Detecting Workload Changes in a Sequence of Request Flow Graphs

Leman Akoglu
Computer Science
Carnegie Mellon University
lakoglu@andrew.cmu.edu

Charles Jackson
Biomedical Engineering
Carnegie Mellon University
charlesjackson@cmu.edu

Tomer Shiran
Electrical and Computer Engineering
Carnegie Mellon University
tshiran@andrew.cmu.edu

April 27, 2008

Abstract

Enterprise-class products, especially distributed systems, are difficult to troubleshoot in the field. Skyrocketing costs per incident are forcing vendors to invest in automation of troubleshooting processes. This project aims to combine granular instrumentation of a distributed storage system with machine learning and stream mining algorithms to detect workload changes in the system's runtime behavior. We develop an auxiliary subsystem that processes the runtime traces of the main system and alerts for anomalies on the fly. Our method further detects sequential patterns of these traces which would help to compress past data and provide system administrators with information about the normal flow of the system.

1 Introduction

Enterprise-class products are very difficult to troubleshoot in the field. This is especially true for distributed systems, as well as systems with many external dependencies. Support costs (measured in cost per incident, or time per incident) are skyrocketing and vendors are desperately looking for ways to reduce them by automating as much of the troubleshooting process as possible.

One way to reduce support costs is by automatically detecting known problems when they reappear. For example, if customer A's issue is diagnosed and then customer B experiences the same issue, the system should automatically be able to diagnose the issue for customer B without involving any human investigation.

Another way to reduce support costs is by detecting problems without relying on a predefined collection of known problems. Even if a customer experiences an issue that was not previously experienced by other customers, it may be possible to detect anomalous behavior in the system which could indicate both the existence of a problem as well as a potential cause.

This project is conducted on Ursa Minor [1], a prototype distributed storage system. Figure 1 outlines the main components of the system, including a custom NFS server that utilizes the client library of Ursa Minor so that unmodified clients can communicate with the storage system. Any results achieved by our research will most likely apply to many different classes of distributed systems, including storage systems, enterprise applications (ie, ERP, CRM), multi-tier Web applications and databases.

Ursa Minor, like most servers in a client-server environment, processes requests that it receives from clients. In this project, the client always communicates with the NFS[10] server of Ursa Minor, so each request corresponds to an NFS procedure (eg, READ, WRITE, COMMIT). As described in [17], the instrumentation framework in Ursa Minor is capable of generating a request flow graph for every request. The nodes of a request flow graph represent various instrumentation points in the source code of Ursa Minor, whereas the edges represent causal relationships between instrument points.

Figure 2 shows a very simple request flow graph representing an NFS read request that was sent to Ursa Minor's NFS server by an external client. As shown in the graph, NFS3_READ_CALL_TYPE_DEFAULT was the first instrumentation point that was reached during the handling of this request, indicating that the NFS server recognized that the request was a READ procedure. The NFS server was able to service the request from its own internal cache, so the next instrumentation point that was reached was NFS_CACHE_BLOCK_OP_TYPE_NFSCACHE_READ_HIT. Finally, the NFS server responded to the client, as indicated by the last instrumentation point,

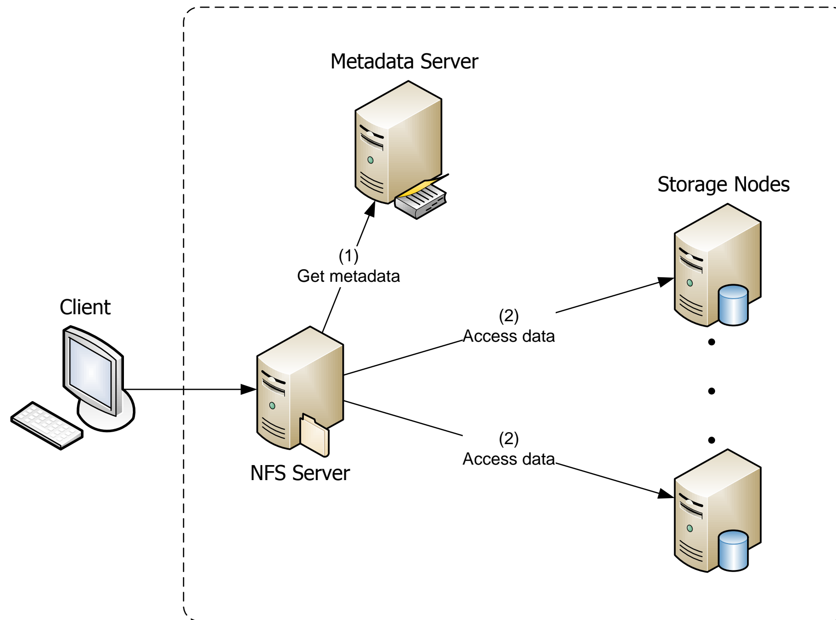

Figure 1: The high-level architecture of Ursa Minor, a cluster-based storage system. The NFS server is, in fact, a client of Ursa Minor, although in this project our client will use the NFS server as an interface to Ursa Minor.

NFS3_READ_REPLY_TYPE_DEFAULT. Each edge is annotated with the latency between the two instrumentation points (in fact, it shows an average time, as explained later in this report). The latencies in the figures are always labeled "R" (for "response time").

Figure 3 shows a slightly more complex request flow graph, representing an NFS read request that the NFS server was unable to serve from its cache. This request resulted in a cache miss at the NFS server, as indicated by the instrumentation point NFS_CACHE_BLOCK_OP_TYPE-_NFSCACHE_READ_MISS. As a result, the NFS server had to retrieve the requested data from one of the system's storage nodes. Each storage node has an internal cache, also known as a front-end, and the request flow graph of Figure 3 shows that this request was serviced from the storage node's cache, as indicated by instrumentation point FRONTEND_BLOCK_OP_START_TYPE-_FRONTEND_READ_HIT. It is worth noting that unlike the preceding request flow graph, this graph exhibits parallelism (ie, multiple threads). Therefore, some nodes branch out into multiple nodes. Another interesting point in this graph, is that the edge connected to FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_READ_HIT exhibits zero latency. The reason this edge does not have an actual latency is because it connects instrumentation points on different machines. Since it is not possible to compare real time between different clocks, all edges

that traverse machine boundaries do not have an actual latency annotation.

Figure 4 shows an even more complex request flow graph, representing an NFS read request that both the NFS server and the storage node were unable to serve from their caches. As a result, the front-end requested the data from the storage node's back-end, as indicated by the instrumentation point BACKEND_BLOCK_OP_START_TYPE_BACKEND_READ.

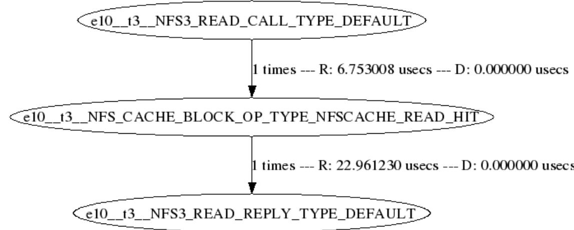

Figure 2: A request flow graph representing an NFS read request that hit in the NFS server cache, and therefore only involved the NFS server (and not other entities in the system).

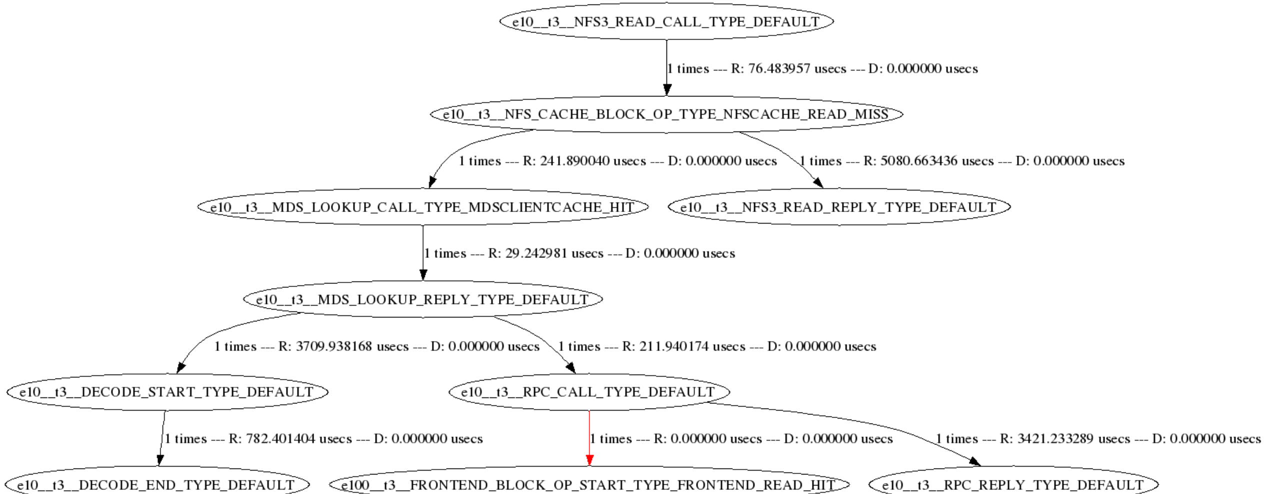

Figure 3: A request flow graph representing an NFS read request that missed in the NFS server cache but hit in the cache of the storage node's front-end, and therefore involved both the NFS server and the front-end.

The existing implementation of Ursa Minor's tracing infrastructure includes an aggregation phase which takes place after the request flow graphs are collected. In this phase, the system aggregates "identical" edges in a request flow graph into one edge. For example, two edges that both connect instrumentation points A and B are aggregated into one edge. As a result, an aggregated graph never has more than one occurrence of the same edge. The latency of an edge

Figure 4: A request flow graph representing an NFS read request that missed in the NFS server's cache as well as the front-end's cache, and therefore had to be serviced by the back-end (ie, from the disk).

in an aggregated request flow graph is the average latency of all the edges in the unaggregated graph that were combined to form that edge.

Figure 5 illustrates the aggregation process. The unaggregated graph is shown on the left, while the aggregated graph is shown on the right. The unaggregated graph includes two edges from node A to node B (remember that each node represents a particular execution of an instrumentation point), whereas the aggregated graph has only one edge from node A to node B. Referring to the aggregated graph, the edge between A and B is annotated with two numbers. The first represents the average latency, which is 3 (ie, the average of 2 and 4), whereas the second represents the number of edges from A to B in the unaggregated graph.

We believe that this aggregation process is both advantageous and disadvantageous. The advantage of the aggregation is that it significantly reduces the sizes of the request flow graphs. Even after aggregation, graphs can include up to 30-40 different instrumentation points (out of a total of about 200 instrumentation points in the system), so the aggregation can be viewed as a pre-processing step that reduces the dimensionality of the data. For example, if a thread spawns 20 threads to write 20 blocks in parallel, it makes sense to avoid a fan-out of 20 and instead, simply record the number as annotation on the corresponding edge. The disadvantage of the aggregation is that it loses information. That is, the resulting graph clearly does not represent all the details of the actual request that are available in an unaggregated graph.

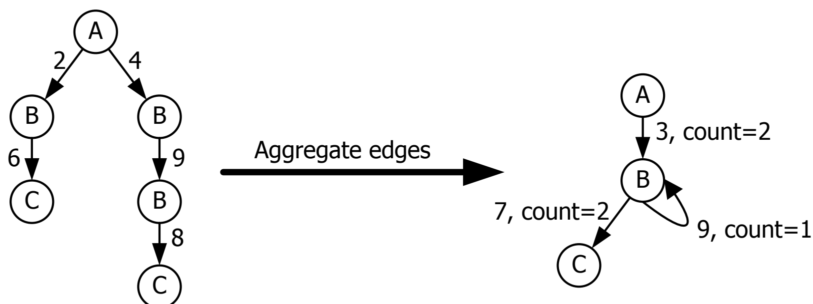

Figure 5: A sample aggregation of a graph that has three instrumentation points (A, B and C).

2 Problem Definition

The major aim of the project is to automatically and effectively detect workload changes and anomalies during the runtime of the system decribed in Section 1. Moreover we want our method

to be efficient as it is developed as a subsytem to run cooperatively along with the core computational loop of the main system. In particular, the auxiliary subsystem should be able to (a) trace the real-time series data (request flow graphs) produced during regular flow of the system (b) detect workload changes (c) alert for anomalous behaviour (d) discover frequent patterns on the fly.

3 What's New in this Report?

In the first half of the semester, we developed a suite of tools for automatically generating classifiers for request flow graphs. With these tools, there is no need for a domain expert to actually label the request flow graphs. This approach has been very effective and provides many advantages over manual labeling (eg, it is less expensive and more resilient to changes in the system).

Our initial goal was to detect workload changes in a sequence of classified request-flow graphs. During the second half of the semester, after analyzing a large amount of actual data, we concluded that we would be able to achieve better results by detecting workload changes without first classifying the request-flow graphs. By working with the raw request-flow graphs (rather than their labels), we can employ more sophisticated pattern recognition techniques. For example, we can utilize the similarity between request-flow graphs when determining when the workload changes. With that said, being able to classify request-flow graphs is still valuable. For example, after detecting different workload intervals, the classifier can be used to produce meaningful descriptions of the intervals that a user can easily understand (eg, "20-30% writes, 40-50% reads that missed in the NFS cache and 30-40% reads that hit in the NFS cache"). This report describes our revised approach as well as our current results.

Tomer will likely continue working in the same area during the summer. This work will include developing an online tracing mechanism for Self-* and improving our current work to produce better results.

4 Survey

Tomer, Charles and Leman each summarized 3 papers:

- Tomer: [20], [2], [5]
- Charles: [9], [4], [8]

- Leman: [7], [18]/[19], [12]

Note that none of these papers were read prior to working on this project. The project does not overlap with prior research of any of us. It is possible that we will continue working on this topic after completing the project, though.

Our work is closely related to Spectroscope [15], which is an unsupervised machine learning tool that uses K-means Clustering [3] on collections of request flow graphs. It has been used to categorize and difference system behavior of Ursa Minor. Raja Sambasivan is currently working on this topic in CMU's Parallel Data Lab (PDL). Raja has provided us helpful guidance throughout our work.

4.1 Tomer's Papers

Yuan et. al.[20] consider the problem of known problem diagnosis. Most existing techniques for diagnosing known problems involve either manually "translating" a textual description of a symptom into a description of the problem (eg, a troubleshooting document), or automatically translating system state information into a description of a known problem. The main disadvantage of the first approach is that its effectiveness is highly variable, and it is often difficult for the user to describe the symptom(s). The main disadvantage of the second approach is that accurately isolating abnormal system states is usually non-trivial due to the large number of possible states even on a standard personal computer (PC). In addition, state-based diagnosis will not work when the root cause is not contained in the collected system states. In this paper, Yuan et. al. propose an approach that translates system behavior information into a description of a known problem. Their results suggest that there does exist at least some correlation between low-level system behavior and high-level problem diagnosis.

The system described by Yuan et. al. includes three main components: tracer, classifier and known problem database. The tracer is a kernel-mode driver that intercepts and records system calls in the Windows operating system. The classifier is responsible for predicting the root cause (class) of a new trace (test data) based on the previous traces with known root causes (labeled training data). An n-gram model[16] is used for feature representation of system call sequences, because n-grams are a simple way of extracting features from sequential data while maintaining its sequential information. Each system call sequence is converted into a bit vector in which each bit corresponds to a single n-gram. The classifier uses Support Vector Machines, or SVMs[3], to classify system call sequences. The decision to use SVMs was due to the high dimensionality of the data, and the authors also propose various noise filtering and canonicalization techniques in

order to reduce the dimensionality prior to classification. For instance, they impose a uniform ordering of events (events in different threads are not interleaved – causality is completely ignored) and discard subsequences that are not frequent in traces of a given problem.

Magpie[2] is a toolchain for automatically extracting a system's workload under realistic operating conditions. Magpie is less intrusive and somewhat less application-specific than other approaches to request tracking, such as Stardust[17] and Pip[14], because it is able to correlate events using an external application-specific schema that specifies the event relationships. The application-specific schema defines the application's requests, but does not affect the collection of event traces. However, it is not clear if most applications actually have sufficient event tracing in place, and Barham et. al. even provide an example in which a database application required additional instrumentation.

Upon detecting a completed request, Magpie converts the request into a canonical form, and a representative set of request types is then identified by clustering the canonical request forms. Magpie considers this representative set, together with the relative frequencies of the requests, as a model of the workload that can then be used for performance analysis purposes. The paper also shows how Magpie can be used to detect anomalies. Any cluster consisting of a significantly small number of requests is considered an anomaly. Barham et. al. provide one example of anomaly detection in which an issue was discovered in a 3Com Ethernet driver. Unfortunately, Magpie does not provide any help in analyzing a request and understanding the semantic difference between different clusters.

Pinpoint[5] introduces a dynamic analysis methodology that automates problem determination in large-scale distributed systems in an application-agnostic way (instrumentation is only required in the middleware or operating system). Client requests are tagged as they enter the system, and the tag travels through the system along with the request. The instrumentation in the middleware creates log records that specify which components of the system were involved in the processing of each request. Pinpoint also records whether the request succeeded. According to the paper, this is the only application-specific aspect in Pinpoint, and in many cases "external failure detection" is possible, in which the result (success or failure) can be determined externally (eg, by examining the error code in an HTTP response). After applying request tracing and failure detection to client requests, Pinpoint transposes the data (for clustering purposes) so that it is indexed by components. It creates a bit vector for each component, indicating which requests involved that component. Another bit vector is created for failures, indicating which requests failed.

Pinpoint's analysis subsystem performs data clustering on the resulting vectors. The interesting result is the set of components clustered with the failure data point (ie, the vector representing the failed requests). These are the components whose occurrences are most correlated with failures, and hence where the root cause is likely to lie. The paper does not describe the actual clustering algorithm in detail. It only mentions that a hierarchical clustering method, unweighted pair-group method using arithmetic averages (UPGMA), was used, and distances between components were calculated based on the Jaccard similarity coefficient (ie, the distance between two points is based on the ratio of the number of requests they appear in together out of all the requests the two points appear in total).

4.2 Charles' Papers

Mannila et. al.[9] present an algorithm for finding patterns within the sequences, which they refer to as episodes. An episode is defined as a collection of events that occurs within a given time span, and which may or may not have restrictions on the order of these events. The approach is to find episodes that occur with some minimum frequency, and the algorithm works by first finding small episodes that occur with the required frequency, and then combining these into larger and larger super-episodes. For example, suppose we have events A and B. We begin by finding the frequency of episodes AA, AB, BA, and BB. If all of these exceed the minimum frequency threshold, we begin combining them together until we achieve the largest episodes possible without falling below the minimum frequency threshold. Association rules can then be formed from these frequent episodes, describing which events occur together, and possibly putting temporal restrictions on their order.

Brin et. al.[4] take a different viewpoint in that they look for implication rules rather than association rules. The distinction is important because co-occurrence does not always mean implication. Using an example from their paper, census data shows a high proportion of people who have 1) past active duty in the military, but 2) no service in Vietnam. From this, we could generate a rule with high confidence saying that past active duty in the military implies no service in Vietnam. This is clearly nonsensical because past active duty in the military would actually increase the chance that a person served in Vietnam. The authors solve this problem by using implication rules that are based on conviction, rather than using association rules based on confidence and co-occurrence. In their experiment on census data, 23712 rules were generated with varying conviction values. Those with conviction ∞ were self-evident and uninteresting, such as "men don't give birth". However rules of lower conviction tend to be more insightful and

informative, such as “African-Americans reside in the same state they were born”, which had conviction 1.28.

Lane et. al.[8] address the problem of anomaly detection using instance-based learning techniques. Their work focuses on identifying users based on the command-line inputs they use, and specifically trying to detect when an imposter (unauthorized user) enters the system. The basic method is to take the sequence of a user’s most recent commands and compare these with sequences from the historical data of that user. A sequence is given a similarity score based on the sequence in the historical data that most closely matches it. If we repeat this at regular time periods then we get a stream of similarity scores and we require these to stay below some threshold. A mean-value filter is used to reduce the noise resulting from the natural variation in a user’s actions. There is a trade-off here because a long window for the filter will result in better noise suppression, but also extends the time during which an imposter can go unnoticed. The authors used a relatively long window because they were primarily concerned with long-term attacks such as industrial data theft, reasoning that short-term attacks can instead be handled by matching to known attack signatures.

4.3 Leman’s Papers

Kuramochi and Karypis [7] propose an algorithm, namely FSG, for finding all connected subgraphs that appear frequently in a large graph database. They use a level-by-level expansion approach, which extends subgraphs by adding one edge at each step in order to generate the candidate subgraphs. FSG initially enumerates all the frequent single and double edge graphs. Then, based on those two sets, it enters the main computational loop. During each iteration, it first generates candidate subgraphs whose size is greater than the previous frequent ones by one. Next, it counts the frequency of each of these candidates, and prunes subgraphs that do not satisfy the support constraint. Support of a subgraph g is defined as the number of graphs in the database in which g is a subgraph. The support constraint is basically the minimum number of graphs a subgraph should exist to be denoted as frequent. They use canonical labeling to sort graphs in a unique and deterministic way and graph isomorphism for determining whether a graph is contained in another. This algorithm essentially has two downsides: (1) candidate generation, that is the generation of size (k+1) subgraph candidates from size k frequent subgraphs, is very complicated and; (2) subgraph isomorphism test is known to be an NP-complete problem, therefore pruning false positives is costly.

Yan and Han [18] propose another method, gSpan, for frequent subgraph mining without using

candidate generation. They build a new lexicographic order among graphs, map each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, the algorithm adopts the depth-first search strategy to mine frequent connected subgraphs. gSpan is reported to run in 10 seconds on a chemical compound dataset with 6.5% minimum support while FSG completes the same task in 10 minutes. A given graph could have multiple DFS trees; by defining a DFS lexicographic order, gSpan reduces the problem of mining frequent connected subgraphs to mining their corresponding minimum DFS codes. The algorithm restricts the way a graph is extended by one edge such that backward edges can only grow from the rightmost vertex while forward edges can grow from vertices on the rightmost path and show that performing rightmost extension guarantees the completeness of the mining result. During the main computational loop, if a DFS code of a subgraph is not a minimum code, it stops searching further as it would mean that graph and all its descendants have been generated and discovered before. Yan and Han [19] later developed an improved version of that algorithm, named CloseGraph, by introducing novel concepts such as early termination and equivalent occurrence which helps pruning the search space substantially.

Nijssen and Kok [12] approach the problem from another perspective, in which for the search for substructures, they first consider the simple structures such as paths, and then transform paths into trees and trees into graphs. This way, they make use of efficient algorithms for simple substructures and use the advanced algorithms when they are really needed. They call this approach a "quickstart principle" which would work in practice as frequent substructures are relatively simple in most "real-world" graphs. Combining frequent path, tree and graph algorithms into one algorithm helps make use of efficient algorithms, such as isomorphism for paths and trees, which do not have a polynomial time when used on generic graphs. Looking at our problem, we observe that the request flow graphs are mostly tree-like structures without many cycles. Therefore, we decided to make use of this algorithm, named Gaston, which could presumably find all the frequent subgraphs – mostly paths and small-depth trees in our case – more efficiently, without delving too much into details of complex graph problems.

5 Proposed Method

Our ultimate goal is to effectively detect workload changes in a real-time series of request flow graphs. We originally planned approach this goal in two steps, both of which we believe are novel:

1. Automatically generate a classifier for request flow graphs
2. Detect workload changes in a sequence of request flow graphs

As described earlier, we realized that it would be better to detect workload changes based on the actual request flow graphs, rather than a sequence of labels. Therefore, given a sequence of request flow graphs, we first divide the sequence into workload intervals by detecting workload changes, and then use the classifier to provide a meaningful description of each interval to the user.

In this section we describe our motivation and method for automatically generating a classifier for request flow graphs. We then describe our method for detecting workload changes. It is worth noting that an automatically-generated classifier for request flow graphs is valuable not only in the this context. In fact, our approach will likely be adopted by the Self-* team for research purposes as well.

5.1 Automatically generating a classifier for request flow graphs

When we began working on this project, we realized that it would be very difficult to manually label or understand large request flow graphs. In fact, feedback from the Self-* development team revealed that it is very difficult even for an expert to understand what a large request flow graph represents. We also realized that even if someone were to manually analyze request flow graphs and label them, these labels would become obsolete as soon as the system is updated, because code paths change and instrumentation points are added and removed. For all these reasons, we decided to investigate an alternative labeling approach in which we actually generate the labeled graphs instead of manually labeling unlabeled graphs.

During the second half of the semester, we discovered that the number of unique structures for request flow graphs is not very large. In our benchmarks we saw only 500 different structures, and our assumption is that Ursa Minor's current instrumentation generates a maximum of several thousand different structures. Since a one-hour benchmark may result in millions of request flow graphs, it is more efficient to run the classifier only when a new graph structure is detected (which can be easily done with simple hashing).

We use the following three-step method to generate a classifier for request flow graphs:

- Define a set of classes based on some understanding of the system's architecture and domain. Since we cannot support a large number of classes in the scope of this project, we selected a collection of classes that seemed interesting and for which we could generate request flow

graphs using a black box approach (ie, without changing any code in Ursa Minor). We currently support the following classes: create, remove, write, read_nfs_hit, read_fe_hit and read_miss.

- For each class, generate requests of that class by running a custom client application.
- Construct a classifier based on the labeled request flow graphs.

In this section we discuss these steps in detail.

5.1.1 Generating requests for a specific class

We developed a client application[1] (label.pl) that interacts with Ursa Minor by performing IO operations within a mounted NFS directory. The standard NFS client translates such operations into various NFS procedures, such as READ, WRITE and COMMIT. With our client application, generating requests for the *read_miss* class, for example, is as simple as running the following command:

```
./label.pl -c read_miss
```

The client application tells Ursa Minor when to enable and disable tracing, thereby enabling the client to perform various "preparation operations" in order to achieve the desired results. The client application implements different strategies for each class of request flow graphs, some of which depend on the sizes of the system's cache layers (the NFS server cache is naturally smaller than the front-end cache):

create The client creates a few thousand files within the mounted directory. The files are given arbitrary names, although these names do not affect the request flow graphs because the NFS protocol uses a separate LOOKUP procedure to resolve paths to handles.

remove The client deletes a few thousand files within the mounted directory. These files are created according to the strategy used to induce NFS create requests, but tracing is enabled only after all the files have been created.

write The client first creates a large file that occupies most of the system's cache, and then enables tracing. The client begins writing to a smaller file using write operations of random

[1]All of the applications discussed in this section, as well as several other axillary applications, were developed by us solely for the purpose of this project.

sizes at random offsets. In some cases, the large file is evicted from cache in order to make space for the smaller file.

read_nfs_hit The client first creates a file that fits entirely in the NFS server cache. The state of the system's cache layers at this stage is shown in Figure 6. The client also flushes the client-side NFS cache at this stage. It then enables tracing and begins reading from that file using sequential read operations of random sizes. These read operations are guaranteed to hit in the NFS server cache, because the file is smaller than the NFS server cache, and the NFS server does not evict blocks from its cache unless the cache overflows.

read_fe_hit The client first creates a file that fits entirely in the NFS server cache. The client then writes a relatively large file, which is larger than the NFS server cache but smaller than the front-end cache. As a result, the original file is entirely evicted from the NFS server cache, but still fits in the front-end cache. The state of the system's cache layers at this stage is shown in Figure 7. The client also flushes the client-side NFS cache at this stage. It then enables tracing and begins reading the original file using sequential read operations of random sizes. Since the original file is in the front-end cache but not in the NFS server cache, these read operations miss in the NFS server cache but hit in the front-end cache (note that we keep a minimal distance between read operations, in order to avoid hitting the NFS server cache due to prefetching triggered by previous reads).

read_miss The client first creates a file that fits entirely in the NFS server cache. It then writes a large file, which is larger than both the NFS server cache and the front-end cache. As a result, the original file is entirely evicted from both cache layers and moved to the storage node's hard disk. The state of the system's cache layers at this stage is shown in Figure 8. The client also flushes the client-side NFS cache at this stage. It then enables tracing and begins reading the original file using sequential read operations of random sizes. Since the original file is not cached at any layer, these read operations miss in both the NFS server cache and the front-end cache (again, we keep a minimal distance between read operations).

After collecting the trace file of a given class of request flow graphs, we run a set of custom applications in order to "purify" the data and convert it into a format that is suitable for machine learning and graph mining algorithms.

The filter.pl application filters request flow graphs based on regular expression matching. This post-processing step is important, because in some cases we cannot induce request flow graphs of a given class without inducing other request flow graphs as well. For example, consider the

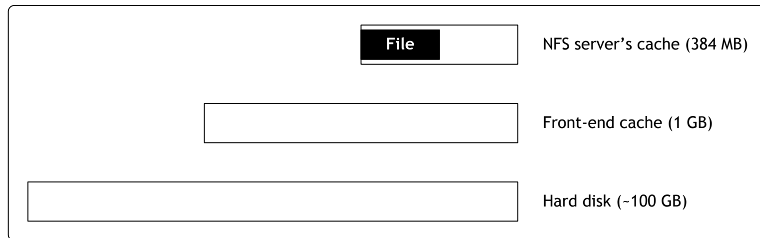

Figure 6: The state of the system's storage layers just before the client begins issuing NFS read requests that will hit in the NFS server cache.

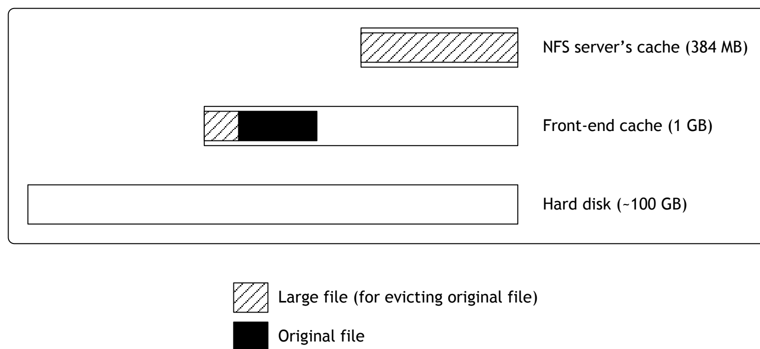

Figure 7: The state of the system's storage layers just before the client begins issuing NFS read requests that will miss in the NFS server cache but hit in the front-end cache.

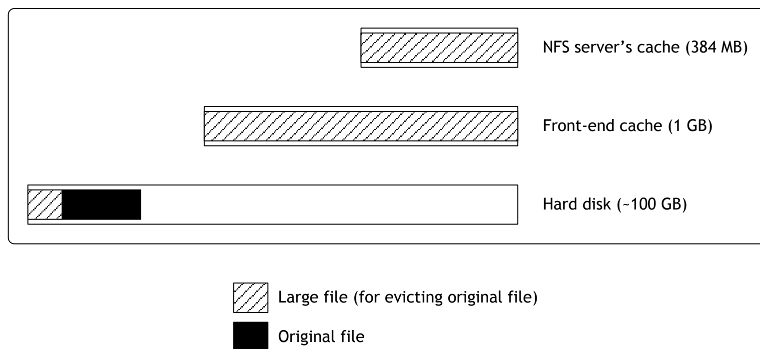

Figure 8: The state of the system's storage layers just before the client begins issuing NFS read requests that will miss in both the NFS server cache and the front-end cache (and thus require disk accesses).

create class. In order to induce such graphs, we must create a large number of files. In order to create a file, we call the client operating system's `open` system call, which accepts a string representing the file's path (we do not want to issue direct remote procedure calls, or RPCs, to the NFS server). In NFS, the `open` system call triggers not only an NFS create request, but also an NFS lookup request in order to resolve the path to an NFS handle. From the point of view of the client application, these requests cannot be distinguished, so we cannot enable tracing only for the NFS create request. However, from the point of view of Ursa Minor (specifically, the NFS server), these are two separate requests. Therefore, we use the filter.pl script to remove all the request flow graphs that represent NFS lookup requests.

The convert.pl application processes a single trace file (typically containing thousands of request flow graphs) and generates three files:

trace.**nodes.txt** : This file contains an index of all the nodes (ie, instrumentation points) that exist in the given trace file. Each line represents a node. For example, the content of this file for the $read_nfs_hit$ class is:

```
0 e10__t3__NFS3_READ_CALL_TYPE_DEFAULT
1 e10__t3__NFS3_READ_REPLY_TYPE_DEFAULT
2 e10__t3__NFS_CACHE_BLOCK_OP_TYPE_NFSCACHE_READ_HIT
```

trace.**edges.txt** : This file contains an index of all the edges that exist in the given trace file. Each line represents an edge, specifying the numeric indices of the two nodes that make up that edge. For example, the content of this file for the $read_nfs_hit$ class is:

```
0 0 2
1 2 1
2 0 1
```

The first line, for example, represents an edge between NFS3_READ_CALL_TYPE_DEFAULT and NFS_CACHE_BLOCK_OP_TYPE_NFSCACHE_READ_HIT.

trace.**graphs.txt** : This file contains the actual request flow graphs that exist in the given trace file. Each line represents a graph, and each column represents a specific edge. One additional column is used for the total latency (ie, response time) of the request. The content of this file for the $read_nfs_hit$ class is:

```
5.735294 15.255682 0 20.990976
12.447861 0 26.239973 26.239973
4.933155 65.228944 0 70.162099
11.844920 0 22.991310 22.991310
...
```

This example shows four request flow graphs, each consisting of only two edges (this example shows the simplest graph structure that we encountered – many of the graphs included a much larger number of nodes and edges). The numbers in each column (other than the last one) represent the average latency associated with the corresponding edge (zero indicates that the edge does not exist in the graph).

The combine.pl application is also used for post-processing, but it serves a completely different role and is far more complicated. Since not all trace files contain every possible node and edge, we must create global node and edge indices so that graphs from different trace files can be compared. The combine.pl application uses several hash tables to create these global indices based on all the "local," class-specific *trace*.node.txt and *trace*.edge.txt indices, and then translates all *trace*.graphs.txt files so that they refer to the global indices rather than the class-specific ones. The combine.pl application was carefully implemented so that adding new classes does not require existing *trace*.graphs.txt files to be re-translated.

Although we discuss classification of unlabeled graphs later in this report, it is worth mentioning at this point that the combine.pl application is executed on all unlabeled request flow graphs before they can be classified. It not only converts the representation of the graphs so that they refer to the global node and edge indices, but also adds any nodes and edges that do not already exist in the global indices. Since the ultimate goal of our research requires real-time classification of request flow graphs, we designed the combine.pl application so that it can efficiently process request flow graphs on the fly (ie, in real-time).

5.1.2 Constructing a classifier based on the labeled request-flow graphs

Constructing a frequent subgraph classifier

One common approach to classification is using frequent itemset analysis on the given dataset of classes and classifying new examples based on the number of types of those frequent items they contain. In our case, we are trying to classify graphs, for which we currently have 6 well-defined classes. For each class, we have a collection of graphs which were carefully induced so that we

know their true labels (ie, classes). These graphs form our training set for frequent subgraph analysis.

In order to extract the frequent subgraphs of a particular class, we make use of an existing software package called Gaston [11]. Gaston is described as a unified GrAph, Sequences and Tree extractiON algorithm. The reason we chose the Gaston algorithm over other algorithms in the literature, such as gSpan[18] or closeGraph[19], is because of the relatively simple structure of the request flow graphs. In most of our training sets, these graphs did not exhibit very complex structures; that is, they were mostly acyclical, with very few back edges forming cycles. Therefore, Gaston is well suited for our needs.

Realizing that in practice most of the frequent graphs are basic structures, Gaston uses a "quickstart" approach to organize the search space efficiently. Gaston then finds all frequent subgraphs by using a level-wise approach in which it first considers simple paths, then more complex trees and, finally, complex cyclic graphs.

The only parameter the algorithm requires is a threshold percentage value. A subgraph is considered to be "frequent" if it occurs in at least that many graphs of a given class. What it returns is all the frequent subgraphs with one edge, two edges and so forth. For example, if a frequent subgraph with two edges occurs in the training data, its own two subgraphs (with one edge in each) are also returned in the frequent subgraph set.

Dividing the labeled data into train and test sets, we first run the algorithm on the training set in order to extract the frequent subgraphs. Next, for each test (ie, unlabeled) graph and for all frequent subgraphs of each class, we perform a containment test. If the test graph contains a given frequent subgraph, we increment the test graph's hit-count for the class that the frequent subgraph belongs to (we maintain c hit-counts for the test graph during this process, where c is the number of classes). In the end, for each class we divide the test graph's hit-count for that class by the total number of frequent subgraphs in the class, and refer to the result as the "hit-ratio." We assign the test graph to the class corresponding to the test graph's highest hit-ratio. The experiment results are reported in Section 6.

Constructing "classic" machine learning classifiers

We have two classic machine learning techniques that we use: the first is naive Bayes, and the second is nearest neighbor. Our feature set for both of these methods is based solely on the edges present in a request graph. Specifically, for any two nodes, we denote a 1 if there is an edge present between them (regardless of how many edges and regardless of the latency), and a 0 if

no edge is present. Thus our set of features is a binary vector with an entry of 0 or 1 denoting the occurrence of each possible edge. Although the number of possible edges is the square of the number of nodes, in practice only a small subset of these edges actually occur, and our feature vector only needs to consider this subset.

In the naive Bayes classifier, we assume that all features are conditionally independent given the class. Although we cannot be sure that this assumption holds, naive Bayes has been successful in many different applications, including applications where the conditional independence assumption clearly does not hold. During the training phase of this classifier, we compute the probability that any edge will be present for a given class. This is done by taking the training examples for that class and simply finding the proportion of instances in which that edge was present. The output of our training phase is a matrix of the classes and the edges (features), with the entries of the matrix being the probability with which an edge is present for any class.

In the testing phase of the classifier, we take the observed edges, and compute the likelihood of these edges being present under each of the classes. We then choose the class that gives the maximum likelihood. For a true Bayesian classifier, we should also consider the prior distribution of the classes. However, in the absence of information about this prior, we assume that all classes are equally likely.

Our second classifier is a nearest neighbor classifier. Here, given a request graph, we look for the closest request graph in our training data. We then assume that these two request graphs are of the same class, and thus output the label from the training data. To calculate the distance between two request graphs, we take their feature vectors and simply count the number of indices at which their entries differ.

For both the naive Bayes and the nearest neighbor classifier, we could make our feature set more complex by considering the latencies of the edges, or by considering dependencies between them. However, our preliminary results show that the stated methods are sufficient to distinguish between the classes that we have acquired so far. Nevertheless, as we increase the number of classes in our dataset, we may consider these extensions.

5.2 Detecting workload changes in a sequence of request flow graphs

Given our sequence request flow graphs, we now wish to identify the points where a change in the workload or performance occurs. This will give us a better understanding of system behavior, and could be used to modify the system in real-time, resulting in higher efficiency and better resource allocation. It also facilitates the recognition of anomalies – sequences of graphs that

fall outside of the regularly observed behavior – which may indicate that a problem is occurring within the system.

In our initial proposal we had approached this from a pattern recognition and prediction angle. The idea was to generate either association rules as done in [9], or implication rules as done in [4]. These would be used to predict upcoming request flow graphs, and if a sequence of graphs consistently contradicted our predictions then we would flag an anomaly.

Our revised proposal generalizes this approach. Rather than specifically generate implication rules to predict a request flow graph, we propose to observe a set of statistical features over time. At regular intervals we will compute a consistency score that reflects how similar the current features are to their historical values. If some or all of the features are well outside of their usual range then the consistency score will be low, and thus we will flag a change in the workload or performance of the system. As done in [8], we make this method more robust to noise by maintaining a moving average of the consistency score, only flagging a change in workload/performance when this moving average falls below some threshold.

The main task here is finding features that will reflect a change in workload or performance. A natural choice is to maintain a count of the occurrences of each request type over a given window (or even over several windows of different length). It is reasonable to assume that a change in workload will change the types of request flow graphs that we observe. Although a request flow graph provides timing information (eg, overall latency, average latency per edge), we decided to focus only on the structure of the graphs. There are typically several million request flow graphs with several hundred unique structures in an hour-long workload. Ursa Minor is currently not capable of providing real-time request traces, so our analysis is entirely offline. However, since future work includes detection of workload changes in real-time (ie, online detection), we focused on designing methods that are suitable for such scenarios.

When given a sequence of request flow graphs, we first convert it to a sequence of numbers. Each graph is converted to a number by hashing its structure (the structure is essentially a bit vector that indicates which edges exist in the request flow graph). Figure 9 is an illustrative plot for an artificial (and unrealistic) workload. In this example, the workload changes twice, resulting in three intervals (each of which is unique). In this example, the workload changes are easy to identify. In reality, such plots are obviously much more difficult to understand, as they may consist of hundreds of different graph structures, millions of request flow graphs and a great deal of "randomness".

Having the hash values for the request flow graphs over time, we want to detect intervals

which correspond to sequences of different types of graphs, which in turn means different system behaviour. To do that, we scan the traces keeping track of histograms over a specific length time window. A histogram for an interval holds the number of occurences of each of the possible request flow graphs during that particular time interval. Figure 10 shows an example for the structure of the histograms we use.

Next, we slide the window over time computing new histograms for each time window. In order to detect a workload change, we compute the distance between contiguous histograms and sign the time step in which the distance gets above a pre-specified threshold value. We use the following formula *****please cite***** to measure the distance between two histograms.

$$d^2_{hist}(\vec{x},\vec{y}) = (\vec{x}-\vec{y})^T \times A \times (\vec{x}-\vec{y}) = \Sigma_i^h \Sigma_j^h a_{ij}(x_i - y_i)(x_j - y_j)$$

where $\vec{x}$ and $\vec{y}$ represent the $n \times 1$ histogram vectors for which the distance is measured, n is the number of different request flow graphs and matrix $\mathbf{A}$ is the similarity matrix with entries a_{ij} indicating the similarity between request flow graphs i and j.

Next, we discuss how the similarity matrix of request flow graphs is computed.

5.3 Similarity of Request Flow Graphs

In order to find the similarity between two given graphs, we started first defining a distance between them. The distance between two graphs can be the number of nodes/edges/labels that need to be added/deleted/changed to convert one graph to the other. This is similar in analogy to the number of characters that need to be shifted, inserted or deleted to match one string to another, that is known as the *string editing distance*.

One way to represent graphs with edge and node labels as a string is to construct its Depth First Search (DFS) tree and concatinate the edge/node labels in the order in which they are visited. We call such string representation of a graph as its *DFS-code*. In our experiments, we decided to use only the node labels to construct the DFS-code of the graphs.

There exists several DFS trees for a given graph depending on the starting node and the order in which the adjacent nodes are visited during the recursive calls of the DFS procedure. Our convention is to enumarate the nodes of the graph and visit the adjacent nodes of a particular node in increasing node number order. That is if a node has k neighbors with unique numbers $v_1 < v_2 < \ldots < v_k$, we recurse on the neighbor with the smallest number, that is v_1 (if it is already visited, we go on with the neighbor with the next largest number, and so on). And

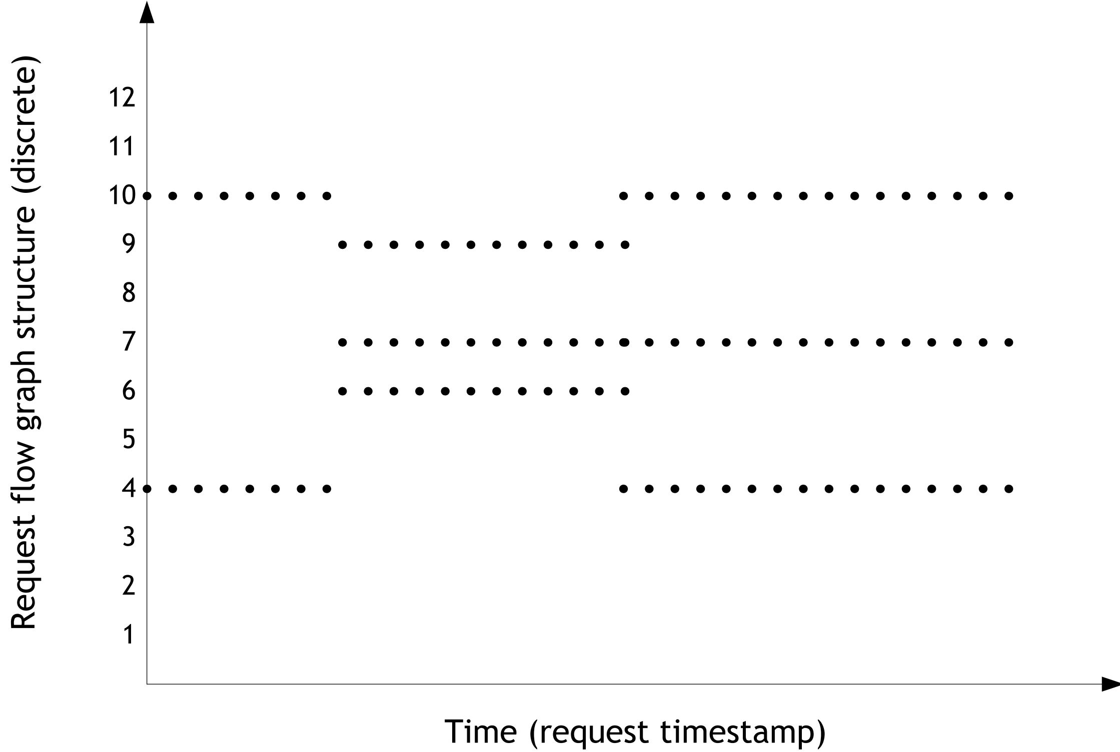

Figure 9: Each request flow graph is hashed according to its structure. The Y axis represents the hash value of a request flow graph, and the actual value for each structure is insignificant (as long as graphs with the same structure are hashed to the same value). The X axis represents the time at which a request first reached the NFS server.

thanks to the structure of the request flow graphs, all the graphs has a single root node to which none of the nodes point and so we always have a certain point to start the DFS.

After obtaining the DFS-codes of all the different flow graphs, we find the string edit distance between those strings for each pair of graphs. All in all, our computations take linear time with respect to the number of edges for DFS, and $\bigodot(n^2)$ time for distance measurement where n is the number of different flow graphs in a specific benchmark. Since the number of edges for a flow graph is in order of tens and number of different flow graphs are in the order hundreds, the running time is reasonable.

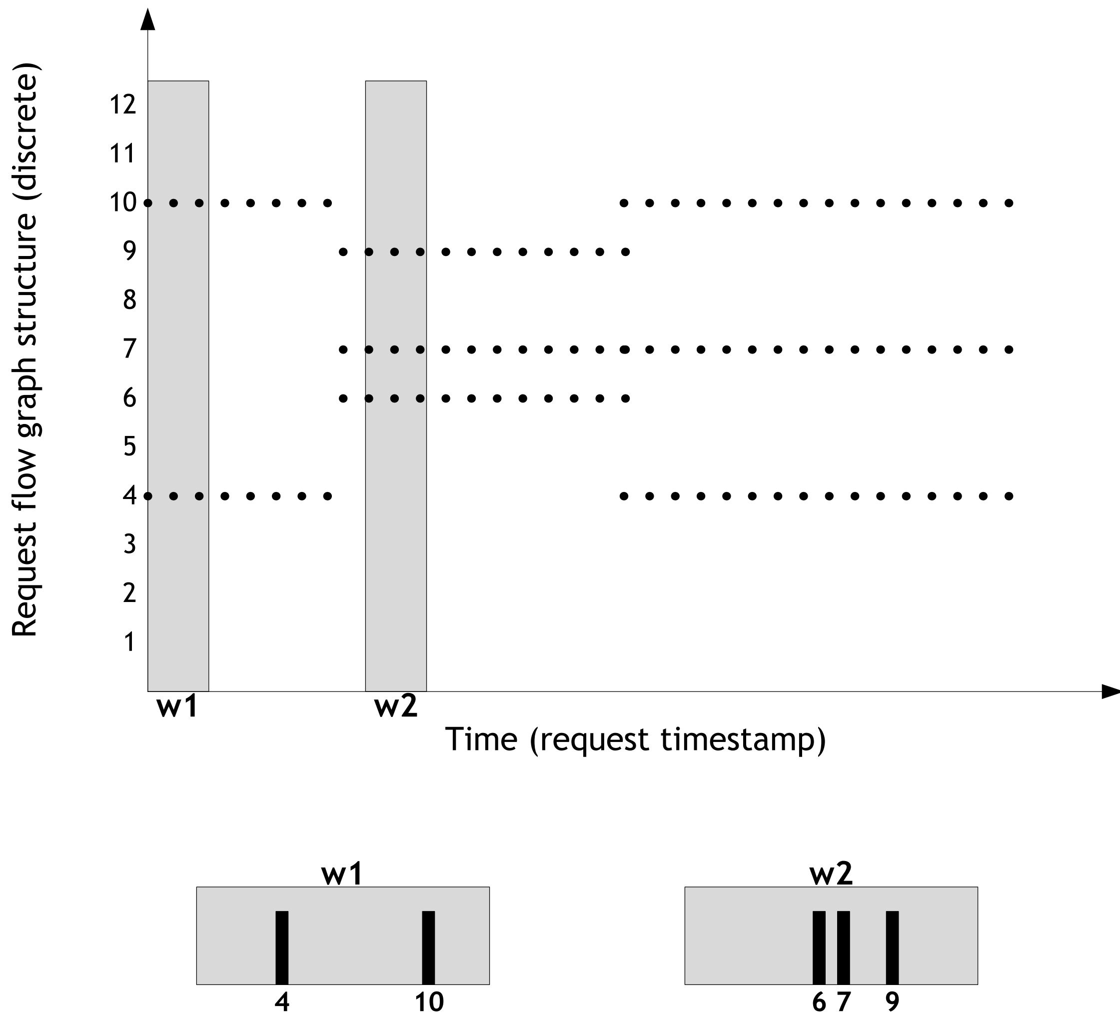

Figure 10: $w1$ and $w2$ are two different positions of the scanning window. The corresponding histograms are shown at the bottom.

6 Experiments and Results

6.1 Automatically generating a classifier for request flow graphs

6.1.1 Frequent subgraph classification

The software package Gaston expects its input in the following format:

```
t #1
v 0 3
v 1 1
v 2 3
v 3 2
v 4 1
e 0 1 31
e 1 2 13
e 0 3 32
e 3 4 21
```

t denotes the beginning of a new graph, lines starting with a `v` represent vertices (`v vertex_id vertex_label`) and lines starting with a `e` represent edges (`e source_vertex_id dest_vertex_id edge_label`). In our case, due to the aggregation of graphs which was discussed in the Section 1, we do not have multiple nodes with the same label.

Given a collection of graphs, and a threshold value above which a subgraph is reported as "frequent," the output of the Gaston program looks like the following:

```
# 9119
t 1
v 0 0
v 1 14
e 0 1 1
# 9119
t 2
v 0 0
v 1 14
v 2 15
```

```
e 0 1 1
e 1 2 7
```

Here, the value after the # is the actual number of graphs in the dataset in which the given subgraph occurs. Notice that in this example, the first graph is a proper subgraph of the second one. The algorithm returns all subgraphs of a frequent subgraph, as apparent in the algorithm: if a graph is frequent, there is chance that its supergraphs are frequent; otherwise (ie, the graph is not frequent), there is no possibility that its supergraphs are frequent.

After obtaining the frequent subgraphs of every class, we need to determine how many of them a given test graph contains. For the containment test, we use the same frequent subgraph extraction program. That is, given the test graph and a particular frequent subgraph for which we want to see if it occurs in the test example, we put these two graphs in a file and run the algorithm with an absolute threshold value of 2, meaning that we are interested in subgraphs that occur in both of these graphs. We conclude that if the algorithm returns a frequent graph with the same number of edges as the frequent graph we started with, then these two graphs are the same, so the test graph contains that frequent subgraph. We must explicitly count the edges of the frequent subgraphs that are returned, because as stated earlier, all the subgraphs of the frequent subgraph are returned and we want to find the maximal one.

With our current 6 classes, for a small test set of 12 test graphs (two graphs per class), we obtained the following results (*create* is 1, *remove* is 2, *read_fe_hit* is 3, *read_nfs_hit* is 4, *read_miss* is 5, and *write_no_evict* is 6):

```
1 100 - 0 - 0 - 0 - 0 - 0
2 100 - 0 - 0 - 0 - 0 - 0
3 0 - 0 - 100 - 100 - 26 - 0
4 0 - 0 - 100 - 100 - 26 - 0
5 0 - 0 - 60 - 0 - 100 - 0
6 0 - 0 - 60 - 33 - 100 - 0
7 0 - 0 - 0 - 33 - 0 - 0
8 0 - 0 - 0 - 100 - 0 - 0
9 0 - 100 - 0 - 0 - 0 - 0
10 0 - 100 - 0 - 0 - 0 - 0
11 0 - 0 - 0 - 0 - 0 - 100
12 0 - 0 - 0 - 0 - 0  100
```

The values in each row are the hit-ratios of a specific test graph for each of the six classes. For example, the first graph contains all the frequent subgraphs of class 1 and none of those of class 2, 3, 4, 5 and 6. As a result, we conclude that it belongs to class 1. The third graph is problematic because it contains all the frequent subgraphs of class 3 and 4 (this is because the frequent subgraphs of class 4 are proper subgraphs of the frequent subgraphs of class 3). In other words, the frequent subgraphs of class 3 are larger and they contain those of class 4. Therefore, a test graph of class 3 is guaranteed to contain the frequent subgraphs of class 4 and we end up with equal hit-ratios.

To solve this problem, we give higher priority to larger subgraphs. In addition to the hit-ratio of a graph for a given class, we calculate the average size of the frequent subgraphs of that class that it contains. In the case of equal hit-ratios for different classes, we determine that the test graph belongs to the class with the larger subgraphs.

After assigning these average values to the subgraphs, the first four lines of the preceding result become:

```
1 100 8.45 - 0 0 - 0 0 - 0 0 - 0 0 - 0 0
2 100 8.45 - 0 0 - 0 0 - 0 0 - 0 0 - 0 0
3 0 0 - 0 0 - 100 3.86 - 100 1.33 - 26 0.88 - 0 0
4 0 0 - 0 0 - 100 3.86 - 100 1.33 - 26 0.88 - 0 0
```

The values after the hit-ratio values are the average sizes of the subgraphs. Here, we correctly classify test graphs 3 and 4 as class 3, since their average sizes for class 3 subgraphs are greater than their average sizes for class 4 subgraphs. After applying this technique, we were able to correctly classify all of the test graphs.

6.1.2 "Classic" machine learning classification

The input format for this classifier has assigned each observed edge with an ID. Each request graph is then represented as a vector of the form `[0 1 1 0 0 0 1 ...]`, where a `1` at position i indicates that edge i is present in this graph. Our naive Bayes approach measures the probability of any given edge appearing in the request flow graph of a given class, and then classifies according to the maximum likelihood. Our training set consisted of approximately 8000 graphs for each labeled class, and thus we were able to characterize these probabilities with a high degree of accuracy. Although many edges appeared in the graphs of several classes, there were some edges that were almost never observed for a particular class, and these edges provide the most

discriminatory information. As a result, we were able to classify our entire testing set correctly using this method. It remains to be seen whether the method will continue to work as the number of classes increases and we encounter new graph structures when running more realistic benchmarks.

The nearest neighbor method was also able to classify all of the classes correctly provided that we had a sufficient number of training examples. As described earlier, our distance metric simply counted the number of positions at which the request graphs differed. Although there were many different structures for each class, they tended to be similar to each other (according to our distance metric) and thus this method worked effectively. Once again, it remains to be seen whether it will continue to work as the number of classes increase and become more similar.

6.2 Detecting workload changes in a sequence of request flow graphs

It is hard to evaluate methods for detecting workload changes and anomalies in a real-world (ie, production) environment, because in most cases there is no other way to determine whether the reported changes actually occurred, and whether there were changes that the system did not report. However, we can evaluate our method using various benchmarks for which we can determine (or "intelligently guess") when workload changes occur. As an example, consider a benchmark that has four phases: creating a large number of files, writing to those files, reading from those files and deleting those files. Our pattern recognition method should be able to detect three points in which the workload changes:

- From creating files to writing files
- From writing files to reading files
- From reading files to deleting files

Although this is a very simple example, detecting the workload changes is not trivial. While writing files, the NFS client issues short sequences of WRITE procedures, where each sequence is followed by a COMMIT procedure that instructs the NFS server to commit the data to stable (persistent) storage. Our pattern recognition method should be able to detect this pattern and label the entire write-phase as one type (that contains repetitive patterns of writes/commits). In addition, each of these phases will include LOOKUP and GETATTR procedures.

We will start by evaluating our pattern recognition method on our own custom "benchmarks" in order to establish that the method indeed behave according to our expectations. For example,

we will develop a simple benchmark with "obvious" patterns and make sure that we are able to recognize those patterns.

We then plan to evaluate our method on a collection of popular benchmarks, some of which more closely resemble real-world workloads. For example, we will use a benchmark that builds the Linux kernel within an NFS directory. Although it is not as obvious as the preceding benchmark, we would like to be able to detect the different phases of a build, and possibly other interesting points in time (eg, when the NFS server cache becomes full). The following list consists of benchmarks that we plan to use, although it is likely to change as we learn more about these and other benchmarks in the next few weeks:

IOzone workload[6] IOzone is a general file system benchmark that can be used to measure streaming data access (eg, for data mining). For our experiments, IOzone measures the performance of 64 KB sequential writes and reads to a single 2 GB file. IOzones performance is reported in megabytes per second read.

"Linux build" development workload The "Linux build" workload measures the amount of time to clean and build the source tree of Linux kernel 2.6. The benchmark copies the source tree onto a target system, then cleans and builds it. The results provide an indication of storage system performance for a programming and development workload. The source tree consumes approximately 200 MB.

Postmark workload[13] : Postmark is a file system benchmark designed to emulate small file workloads such as e-mail and netnews. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append. The configuration parameters used were 100000 files, 50000 transactions, and 224 subdirectories. All other parameters were left as default. Postmark's performance is reported in transactions per second (TPS).

If time permits, we will try to evaluate additional scenarios, such as:

- Adding/removing clients to/from the system, where each client performs a completely different workload.
- Adding a misconfigured client, such as one that is issuing synchronous write requests (instead of asynchronous writes with periodic commits). This was an actual misconfiguration that we encountered while working on the project, and the cause for the performance hit

was identified only after several hours of investigation (it turned out that the client had mounted Ursa Minor's NFS server with the `-o sync` option).

- Issuing a single pattern of requests, periodically slowing down one period's worth of requests. Our method should be able to detect the slower period.

Note that the last two scenarios require the pattern recognition method to consider the overall latency (ie, response time) of each request in addition to its class (unless we change our classifier to differentiate between identical requests with different latencies). Also note that these are just examples of possible scenarios, and are likely to change as we receive additional feedback from the Self-* team.

As we work on our pattern recognition methods, we will need to successfully classify many graph structures that we did not previously encounter. The benchmarks that we plan to run will likely hit a much larger number of instrumentation points than the simple benchmarks that we have been using so far. Therefore, we will continue to evolve our classification methods by adding more classes and refining the algorithms.

7 Conclusions

8 Responsibilities

The responsibilities listed below have naturally throughout the semester. During the second half of the semester, we discovered a severe bug in the tracing mechanism of Ursa Minor. Specifically, the traces were not actually ordered by time. In fact, they were ordered first by the top-level request type, and only then by time (within the group of request flow graphs having the same top-level request type). Since our planned work was meaningless in this state, we had no choice but to correct the relevant Ursa Minor code. As a result, we had to change our responsibilities and cut back in some of the other work that we had planned.

February 15-29

- **TS:** Develop tools for generating labeled data for simple request types.
- **LA:** Implement classifier using frequent subgraphs and test on synthetic data.
- **CJ:** Implement alternative classifier using machine learning techniques.

March 1-20

- **TS:** Develop tools for generating labeled data for more advanced request types.

• **TS:** Develop tools for cleaning data, constructing global index files and converting formats.

• **LA:** Test frequent-subgraph classifier on simple labeled data, and select appropriate classifier thresholds.

• **CJ:** Test classifiers with different machine learning approaches on simple labeled data, and select appropriate classifier parameters.

March 20-April 10

• **TS:** Prepare unlabeled data (by developing and/or running various benchmarks) to be used for simple pattern recognition experiments.

• **TS:** Develop tools for generating labeled data for any major request types that show up in unlabeled data.

• **LA:** Refine frequent-subgraph-classifier and test on more advanced labeled data.

• **LA:** Classify sequence of unlabeled benchmark data using frequent subgraph classifier.

• **CJ:** Test on more advanced labeled data and decide on the best machine learning techniques.

• **CJ:** Classify sequence of unlabeled benchmark data using selected machine learning approaches.

April 10-25

• **CJ:** Implement and test algorithm to detect workload/performance changes.

• **TS:** Prepare unlabeled data (by running various benchmarks) to be used for more advanced pattern recognition experiments.

• **TS:** Add timestamps to Ursa Minor's tracing output, and fix the ordering bug.

• **TS:** Evaluate results of pattern recognition experiments.

• **LA:** Implement an algorithm to determine distance/similarity of request flow graphs.

April 25-...

• **All:** Tidy code, prepare final report and poster.

References

[1] Michael Abd-El-Malek, II William V. Courtright, 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. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 255–264, 1997.

[5] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[6] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[7] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[8] T. Lane and C.E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[9] H. Mannila, H. Toivonen, and A.I. Verkamo. Efficient algorithms for discovering association rules. *KDD-94: AAAI Workshop on Knowledge Discovery in Databases, July*, 1994.

[10] NFS Version 3 Protocol Specification. `http://www.faqs.org/rfcs/rfc1813.html`.

[11] Siegfried Nijssen and Joost N. Kok. The gaston tool for frequent subgraph mining. In *Proceedings of the International Workshop on Graph-Based Tools, Grabats 2004, Rome, Italy, October 2, 2004*. Elsevier, 2004.

[12] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, New York, NY, USA, 2004. ACM.

[13] Postmark. `http://packages.debian.org/stable/utils/postmark`.

[14] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[15] Raja R. Sambasivan, Alice X. Zheng, Eno Thereska, and Gregory R. Ganger. Categorizing and differencing system behaviours. In *HotAC II: Hot Topics in Autonomic Computing on Hot Topics in Autonomic Computing*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[16] C. Y. Suen. n-Gram Statistics for Natural Language Understanding and Text Processing. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-1(2):164–172, April 1979.

[17] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 3–14, New York, NY, USA, 2006. ACM.

[18] J. Han X. Yan. gspan: Graph-based substructure pattern mining. In *In Proc. IEEE ICDM02*, page 721724, 2002.

[19] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, New York, NY, USA, 2003. ACM.

[20] 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. *SIGOPS Oper. Syst. Rev.*, 40(4):375–388, 2006.

Contents

1 Introduction 2

2 Problem Definition 6

3 What's New in this Report? 7

4 Survey 7
- 4.1 Tomer's Papers . . . 8
- 4.2 Charles' Papers . . . 10
- 4.3 Leman's Papers . . . 11

5 Proposed Method 12
- 5.1 Automatically generating a classifier for request flow graphs . . . 13
  - 5.1.1 Generating requests for a specific class . . . 14
  - 5.1.2 Constructing a classifier based on the labeled request-flow graphs . . . 18
- 5.2 Detecting workload changes in a sequence of request flow graphs . . . 20
- 5.3 Similarity of Request Flow Graphs . . . 22

6 Experiments and Results 25
- 6.1 Automatically generating a classifier for request flow graphs . . . 25
  - 6.1.1 Frequent subgraph classification . . . 25
  - 6.1.2 "Classic" machine learning classification . . . 27
- 6.2 Detecting workload changes in a sequence of request flow graphs . . . 28

7 Conclusions 30

8 Responsibilities 30