Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues

Mark D. Syer[1], Zhen Ming Jiang[2], Meiyappan Nagappan[1], Ahmed E. Hassan[1], Mohamed Nasser[3] and Parminder Flora[3]
Software Analysis and Intelligence Lab[1], Department of Electrical Engineering & Computer Science[2], Performance Engineering[3]
School of Computing, Queen's University, Canada[1], York University, Canada[2], BlackBerry, Canada[3]
mdsyer@cs.queensu.ca, zmjiang@cse.yorku.ca, {mei, ahmed}@cs.queensu.ca

***Abstract*—Load tests ensure that software systems are able to perform under the expected workloads. The current state of load test analysis requires significant manual review of performance counters and execution logs, and a high degree of system-specific expertise. In particular, memory-related issues (e.g., memory leaks or spikes), which may degrade performance and cause crashes, are difficult to diagnose. Performance analysts must correlate hundreds of megabytes or gigabytes of performance counters (to understand resource usage) with execution logs (to understand system behaviour). However, little work has been done to combine these two types of information to assist performance analysts in their diagnosis. We propose an automated approach that combines performance counters and execution logs to diagnose memory-related issues in load tests. We perform three case studies on two systems: one open-source system and one large-scale enterprise system. Our approach flags $\leq$0.1% of the execution logs with a precision $\geq$80%.**

***Keywords*-Performance Engineering; Load Testing; Performance Counters; Execution Logs**

I. Introduction

The rise of ultra-large-scale (ULS) software systems (e.g., Amazon.com, Google's GMail and AT&T's infrastructure), poses new challenges for the software maintenance field [1]. ULS systems require near-perfect up-time and potentially support thousands of concurrent connections and operations. Failures in such systems are more likely to be associated with an inability to scale, than with feature bugs [2], [3]. This inability to meet performance demands has led to several high-profile failures, including the launch of Apple's MobileMe [4] and the release of Firefox 3.0 [5], with significant financial and reputational repercussions [6], [7].

Load testing has become a critical component in the prevention of these failures. Performance analysts are responsible for performing load tests that monitor how the system behaves under realistic workloads to ensure that ULS systems are able to perform under the expected workloads. Such load tests allow analysts to determine the maximum operating capacity of a system, validate non-functional performance requirements and uncover bottlenecks. Despite the important of load testing, current load test analysis techniques require considerable manual effort and a high degree of system-specific expertise to review hundreds of megabytes or gigabytes of performance counters (to understand resource usage) and execution logs (to understand system behaviour) [2], [8], [9].

Performance analysts must also be aware of a wide variety of performance issues. In particular, memory-related issues, which may degrade performance (by increasing memory management overhead and depleting the available memory) and cause crashes (by completely exhausting the available memory), are difficult to diagnose. Memory-related issues can be broadly classified as transient or persistent. *Transient memory issues* (memory spikes) are large increases in memory usage over a relatively short period of time. *Persistent memory issues* are steady increases in memory usage over time. Persistent memory issues can be further divided into memory bloat (caused by inefficient implementations) and memory leaks (caused by a failure to release unneeded memory). Such issues have led to high profile failures, including the October 22, 2012 failure of Amazon Web Services (caused by a memory leak) that affected thousands of customers [10].

We present a novel approach to support performance analysts in diagnosing memory-related issues in load tests by combining performance counters and execution logs. First, we abstract the execution logs into execution events. We then combine the performance counters and execution events by discretizing them into time-slices. Finally, we use statistical techniques to identify a set of execution events corresponding to a memory-related issue.

Our approach focuses on the *diagnosis*, as opposed to the *detection*, of memory-related issues. Performance analysts could use a variety of existing techniques to detect issues prior to using our approach for diagnosis. For example, performance analysts may plot memory usage over time to determine whether there are any persistent memory issues (where the memory usage continually increases) or compare the minimum, mean and maximum memory usage to determine whether there are any transient memory issues.

This paper makes two main contributions:

1) Existing load test analysis techniques use either execution logs or performance counters. We present the first approach that combines both sources to diagnose memory-related issues in load tests.
2) Our approach is fully automated and scales well with large-scale enterprise systems, flagging $\leq$0.1% of the log lines for further analysis by system experts with a precision $\geq$80%.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.22

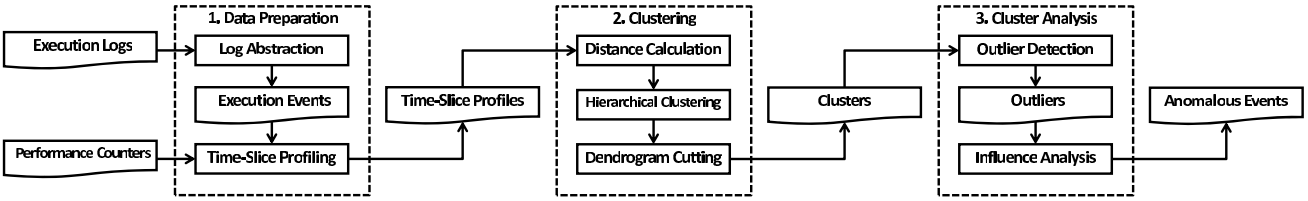

Fig. 1. Overview of Our Approach.

This paper is organized as follows: Section II provides a motivational example of how our approach may be used in practice. Section III describes our approach in detail. Section IV presents the setup and results of our case studies and Section V discusses how changes to the input data impact our results. Section VI outlines the threats to validity of our case studies. Section VII presents related work. Finally, Section VIII concludes the paper and presents future work.

II. Motivational Example

Jack, a performance analyst, performed a 24 hour load test on a ULS system. He discovers that there may be a persistent memory issue in the system based on a visual inspection of memory usage, which continues to increase throughout the entire load test. However, in order to properly report this issue to the developers, Jack must understand the underlying cause (i.e., the usage scenario and functionality that causes this issue). This is a very challenging task because Jack needs to correlate the 3 million log lines and 2,000 samples of memory usage that were collected during the load test.

Jack is introduced to a new load test analysis approach to help him diagnose the memory issue uncovered during the load test. When applied to the execution logs and performance counters that Jack collected, this approach flags 10 events, less than 0.001% of the log lines, for further analysis. These events directly correspond to specific usage scenarios and functionality within the system.

The new approach has produced a much smaller data set that Jack is able to manually analyze. Jack analyzes this much smaller set of execution events and concludes that this memory issue is caused by a particular usage scenario where an error handler fails to release memory back to the system once the error has occurred (i.e., a memory leak). Jack reports this issue, along with the associated events, to the developers.

III. Approach

This section outlines our approach to diagnose memory-related issues in load tests by combining and leveraging the information provided by performance counters and execution logs. Figure 1 provides an overview of our approach. We describe each step in detail below.

The first step in our approach is data preparation. In this step we abstract the execution logs into execution events and combine the performance counters and execution events by discretizing them into time-slices. Each time-slice represents a period of time where we can measure the number of execution events that have occurred and the change in memory usage. The second step is to cluster the time-slices into groups where a similar set of events have occurred. The third step in our approach is to identify the events that are most likely to correspond to the functionality that is causing the memory issue by analyzing the clusters.

We will demonstrate our approach with a working example of a real-time chat application.

Input Data

Execution Logs: Execution logs describe the occurrence of important events in the system. They are generated by output statements that developers insert into the source code of the system. These output statements are triggered by specific execution events (e.g., starting, queueing or completing a job or encountering a specific error). Execution logs record notable events at runtime and are used by developers (to debug a system) and system administrators (to monitor the operation of a system).

The second column of Table I presents the execution logs from our working example. These execution logs contain both static (e.g,. `starts a chat`) and dynamic (e.g., `Alice` and `Bob`) information.

Performance Counters: Performance counters describe system resource usage (e.g., CPU usage, memory usage, network I/O and disk I/O). Memory usage may be measured by a number of different counters, such as 1) the amount of allocated virtual memory, 2) the amount of allocated private (non-shared) memory and 3) the size of the working set (amount of memory used in the previous time-slice). Performance analysts must identify and collect the set of counters that are relevant to their system. Each of these counters may then be analyzed independently. Performance counters are sampled at periodic intervals by resource monitoring tools (e.g., PerfMon) [11].

Table II presents the performance counters (i.e., memory usage) for our working example. A transient memory issues is seen at 00:12 (i.e., memory spikes to 100).

TABLE II
Performance Counters

Time	Memory (MB)
00:00	10
00:04	15
00:08	20
00:12	100
00:16	20
00:20	10

TABLE I
ABSTRACTING EXECUTION LOGS TO EXECUTION EVENTS

Time	Log Line	Execution Event	Execution Event ID
00:01	Alice starts a chat with Bob	USER starts a chat with USER	1
00:01	Alice says 'hi' to Bob	USER says MSG to USER	2
00:02	Bob says 'hello' to Alice	USER says MSG to USER	2
00:05	Charlie starts a chat with Dan	USER starts a chat with USER	1
00:05	Charlie says 'here is the file' to Dan	USER says MSG to USER	2
00:06	Alice says 'are you busy?' to Bob	USER says MSG to USER	2
00:08	Initiate file transfer (Charlie to Dan)	Initiate file transfer (USER to USER)	3
00:13	Complete file transfer (Charlie to Dan)	Complete file transfer (USER to USER)	4
00:14	Charlie ends the chat with Dan	USER ends the chat with USER	5
00:17	Bob says 'yes' to Alice	USER says MSG to USER	2
00:18	Alice says 'ok, bye' to Bob	USER says MSG to USER	2
00:18	Alice ends the chat with Bob	USER ends the chat with USER	5

1. Data Preparation

The first step in our approach is to prepare the execution logs and performance counters for automated, statistical analysis. Data preparation is a two-step process. First, we remove implementation and instance-specific details from the execution logs to generate a set of execution events. Second, we count the number of execution events and the change in memory usage over each sampling interval.

Log Abstraction: Execution logs are not typically designed for automated analysis. Each occurrence of an execution event results in a slightly different log line, because each log line contains static components as well as dynamic information (which may be different for each occurrence of the execution event). Therefore, we must remove this dynamic information from the log lines prior to our analysis in order to identify similar execution events. We refer to the process of identifying and removing dynamic information from a log line as "abstracting" the log line.

Our technique for abstracting log lines recognizes the static and dynamic components of each log line using a technique similar to token-based code cloning techniques [12]. In addition to preserving the static components of each log line, some dynamic information is also partially preserved. This is because some dynamic information may be relevant to memory-issues (e.g., the size of a queue or file). Therefore, this dynamic information is partially preserved by abstracting the numbers into ranges (e.g., quantiles or the order of magnitude).

In order to verify the correctness of our abstraction, many execution logs and their corresponding execution events have been manually reviewed by system experts.

Table I presents the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event for automated analysis and brevity) for the log lines in our working example.

Time-Slice Profiling: We combine performance counters with the execution events using time stamps. When a log line is generated or a performance counter is sampled, the log line or performance counter is written to a log/counter file along with the date and time of generation/sampling.

Although performance counters and execution logs both contain time stamps, combining these two is a major challenge. This is because performance counters are sampled at periodic intervals, whereas execution logs are generated continuously. Therefore, we must discretize the execution logs such that they co-occur with the performance counters.

Discretization also allows us to account for the delayed impact of some functionality on the performance counters. For example, there may be a slight delay between when a log line is generated and when the associated functionality is executed. Discretization also helps to reduce the overhead imposed on the system during load testing because the performance counter sampling frequency can be reduced.

During the period of time between two successive samples of the performance counters (i.e., a "time-slice"), zero or more log lines may be generated by events occurring within the system. For example, a log line may be generated when a new work item is started, queued or completed or a specific error is encountered during the time-slice. The execution events are then discretized by creating a profile for each time-slice. This profile is created by counting the number of times that each type of execution event occurred during the time-slice and by calculating the change in memory usage between the start and end of the time-slice. Therefore, each time-slice has a profile with two components: 1) a log activity component, which is a count of each execution event that has occurred during the time-slice and 2) a memory delta over the time-slice. We refer to the process of connecting the performance counters with the execution events as "profiling" the time-slices.

Our profiling technique is agnostic to the contents and format of the performance counters and execution logs. We do not rely on transaction/thread/job IDs and we do not assume any tags other than a time stamp.

Table III shows the results of profiling the time-slices from our working example.

TABLE III
TIME-SLICE PROFILES

	Log Activity (Execution Event ID)					Memory
Time	1	2	3	4	5	Delta
00:04	1	2	0	0	0	5
00:08	1	2	0	0	0	5
00:12	0	0	1	0	0	80
00:16	0	0	0	1	1	−80
00:20	0	2	0	0	1	−10

2. Clustering

The second step in our approach is to cluster the time-slice profiles into groups with similar log activity (i.e., where a similar set of events have occurred). This is because we expect that similar log activity should lead to similar memory deltas. Memory-related issues will impact these memory deltas. We have automated the clustering step using robust statistical techniques to account for the size of the data.

Distance Calculation: Each time-slice profile is represented by one point in a multi-dimensional space. Clustering procedures rely on identifying points that are "close" in this multi-dimensional space. Therefore, we must specify how distance is to be measured in this space. Larger distance between two points imply a greater dissimilarity between the time-slice profiles that these points represent. We calculate the distance between the log activity component of every pair of time-slice profiles. This produces a distance matrix.

We use the Pearson distance, as opposed to the many other distance measures [13]–[15], as this measure often results in a clustering that is closer to the true clustering [14], [15].

We first use the Pearson correlation to calculate the similarity between two profiles. This measure ranges from -1 to +1, where a value of 1 indicates that two profiles are identical, a value of 0 indicates that there is no relationship between the profiles and a value of -1 indicates an inverse relationship between the profiles (i.e., as the occurrence execution logs increase in one profile, they decrease in the other).

$$\rho = \frac{n\sum_i^n x_i \times y_i - \sum_i^n x_i \times \sum_i^n y_i}{\sqrt{(n\sum_i^n x^2 - (\sum_i^n x)^2) \times (n\sum_i^n y^2 - (\sum_i^n y)^2}} \quad (1)$$

where x and y are the log activity components of two time-slice profiles and n is the number of execution events. We then convert the Pearson correlation to the Pearson distance.

$$d_\rho = \begin{cases} 1 - \rho & \text{for } \rho \geq 0 \\ |\rho| & \text{for } \rho < 0 \end{cases} \quad (2)$$

Table IV presents the distance matrix produced by calculating the Pearson distance between every pair of time-slice profiles in our working example.

TABLE IV
DISTANCE MATRIX

	00:04	**00:08**	**00:12**	**00:16**	**00:20**
00:04	0	0.333	0.408	0.408	0.667
00:08	0.333	0	0.612	0.612	0.167
00:12	0.408	0.612	0	0.25	0.408
00:16	0.408	0.612	0.25	0	0.388
00:20	0.667	0.167	0.408	0.388	0

Hierarchical Clustering: We cluster the time-slice profiles (i.e., to group time-slices where a similar set of logs have occurred) using the distance matrix and an agglomerative, hierarchical clustering procedure. This procedure starts with each profile in its own cluster and proceeds to find and merge the closest pair of clusters (using the distance matrix), until only one cluster (containing everything) is left. Every time two clusters are merged, the distance matrix is updated.

Hierarchical clustering updates the distance matrix based on a specified linkage criteria. We use the average linkage, as opposed to the many other linkage criteria [13], [16], as this linkage is the most appropriate when little information about the expected clustering (e.g., the relative size of the expected clusters) is available. Every time two clusters are merged, the average linkage criteria removes the merged clusters from the distance matrix and adds the new cluster by calculating the distance between the new cluster and all existing clusters. The distance between two clusters is the average distance (as calculated by the Pearson distance) between the profiles of the first cluster and the profiles of the second cluster [13], [16].

Figure 2 shows the dendrogram produced by hierarchically clustering the time-slice profiles using the distance matrix (Table IV) from our working example.

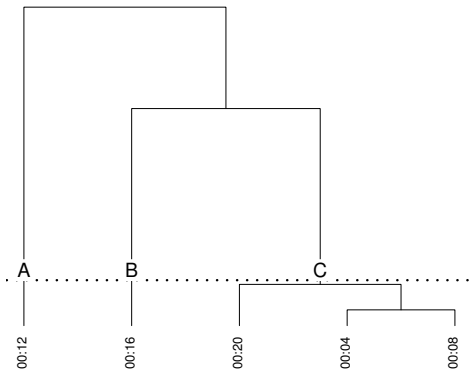

Fig. 2. Sample Dendrogram.

Dendrogram Cutting: The result of a hierarchical clustering procedure is a hierarchy of clusters that are typically visualized using hierarchical cluster dendrograms (e.g., Figure 2). These are binary tree-like diagrams that show each stage of the clustering procedure as nested clusters [16].

To complete the clustering procedure, the dendrogram must be cut at some height. This results in a clustering where each time-slice profile is assigned to only one cluster. Such cutting of the dendrogram is done either by manual (visual) inspection or by statistical tests (referred to as stopping rules).

Although a visual inspection of the dendrogram is flexible and fast, it is subject to human bias and may not be reliable. We use the Calinski-Harabasz stopping rule, as opposed to the many other stopping rules [17]–[21], as this rule is commonly referred to as the most accurate [19]. The Calinski-Harabasz stopping rule is a pseudo-F-statistic, which is a ratio reflecting within-cluster similarity and between-cluster dissimilarity. The optimal clustering will have high within-cluster similarity (i.e., the time-slice profiles within a cluster are very similar) and a high between-cluster dissimilarity (i.e., the time-slice profiles from two different clusters are very dissimilar).

The horizontal line in Figure 2 shows how the Calinski-Harabasz stopping rule is used to cut the dendrogram from our working example into three clusters. Cluster A contains one time-slice profile (00:12), cluster B contains one (00:16) and cluster C contains three (00:20, 00:04 and 00:08).

TABLE V
SCORING TECHNIQUES FOR IDENTIFYING OUTLYING CLUSTERS

	Transient Memory Issues	**Persistent Memory Issues**	
	Memory Spike	*Memory Bloat*	*Memory Leak*
Motivation	Functionality causing a memory spike is characterized by a higher than average memory delta in small clusters or a combination of a higher than average memory delta and higher than average memory delta standard deviation in larger clusters. The standard deviation component is scaled with the cluster size to emphasize the standard deviation component in larger clusters.	Functionality causing memory bloat is characterized by a higher than average memory delta.	Functionality causing a memory leak is characterized by a combination of a higher than average memory delta and a higher than average memory delta standard deviation. The standard deviation component is added because functionality with variable memory deltas may indicate a memory leak.
Score	$spike_i = \frac{n_i \times \sigma_i}{\max(n \times \sigma)} + \frac{\mu_i}{\max \mu}$	$bloat_i = \frac{\mu_i}{\max \mu}$	$leak_i = \frac{\sigma_i}{\max \sigma} + \frac{\mu_i}{\max \mu}$
	$\mu_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \Delta memory_{i,j}$ $\quad$ $\sigma_i = \sqrt{\frac{1}{n_i - 1} \sum_{j=1}^{n_i} (\Delta memory_{i,j} - \mu_i)^2}$ where i is the cluster number, n_i is the size of cluster i, μi is the average memory delta across each time-slice profiles, j, in cluster i, σ_i is the standard deviation of the memory deltas of cluster i and $spike_i$, $bloat_i$ and $leak_i$ are the scores assigned to cluster i. n, μ and σ are vectors containing the cluster size, average memory delta and standard deviation of the memory deltas of all clusters. The standard deviation of a cluster with one time-slice profile is arbitrarily assigned the maximum standard deviation (i.e., such that the first component of the $spike$ and $leak$ scores is equal to one).		

3. Cluster Analysis

The third step in our approach is to identify the log lines that correspond to the functionality that is responsible for the memory issue exhibited by the data set. First, outlying clusters are detected. Second, the key log lines of the outlying clusters are identified. As in our previous step, we have used robust statistical techniques to automate this step.

Outlier Detection: We identify outlying clusters by examining the memory deltas from each of the time-slice profiles within each of the clusters. Outlying clusters will contain time-slice profiles that have a significant impact on memory usage (as evidenced by the memory deltas). Given the wide variety of memory-related performance issues, identifying time-slice profiles that have the "right" impact on memory usage is a major challenge.

After discussions with system experts and a review of memory-related issue reports from an enterprise system, we developed scoring techniques to identify clusters that contain evidence of these memory issues. Table V presents the scoring technique used to identify transient memory issues (i.e., memory spikes) and persistent memory issues (i.e., memory bloat or memory leaks).

We calculate either the memory spike ($spike$) or memory bloat ($bloat$) and memory leak ($leak$) scores for each cluster, depending on whether a transient or persistent memory issue is detected. Outlying clusters are identified as having a score that is more than twice the standard deviation above the average score (i.e., a z-score greater than 2). The z-score is the number of standard deviations a data point is from the average.

Table VI presents the $spike$ score for each of the clusters in our working example (i.e., each of the clusters that were identified when the Calinski-Harabasz stopping rule was used to cut the dendrogram in Figure 2). We calculate the $spike$ score because a transient memory issues is seen at 00:12 (i.e., memory spikes to 100 at 00:12).

TABLE VI
Spike SCORES FOR FIGURE 2 CLUSTERS

Cluster	$Spike$ **Score**
Cluster A	2
Cluster B	0
Cluster C	1
μ_{spike}	1
σ_{spike}	1

From Table VI, we find that the average $spike$ score (μ_{spike}) is 1 and the standard deviation in the $spike$ scores (σ_{spike}) is 1. Therefore, no clusters are identified as outliers (i.e., there are no $spike$ scores $\geq \mu_{spike} + 2 \times \sigma_{spike} = 3$). However, outliers are extremely difficult to detect in such small data sets. Therefore, for the purposes of this working example, we will use one standard deviation (as opposed to two standard deviations). Consequently, we identify Cluster A as an outlying cluster (as it is one standard deviation above the average).

Influence Analysis: We perform an influence analysis on the outlying clusters to determine which execution events differentiate the time-slice profiles in outlying clusters from the average ("normal") time-slice profile. These execution events are most likely to be responsible for the cause of memory-related issues.

We first calculate the centre of the outlying cluster and the universal centre. The centre of the outlying cluster is calculated by averaging the count of each event in all the time-slice profiles of a cluster. Similarly, the universal centre is calculated by averaging the count of each event in all the time-slice profiles of all clusters. These centres represent the location, in an n-dimensional space (where n is the number of unique execution events), of each of the clusters, as well as the average ("normal") time-slice profile.

We then calculate the Pearson distance (Equation 1 and Equation 2) between the centre of the outlying cluster and the universal centre. This "baseline" distance quantifies the difference between the time-slice profiles in outlying clusters and the universal average time-slice profile.

We then calculate the change in the baseline distance between the outlying cluster's centre and the universal centre with and without each execution event. This quantifies the influence of each execution event. When an overly influential execution event is removed, the outlying cluster becomes more similar to the universal average time-slice profile (i.e., closer to the universal center).

Therefore, overly influential execution events are identified as any execution event that, when removed from the distance calculation, decreases the distance between the outlying cluster's centre and the universal centre by more than twice the standard deviation above the average change in distance.

Table VII presents the change in the distance between Cluster A and average ("normal") time-slice profile when each event is removed from the distance calculation.

TABLE VII
IDENTIFYING OVERLY INFLUENTIAL EXECUTION EVENTS

Event ID	Δd_ρ
1	4.267×10^{-2}
2	1.999×10^{-1}
3	-3.774×10^{-1}
4	1.487×10^{-1}
5	4.267×10^{-2}
$\mu_{\Delta d_\rho}$	1.131×10^{-2}
$\sigma_{\Delta d_\rho}$	2.278×10^{-1}

From Table VII, the average Δd_ρ ($\mu_{\Delta d_\rho}$) is 1.131×10^{-2} and the standard deviation in Δd_ρ ($\sigma_{\Delta d_\rho}$)is 2.278×10^{-1}. Therefore, no clusters are identified as outliers (i.e., no Δd_ρ is $\geq \mu_{\Delta d_\rho} - 2 \times \sigma_{\Delta d_\rho} = -4.442 \times 10^{-1}$). Again, for the purposes of this example, we will use one standard deviation (as opposed to two standard deviations). Therefore, we identify event 3 as overly influential. This flagged event corresponds to initiating the transfer of a file. Performance analysts and developers now have a concrete starting point for their investigation of this memory issue.

IV. CASE STUDY

This section outlines the setup and results of our case study. First, we provide an overview of the subject systems. We then present a case study using a Hadoop application. Finally, we discuss the results of an enterprise case study.

A. Subject Systems

Our case studies use performance counters and execution logs from two systems. Table VIII outlines the systems and data sets used in our case study.

Hadoop Case Study: Our first system is an application that is built on Hadoop. Hadoop is an open-source distributed data processing platform that implements the MapReduce data processing framework [22], [23].

MapReduce is a distributed data processing framework that allows large amounts of data to be processed in parallel by the nodes of a distributed cluster of machines [23]. The MapReduce framework consists of the Map component, which divides the input data amongst the nodes of the cluster, and the Reduce component, which collects and combines the results from each of the nodes.

Enterprise System Case Study: Our second system is a large-scale enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system's architecture, however the system is responsible for simultaneously processing thousands of client requests and has very high performance requirements.

B. Hadoop Case Study

Our Hadoop case study focuses on the WordCount application [24]. The WordCount application is a standard example of a Hadoop application that is used to demonstrate the Hadoop MapReduce Framework.

WordCount: The WordCount application [24] reads one or more text files (a corpus) and counts the number of times each unique word occurs within the corpus. The output is one or more text files (depending on the number of unique words in the corpus), with one unique word and the number of times that word occurs in the corpus per line.

Load Test Configuration: We load test the Hadoop WordCount application on our cluster by attempting to count the number in times each unique word occurs in two 150MB files and one 15GB file. Linefeeds and carriage-returns are removed from one of the 150MB files so that the file is composed on one line.

According to the official Hadoop documentation: *as the Map operation is parallelized the input file set is first split to several pieces called FileSplits. If an individual file is so large that it will affect seek time it will be split to several Splits. The splitting does not know anything about the input file's internal logical structure... [25].*

Input files are split using linefeeds or carriage-returns [24], [26]. Therefore, attempting to read the one-line 150MB file (which lacks linefeeds and carriage-returns) will result in a persistent memory issue (i.e., memory bloat as the application attempts to read the entire file into memory).

TABLE VIII
CASE STUDY SUBJECT SYSTEMS.

	Hadoop	Enterprise System	
Application domain	Data processing	Telecom	
License	Open-source	Enterprise	
Memory issue	Memory bloat	Memory leak	Memory spike
Load test duration	49.2 minutes	17.5 hours	45.5 hours
Number of log lines	5,303	2,685,838	182,298,912
Number of flagged events	1	10	4
Reduction in analysis effort	1 - 0.019%	1 - 0.00037%	1 - 0.0000022%
Precision	100%	80%	100%

Application Failure: During the load test, the application fails prior to processing the input files. As expected, the cause of the failure is a memory-related issue. The following log line (an error message) is seen in Hadoop's execution logs, indicating that the WordCount application has a memory issue (i.e., the application is trying to allocate more memory than available in the heap): `FATAL org.apache.hadoop.mapred.TaskTracker: Task: attempt_id - Killed: Java heap space`

Further, a plot of memory usage (i.e., memory heap usage) for the WordCount application on the virtual machine with the failure (Figure 3) clearly shows a persistent memory issue.

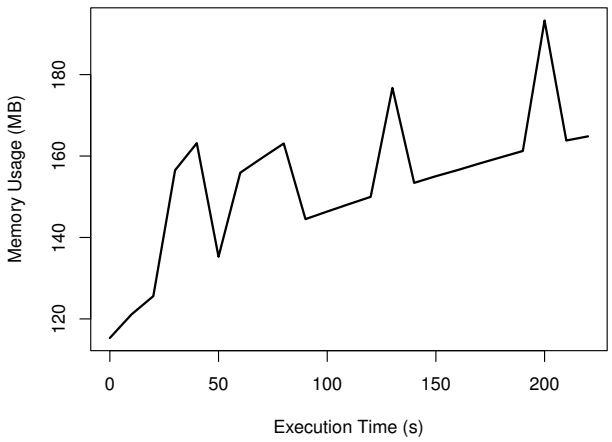

Fig. 3. Memory Usage

Applying Our Approach: We apply our approach to the execution logs and performance counters collected during our load test. Our approach identifies the following log line as most likely to be responsible for this issue and flags it for further analysis: `INFO org apache hadoop mapred TaskTracker attempt_id hdsf file start 1 end 8`

Results: Our approach flagged only one log line for expert analysis. As expected, this log line relates to data input. Further, the numbers in this log line (e.g., "start *1* end *8*") relates to the order of magnitude of the seek position (`start`) within the file and the number of bytes (`end`) to be read. As expected, our approach has flagged the execution event where the data is read and split amongst the nodes of the cluster for processing [26]. Our approach managed to reduce 5,303 log lines into 1 execution event (a $100\% - 0.019\% = 99.981\%$ reduction in analysis effort) with a precision of 100% (i.e., this log line is relevant to the memory issue).

C. Enterprise System Case Study

Although the results of our Hadoop study were promising, we apply our approach to two load tests of our Enterprise System to examine the scalability of our approach. Similar to our Hadoop data set, these load tests have exposed memory issues, however, they are significantly larger than our Hadoop data set. From Table VIII, we find that our enterprise case studies comprise of data sets that contain 500 times more log lines from load tests that are 20 times longer that our Hadoop data set.

We perform two case studies using the performance counters and execution logs collected during two separate load tests of the system that have been performed and analyzed by system experts. The first load test exposed a memory leak caused by a specific piece of functionality. The second load test exposed a memory spike caused by rapidly queueing a large number of work items. Each of these load tests have been analyzed by system experts and their conclusions have been verified by at least one additional expert. Therefore, we are confident that the true cause of the memory issues have been correctly identified.

Our approach was applied to the performance counters and execution logs collected during each of these load tests. We flag 10 log lines (0.00037% of the 2,685,838 log lines) and 4 log lines (0.0000022% of the 182,298,912 log lines) for the memory leak and memory spike data sets respectively. We correctly identified 8 log lines (80% precision) and 4 log lines (100% precision) for the memory leak and memory spike data sets respectively. These log lines correspond directly to the functionality and usage scenario that system experts have determined to be the cause of the memory issue. For confidentiality reasons, we cannot disclose the log lines our approach has flagged or the functionality and usage scenario to which they correspond. However, our results have been independently verified by system experts.

V. DISCUSSION

Although our approach has been fully automated to analyze the execution logs and performance counter generated during a load test, how this data is generated and collected is open to the system's developers and load testers. Developers must ensure that they are inserting accurate execution logs that cover the system's features and load testers must specify how often the performance counters are sampled (i.e., the sampling interval).

The sampling interval of our data sets range from 5 seconds (Hadoop data set) to 30 seconds and 3 minutes (Enterprise data sets). To explore how varying the sampling interval would impact our results, we simulate longer sampling intervals in

the data set where a memory leak was found in our Enterprise System. The sampling interval for this data set is 30 seconds, however we simulate longer sampling intervals by merging successive time-slice profiles. For example, a 60 second sampling interval can be created by counting the execution events and calculating the memory delta over two successive 30 second intervals. Using this approach, we simulate 60, 90, 120, 150 and 180 second sampling intervals.

Table IX presents how the number of flagged events, the precision (the percentage of correctly flagged events) and the recall is impacted by an increased sampling interval. As we do not have a gold standard data set, we calculate recall using the best results in Table IX. Sampling intervals between 90 and 150 seconds correctly flag 13 events. Hence, we measure recall as the percentage of these 13 events that are flagged.

TABLE IX
IMPACT OF INCREASING THE SAMPLING INTERVAL

	Sampling Interval					
	30s	60s	90s	120s	150s	180s
Flagged events	10	5	15	15	15	2
Precision (%)	80	100	87	87	87	0
Recall (%)	62	38	100	100	100	0

From Table IX, we find that sampling intervals between 90 seconds and 150 seconds flag events with high precision and recall. However, performance analysts may need to tune this parameter based on the duration of their load tests and the sampling overhead on their system.

VI. THREATS TO VALIDITY

A. Threats to Construct Validity

1) Monitoring Performance Counters: Our approach is based on the ability to identify log lines that have a significant impact on memory usage. This is based on the assumption that memory is allocated when requested and the allocation of memory can be reliably monitored. Our approach should be able to correctly identify the cause of memory issues in any system that shares this property. To date, we have not yet encountered a system where this property does not hold.

2) Timing of Events: Our approach is also based on the ability to combine the performance counters (specifically memory usage) and execution logs. This is done using the date and time from each log line and performance counter sample. However, large-scale software systems are often distributed, therefore the timing of events may not be reliable [27]. However, the performance counters and execution logs used in our case studies were generated from the same machine. Therefore, there are no issues regarding the timing of events. System experts also agree that this timing information is correct.

The timing of events may also be impacted if the time stamps in the performance counters and execution logs do not reliably reflect when the counters/logs were sampled/generated. However, we have found that the time stamps reflect the times that the counters/logs were sampled/generated, as opposed to the time the counters/logs were written to a file. Therefore, the time stamps of the performance counters and execution logs reliably reflect the true order of the events.

Our approach should only be used when the performance counters and execution logs are reliably collected.

3) Evaluation: We have evaluated our approach by determining the precision with which our approach flags execution events (i.e., the percentage of flagged events that are relevant to the memory issue). While system analysts have verified these results, we do no have a gold standard data set. Further, complete system knowledge would be required to identify all of the execution events that are relevant to a particular issue. Therefore, we cannot calculate the recall of our approach. However, our approach is intended to help performance analysts diagnose memory-related issues by flagging execution events for further analysis (i.e., to provide analysts with a starting point). Therefore, our goal is to maximize precision so that analysts have confidence in our approach. In our experience, performance analysts agree with this view. Additionally, we were able to identify at least one event that was relevant to the memory issue at hand in all three case studies.

B. Threats to Internal Validity

1) Selecting Performance Counters: Our approach requires performance counters measuring memory usage. However, memory usage may be measured by a number of different counters including, 1) allocated virtual memory, 2) allocated private (non-shared) memory and 3) the size of the working set (amount of memory used in the previous time-slice). Performance analysts should sample all of the counters that may be relevant. Once the load test is complete, performance analysts can than detect whether memory-related issues are seen in any of the counters and use our approach to diagnose these issues. However, performance analysts may require system-specific expertise to select an appropriate set of performance counters.

2) Execution Log Quality/Coverage: Our approach assumes that the cause of any memory issue is manifested within the execution logs (i.e., there are log lines associated with the exercise of this functionality). However, it is possible that there are no execution logs to indicate when certain functionality is exercised. Therefore, our approach is incapable of identifying this functionality in the case that they cause memory issues. Further, our approach is incapable of identifying functionality that does not occur while performance counters are being collected (e.g., if the system crashes). However, this is true for all execution log based analysis, including manual analysis.

This issue may be mitigated by utilizing automated instrumentation tools that would negate the need for developers to manually insert output statements into the source code. However, we leave this to future work as automated instrumentation imposes a heavy overhead on the system [28].

C. Threats to External Validity

1) Generalizing Our Results: The studied software systems represent a small subset of the total number of software systems. Therefore, it is unclear how our results will generalize to additional systems, particularly systems from other domains (e.g., e-commerce). However, our approach does not assume any particular architectural details.

VII. Related Work

Our approach is a form of dynamic program analysis, however much of the existing work on dynamic analysis focuses on the functional behaviour of a system (except for some work on visualizing threads [29], [30]), whereas we focus on the performance of a system. Cornelissen et al. present an excellent survey of dynamic analysis [31]. Performance testing, load test analysis, performance monitoring and software ageing are the closest areas of research to our work.

A. Performance Testing

Grechanik et al. propose a novel approach to performance testing based on black-box software testing [32]. Their approach analyzes execution traces to learn rules describing the computational intensity of a workload based on the input data. An automated test script then selects test input data that potentially expose performance bottlenecks (where such bottlenecks are limited to one or few components). Our approach is not limited to finding performance bottlenecks and relies on existing testing infrastructure.

B. Load Test Analysis Using Execution Logs

Jiang et al. mine execution logs to determine the dominant (expected) behaviour of the application and to flag anomalies from the dominant behaviour [9]. Their approach is able to flag $<0.01\%$ of the execution log lines for closer analysis. Our approach does not assume that performance problems are associated with anomalous behaviour.

Jiang et al. also flag performance issues in specific usage scenarios by comparing the distribution of response times for the scenario against a baseline derived from previous tests [2]. Their approach reports scenarios that have performance problems with few false positives (77% precision). Our approach does not rely on baselines derived from previous tests.

C. Load Test Analysis Using Performance Counters

Load test researchers have also used performance counters to detect performance problems and identify the probable cause of performance regressions [8], [28], [33]–[35].

Foo et al. use association rule mining to extract correlations between the performance counters collected during a load test [33]. The authors compare the correlations from one load test to a baseline to identify performance deviations. Nguyen et al. use control charts to identify load tests with performance regressions and detect which component is the cause of the regression [34], [35]. Malik et al. have used principle component analysis (PCA) to generate performance signatures for each component of a system. The authors assess the pair-wise correlations between the performance signatures of a load test and a baseline to identify performance deviations with high accuracy (79% accuracy) [8], [28].

Our approach does not rely on performance baselines derived from previous tests. Further, our approach can pinpoint the cause of performance problems at a much lower level (i.e., execution log level) compared to Nguyen et al. [34], [35] and Malik et al. [8], [28] (i.e., the component level). Finally, our approach focuses on *diagnosing* (i.e., discovering the cause) memory issues that have already been *detected*.

In our previous work, we proposed an approach to identify performance deviations in thread pools using performance counters [36], [37]. Our approach identified performance deviations (e.g., memory leaks) with high precision and recall. However, we did not make use of execution logs. Hence, we could not identify the underlying cause of these deviations.

D. Performance Monitoring

Research in automated performance monitoring has developed application signatures based on performance counters that can be used to detect changes to the performance of an application as it evolves over time [38]–[40]. However, these methodologies require a baseline model of the application's performance in order to characterize changes resulting from software evolution and maintenance.

E. Software Ageing Monitoring

Work in software ageing has developed monitoring techniques to detect the effects of software ageing. Software ageing is defined as the progressive degradation of a system's performance during its operational lifetime [41]. This degradation is typically caused by resource exhaustion. Researchers have noted the importance of memory issues in software ageing; memory is the most cited cause of software ageing [42]–[50].

VIII. Conclusions and Future Work

In this paper, we proposed a novel approach to combine the information provided by both performance counters and execution logs. Our approach is intended to help performance analysts diagnose memory-related performance issues.

We performed three case studies using Hadoop and an Enterprise System. Each of our case studies investigated a different memory issue (i.e., memory bloat, memory leak and memory spike). We have shown that our approach can correctly diagnose memory issues.

Although our approach performed well in diagnosing memory issues, we intend to explore our ability to diagnose other performance issues (e.g., CPU spikes).

Acknowledgement

We would like to thank BlackBerry for providing access to the enterprise system used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of BlackBerry's products.

References

[1] S. E. Institute, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.

[2] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Proceedings of the International Conference on Software Maintenance*, Sep 2009, pp. 125–134.

[3] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, Dec 2000.

[4] "Steve Jobs on MobileMe," www.arstechnica.com/apple/2008/08/steve-jobs-on-mobileme-the-full-e-mail/, Last Accessed: 17-Apr-2013.

[5] "Firefox Download Stunt Sets Record For Quickest Meltdown," www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/, Last Accessed: 17-Apr-2013.

[6] "IT Downtime Costs $26.5 Billion In Lost Revenue," www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441, Last Accessed: 17-Apr-2013.

[7] "The Avoidable Cost of Downtime," http://www.arcserve.com/us/lpg/costofdowntime.aspx, Last Accessed: 17-Apr-2013.

[8] H. Malik, "A methodology to support load test analysis," in *Proceedings of the International Conference on Software Engineering*, May 2010, pp. 421–424.

[9] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the International Conference on Software Maintenance*, Oct 2008, pp. 307–316.

[10] "Summary of the October 22, 2012 AWS Service Event in the US-East Region," http://aws.amazon.com/message/680342/, Last Accessed: 17-Apr-2013.

[11] "PerfMon," www.technet.microsoft.com/en-us/library/bb490957.aspxx, Last Accessed: 17-Apr-2013.

[12] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance and Evolution*, vol. 20, no. 4, pp. 249–267, Jul 2008.

[13] I. Frades and R. Matthiesen, "Overview on techniques in cluster analysis," *Bioinformatics Methods In Clinical Research*, vol. 593, pp. 81–107, Mar 2009.

[14] A. Huang, "Similarity measures for text document clustering," in *Proceedings of the New Zealand Computer Science Research Student Conference*, Apr 2008, pp. 44–56.

[15] N. Sandhya and A. Govardhan, "Analysis of similarity measures with wordnet based text document clustering," in *Proceedings of the International Conference on Information Systems Design and Intelligent Applications*, Jan 2012, pp. 703–714.

[16] P.-N. Tan, M. Steinbach, and V. Kumar, *Cluster Analysis: Basic Concepts and Algorithms*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., 2005.

[17] T. Calinski and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics*, vol. 3, no. 1, pp. 1–27, Jan 1874.

[18] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, 1st ed. John Wiley & Sons Inc, 1973.

[19] G. W. Milligan and M. C. Cooper, "An examination of procedures for determining the number of clusters in a data set," *Psychometrika*, vol. 50, no. 2, pp. 159–179, Jun 1985.

[20] R. Mojena, "Hierarchical grouping methods and stopping rules: An evaluation," *The Computer Journal*, vol. 20, no. 4, pp. 353–363, Nov 1977.

[21] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, no. 1, pp. 53–65, Nov 1987.

[22] "Hadoop," www.hadoop.apache.org/, Last Accessed: 17-Apr-2013.

[23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008.

[24] "MapReduce Tutorial," http://hadoop.apache.org/docs/stable/mapred_tutorial.html, Last Accessed: 17-Apr-2013.

[25] "HadoopMapReduce," http://wiki.apache.org/hadoop/HadoopMapReduce, Last Accessed: 17-Apr-2013.

[26] "TextInputFormat," http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/TextInputFormat.html, Last Accessed: 17-Apr-2013.

[27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul 1978.

[28] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Automatic comparison of load tests to support the performance analysis of large enterprise systems," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Mar 2010, pp. 222–231.

[29] S. P. Reiss, "Efficient monitoring and display of thread state in java," in *Proceedings of the International Workshop on Program Comprehension*, May 2005, pp. 247–256.

[30] ——, "Controlled dynamic performance analysis," in *Proceedings of the International Workshop on Software and Performance*, Jun 2008, pp. 43–54.

[31] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, Sep 2009.

[32] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proceedings of the International Conference on Software Engineering*, Jun 2012, pp. 156–166.

[33] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, K. Martin, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Proceedings of the International Conference on Quality Software*, Jul 2010, pp. 32–41.

[34] T. H. D. Nguyen, "Using control charts for detecting and understanding performance regressions in large software," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, Apr 2012, pp. 491–494.

[35] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated verification of load tests using control charts," in *Proceedings of the Asia-Pacific Software Engineering Conference*, Dec 2011, pp. 282–289.

[36] M. D. Syer, B. Adams, and A. E. Hassan, "Industrial case study on supporting the comprehension of system behaviour," in *Proceedings of the International Conference on Program Comprehension*, jun 2011, pp. 215–216.

[37] ——, "Identifying performance deviations in thread pools," in *Proceedings of the International Conference on Software Maintenance*, sep 2011, pp. 83–92.

[38] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2008, pp. 452–461.

[39] ——, "Automated anomaly detection and performance modeling of enterprise applications," *Transactions on Computer Systems*, vol. 27, no. 3, pp. 6:1–6:32, Nov 2009.

[40] N. Mi, L. Cherkasova, K. Ozonat, J. Symons, and E. Smirni, "Analysis of application performance and its change via representative application signatures," in *Proceedings of the Symposium on Network Operations and Management*, Apr 2008, pp. 216–223.

[41] D. L. Parnas, "Software aging," in *Proceedings of the international conference on Software engineering*, May 1994, pp. 279–287.

[42] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A methodology for detection and estimation of software aging," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov 1998, pp. 283–292.

[43] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, Sep 2010.

[44] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of software aging in a web server," *Transactions on Reliability*, vol. 55, no. 3, pp. 411–420, Sep 2006.

[45] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proceedings of the International Workshop on Software Aging and Rejuvenation*, Nov 2008, pp. 1–6.

[46] A. Macedo, T. B. Ferreira, and R. Matias, "The mechanics of memory-related software aging," in *Proceedings of the International Workshop on Software Aging and Rejuvenation*, Nov 2010, pp. 1–5.

[47] R. Matias, B. Evangelista, and A. Macedo, "Monitoring memory-related software aging: An exploratory study," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, Nov 2012, pp. 247–252.

[48] R. Matias and P. J. F. Filho, "An experimental study on software aging and rejuvenation in web servers," in *Proceedings of the International Computer Software and Applications Conference*, Sep 2006, pp. 189–196.

[49] Q. Ni, W. Sun, and S. Ma, "Memory leak detection in sun solaris os," in *Proceedings of the International Symposium on Computer Science and Computational Technology*, Dec 2008, pp. 703–707.

[50] M. Shereshevsky, J. Crowell, B. Cukic, V. Gandikota, and Y. Liu, "Software aging and multifractality of memory resources," in *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2003, pp. 721–730.