Categorizing and differencing system behaviours

Raja R. Sambasivan, Alice X. Zheng, Eno Thereska, Gregory R. Ganger
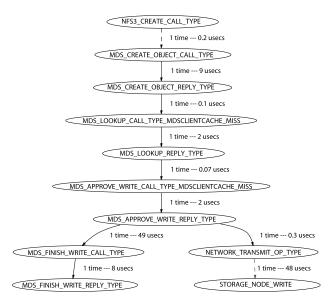*Carnegie Mellon University*

Abstract

Making request flow tracing an integral part of software systems creates the potential to better understand their operation. The resulting traces can be converted to per-request graphs of the work performed by a service, representing the flow and timing of each request's processing. Collectively, these graphs contain detailed and comprehensive data about the system's behavior and the workload that induced it, leaving the challenge of extracting insights. Categorizing and differencing such graphs should greatly improve our ability to understand the runtime behavior of complex distributed services and diagnose problems. Clustering the set of graphs can identify common request processing paths and expose outliers. Moreover, clustering two sets of graphs can expose differences between the two; for example, a programmer could diagnose a problem that arises by comparing current request processing with that of an earlier non-problem period and focusing on the aspects that change. Such categorizing and differencing of system behavior can be a big step in the direction of automated problem diagnosis.

1. Introduction

Autonomic computing is a broad and compelling vision, but not one we are close to realizing. Many of the core aspects require the ability to understand runtime system behaviour more deeply that human operators and even developers are usually able to. For example, healing a performance degradation problem observed as increased request response times requires understanding where requests spend time in a system and how that has changed; without such knowledge, it is unclear how potential corrections or the root cause can be identified, or even which parts of a complex distributed service to focus on. The (dearth of) tools available to distributed service developers for understanding and debugging the behaviour of their systems offers little hope for those seeking to automate diagnosis and healing—if we don't understand how to do such things with the direct involvement of the people who build a system, why will it be possible without them?

This paper discusses a new approach to understanding the behaviour of a distributed service based on analysis of

Figure 1: Example request flow graph. This graph represents the processing of a CREATE request in Ursa Minor, a distributed storage service. Nodes are identified by an instrumentation point (e.g., STORAGE_NODE_WRITE) and any important parameters collected at that instrumentation point. Edges are labeled with a repeat count (see Section 2) and the average latency seen between the connected points. The tracing infrastructure includes information that allows a request's processing to be stitched together from the corresponding activity across the distributed system—for example, the CREATE request involves activity in the NFS front-end, the metadata service (MDS), and a storage-node.

end-to-end request flow traces. The rest of this Section describes such traces and discusses how they can be used to attack performance debugging problems. The remainder of the paper describes an unsupervised machine learning tool (called Spectroscope[1]) that uses *K*-means Clustering [4] on collections of request flow graphs, discusses challenges involved with such a tool, and early experiences with using it to categorize and difference system behaviour. Spectroscope is being developed iteratively while being used to analyze and debug performance problems in a prototype distributed storage service called Ursa Minor [1], so examples

---

[1] Spectroscopes can be used to help identify the properties (e.g., temperature, makeup, size) of astronomical bodies, such as planets and stars.

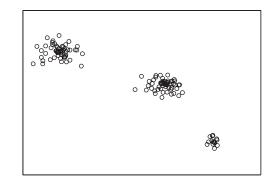from these experiences are used to illustrate features and challenges.

End-to-end request flow traces: Recent research has demonstrated efficient software instrumentation systems that allow end-to-end per-request flow graphs to be captured [3, 8]—that is, graphs that represent the flow of a request's processing within and among the components of a distributed service, with nodes representing trace points reached and edge labels corresponding to inter-tracepoint latencies. See Figure 1 for an example.

The collection of such graphs offers much more information than traditional performance counters about how a service behaved under the offered workload. Such counters aggregate performance information across an entire workload, hiding any request-specific or client-specific issues. They also tend to move collectively with a performance problem, when that problem reduces throughput by slowing down clients; so, for example, when a latency problem causes clients to block, throughput can drop throughout the system with no indication in the counters other than a uniform decrease in rate.

If the important control flow decisions are captured, request flow graphs contain all of the raw data needed to expose the locations of throughput or latency problems. They also show exact flow of processing for individual requests that may have suffered and workload characteristics of such requests. Of course, the challenge is extracting the information needed to diagnose problems from this mass of data.

Understanding system behaviour by categorizing requests: One collection of insights can be obtained by categorizing requests based on similarity of how they are processed. Doing so compresses the large collection of graphs (e.g., one for each of 100s to 1000s of requests per second) into a much smaller collection of request types and corresponding frequencies. The types that occur frequently represent common request processing paths, which allows one to understand what the system does in response to most requests. The types that occur infrequently and requests that are poor matches for any type are outliers, worth examining in greater detail, particularly if they are part of the problem being diagnosed. Figure 2 shows a hypothetical example that illustrates the utility of categorizing via clustering.
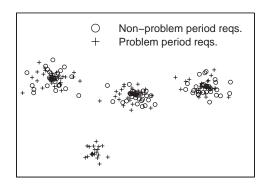
One of the earliest exploratory uses of Spectroscope illustrated the utility of this usage mode. Spectroscope was applied to activity tracking data collected while Ursa Minor was supporting a large compilation workload. The output consisted of a large number of clusters, analysis of which revealed that most were composed of READDIR requests that differed only in the number of cache hits seen at Ursa Minor's NFS front-end. For example, READDIR requests that constituted one cluster saw 16 cache hits, while READDIR requests that constituted another saw only 2 cache hits. Further investigation revealed a performance bug in which, in

Figure 2: Example of categorizing system behaviour. Clustering exposes modes of activity (clusters) when a workload is run against an instrumented system; by analyzing requests in each cluster, one can categorize (label) each mode of activity. This illustration shows three clusters corresponding to the following three categories (from left to right): READs that hit in cache, READs that require disk accesses, and a small number of READs that take an unusually long time to complete. By determining how requests in the third cluster differ from those in the first two, one can localize the portion of the code responsible for inducing the high latencies.

servicing a READDIR, the NFS front-end would iteratively load the relevant directory block into its cache, memcpy one the many requested entries from the cache block to the return structure, and then release the cache block; this tight loop would terminate only when all relevant entries were copied. Each iteration of the loop (one per directory entry) showed up as a cache hit in the activity traces. The fix to this performance bug involved simply shifting the cache block get and release actions from inside the loop. But, most importantly, this example illustrates the value of exposing the actual request flows.

Diagnosing performance changes by differencing system behaviour: Another interesting collection of insights can be obtained by *differencing* two sets of request flow graphs. Identifying similarities and differences between the per-request graphs collected during two periods of time can help with diagnosing problems that are not always present, such as a sporadic or previously unseen problem ("why has my workload's performance dropped?"). Comparing the behaviour of a problem period with that of a non-problem period highlights the aspects of request processing that differ and thus focuses diagnosis attention on specific portions of the system. Categorizing both sets of requests together enables such differencing—one simply computes two frequencies for each type: one for each period. Some types may have the same frequencies for both periods, some will occur in only one of the two periods, and some will occur more in one than the other. This information tells one what changed in the system behaviour, even if it doesn't explain why, focusing diagnosis attention significantly. Figure 3

Figure 3: Example of differencing system behaviour. Clustering the request flow graphs of two workloads (or one workload during two time periods) exposes differences in their processing. This illustration shows clusters for two datasets: the first collected during the existence of some performance problem and the second collected during its absence. The results show four clusters, three of which are equally comprised of requests from both datasets. One cluster, however, is made up of requests from just the problem dataset and is likely to be indicative of the performance problem. Determining how these requests differ from requests in the other clusters can help localize the source of the problem.

shows a hypothetical example that illustrates the results of such clustering.

Early experiences accumulated by injecting performance problems into Ursa Minor have shown that this usage mode holds promise. In one particular experiment, two activity tracking datasets were input into Spectroscope. The first consisted of a recursive search through a directory tree that resulted in a 100% cache hit rate at Ursa Minor's NFS server. The second consisted of the same search, but with the cache rigged so that that some of the search queries hit in cache while others missed. In addition to generating clusters composed of other system activity (e.g., attribute requests, etc.), Spectroscope correctly generated a cluster composed of READs from both datasets and another cluster composed of just READs that missed in cache from the second dataset.

2. Creating a Spectroscope

Spectroscope uses unsupervised learning techniques (specifically K-means clustering) to group similar request flow graphs. Since categories for most complex services will not be known in advance, it is necessary to perform the final part of categorization (attaching labels to each cluster) manually. The utility of Spectroscope depends on three items: the quality of the clustering algorithm used, the choice of feature set, and the quality of the visualization tools employed to help programmers understand the cluster-

ing results. Many exploratory studies are required to identify the best choices for each. This section describes our initial approach to these components of Spectroscope.

2.1. Current clustering implementation

A version of K-means Clustering [4] is currently implemented in Spectroscope. It takes as input a fixed-size table, in which each row represents a unique request and each column a feature. It also optionally accepts a weight indicating the number of times a given unique request was seen. When the goal is to difference system behaviour, Spectroscope has access to a label for each request indicating the dataset to which it belongs. These labels are only used to annotate the final results and are not used during clustering.

The K-means Clustering algorithm implemented in Spectroscope proceeds as follows. First, K random requests are chosen as cluster centers. Each request computes its Euclidian distance to the cluster centers and joins the closest cluster. New cluster centers are then computed as the average of all feature values of requests in each cluster and the process is repeated. Repetition stops when cluster assignment no longer changes. Though K-means will always converge, only a local optimum is guaranteed. As such, the entire process is repeated several times and the best local optimum (i.e., the assignment that best minimizes intra-cluster distance while maximizing inter-cluster distance) is chosen.

In addition to determining the best cluster assignment, Spectroscope must also automatically determine the best number of clusters. The programmer cannot be relied upon to guess this value because doing so would require detailed prior knowledge of the system behaviour.[2] Spectroscope determines the best number of clusters by finding the best cluster assignment for a range of values of K and choosing the one that yields the best global cluster assignment.

2.2. Choosing the feature set

The choice of feature set over which clustering is performed dictates the type of problems Spectroscope can help diagnose. Several options are being explored; the advantages and tradeoffs of each are described below.

Instrumentation points: In this option, each feature in the input table is an unique instrumentation point/value combination that appears in the request flow graphs input to Spectroscope. Column values indicate whether or not the corresponding instrumentation point was observed for a given request. This is both the least expensive and least powerful method of clustering, as it does not utilize the structure of the graphs, or the edge labels.

---

[2]However, the programmer may possess enough intuition to provide a best-guess value, which can be used as a starting point by Spectroscope.

The simulated performance problem mentioned in the introduction, in which cache misses were injected into Ursa Minor, is an example of a problem for which clustering on the instrumentation points is sufficient to help the programmer diagnose the problem. Differences between the instrumentation point names themselves reveal the source of the problem and so no additional detail is required.

Structure+latencies: This mode of clustering is similar to clustering on instrumentation points, except that each feature is an unique edge that is traversed by requests in the input dataset(s). Each edge corresponds to an ordered pair of instrumentation points. Column values indicate the average latency incurred when traversing the corresponding edge for a given request. Clustering on structure+latencies is more expensive than clustering on instrumentation points, partially because many more requests are unique when this mode of clustering is used.

Clustering on structure+latencies is more powerful than just clustering on instrumentation points in that it utilizes the structure of the request path as well as the edge latencies. This feature set is useful when debugging performance problems that arise because a particular component or piece of code is taking longer than expected to perform its work. A classic example of this scenario was seen while building Ursa Minor; specifically, a hash table responsible for storing mappings from filehandles to object names on disk would routinely store all of its mappings in one bucket. As the number of objects in the system grew, queries to the hash table, necessary to determine the location of objects on disk, would take progressively longer to complete. Using Spectroscope on this problem should reveal clusters differentiated only by the latency of the edge that bridges the location of the hash table. This information would allow the programmer to localize the problem and fingerpoint the hash table as the root cause.

Structure+counts: This mode of clustering is identical to clustering on structure+latencies, except that, instead of average latency, the values of each column in the input table are counts of the number of times the corresponding edge was traversed by a given request.

Clustering on structure+counts is useful when debugging problems related to unintended looping, or certain forms of reduction in parallelism. For example, the READDIR performance problem mentioned in the introduction was revealed when clustering using this feature set.

2.3. Presenting the clustering results

The choice of clustering algorithm and feature set are integral to the quality of results returned by Spectroscope, however, in the end, Spectroscope will be useless unless the programmer is able to understand how requests placed in different clusters differ, since it is this knowledge that will help him localize the source of a given performance problem. As a result, good visualization tools are required to help the programmer interpret the clustering results.

There are two main questions that need to be addressed with regards to visualization. First, how should clusters be summarized when being shown to the programmer? Initially, Spectroscope visualized clusters as average graphs constructed by using all of the unique edges contained in the cluster. However, it quickly became apparent that such average graphs are counter-intuitive and confusing, as they usually contain false paths that cannot occur in practice. Currently, calling context trees [2] (CCTs) are being explored as a more intuitive way of summarizing clusters; such CCTs are constructed by iterating from the leaves of each request flow graph to the root and merging pairs of graphs only if the path from the current node to the root for both is identical. As such, CCTs represent every unique path in the request flow graphs they represent without exhibiting redundant information or false paths.

Second, are there ways to help the programmer determine how requests assigned to one cluster differ from requests assigned to others? In many cases, to understand why a request is assigned to a given cluster, it is most useful to compare it to requests assigned to the most similar distinct cluster. For example, consider a cluster that consists entirely of WRITEs. The most knowledge about why requests in this cluster are distinct can be gained by comparing requests in it to requests in another cluster also comprised entirely of WRITEs. To facilitate such comparisons, the utility of a similarity matrix, which identifies Euclidian distances between pairs of cluster centers is being explored. Additionally, to help the programmer determine how two request graphs or CCTs differ, a tool that overlays two graphs on top of one another and highlights the nodes and edges that are most different is being considered.

3. Further challenges & questions

Our initial experiences using Spectroscope has convinced us that it can be a very useful aid in helping debug performance problems and understand system behaviour. However, in further developing Spectroscope, there are several challenges that must be addressed. Most importantly, heuristics must be determined that will allow the quality of the clustering results to be evaluated independently of the end-to-end process of debugging the relevant problem. Without this separation, it will be extraordinarily difficult to develop and/or evaluate new clustering algorithms for use with Spectroscope. Perhaps just as important is the challenge of maximizing generality. In the previous section, different feature sets were chosen as inputs to the clustering algorithm based on the type of performance problem expected. Ideally, this would not be the case; all possible fea-

tures (e.g., structure+counts+latencies) would be used and Spectroscope would be run without thought to the type of problem being debugged. However, instance-based learners, such as K-means Clustering, suffer from the "Curse of Dimensionality" [4]; adding irrelevant features will only serve to decrease the quality of the clustering results. The rest of this section discusses other open questions and challenges with regards to further development of Spectroscope.

Will clustering using Spectroscope scale for large problem sizes? Clustering algorithms, in general, are very computationally expensive. For example, naive implementations of K-means require $O(KdN)$ time to complete one iteration, where K is the number of clusters, d is the number of features, and N is the number of requests. This does not include the number of iterations necessary for the algorithm to converge, or that necessary to find the best number of clusters, or guarantee that the algorithm did not fall into a bad local minima. Unfortunately, a simple 1-minute run of the IoZone benchmark [6] can generate over 46,000 unique requests when clustering on structure+latencies. As such, to make Spectroscope usable for large workloads, it will be necessary to explore and utilize the fastest possible clustering techniques available. Additionally, sampling may be necessary to further reduce the number of unique requests.

How can prior expectations or intuition be used to help direct the results output by Spectroscope? Spectroscope currently uses an undirected approach to help programmers diagnose performance problems. Specifically, it categorizes requests based on the most discriminating differences between them, without consideration of the type of differences the programmer will find most useful in order to solve the problem at hand. Directed approaches, which may be more useful, will require finding ways to mesh the unsupervised machine learning algorithms used by Spectroscope with known models or expectations.

Where must instrumentation be placed in order to identify changes in system behaviour due to various types of performance problems? In developing Spectroscope, it has been so far assumed that the underlying end-to-end tracing mechanism collects all of the information required to help diagnose complex performance problems. Clearly, this is not guaranteed. As such, we are interested interested in developing a taxonomy of performance problems and the instrumentation required to diagnose them.

4. Related work

There has been a large amount of work in the area of problem diagnosis of complex distributed systems in the past few years; such work can be split into two broad categories: expectation-based methods and statistical methods.

Pip, an example of the former, requires programmers to manually create categories of valid system behaviour (by writing expectations) and flags differences when the observed system behaviour does not match [7]. Conversely, Spectroscope does not require explicitly written expectations; instead, it learns categories of system behaviour automatically and leaves it to the programmer to determine which are valid and which require further investigation. Though we appreciate the formality of Pip's approach, we believe our approach is worth exploring, especially for large, complex distributed systems, for which it is hard to know the different categories of behaviour apriori. Pip also allows for expectations to be generated automatically based on observed system behaviour. When used in this mode, Spectroscope can be used to complement Pip by helping difference two sets of automatically generated expectations.

In general, statistical approaches to performance debugging do not attempt to help characterize system behaviour. For example, Cohen et al. attempt to aid root cause diagnosis by using statistical techniques to identify the low-level metrics that are most correlated with SLO violations [5].

Acknowledgements

References

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

[2] G. Ammons, et al. Exploiting hardware performance counters with flow and context sensitive profiling. ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 1997.

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

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

[5] I. Cohen, et al. Correlating instrumentation data to system states: a building block for automated diagnosis and control. Symposium on Operating Systems Design and Implementation. USENIX Association, 2004.

[6] W. Norcott and D. Capps. IoZone filesystem benchmark program, 2002. www.iozone.org.

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

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