor. Sections 2.2, 2.3, and 2.4 describe the performance problems and how they were diagnosed using Spectroscope. Finally Section 2.5 presents a high-level summary.

2.1 Ursa Minor

Ursa Minor is a prototype distributed storage service being developed to serve as a testbed for autonomic computing [18] research. It is a direct-access storage system built as per the NASD model [11]. Details about its implementation can be found in Abd-El-Malek et al. [1]; only the relevant details are discussed here.

Figure 3 illustrates the Ursa Minor architecture, which is comprised of potentially many NFS servers, storage nodes (SNs), metadata servers (MDSs), and end-to-end-trace servers. To access data, clients 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. Upon obtaining this metadata, clients are free to access the storage nodes directly.

The components of Ursa Minor are usually run on separate machines within a datacenter. Though Ursa Minor supports an arbitrary number of components, it is usually run in a simple 5-machine configuration during development and testing. Specifically, one NFS server, one metadata server, one trace server, and two storage nodes are used; one storage node stores data, while the other stores metadata. This is the configuration in which all of the problems described in this section were observed and diagnosed.

2.2 Problem 1: Create behaviour

By visualizing system behaviour induced by running benchmarks such as `postmark-large` and `SFS97`[1], we were able to identify an architectural issue with the manner in which CREATEs are handled. The visualization, which presented categories ranked by average response time, showed that the highest ranked ones were CREATEs. Examining these categories revealed two irregularities. First, to serve a CREATE operation, the metadata server was executing a tight inter-component loop with a storage node. Each iteration of the loop required on the order of milliseconds, meaning this loop greatly affected response times. Second, requests in the various categories that contained CREATEs differed only in the number of times they executed this loop. CREATEs issued at the beginning of the benchmark executed the loop only a few times, whereas those issued near the end executed the loop dozens of times and, hence, incurred response times on the order of 100s of milliseconds.

Conversations with the metadata server's developer have led us to the root cause. When servicing a CREATE, the metadata server must traverse a B-tree in order to find a location in which to insert the created item's metadata. As the number of objects in the system grows, the probability that the insertion will require non-leaf pages to be recursively split up to the root increases. Each step of the recursion manifests as an iteration of the loop observed in the request-flow graphs. The developer is currently considering increasing the page size as a means of ameliorating this behaviour.

Identification of this interesting CREATE behaviour is an instance of how simply visualizing request-flow graphs can help identify architectural issues in a distributed system. Though simple per request-type performance counters could have revealed that CREATEs are the most expensive operations in Ursa Minor, they would not have shown the reason they are expensive is because of a tight inter-component loop.

2.3 Problem 2: MDS configuration

Every night many benchmarks are run on the latest Ursa Minor build. After one large code check-in, performance of many of these benchmarks decayed significantly. Using Spectroscope, we were able to determine that the root cause of the problem was a change in the metadata server's configuration file. Where before the check-in, this file specified that all metadata should be written to a dedicated storage node, after the check-in, the file specified that all metadata should be written to the same storage node as that used for regular data. As such, the load on the data storage node increased, resulting in slower performance for all requests that accessed it.

To diagnose this problem using Spectroscope, we obtained the request-flow graphs generated by one run of `postmark-large` on Ursa Minor from before the check in and another from after. The graphs from these two periods were fed into Spectroscope to identify the changes in behaviour that most accounted for the change in performance. As output, Spectroscope identified many structural-mutation categories (i.e., categories containing structural mutations) and corresponding candidate originator categories. We observed these mutations and originators differed only in the storage node they accessed. Additionally, we realized that all of the structural mutations were requests that incurred metadata accesses. As developers of Ursa Minor, this information was enough to lead us to the root cause.

[1]Details about these benchmarks can be found in Section 5.1

2.4 Problem 3: Metadata prefetching

A few years ago, several graduate students, including one of the authors of this paper, tried to add *server-driven metadata prefetching* functionality to Ursa Minor [14]. This feature was intended to intelligently prefetch metadata to clients on every metadata access, in hopes of squashing future accesses. Since metadata accesses are relatively expensive—they can incur both a network and processing overhead—reducing the number of these accesses, we believed, would greatly reduce response time and improve performance on closed workloads. Once the feature was implemented, however, the expected performance improvement was not observed. Though use of prefetching did increase the number of cache hits at the client's metadata cache, it actually increased the overall runtime of benchmarks such as `linux-build`, `postmark-large`, and `grep`. The graduate students attempted to determine why the expected performance benefit was not observed for a few months after the implementation was finished, but eventually gave up and moved on.

To diagnose this problem using Spectroscope, two sets of request-flow graphs were obtained: the first was generated by running `postmark-large` on Ursa Minor without prefetching and the second by running the same benchmark with prefetching turned on. When fed these inputs, Spectroscope identified many structural-mutation categories containing requests that incurred metadata server accesses. All of these structural mutations differed from their candidate originators in that they incurred several calls to the database used to store metadata; each of these calls added a small fixed amount to the mutation's response time. Further analysis revealed that these database calls reflected the extra work performed by metadata accesses to prefetch metadata to clients.

The above 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. This imbalance was exacerbated by a limitation of Ursa Minor's architecture and by a poor design choice made when implementing the prefetching functionality. With regard to the first, all WRITE operations require metadata accesses; for consistency reasons, they cannot be squashed, even if the metadata they require is cached at the client. With regards to the latter, the prefetching implementation, since it was server driven, did not know which metadata the clients had already cached. As a result, in certain cases the extra cost of prefetching was duplicated many times.

2.5 Summary

The problems described in this section highlight a very important fact: Spectroscope is not designed to automate performance diagnosis. To do so for an arbitrary performance problem would be a monumental task. Automation is the eventual goal, but Spectroscope provides a reasonable first step: It helps developers reason about the behaviour of a distributed system in order to aid them in their diagnosis efforts.

3 Tracing architecture

The request-flow graphs utilized by Spectroscope are obtained from a modified version of Stardust [31], Ursa Minor's end-to-end tracing architecture. Originally designed for workload modeling purposes, it has been modified to support diagnosis tasks instead. Stardust is similar to other white-box end-to-end tracing solutions, such as Magpie [3] and X-Trace [10]. Section 3.1 describes how Stardust collects traces and stitches them together to create request-flow graphs. Section 3.2 describes the key design decisions and features of Stardust. Finally, Section 3.3 describes an example request-flow graph.

3.1 Overview

Stardust is a white-box tracing framework that explicitly ties activity induced within various components of the distributed system to the initiating request. It differs from black-box approaches to tracing, such as Project 5 [2] and Pinpoint [7], as it allows trace points to be defined within components and identifies explicit causal relationships between trace points without inferring them. Unlike Whodunit [6], and DARC [32], which automatically instrument function names, Stardust requires programmers to manually insert trace points at key areas of interest. Trace points can capture a *trace-point name*, a *semantic label*, for example "cache hit" or "cache miss," and *low-level parameters*, such as the values of parameters sent to the function in which the trace point resides.

Stardust employs a two-step process to generate request-flow graphs. In the first step, which occurs during runtime, activity records of trace points executed by requests are collected and sent to a central server, which stores this data in a SQLite database. In addition to the trace-point name, semantic label, and parameter values, activity records contain a per-request identifier called a *breadcrumb* and a timestamp. The breadcrumb allows activity records generated on behalf of the same request to be tied together. The timestamp allows these records to be ordered. Since Stardust does not assume synchronized timestamps across machines, an explicit happens-before relationship [20] is created to order inter-component activity; this is done by generating a new breadcrumb for the request and inserting a "old breadcrumb happens before new breadcrumb" relationship in the database.

In the second step, request-flow graphs are generated by finding the breadcrumbs generated on behalf of each request and *stitching* together the activity records that contain them. Graphs are generated in DOT format [12]. Currently, this step must be performed offline after the system has shut down; however, it could also be performed online.

3.2 Design and features

This section lists the key features and design decisions made in modifying Stardust to support diagnosis.

Definition of initiating request: Activity performed in a distributed system can be attributed either to the request that placed the corresponding data in the system, or the request in whose context the activity is performed. That is, there are two possible *initiating requests* for any activity performed in the system. For example, the initiating request of a cache eviction can either be defined as the request that initially inserted the item being evicted, or the request on whose causal path the item is evicted. Since Stardust was designed for workload modeling purposes, it originally used the former definition. For diagnosis, however, we have found that the latter definition is more useful, as it allows for creation of more intuitive request-flow graphs. For example, under the latter definition, requests-flow graphs always end with the last trace point executed before a response is sent to the client, whereas under the former, requests-flow graphs often contained an arbitrary number of trace points that were executed *after* the response was sent.

Automatic RPC instrumentation: Stardust hooks into Ursa Minor's `RPCGEN` mechanism to automatically instrument RPC calls and replies. This yields component-level instrumentation granularity automatically.

Support for expressing parallelism and joins: Request-flow graphs generated by Ursa Minor make apparent the structure of requests. Parallelism and joins are exposed to impart as much information as possible to the developer. For example, during a striped read the distributed system should read data from all of the stripes, each of which is located on a different storage node, in parallel. Failure to do so is a design issue that is easily exposed by request-flow graphs that show request structure. Similarly, joins allow developers to see where the request must wait for slow excessively components before completing. For the case of a striped read, the join might expose the fact that one storage node always takes much longer to reply than the others.

Benchmark	Tracing off	Tracing on	Ovhd
`linux build`	1,716s	1,723s	0.4%
`IoZone`	2,583s	2,632s	1.9%
`postmark large`	49tps	46tps	6.1%
`SFS97 355 Ops/s`	1.5 Msec/op	1.6 Msec/op	6.7%

Table 1: **Effect of end-to-end tracing on Ursa Minor's performance.** Performance numbers for `linux-build`, `IoZone`, `postmark-large`, and `SFS97` are shown for two cases: the first with tracing disabled, and the second with tracing enabled. For the latter case, a request-sampling percentage of 20% was used. The maximum performance difference observed 6.7%. Configuration details for the benchmarks used can be found in Section 5.1. The results of 3 runs were averaged to obtain the above numbers.

To support expression of parallelism and joins in the request-flow graphs, Stardust's instrumentation API exposes the following functions:

- `new bcs = get_one_to_many(original bc)`
- `new bc = get_many_to_one(original bc)`

Developers can use the former function to express parallelism; one new breadcrumb is inserted in each concurrent thread and a happens-before relationship is inserted between each new breadcrumb and the original breadcrumb. Conversely, the latter allows developers to express join operations; one new breadcrumb is generated and a happens-before relationship is inserted between the breadcrumbs contained in the threads invoked in the join and the new breadcrumb.

Support for request-sampling: To minimize the effect on performance, activity records are not sent to the database on the critical path of requests. Rather, generated records are stored in a 64MB buffer, which is emptied periodically by a flusher thread. If the buffer is full, newly generated records are dropped, as blocking until the buffer empties will significantly affect performance. A dropped record might mean that an entire subpath of a request-flow graph will be lost, so it is beneficial to minimize this event. To this end, Stardust employs request-level sampling. When a request enters the system, a decision is made whether or not to collect its activity records. Currently, Stardust utilizes a request-sampling rate between 10% and 20%. Table 1 shows Stardust's effect on run-time performance when a request-sampling rate of 20% is used; the maximum performance degradation observed is 6.7%.

Support for critical-path extraction: Stardust's stitching mechanism can be configured to extract only the critical paths of sampled requests.

3.3 Request-flow graphs

Figure 1 shows a request-flow graph of a striped READ. Parallel paths in this graph represent concurrent activity. Nodes represent trace-point names concatenated with the optional semantic label. Edges are labeled with the latency between execution of the parent and child trace points, in wall-clock time. This helps identify where time is spent processing a request.

4 Spectroscope design and implementation

Spectroscope is written as a Perl application that interfaces with MATLAB when performing statistical computations. It takes as input request-flow graphs specified in the DOT language [12]. At its core, Spectroscope expects that requests that take similar paths through a distributed system should incur similar response times and edge latencies and that deviations from this expectation indicate performance problems. In a distributed storage system this expectation is valid because the time required to a process a request is largely determined by the caches and disks that it visits, not by computation.

The rest of this section is organized as follows. Section 4.1 describes how Spectroscope helps developers build intuition about system behaviour by visualizing request-flow graphs. Section 4.2 describes how Spectroscope helps diagnose performance changes by comparing system behaviours. Section 4.3 describes how mutations can be explained by identifying the low-level parameter differences between them and their originators.

4.1 Visualizing system behaviour

To help developers build intuition about the behaviour induced by a workload, Spectroscope presents a visualization of the unique paths taken by requests through the system. To do so, it takes as input the generated request-flow graphs and bins them into *categories* of unique paths. Unique paths are determined by encoding request-flow graphs into strings via a depth-first traversal—strings that differ represent unique paths. Binning to aggregate information is necessary because distributed storage systems can service 100s to 1000s of requests per second, so it is infeasible to visualize all of the request-flow graphs generated by anything other than the most trivial workloads. Request-flow graphs are binned based on path because of the expectation that requests that take similar paths through the distributed system should incur similar costs.

For each category, Spectroscope identifies the number of requests it contains, average response time and variance. To identify where time is spent by requests within a category, Spectroscope also computes average edge latencies and corresponding variances. As output, Spectroscope shows DOT graphs of the path taken by requests in each category. Each graph is overlaid with the computed statistical information. The categories can be ranked either by average response time, or variance in response time—both have proven useful for helping identify performance problems. Categories that exhibit high average response times are obvious points for optimization. Alternatively, high variance in response time is an indication of contention for a resource.

Though binning request-flow graphs into categories reduces the amount of data presented to the developer by several orders of magnitude—for example, `linux-build`, which generates over 280,000 request-flow graphs, yields only 457 categories—this is still too much detail to present if the developer is trying to gain a coarse-grained sense of system behaviour in response to a workload. To enable such coarser-grained views, Spectroscope supports *zooming* functionality.

Initially, Spectroscope presents a completely zoomed-out view in which categories are created by binning filtered versions of request-flow graphs that only expose the components accessed; nodes represent RPC call and reply events, and edge latencies represent time spent within the component. These zoomed-out graphs are both smaller in size than their fully detailed counterparts and induce fewer categories. As such, the zoomed-out view allows the developer to more easily gain a global sense of system behaviour. He can identify which components tend to be accessed and how much time is spent in each without being encumbered by the details of the paths taken by requests within them. In the zoomed-out view, only 194 categories are generated by `linux-build`. If the developer notices interesting behaviour in the zoomed-out view (e.g., a category in which a component traversal takes a long amount of time), he can choose to "zoom in" by exposing the instrumentation within that component, thus revealing the paths taken by requests within it.

4.2 Comparing system behaviours

When comparing system behaviours, Spectroscope takes as input activity from a *non-problem period* and a *problem period*. It identifies behaviour in the problem period that is different from the non-problem period and, for each of these mutations, identifies their candidate originators—what the mutations could have looked like in the non-problem period. Finally, Spectroscope ranks the observed mutations by how much they contribute to the overall change in performance.

Ranking is necessary for two reasons. First, the performance problem might have multiple root causes, each of which represents 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. This is because requests directly affected by the root cause will effect changes in system resource usage, and hence, induce even more mutations. To illustrate this point, consider a newly introduced bug in a cache that causes more requests to miss in that cache than before. The requests directly affected by this bug will use more space in the next cache in the system, forcing some requests that used to hit in this next-level cache to also start missing. 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.

There are several challenges involved in comparing system behaviours, listed below.

Mutations must be automatically identified: Spectroscope currently considers two possible mutations in behaviour. *Structural mutations* represent requests that have changed in the path they take through the distributed system. *Response-time mutations* represent requests that take the same path, but which have changed in the time needed to service them (i.e., their response time).

The candidate originators must be automatically identified: Identifying candidate originators serves two purposes. First, it allows developers to compare the path and costs of a mutation to its corresponding path and costs during the non-problem period, thus aiding his intuition. Second, it allows mutations to be ranked by how much they affected the change in performance. Heuristics are needed to identify the candidate originators of structural mutations; candidate originators of response-time mutations are requests from the non-problem period that belong to the same category as the mutation.

Natural variations must be distinguished from true behaviour changes: It is unreasonable to assume two periods of activity will yield absolutely identical behaviour. There will always be some natural variance in the response time of requests and in the path taken by requests through the system. Statistical tests are required to disambiguate this natural variance from true behaviour changes.

4.2.1 Identifying Response-time mutations

Spectroscope uses the Kolomogrov-Smirnov two-sample, non-parametric hypothesis test [22] to automatically identify response-time mutations. In general, two-sample hypothesis tests take as input two sample distributions and determine whether the null hypothesis, that both are generated from the same underlying distribution, is false. The decision of whether to reject the null hypothesis is made so as to keep the false-positive rate less than a preset threshold. 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 Kolomogrov-Smirnov test was chosen specifically because it identifies differences in both the shape of a distribution and its median. Pinpoint explored the use of the Man-Whitney U test, which looks for only differences in the median [7].

The Kolomogrov-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. Categories that contain too few requests to run the test accurately are not marked as response-time mutations by default. To help identify what specific interaction or component accounts for the change in response time, 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.

When first using the Kolomogrov-Smirnov test, we often found it was too sensitive to changes in edge latency and response time. It would identify categories as containing response-time mutations even if the change in average response time was very small, and not worth diagnosing. To correct this problem, response times, which had been specified in microseconds, are now rounded to the nearest millisecond before being input into the test.

4.2.2 Identifying structural mutations

Since Spectroscope assumes the same workload was run in both the non-problem period and the problem-period, 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. As such, categories that contain more requests from the problem period than from the non-problem period are labeled as containing structural mutations and those that contain less are labeled as containing originators. Before structural mutation categories and originator categories can be identified in this manner, however, it is necessary to determine whether these changes in frequency actually represent a change in system behaviour, or are just due to natural variance. This is currently accomplished by using a threshold.

Spectroscope identifies a category as containing structural mutations only if it contains `SM_THRESHOLD` more problem-period requests than non-problem-period requests. Similarly, categories are identified as containing originators if they contain `O_THRESHOLD` more non-problem-period requests than problem-period requests. In our experience diagnosing problems observed in Ursa Minor's standard 5-component configuration, we have found that good values for these thresholds range between 10 and 50. It is likely, however, that developers will have to experiment with different values for these thresholds when diagnosing problems that arise in different configurations, or in different classes of workloads than those currently used to test Ursa Minor. To help alleviate some of the fragility associated with using thresholds, a χ^2 statistical hypothesis test [34] was implemented as a first-pass filter for determining whether overall structural behaviour had changed. However, we found that there is always enough structural variance in request structure for such tests to always reject the null hypothesis.

Once the total set of structural-mutation categories and originator categories are identified, Spectroscope must identify the candidate originators for each structural mutation. That is, it must identify the set of originator categories that are likely to have contributed requests to a given structural-mutation category during the problem period. Preferably, these candidate originators should also be ranked in a meaningful way. Several heuristics are used to achieve this and are described below; Figure 4 shows how they are applied.

First, the total list of originator categories is pruned to eliminate categories which contain requests 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; since Spectroscope assumes the same workload is executed during the problem and non-problem period, it is safe to assume that requests of different high-level types cannot be originator/mutation pairs.

Second, originator categories for which the decrease in request-count between the problem and non-problem periods is less than the increase in request-count of the structural-mutation category are also pruned. This reflects a 1 to unique N assumption between candidate originator and mutation categories. It is made because, in the common case, structural mutations will exhibit a common subpath until the point at which

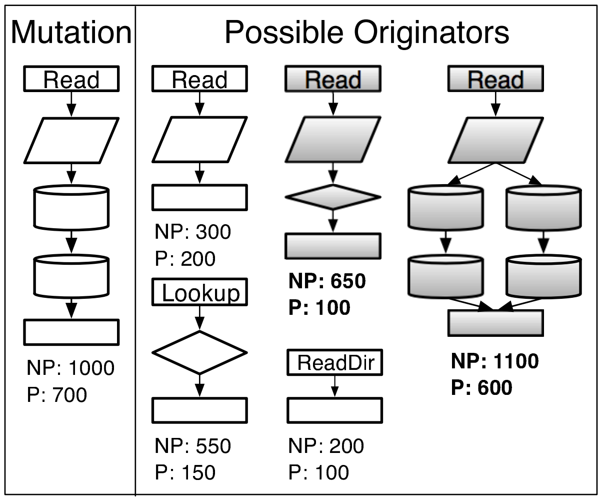


Figure 4: **How the candidate originator categories of a structural-mutation category are identified.** The shaded originator categories shown above will be identified as the candidate originators of the structural-mutation category shown. The originator categories that contain LOOKUP and READDIR requests will not be identified as candidates because their constituent requests are not READS. The top left-most originator category contains READS, but will not be identified as a candidate because the decrease in frequency of its constituent requests between the problem (P) and non-problem period (NP) is less than the increase in frequency of the structural-mutation category.

they are affected by the problem and many different subpaths after that point. 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. 1 to unique N instead of 1-N is assumed because of the large amount of instrumentation present in the system, minimizing the probability of requests from two different originator categories contributing requests to the same structural-mutation category. Since the 1 to unique N assumption may not hold for certain performance problems, the developer can optionally choose to bypass this heuristic when identifying the candidate originator categories.

Finally, the remaining originator categories are identified as the candidates and ranked according to probability of being a "true" originator of the structural mutation. Specifically, they are ranked according to their similarity in structure to the mutation. This reflects the heuristic that originators and structural mutations are likely to resemble each other. Similarity in structure is calculated by computing the normalized string-edit distance between each of the candidate originator categories and the structural mutation. Though computation of normalized string-edit distance is relatively expensive—$O(K^2)$ time is required to calculate the edit distance between two strings of length K—the cost can be amortized by storing a database of edit distances between previously observed paths taken by requests. This database can be reused and added to whenever Spectroscope is run.

4.2.3 Ranking

Both structural-mutation categories and response-time-mutation categories are ranked 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 it contains times the change in average response time in the problem period between it and its candidate originator categories. If more than one candidate originator 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 contained in the category 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 originators and response-time mutations, or both structural and response-time mutations. Such categories are split into multiple 'virtual categories,' each of which is ranked independently.

4.3 Explaining mutations

In many cases, it is possible to identify why a mutation occurred by identifying the differences in low-level parameters seen by the mutation and its originator. Knowledge of these differences can further help the developer identify the root cause, or may themselves be the root cause. For example, a response-time mutation might be caused by a component sending more data in its RPCs than during the non-problem period. Similarly, a structural mutation might be caused by a client which, during the problem period, issues WRITEs at a smaller granularity than that natively supported by the distributed storage system. To service these small WRITEs the storage system must first retrieve data from the storage-nodes (or disk) at its native granularity and then insert the client's data at the requested offset. Such *read-modify writes* can severely reduce the performance of a storage system, as every WRITE incurs an extra READ.

Spectroscope allows developers to pick a mutation category and candidate originator category for which they wish to identify low-level differences. Given these categories, Spectroscope induces a regression tree [4] that shows low-level parameters and associated values that best separate requests in these categories. Each path from root to leaf represents an independent explanation, in terms of parameters and associated values, of why the mutation occurred. Since developers may already possess some intuition about what differences are important, we envision them utilizing Spectroscope's ability to explain mutations interactively. If the developer does not like the set of explanations chosen, he can select a new set of explanations by simply removing the parameter chosen as the root 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 portion of request structure that is common between requests in both categories. Since low-level parameters are expected to differ once the mutation starts to differ, structurally or edge latency-wise, from its originator, the template describes only the common structure until the point of the first difference as observed by the depth-first traversal. Second, a table in which rows represent requests and columns represent associated parameter values is extracted by iterating through each request in the categories and extracting parameters observed in portions of requests that fall within the template. Each row is labeled as belonging to the problem or non-problem period. In creating this table, 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.

5 Evaluation

This section presents a quantitative evaluation of Spectroscope. A qualitative evaluation of Spectroscope's utility in aiding diagnosis efforts was presented in Section 2. Section 5.1 describes the experimental setup. Section 5.2 evaluates the expectation that requests that take the same path through a distributed system will incur similar response times. Section 5.3 evaluates Spectroscope's ability to automatically identify response-time mutations and structural mutations. Section 5.4 shows that the low-level parameters that differentiate a given mutation and originator can be automatically identified.

5.1 Experimental setup

All experiments described in this section were run using a simple 5-component configuration of Ursa Minor, as shown in Figure 3. It consists of one NFS server, one metadata server, one Stardust trace collection server, and two storage nodes. The NFS server and metadata server were co-located on one machine; all other components were run on separate machines. The machines used were Dell SuperMicro 6014HT servers with two Intel 3.00GhZ Xeon processors and 2GB of RAM.

Three benchmarks were used in the evaluation. For all benchmarks, a request-sampling rate of 20% was used.[2] Both `SM_THRESHOLD` and `O_THRESHOLD` were set to 10.

`Linux-build`: This benchmark consists of copying the linux 2.6.32 tarball to Ursa Minor and building it from scratch.

`Postmark-large`: This synthetic benchmark is designed to evaluate the small file performance of storage systems. Such small-file workloads can be induced by e-mail applications and web-based commerce [17]. For this evaluation, Postmark has been configured to utilize 224 subdirectories, 50,000 transactions, and 100,000 files. This configuration yields a more strenuous workload than the default settings.

`SPEC SFS 97 V3.0 (SFS97)`: This synthetic benchmark is the industry standard for measuring storage system scalability and performance [29]. It applies a periodically increasing load of NFS operations to a storage system's NFS server and measures the average request response time. For this evaluation, SPEC SFS was configured to generate load between 50 and 350 operations/second in increments of 50 operations/second. 350 operations/second represents the limit of the 5-component configuration of Ursa Minor.

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

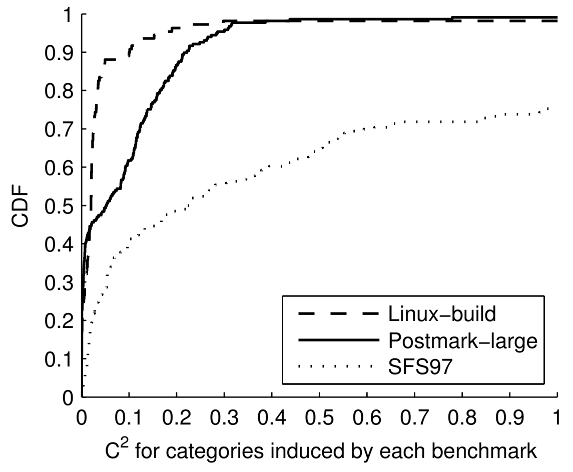
5.2 Requests that take the same path will incur similar response-times

When comparing system behaviours, Spectroscope relies on the expectation that requests that take the same path through a distributed storage system should incur similar costs (response times and edge latencies) and that deviations from this expectation indicate behaviour changes. Spectroscope's ability to identify response-time mutations is most sensitive to this expectation, whereas only the ranking of structural mutations is affected.

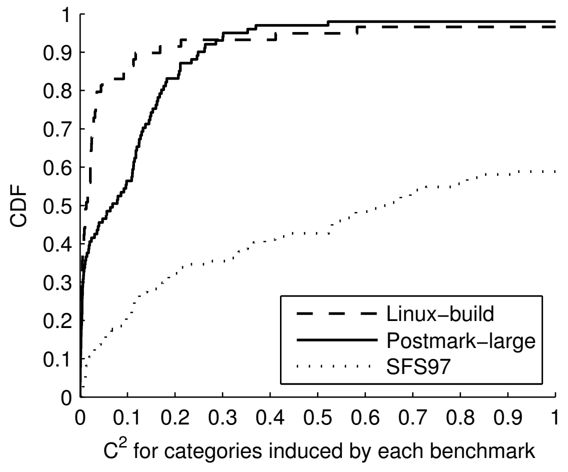
For response-time mutations, both the number of false negatives (i.e., categories incorrectly identified as **not** containing mutations) and false positives (i.e., categories incorrectly identified as containing mutations) when comparing system behaviours will increase with the number of categories that do not meet the expectation. The former will increase when categories do not meet the expectation due to high variance in response time, as this will reduce the Kolomogrov-Smrinov test's power to differentiate true behaviour changes from natural variance. The latter will increase when categories do not meet the expectation because requests within them exhibit similar response times during a single run of the system, but different response times across runs. It is important to note that both cases are likely to be caused by latent performance problems within the system and thus should be investigated in of themselves.

Figure 5(a) shows the CDF of the squared coefficient of variation (C^2) of response time for large categories induced by `linux-build`, `Postmark-large`, and `SFS97`. 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, while those with C^2 greater than one exhibit high variance [13]. As such, C^2 for a category is a measure of how well it meets the similar paths/similar categories expectation. *Large categories* contain over 10 requests; Table 2 shows that these categories account for only 17%—29% of all categories, but contain over 99% of all requests. Categories containing a smaller number of requests are not included because the C^2 statistic is not meaningful for them; including these categories unfairly biases the results optimistically.

[2]20% is the minimum sampling rate needed to diagnose problems in `linux-build`. Lower sampling rates could be used for larger benchmarks.



(a) C^2 for categories induced when using full request-flow graphs



(b) C^2 for categories induced when graphs are filtered to include only component traversals

Figure 5: **CDF of C^2 for large categories induced by various benchmarks.** These graphs show the squared coefficient of variation of response-time, a normalized measure of variance defined as $(\frac{\sigma}{\mu})^2$, for large categories induced by `linux-build`, `postmark-large`, and `SFS97`. Requests in categories for which C^2 is less than 1 exhibit low variance in response time, whereas those in categories for which C^2 is greater than one exhibit high variance. Figure 5(a) shows that when the full request-flow graphs are used, 97% of categories induced by `linux-build` and `postmark-large` exhibit C^2 less than 1. For `SFS97`, 73% of categories induced exhibit C^2 less than 1. Figure 5(b) shows that when the request-flow graphs are filtered to show only component traversals, 59% of categories induced by `SFS97` exhibit C^2 less than 1.

For all of the benchmarks, at least 73% of the large categories induced exhibit C^2 less than 1. C^2 for all the categories generated by `postmark-large` is small. 98% of all categories exhibit C^2 less than 1 and the maximum C^2 value observed is 7.12. The results for `linux-build` are slightly more heavy-tailed; 97% of all categories exhibit C^2 less than 1, but at the 99% percentile $C^2 = 325.23$ and the maximum value C^2 value observed is 724.9. The categories induced by `SFS97` display the highest variance of the 3 benchmarks. Only 73% of all categories exhibit C^2 less than 1. C^2 remains under 10 until the 97^{th} percentile and at the 99^{th}, $C^2 = 381$. The maximum C^2 value observed is 1292. We believe that the large tail observed in `linux-build` and `SFS97` is motivation for creating a tool to analyze the main sources of variance in a distributed system, as we are interested in identifying the latent problems they represent.

Ursa Minor's `rpcgen` mechanism automatically instruments RPCs; additional instrumentation within components has been added by the developer. Though, for most components, developer's of Ursa Minor have undertaken the effort necessary to add such intra-component instrumentation, other systems may not posses such granularity. For example, X-Trace for Hadoop [19], Pinpoint [7], and Dapper [9] only capture inter-component activity (e.g., only RPCs are instrumented). This more limited instrumentation granularity means that paths taken within components cannot be disambiguated. Intuitively, one would expect the squared coefficient of variation for the induced categories to be larger. In the worst case, the variance within categories might be too large to make use of statistical tests to properly identify response-time mutations.

Figure 5(b) shows the CDF of the squared coefficient of variation when the request-flow graphs are filtered so as to include only component traversals. As expected, zooming out reduces the number of categories created, but increases variance within each category. `linux-build` and `postmark-large` are not

	`linux build`	`postmark large`	`SFS97`
# of categories	457	748	1,186
large categories	23.9%	29.0%	17.4%
# of requests sampled	283,510	126,245	405,278
requests in large categories	99.7%	99.2%	98.9%

Table 2: **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 5(a). Though many categories are generated, most contain only a small number of requests. *Large categories*, which contain more than 10 requests, account for between 17-29% of all categories generated, but contain over 99% of all requests.

affected much, but, the percentage of categories induced by `SFS97` that exhibit C^2 less than 1 drops from 73% to 59%. To reduce C^2 when only component-granularity graphs are available, parameters contained in RPCs that are representative of the amount of work to be done within a component should be added to the request-flow graphs. Doing so would disambiguate requests that place different demands on the relevant component and thus reduce C^2.

5.3 Response-time and structural mutations are identified

Spectroscope's ability to automatically identify response-time mutations and structural mutations was evaluated by injecting known performance problems in Ursa Minor and judging how well Spectroscope was able to identify the resulting mutations. The non-problem period behaviour was obtained by simply running `linux-build` on Ursa Minor, whereas the problem period behaviour was obtained by running `linux-build` against Ursa Minor with the performance problem injected; `linux-build` was used, as opposed to the other benchmarks, because it represents a real-world workload. Section 5.3.1 evaluates response-time mutation identification and Section 5.3.2 evaluates structural mutation identification.

For all of the experiments, the *normalized discounted cumulated gain* (nDCG) [15] is used to evaluate the quality of Spectroscope's output. nDCG is a metric commonly used in information retrieval. It captures the notion that, for a ranked list, highly-ranked false positives (e.g., those ranked in the top 10) should affect the judged quality of the results more than low-ranked ones (e.g., those that appear at the end of the list). As such, nDCG naturally aligns with the way developers will utilize the ranked list of mutated categories output by Spectroscope. They will naturally focus their diagnosis efforts on higher ranked items first; if these highly-ranked categories are relevant—that is they are useful in helping diagnose the root cause—they will not investigate the lower-ranked ones, some of which may be false positives.

The equations in Figure 6 show how the nDCG is calculated. First, each category in the ranked list is assigned a relevance score of 0 or 1, depending on whether it contains mutations that were induced by the injected problem and would help the developer diagnose the root cause; items assigned a relevance score of 0 are false positives with respect to the induced problem, but in many cases, represent other undiagnosed problems in the system. The *discounted cumulated gain* (DCG) is then calculated by summing the scores, discounted by a value logarithmically proportional to the corresponding category's rank. Finally, the nDCG is calculated by normalizing the DCG by the score that would be returned for an ideal version of the ranked

$$
\begin{aligned}
DCG &= rel_1 + \sum_{i=2}^{N} \frac{rel_i}{log_2(i)} && (1)\\
IDCG &= ideal\_rel_1 + \sum_{i=2}^{N} \frac{ideal\_rel_i}{log_2(i)} && (2)\\
nDCG &= \frac{DCG}{IDCG} && (3)
\end{aligned}
$$

Figure 6: **Equations for calculating the normalized discounted cumulated gain (nDCG).** In Equation 1, the discounted cumulated gain is calculated by assigning a relevance score of 0 or 1 to each item in the ranked list and summing a logarithmically discounted version of these scores. In Equation 2, the DCG of an ideal version of the ranked list, in which false positives are ranked at the end and relevant items are ranked near the beginning, is calculated. Finally, in Equation 3, the nDCG is calculated by normalizing the DCG against the ideal DCG score.

list, in which all relevant categories are ranked at the top and false positives at the bottom. The nDCG can vary from 0 (bad) to 1 (excellent). Since a ranked list that contains only one relevant category, but which is ranked first, will yield a nDCG of 1, both the total number of mutated categories output by Spectroscope and the fraction of those that are relevant are also presented.

5.3.1 Response-time mutations are identified

To evaluate Spectroscope's ability to automatically identify response-time mutations and localize the interactions responsible for the mutation, a 1ms spin loop was injected into the storage nodes' WRITE code path at the point just before the RPC reply is sent. As such, during the problem period, any WRITE requests that accessed a storage node incurred an extra 1ms delay. The request-flow graphs for these WRITEs exhibit this delay in edges whose destination node is marked as a `STORAGE_NODE_RPC_REPLY`. Specifically, In request-flow graphs from the non-problem period, these edges exhibit an average latency of 0.02ms, whereas in the problem period, their average latency increased to 1.02ms. This injected problem increased the run time of `linux-build` from 44 minutes to 51 minutes and is representative of the class of problems for which identification of response-time mutations is useful.

Table 3 summarizes the ranked list of mutated categories output by Spectroscope; it shows that Spectroscope was able to correctly identify the induced response-time mutations. Of the 95 categories identified, 68 were identified as response-time mutation categories and contained WRITEs that that accessed a storage node. Additionally, for each of these categories, at least one $\star \rightarrow$ `STORAGE_NODE_RPC_REPLY` edge was identified as the cause of the mutation. Since these categories would help a developer diagnose the injected problem, they were all deemed relevant. The resulting nDCG was 0.93.

Table 4 shows the positions of relevant categories and provides intuition for why a nDCG of 0.93 was obtained. Though most of the highest-ranked categories are relevant, the 1^{st} and 5^{st} ranked categories are not and thus hurt the NDCG score. Both contain response-time mutations; they are caused by a latent contention problem in the NFS server that yields vastly different response times across runs depending on the order in which concurrent requests are serviced. Many low-ranked categories also are relevant; these categories are affected by the injected problem, but do not contain enough requests to be awarded a high rank.

The remaining false positives all contain structural mutations; the ones ranked higher than 73 are induced either due to variance between runs in the amount of data sent by the linux client during individual WRITE operations, or are induced as a by-product of the unusual CREATE behaviour described in Section 2.2.

Total number of requests sampled	565,295
Total number of categories	605
Number of categories identified as mutations	95
Percent relevant	72% (68)
nDCG	0.93

Table 3: **Evaluation of Spectroscope's ability to identify response-time mutations.** A 1ms spin-loop was injected into Ursa Minor's storage nodes and Spectroscope was used to compare system behaviour before and after this change. Of the 95 total categories identified by Spectroscope as containing mutations, 72% would have helped the developer diagnose the root cause. Since these relevant categories were awarded high ranks, a nDCG of 0.93 was achieved.

Rank										
1		x	x	x		x	x	x	x	x
11	x	x	x	x	x	x	x	x	x	x
21		x	x	x	x	x	x	x	x	x
31	x	x	x	x	x	x		x	x	x
41	x	x	x	x	x		x	x	x	x
51	x	x	x	x	x	x	x	x	x	x
61	x	x	x	x	x	x	x	x	x	x
71	x	x		x						
81										
91										

Table 4: **Visualization of the ranked list of mutated categories output by Spectroscope when evaluating its ability to identify response-time mutations.** Slots that contain x's denote the ranks of relevant categories, whereas blank slots denote false positives. Most high-ranked categories are relevant, thus allowing for a high nDCG score. The false positives awarded the 1^{th} and 5^{th} ranks are caused by already existing latent problems in Ursa Minor.

The ones ranked 73 and lower are induced due to natural variance in the way in which Ursa Minor services requests; it is likely that a smaller value for `SM_THRESHOLD` would eliminate them. Since none of these false positives are assigned high ranks, they are unlikely to mislead the developer during diagnosis of the injected problem.

5.3.2 Structural mutations are identified

To evaluate Spectroscope's ability to identify structural mutations and corresponding candidate originator categories, the NFS server's data cache size was reduced from 384MB to 20MB for the problem-period run. As such, when comparing system behaviours between the non-problem and problem period, one would expect Spectroscope to identify structural-mutation categories that contain requests that missed in the NFS server's cache during the problem period. Candidate originator categories for each of these structural mutation categories should contain requests that hit in cache during the non-problem period. This injected problem is representative of the class of performance problems for which identification of structural mutations is useful.

Table 5 summarizes the results of running Spectroscope. Of the 32 categories identified as containing

Total number of requests sampled	567,657
Total number of categories	622
Number of categories identified as mutations	32
Percent relevant	53% (17)
nDCG	0.83

Table 5: **Evaluation of Spectroscope's ability to identify structural mutations.** Ursa Minor's NFS server's cache size was changed from 384MB to 20MB and Spectroscope was used to compare the system behaviour before and after this change. Of the 32 total categories identified by Spectroscope as containing mutations, 53% would have helped the developer diagnose the cause. Since these relevant categories were awarded high ranks, a nDCG of 0.83 was achieved.

Rank										
1		x		x	x	x	x	x	x	x
11	x	x		x	x		x	x	x	
21		x							x	
31										

Table 6: **Visualization of the ranked list of mutated categories output by Spectroscope when evaluating its ability to identify structural mutations.** Slots that contain x's denote the ranks of relevant categories, whereas blank slots indicate false positives. Most high-ranked categories are relevant, allowing for a high nDCG score. The false positive awarded the 1^{th} rank is caused by the same latent problem in Ursa Minor as that which induced the false positives awarded the 1^{th} and 5^{th} ranks when evaluating Spectroscope's ability to identify response-time mutations.

mutations, 31 were labeled as structural-mutation categories and 1 labeled as a response-time-mutation category. 17 of the structural-mutation categories contain READDIR, WRITE, and READ, and REMOVE operations that missed in cache during the problem period; the highest ranked candidate originator category for each contains requests from the non-problem period that hit in cache. Since these categories would help a developer diagnose the cause of the problem, they were all deemed relevant, yielding a relatively high nDCG score of 0.83.

Table 6 shows the ranks assigned to the mutations identified by Spectroscope. Once again, most high-ranked categories are relevant. False positives assigned the 1^{st} and 3^{rd} ranks hurt the nDCG score the most; the former contains response-time mutations and occurs as a result of the same latent contention problem observed when evaluating Spectroscope's ability to identify response-time mutations. The root cause of the latter is not yet known. Lower-ranked false positives all contain structural mutations and are induced due to natural variance in the way REMOVE operations are serviced by Ursa Minor.

Spectroscope's results when identifying structural mutations both exhibit a lower nDCG score and a higher ratio of false positives to relevant results than those obtained when evaluating response-time mutation identification. This is because the performance impact of changing the NFS server's cache size is not as great as that of adding a 1ms delay to the storage nodes. Less categories and requests are affected, making it harder for Spectroscope to properly rank relevant categories higher than false positives. Additionally, there are less relevant categories overall, allowing for a higher ratio of false positives to relevant results. It is important to note that even though the cache size change affected performance only slightly, Spectroscope was still able to return results that would help developers diagnose the problem.

5.4 Low-level differences are identified

The `IoZone` benchmark used to evaluate Spectroscope's ability to identify the low-level parameters that best differentiate requests contained in a given mutation and originator category. Request-flow graphs for the non-problem period were obtained by simply running `IoZone`. Problem period request-flow graphs were obtained by mounting Ursa Minor's NFS server on the client with the `wsize` option changed from 16KB, the fundamental granularity supported by Ursa Minor's NFS server, to 2KB. As such, during the problem period, clients issued `WRITE` requests containing only 2KB of data, whereas during the non-problem period, each `WRITE` request contained 16KB of data. This change resulted in many read-modify writes.

Spectroscope was used to identify the low-level differences between a category containing read-modify writes and a category containing regular writes. The resulting regression tree indicated that the requests in the two categories were perfectly separated by a request's `count` parameter, which indicates how much data should be written. Specifically, the results indicated that requests with `count` $>$ 2KB should be classified as belonging to the non-problem period, and those with `count` $<$ 2KB should be classified as belonging to the problem period.

6 Related work

There has been a significant amount of work in problem diagnosis in recent years. This section categorizes them.

General purpose tools: Tools such as Pip [27] and TAU [30] are designed with the same overall goal as Spectroscope: they aid diagnosis of general performance problems by helping developers build intuition about system behaviour. Both Pip and Spectroscope compare expected system behaviour to actual behaviour and report the differences. While Spectroscope uses behaviour observed during periods of acceptable performance as the expected behaviour, Pip requires users to manually write these expectations. Expectations must be specified for each component or subcomponent for which behaviour is to be compared, and thus cannot easily be used to help developers understand the overall behaviour of the system. The TAU toolchain is an infrastructure for understanding MPI behaviour via profiling and visualization. TAU can identify the overall latency of K-deep callpaths, an approach resembles Spectroscope's ability to identify the average response time of requests within a category.

Tools for identifying a specific class of problem: Diagnosis tools such as MUVI [21], Pinpoint [7], Project 5 [2], Whodunit [6], and DARC [32] are designed to identify very specific classes of problems and, as such, represent an alternative first step toward the goal of automating problem diagnosis. MUVI uses data-mining techniques to identify cases in linux in which correlated variables, such as a string and associated length, should be held together in a critical section, but are not. Pinpoint uses black-box end-to-end traces to identify failed components in a distributed system. Project 5 uses black-box traces of message passing events to identify the inter-component bottlenecks in a distributed system. Whodunit uses end-to-end tracing to aid profiling efforts in distributed transactional systems. DARC is a single process tool that automates profiling. It identifies the function call subpath that contributes most to latency of a particular class of function executions. This approach is similar to Spectroscope's ability to identify the average edge latencies along the critical path of requests within a category. However, where Spectroscope groups requests by structure, DARC's classes are made of function executions that incur similar latencies.

Tools for identifying anomalies: Many tools exist to identify anomalies in distributed systems; Spectroscope differs from all of these in that it attempts to identify changes in behaviour, not anomalies. The tool described by Xu [35], PeerPressure [33], and Magpie [3], use machine-learning-based approaches. Of these Magpie bears the most relevance to Spectroscope, as its anomaly detection algorithm operates on request-flow graphs obtained from end-to-end traces. Mapgie uses clustering algorithms to group traces and identifies unitary clusters as anomalies. Features used by the clustering algorithm include the serialized request structure and resource-usage metrics. Spectroscope initially also used clustering algorithms to reduce the number of categories [28], but we found that doing so impeded diagnosis efforts, as it made understanding how requests in different categories differ more difficult. It also proved difficult to automatically determine the right number of clusters. It is possible that the former issue can be alleviated by sophisticated distance metrics and feature sets that align with a developer's notion of request similarity.

Many diagnosis tools, such as Ganesha [25] and that described by Kasick [16], utilize black-box metrics and peer comparison to identify anomalous activity in tightly-coupled systems. Though such black-box tools can be scaled easily and readily applied to existing systems, they cannot be used to help developers understand complex system behaviour.

Tools for identifying problems as a reoccurence: Tools such as SLIC [8] and that described by Yuan [36] use machine learning to identify performance problems as reoccurences of those seen in the past. These tools would compliment Spectroscope as well as all other diagnosis tools described in this section.

7 Conclusion

Spectroscope uses end-to-end traces of activity in distributed systems to aid developers in diagnosing performance problems. By comparing the request-flow graphs observed in two periods or system versions, Spectroscope identifies mutations and their originators. Experiences and experiments indicate that Spectroscope can provide significant assistance with this long-standing challenge.

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] 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.

[9] Google technical presentation: It's 11pm, and do you know where your RPC is?, March 2008.

[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] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. *USENIX Annual Technical Conference* (Monterey, CA, June 1999). USENIX Association, 1999.

[12] Graphviz. www.graphviz.com.

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

[14] 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.

[15] Kalervo Jarvelin and Jaana Kekalainen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, **20**(4):442–446. ACM, October 2002.

[16] 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.

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

[18] Jeffrye O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, **36**(1):41–50. IEEE, January 2003.

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

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–565. ACM, 1978.

[21] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *ACM Symposium on Operating System Principles* (Stevenson, MA, 14–17 October 2007), pages 103–116. ACM, 2007.

[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] 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.

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

[30] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, **20**(2):287–311. SAGE, 2006.

[31] 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.

[32] 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.

[33] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 06–08 December 2004), pages 245–258. USENIX Association, 2004.

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

[35] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. *ACM Symposium on Operating System Principles* (Big Sky, MT, 11–14 October 2009), pages 117–132. ACM, 2009.

[36] 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.