Identifying Performance Deviations in Thread Pools

Mark D. Syer, Bram Adams and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{mdsyer, bram, ahmed}@cs.queensu.ca

***Abstract*—Large-scale software systems handle increasingly larger workloads by implementing highly concurrent and distributed design patterns. The thread pool pattern uses pools of pre-existing and reusable threads to limit thread lifecycle overhead (thread creation and destruction) and resource thrashing (thread proliferation). However, these advantages are weighed against performance issues caused by concurrency risks, like synchronization errors or deadlock, and thread pool-specific risks, like poorly tuned pool size or thread leakage. Detecting these performance issues during load testing requires a thorough understanding of how thread pools behave, yet most performance analysts have limited knowledge of the system and are flooded with terabytes of data from load tests. We propose a methodology to identify threads with performance deviations in thread pools. Our methodology ranks threads based on the dissimilarity of their resource usage metrics. A case study on a large-scale industrial software system shows that our methodology can identify threads with performance deviations with an average precision of 100% and an average recall of 76.61%. Our methodology performs very well when ranking long-lived deviations, such as memory leaks, but more work is needed to rank short-lived deviations, such as CPU spikes.**

***Index Terms*—thread-pools; behaviour-based clustering; understanding ULS systems;**

I. Introduction

The rise of ultra-large-scale (ULS) e-commerce and telecommunication systems, like Amazon or Facebook, poses a new challenge for the software maintenance field. ULS systems require near-perfect up-time and potentially must support hundreds or even thousands of concurrent connections and operations. Failures in such systems are more often associated with an inability to scale to performance demands, than with feature bugs [1], [2].

To ensure that ULS systems are able to perform according to the expected workloads, performance analysts have become key players in the development and maintenance of today's enterprises. Performance analysts are responsible for performing load tests on systems and monitoring how the systems behave under realistic workloads. Such load tests allow performance analysts to determine the maximum operating capacity of a system, to validate non-functional requirements and to uncover bottlenecks. A load test typically lasts from several hours to several days and generates terabytes of performance data and execution logs [3].

Despite the crucial role of performance analysts, current research tends to ignore their needs. The current state of the practice requires significant manual review of the performance data and execution logs and a high degree of insight into the system behaviour [2], [4], [5]. Tool support is lacking because most research techniques require either heavy instrumentation (disrupting the work-load) or more detailed log entries [2], [5]–[8] (often unavailable during load tests because collecting detailed logs impacts the system's performance). Resource usage metrics like CPU and memory usage, gathered by hardware sensors, are often the only feasible data source for ULS systems, but are very hard to interpret [3], [9].

To focus our discussion, this paper considers performance deviations in ULS systems that are designed using the thread pool architectural pattern. The thread pool design pattern is a common pattern for designing scalable multithreaded and distributed systems. The thread pool pattern consists of a fixed number of pre-existing and reusable threads that facilitate incoming service requests. Thread pools limit thread lifecycle overhead and can be used to prevent resource thrashing. Despite their potential for scalability, thread pools are hard to configure and test because of concurrency risks like synchronization errors and deadlock and thread pool-specific risks like resource thrashing and thread leakage.

To support analysts in identifying performance deviations in thread pools, we present an iterative, top-down methodology for automatically identifying and ranking deviating behaviour based on resource usage metrics. Our methodology can be applied with very little understanding of the architecture or purpose of the software system under test. Our methodology is based on measuring the level of dissimilarity between the resource usage metrics of threads or groups of threads, then ranking the most dissimilar threads or thread groups.

This paper makes two main contributions:

1) We present a top-down methodology for identifying and ranking the most deviating thread behaviour.
2) We perform a qualitative and quantitative evaluation of our methodology on a large-scale industrial ULS system. Our methodology performs very well when ranking long-lived deviations (e.g., memory leaks), but more work is required to rank short-lived deviations (e.g. CPU spikes).

The paper is organized as follows: Section II describes the current challenges to understanding the behaviour of thread pools. Before Section IV presents our methodology, Section III provides a motivational example of how it may be applied in practice. Section V describes the setup of our case study on a large industrial software system and Section VI presents the results of our qualitative and quantitative evaluation. Section VII outlines threats to validity, whereas Section VIII discusses related work. Finally, Section IX concludes the paper.

II. Thread Pools

The thread pool design pattern is a popular technique for designing scalable multithreaded and distributed systems [10]–[13]. A thread pool is a collection of threads that are available for computational tasks (work items) and that can be recycled. When a thread completes its current task, it returns to the thread pool for reuse instead of terminating. Incoming work items are assigned to available threads in the thread pool or queued until a thread becomes available. Thread pools actually are a special case of a resource pool. Other types of resource pools include pools of memory, pools of database connections and pools of connections to a backend.

Thread pools are particularly useful in server applications where work items are typically short-lived and the number of incoming work items is large. Since creating and destroying a thread has a significant overhead, systems where work items do not take much time to process spend as much time creating and destroying threads as processing work items. The reuse of threads across work items in a thread pool substantially reduces the associated overhead. In addition, by using pre-existing threads, the system becomes more responsive, enabling incoming work items to be assigned and executed almost instantly.

Resources, such as CPU time and memory, are allocated to threads so they can process their items. Creating too many threads causes resource thrashing, where the amount of resources expended to manage the threads increases at the expense of the resources available to process the work items. CPU and memory thrashing are important problems, potentially leading to a significant degradation in system performance [14]. Thread pools enable explicit control of the resources allocated to threads via policies on, for example, the maximum and minimum number and/or lifetime of pooled threads.

Despite their potential for scalability, thread pools are hard to configure and test. For example, synchronization errors might occur when notifications to threads are lost. This can result in threads remaining idle even if there are work items on the queue. Worse, thread pools introduce a special kind of deadlock where all threads in the pool are waiting for a work item on the queue to be processed, but there are no "available" threads to process that work item. Poorly configured thread pools still face the risk of resource thrashing. Finally, thread leakage occurs when a thread finishes processing its work item, but fails to return to the pool. Thread leakage results in fewer threads available to process work items.

A thorough understanding of the behaviour of thread pools is necessary to address these problems. However, understanding is hindered by the explosion of performance data inherent to thread pool systems. For example, the subject system in our case study implements the typical thread pool architecture for servers, described above. There are 320 threads in our case study, each of which contains over 60 micro-threads each. Hence, at the micro-thread level there would be over 20,000 micro-threads during a sixteen hour load test. Five resource usage metrics are collected every seven seconds, amounting, at the micro-thread level, to six millions values collected every hour. Analyzing the evolution of all resource usage metrics for all micro-threads in multiple execution runs is unmanageable. Thus, this paper proposes a scalable methodology for identifying deviations in the performance of threads in a thread pool.

III. Motivational Example

To illustrate how our methodology for understanding the behaviour of thread pools can be applied, consider the following scenario. After years of competition, a large shipping company, Quick Shipping, has decided to buy a smaller rival, Secure Shipping. Secure Shipping had recently developed a proprietary package tracking system that allows their customers to log on from Secure Shipping's website and review the package shipping information. Incoming requests are balanced across four machines and each machine implements the thread pool design pattern to retrieve the requested information from a database.

However, Quick Shipping is a much bigger company and now requires the system to handle five times the previous traffic. To scale the system to meet these new requirements, Quick Shipping must know how the thread pools behave under a five-fold increase in workload. The performance analysts of Quick Shipping's software department monitor the performance of the system and collect resource usage metrics for each thread in each thread pool on each machine. Since the system must be upgraded as soon as possible and very few people are familiar with this proprietary system, the analysts need a fast way to analyze the results and understand the system and hence decide to use our methodology.

Since our methodology is a top-down approach, the analysts first aggregate each machine's thread data into one macro-thread at the machine level, resulting in four abstracted threads. Our ranking scripts then cluster and visualize the four macro-threads to identify relevant commonalities and differences between the four machine's thread pools. Figure 1 shows the results in a dendrogram. Such a graph illustrates the arrangement of clusters produced by our methodology's hierarchical clustering step.

From Figure 1, the analysts immediately notice two groups of behaviour: machine4 and the other three machines. To understand the majority behaviour, the analysts should pick one of machine1, machine2 or machine3 and look at the threads in the thread pool of that machine. To understand deviating thread pool behaviour, possibly indicating a defect or uncommon business scenario, they should focus their attention to machine4. Our methodology identifies and ranks deviating behaviour, however, it is possible that deviating behaviour is actually the expected behaviour. For example, if a system's hard drives are filled during the first hour of a sixteen hour load test, then the deviating behaviour (the behaviour under limited hard drive capacity) will be the majority behaviour.

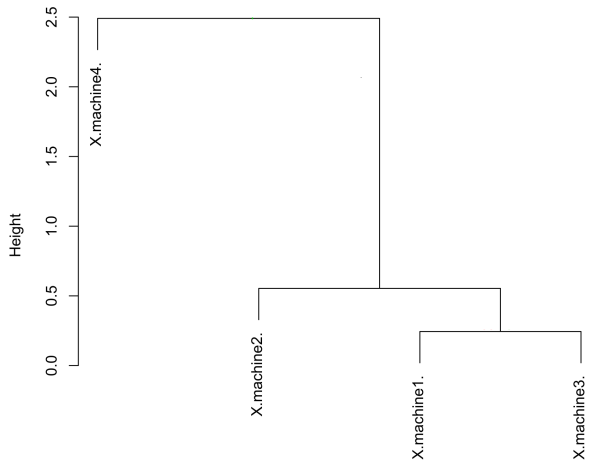

Fig. 1. Hierarchical cluster dendrogram of machine level abstractions.

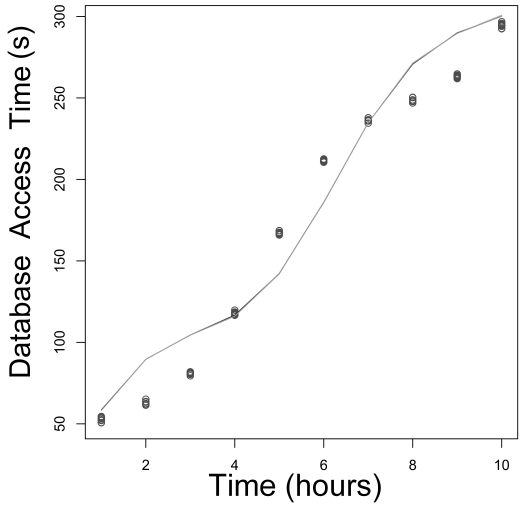

Fig. 2. Database access time plots. The points are actual values for machine4 and the line is the average value of the other three machines' threads.

To find out what exactly causes the deviating behaviour in machine4, the analysts now proceed to perform the same analysis at a lower level, i.e., on the threads of machine4. From the resulting dendrograms, the analysts identify clusters of similar behaviour and select one thread as representative of each cluster. The resource usage metrics of each selected thread is plotted and from the plot of database access time (Figure 2) the analysts quickly determine that competition for access to the database between machine4 and the other machines is affecting the access time. When the database access time of machine4 increases, the database access times of the other machines decrease and vice versa.

Thanks to our iterative, top-down methodology, Quick Shipping is able to quickly understand the system, pinpoint problems, fix them and deploy the system to their customers.

IV. Methodology

We now present in detail our methodology for identifying and ranking performance deviations in thread pools that we illustrated in the previous section. Figure 3 provides a graphical overview of our methodology.

A. Performance Data

Our methodology requires the performance data containing the resource usage metrics of the pooled resources, typically threads, processes or memory buffers. Increasing the number of resource usage metrics allows for a more accurate characterization of the behaviour, but it increases the overhead of performance monitoring and may lead to data redundancy [3], [9]. Typical resource usage metrics include CPU usage and the amount of allocated memory.

B. Metric Abstraction

Our methodology is a top-down approach that determines the level of dissimilarity between different levels of abstracted metrics. This top-down approach allows us to scale our methodology to hundreds or thousands of threads by first identifying dissimilarities at a high-level between few abstractions, before delving into more concrete details.

Performance analysts should group threads into higher-level abstractions by aggregating their resource usage metrics into a single abstracted thread. For example, in a cluster of machines, all pooled threads executing on one machine (or within one data centre) could be aggregated into one macro-abstraction. Our methodology is first applied at this level to detect deviating machines. Afterwards, a deviating machine identified during this iteration could then be more thoroughly examined by repeating our methodology at the level of the pooled threads of the deviating machine, or groups of pooled threads. If threads cannot be grouped by space (machines), they typically can be grouped by time, e.g., all the threads created in slots of one hour.

C. Distance Calculation Between Covariance Matrices

Once abstractions have been defined for a particular system, we can determine the level of dissimilarity between abstractions. The level of dissimilarity, or distance, between two abstractions must take into account the differences in the time-dependent behaviour of the resource usage metrics in each abstraction. This distance must be robust to noise in the performance data. For this reason, we use a statistical approach based on covariance matrices.

The covariance matrix of an abstraction characterizes (1) the variation of each resource usage metric across time and (2) the degree to which two metrics vary together. The covariance matrix for an abstraction is built as follows:

1) The diagonal values, (i, i), contain the statistical variance ${\sigma_i}^2$ of metric i. The variance characterizes the spread of the metric i across time.
2) The off-diagonal values, (i, j), contain the statistical covariance ${\sigma_{ij}}^2$ between each pair of metrics i and j. The covariance characterizes how closely metric i and metric j vary together.

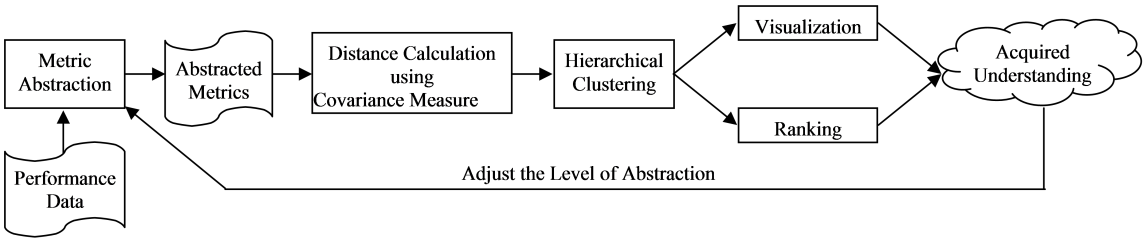

Fig. 3. Methodology overview.

The covariance matrices factor out time, and thus we can compare threads with different execution times. For example, assume that five metrics are measured for threads A and B and thread A is instrumented 100 times, whereas thread B is instrumented 1,000 times. The matrix of resource usage metrics for thread A has dimension 5×100 and the matrix for thread B has dimension $5\times1{,}000$, therefore, element-by-element comparison is not possible without defining a mapping scheme. However, the covariance matrix for both threads A and B has dimension 5×5. Therefore, we can compare load tests of differing lengths.

To determine the distance between the covariance matrices of the resource usage metrics for two abstractions, we use a distance metric proposed by Forstner and Moonen [15]. This metric for covariance matrices has been used successfully in the field of computer vision and image analysis to characterize and compare images [16]–[19]. Forstner and Moonen's distance metric generates a one-dimensional distance for each pair of covariance matrices. The smaller the distance value between two abstractions, the more similar they are. The distance is based on the geodesic distance between the covariance matrices; for more technical details we refer to [15].

D. Hierarchical Clustering

Once the distance between each pair of abstractions has been determined, the abstractions can be clustered to identify and rank similar and deviating behaviour. We use an agglomerative, hierarchical clustering procedure. This procedure starts with each abstraction in its own cluster and proceeds to find and merge the closest pair of clusters (in terms of the distance), until only one cluster (containing everything) is left.

Hierarchical clustering decides which clusters to merge based on a distance metric between clusters of abstractions. As the algorithm for merging clusters, we use Ward's method of clustering. This treats determining which clusters to merge as an analysis of variance problem based on minimizing the sum of squares of the distance between two clusters' centroids if the two clusters are merged [20], [21]. Since the distance between abstractions is a one-dimensional and continuous feature, we use the Euclidean distance as the distance between clusters. The Euclidian distance is the most popular distance metric for continuous features [20].

The likelihood that a cluster in the data is a statistically-valid cluster is assessed by a p-value ($0\leq p\leq100$) [21]. We use the approximately unbiased (au) p-value, which is computed using multiscale bootstrap resampling. In this paper, the au p-value appears to the left of each merge node in the dendrogram [22].

E. Cluster Visualization

Clusters of abstractions are visualized using hierarchical cluster dendrograms. These are binary tree-like diagrams that show each stage of the clustering procedure as nested clusters [20]. The distance between clusters determined in the previous step becomes the height value on the dendrogram, as shown in Figure 1.

F. Ranking Clusters

Once the hierarchical cluster dendrograms have been generated, they are analyzed to identify and rank the most deviating behaviour. Our ranking algorithm was formalized from our experiences analyzing dendrograms. We have attempted to maximize the precision of our ranking algorithm to encourage tool adoption. Our ranking algorithm requires a recursive decent into the dendrogram using the following algorithm:

```
1  ranking=[]
2  rank(clust){
3     /* always consider clusters with one member as deviating, limit the
           number of deviating clusters (with more than one member) to
           approximately 25% of the dendrogram clusters  */
4     if(|clust|==1
           or |clust|<(num_clusters/4-size(ranking))){
5       ranking=[ranking,clust]
6     return
7     }
8
9     s=clust.left    // smaller cluster
10    l=clust.right   // larger cluster
11
12    if(|clust.left|>|clust.right|){
13      s=clust.right
14      l=clust.left
15    }
16
17    if((height(l)-height(s)>=height(clust)/4
           and height(l)-height(s)>=max_height/10)
           or height(clust)>max_height/3
           or (p(clust)<80 and (height(s)>max_height/4
           or height(l)>max_height/4))){
18      rank(s)
19      rank(l)
20    }
21    return
22 }
```

The conditions in line 17 were developed to ensure our ranking methodology continues to recurse when one of the following conditions is true:

- The height difference between the small and large clusters is relatively large
- The height of the cluster is relatively large
- The cluster is not a good fit ($p < 80$) and the height of the small or large cluster is relatively large

Using Figure 1 as an example and assuming all p-values are greater than 80, we have:

1) Starting at the highest level, machine4 vs. machine1/2/3, the first condition is true (line 17), because the difference between the height of machine4 and machine1/2/3 is 1.9, which is greater than 25% of the cluster's height (2.5×25%=0.63) and 10% of the maximum height (2.5 ×10%=0.25). Hence, we recurse into machine4 (line 18)
2) We add machine4 as a deviation (line 5) because it contains only one member. We now recurse into the other cluster, machine1/2/3 (line 18)
3) We stop the algorithm, because none of the conditions are true (line 17)
   - The difference between the height of machine2 and machine1,3 is 0.3, which is greater than the 25% of the cluster's height (0.55×25%=0.14) but less than 10% of the maximum height (2.5×10%=0.25).
   - The height of the smaller cluster, machine2, 0.55 is then 25% of the maximum height (2.5×10%=0.25).
   - The height of the machine1/2/3 node is less than 33% of the maximum height (2.5×33%=0.83)
   - The p-value is greater than 80.
4) Only the machine4 cluster has been ranked as a deviation.

V. Case Study Setup

To validate our methodology for identifying performance deviations of threads in a thread pool, we perform a study on a large-scale industrial software system. In particular, we first qualitatively evaluate our methodology by manually comparing the resource usage metrics of threads identified as deviating to those of other threads. Second, we quantitatively evaluate our methodology by calculating precision and recall values when applying our methodology to a load test with synthetically injected deviations, a common practice for verifying such work. This section discusses our case study set-up and how we evaluate the results of our methodology.

A. Subject System

Our case study uses the performance data of an industrial ultra-large-scale system in the e-commerce domain. For confidentiality reasons, we cannot disclose the specific details of the system architecture, however the system follows a typical thread pool architecture. Although the threads of the thread pool manage their own micro-threads, we ignore this lower level of micro-threads because these metrics are unavailable.

The performance data used in our case study was generated during load tests performed on the system. The test cases of these load tests are representative of typical usage scenarios. Performance data for a sixteen hour load test was available.

B. Performance Data

The performance data used in this case study consists of five resource usage metrics collected for each active thread and shown in Table I. The metrics are sampled approximately every 7 seconds. Every instrumentation point has a tag for the thread ID, a tag for the time of instrumentation and a value for each resource usage metric.

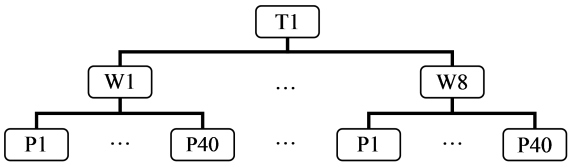

Fig. 4. Top-down approach for analyzing load tests. In our particular case study, abstractions are either a load test (T), a wave (W) or a thread (P).

TABLE I
Resource Usage Metrics Collected For Each Thread

CPU	Percentage of CPU time in use
VirtualBytes	Amount of virtual address space in use
PrivateBytes	Amount of private (non-shared) memory in use
Handles	Number of open file handles
MicroThreads	Number of allocated micro-threads

C. Metric Abstraction

When determining a suitable abstraction of threads, we observe that the load test has a unique internal pattern of load test generation to simulate workload spikes. Approximately every two hours the system is "spiked" with a large number of work items. The thread pool contains forty threads and each thread is assigned to process one work item. The remaining work items remained queued until threads become available. After approximately two hours, the threads complete processing the work items in the queue and the system is "spiked" again. With the exception of the threads cut short at the end of the load test, the lifetimes of the threads are roughly identical. We call a group of threads that are assigned work items at a similar time a "wave." Waves allow for a convenient method of dividing the data set into more manageable subsets based on the start time of threads. Figure 4 shows how our methodology is applied top-down on the load test. We label waves with W and the wave number, the first wave to be created in a load test is W1. Each wave lasts for approximately 2 hours and 5 minutes; the sixteen hour load test has 8 waves.

D. Covariance Matrices

We implemented Forstner and Moonen's metric for covariance matrices as specified in the original paper using R, a software environment for statistical computing [15], [23]. Our implementation takes as input the annotated resource usage file, constructs the covariance matrix for each abstraction, then calculates the distance between the covariance matrices of each pair of abstractions. The output from this stage is a distance matrix with the distances between each pair of abstractions.

E. Hierarchical Clustering

We use pvclust, an R package for hierarchical clustering, to cluster the abstractions [22], [24]. Pvclust uses an agglomerative, hierarchical clustering algorithm using a specified distance metric and clustering method.

The input to this stage is the distance matrix calculated in the previous step. The output is a dendrogram, like the one in Figure 1, which contains the height of each abstraction and the au p-value to the left of each node.

F. Ranking Clusters

The ranking of clusters is based on the dendrogram produced in the previous step. We use the ranking algorithm described in Section IV.F.

G. Qualitative Evaluation

Our first evaluation analyzes the resource usage metric plots of the waves and threads of the sixteen hour load test, T1, to evaluate whether the threads that have been flagged as deviating would also be recognized by humans. We generate these plots by:

1) Selecting the abstractions to visualize based on our methodology's ranking.
2) Normalizing the time scale so that the first measurement of each abstraction shifts to time zero. This is necessary, because abstractions do not have the same start time.
3) Undersampling the metrics by averaging every two minutes to remove single-point anomalies and noise.
4) Plotting the undersampled metrics.
5) Plotting additional details, such as the average and standard deviation, as needed.

The first author of this paper analyzed all flagged abstractions (waves or threads) of the load test. If a flagged abstraction's plots looked similar to those of other abstractions, we count this as a false positive. Similarly, if the plots of an abstraction that was not flagged does not look similar to those of other abstractions, we count this as a false negative.

H. Quantitative Evaluation

Our second evaluation validates our methodology's ability to identify and rank deviations by synthetically injecting typical symptoms of performance deviations into the performance data. Based on previous work and discussions with experts of the subject system, we identified three important types of deviations, CPU spikes, memory leaks and IO leaks [2], [25]. We inject these deviations using the following methodology:

1) We manually examined the resource usage metric plots of a sixteen hour load test and identified a group of twelve threads that exhibit very similar behaviour. We then collected these threads into a pool of "good" threads.
2) We created a pool of twelve "bad" threads by taking each thread from the "good" pool and injecting one of the following deviations into them at a random time:
   - CPU spike - the value of the CPU ramps up during a random time period (1-3 minutes) to a randomly selected maximum (50%-150% CPU usage), then ramps back down;
   - Memory leak - from a random start time to the end of the thread's lifetime the amount of allocated virtual memory ramps up to a randomly selected maximum (between 150%-300% of the previous maximum);
   - IO leak - from a random start time to the end of the thread's lifetime the number of open file handles ramps up to a randomly selected maximum (between 125%-200% of the previous maximum). An IO leak simulates a thread failing to close files it has opened.
3) We then generated the validation performance data by composing threads from the "good" and "bad" thread pools in a structure similar to Figure 4. We created tests with 8 waves and 10 threads per wave. To inject a deviation into a wave we randomly select a thread from the "bad" pool that contains a random type of deviation (e.g., a wave with one injected deviation would have nine threads from the "good" pool and one thread from the "bad" pool). The number and pattern of deviations injected depends upon the focus of the validation:
   - To verify our methodology's ability to identify and rank waves with performance deviations, we randomly injected 2-4 deviations into 0-3 waves. The lower bound of 0 waves prevents us from relying on the fact that there will be at least one deviating wave in the test. The upper bound of 3 waves enforces our assumption that the majority behaviour is the expected behaviour;
   - To verify our methodology's ability to identify and rank threads (inside waves) with performance deviations, we generated performance data by randomly injecting 0-3 deviations into each wave. We chose these bounds for the same reason as outlined above.
4) We then applied our methodology and evaluated our ranking using the precision, recall and k-recall metrics.
5) We repeated Steps 3-4 ten times and calculated the average performance across all ten tests.

The precision and recall metrics that we use depend upon the focus of the validation. To verify our methodology at the wave level, we use the following metrics:

$$
\begin{aligned}
precision_w &= \frac{\#\ correctly\ ranked\ waves}{\#\ ranked\ waves} \\
recall_w &= \frac{\#\ correctly\ ranked\ waves}{\#\ waves\ with\ deviations} \\
k-recall_w &= \frac{\#\ deviating\ threads\ in\ top\ k\ ranked\ waves}{\#\ deviating\ threads\ in\ K}
\end{aligned}
$$

where *K* is the number of deviations in the top *k* waves with the highest number of deviating threads.

To verify our methodology at the thread level, we use the following metrics:

$$
\begin{aligned}
precision_t &= \frac{\#\ correctly\ ranked\ threads}{\#\ ranked\ threads} \\
recal_tl &= \frac{\#\ correctly\ ranked\ threads}{\#\ threads\ with\ deviations} \\
k-precision_t &= \frac{\#\ deviating\ threads\ in\ top\ k\ ranked\ threads}{k}
\end{aligned}
$$

To quantify how well our methodology is able to identify deviations, we use precision and recall metrics and to quantify how well our methodology is able to rank deviations we use the k-precision and k-recall metrics. The closer to 1 these metrics are, the better.

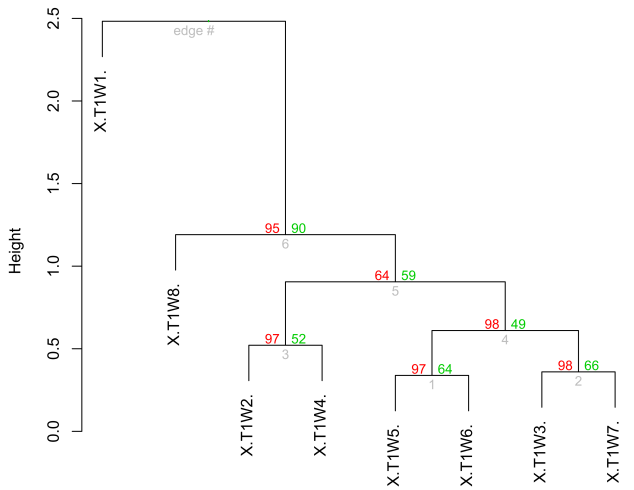

Fig. 5. Dendrogram of the waves of T1. T1W1 is the first wave of this load test and T1W8 is the last wave.

Fig. 6. Plot of resource metric VirtualBytes. T1W1 is plotted in a solid line, T1W8 in a dotted line and T1W5 in a dashed line.

VI. Case Study Results

A. Qualitative Evaluation

Using the approach described in Section V.G, we check that the abstractions (waves and threads) identified and ranked as deviating (or common) actually exhibit deviating (or common) behaviour in their resource usage metrics.

Figure 5 shows the dendrogram at the wave level. From Figure 5, we can identify and rank the most deviating waves as follows:

- T1W1
- T1W8
- T1W2/3/4/5/6/7

We visualize the resource metrics of T1 to verify if T1W1 and T1W8 indeed display deviating behaviour compared to the other waves. The deviations in W1 and W8 are likely caused by ramp-up and ramp-down in the system. Figure 6 shows one such resource metric visualization for the amount of virtual memory allocated. We can see that T1W1 (solid line) deviates the most because of a large spike at the beginning and because the values are higher than T1W5 (dashed line), which was selected to represent the W2/3/4/5/6/7. T1W8 (dotted line) is deviating because its values rise slower than the average and the lifetime of T1W8 is shorter than that of the other waves (load test was cut short). We find the same patterns when examining the resource usage metric plots for amount of allocated private memory and number of open file handles. For confidentiality reasons, we have replaced the actual values by the percentage of the maximum observed value.

The dendrogram in Figure 5 can be used to guide further analysis in two ways:

- We can recommend that the most deviating behaviour be thoroughly examined. In this case, analysts should focus on T1W1, then T1W8.
- Alternatively, we can recommend that the largest clusters be examined because it represents the common behaviour.

We explored T1W1, T1W8 and the common behaviour, but due to space constraints we only discuss T1W1 in detail.

We explore the T1W1 wave and the resulting dendrogram in Figure 7. From Figure 7, we can identify and rank the threads with the most deviating behaviour to be clusters A and B.

To verify if clusters A and B display deviating behaviour, we visualize the resource metrics of representative threads of cluster A, cluster B and the remaining threads of wave T1W1. We analyzed the resource metric visualizations for each resource, but due to space constraints we only show one such resource metric visualization, i.e., the number of micro-threads allocated to each thread.

From Figure 8, we can see that thread 12452 (dashed line), which was selected as the representative thread of cluster B, is deviating the most because of a large spike at the beginning, high variability in the values throughout the load test, with a drop in values towards the end. For similar reasons, thread 7740 (solid line), which has been chosen as a representative of cluster A, deviates from the common behaviour, because it has a large spike at the beginning, high variability in the values (although not as much as thread 12452) and a small drop in values towards the end.

The qualitative evaluation of our methodology has shown that our methodology is able to identify deviating resource usage trends, such as faster/slower growth and higher volatility, at the level of waves and threads.

B. Quantitative Evaluation

Using the approach described in Section V.H, we injected CPU spikes, memory leaks and IO leaks into a pool of "good" threads to quantitatively evaluate the precision and recall of our technique at the wave and thread levels.

We perform this evaluation on ten synthesized tests of eight waves and present the evaluation metrics for each test in Table II. Our methodology has very high precision, recall,

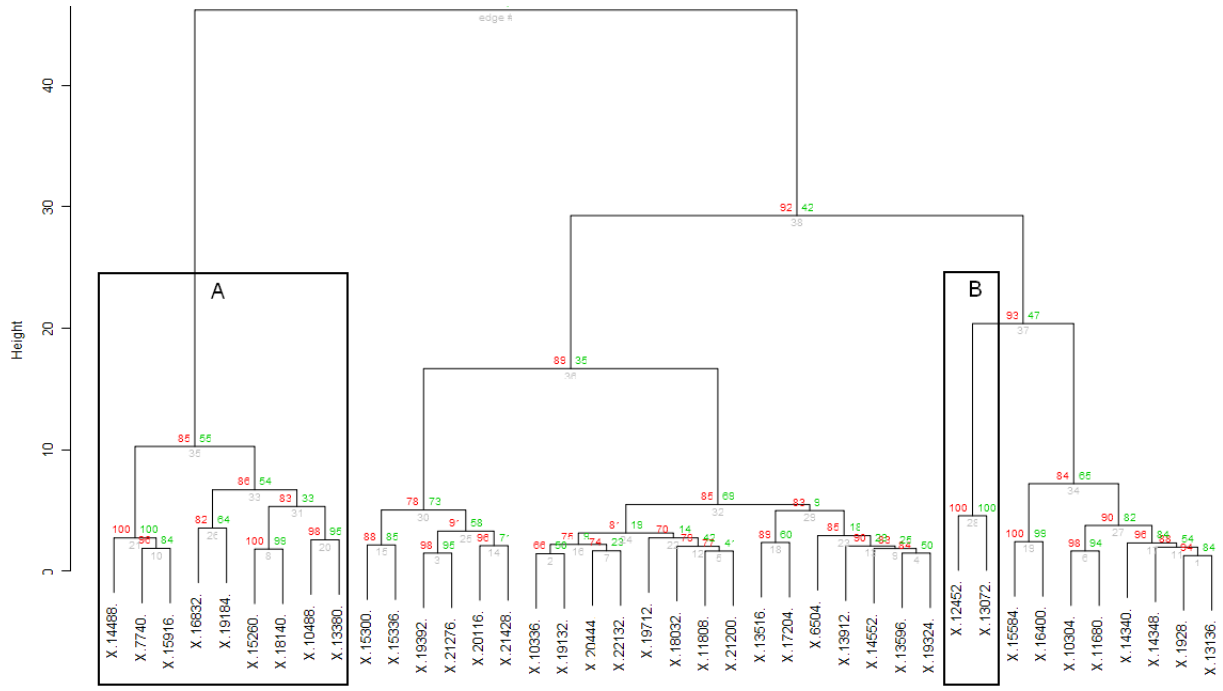

Fig. 7. Dendrogram of T1W1. This is the lowest level in our top-down approach.

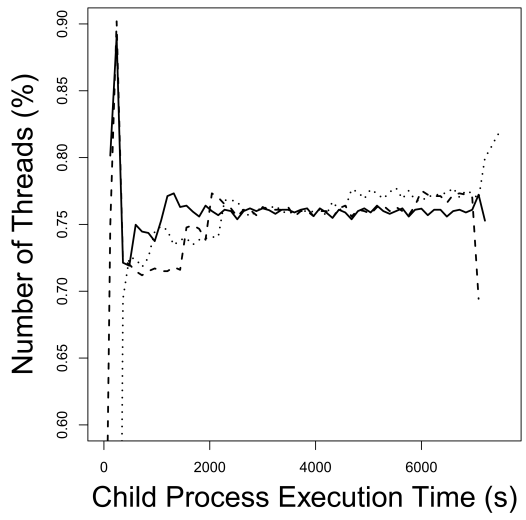

Fig. 8. Plot of resource metric Threads. Thread 7740 (cluster A) is plotted in a solid line, thread 12452 (cluster B) in a dashed line and thread 22132 (remaining threads) in a dotted line.

TABLE II
WAVE LEVEL RANKING

Test	Precision	Recall	1-recall	2-recall	3-recall
1	100%	100%	50%	100%	100%
2	100%	100%	100%	100%	100%
3	100%	100%	66.67%	83.33%	100%
4	100%	100%	100%	100%	100%
5	100%	100%	100%	100%	100%
6	100%	100%	100%	100%	100%
7	100%	100%	50%	100%	100%
8	100%	100%	100%	100%	100%
9	100%	100%	100%	83.33%	100%
10	100%	100%	85.71%	96.55%	100%
Average	100%	100%	86.67%	96.67%	100%

2-recall and 3-recall values. Most of the times, our methodology produces high 1-recall values, however, there are some outlying low 1-recall values.

We explored the cause of these low 1-recall values and determined that ties between two clustered waves is the cause. This issue can be clearly seen in Figure 9. Currently there is no method of recommending Wave7 over Wave8 given Figure 9 and our ranking algorithm (lines 4-5 in our pseudo-code in Section IV.F). A possible approach to address this issue might be to compare the average values of each resource usage metric of the tied waves to the remainder of the waves. For example, in Figure 9 we would compare the average value of each resource usage metric in Wave7 and Wave8 to the average value of each resource usage metric in Wave1-Wave6. Therefore, we could recommend Wave7 over Wave8 based on which wave has the greater difference between the average values of its resource usage metrics and the average values of the resource usage metrics of Wave1-Wave6.

Table III summarizes the results of our evaluation performed at the thread level. We perform this evaluation on ten synthesized tests of eight waves containing ten threads each and calculate the average precision and recall values over all eighty waves. Table III also breaks down the average precision and recall values across the different types of injected deviations.

Our methodology was able to identify 100% of the memory and IO leaks, but none of the CPU spikes, leading to an

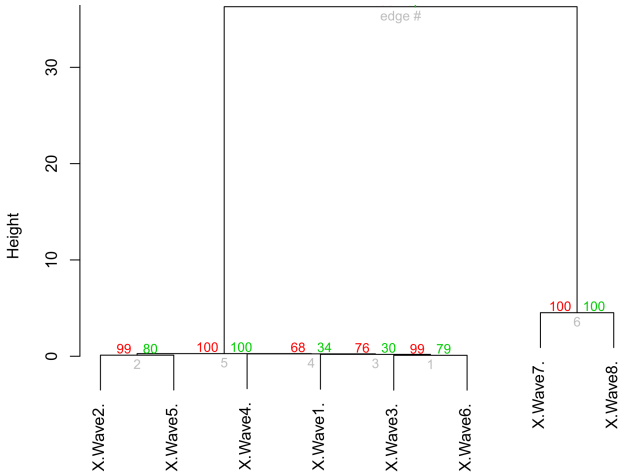

Fig. 9. Problem of tie breaking during ranking.

average recall of 76.61%. Similarly, the k-precision values are impacted by the inability to detect CPU spikes. We believe that this is because CPU spikes manifest themselves in a relatively small portion of the data set and, therefore, have much less impact on the values of the covariance matrix. We intend to address this issue by applying our methodology using a small moving window over the data set and aggregating the rankings produced in each window. Therefore, within these smaller windows, CPU spikes would manifest themselves in a larger proportion of the data set.

TABLE III
AVERAGE THREAD LEVEL RANKING ACROSS ALL TESTS

Average Precision	100%
Precision (CPU Spikes)	100%
Precision (Memory Leaks)	100%
Precision (IO Leaks)	100%
Average Recall	76.61%
Recall (CPU Spikes)	0%
Recall (Memory Leaks)	100%
Recall (IO Leaks)	100%
1-precision	91.52%
2-precision	87.13%
3-precision	76.61%

VII. THREATS TO VALIDITY

The proposed methodology was evaluated against a single industrial software system. Our results may not generalize to other industrial or open-source software systems. We intend to address this by evaluating our methodology against additional industrial and open-source software systems. Yet, access to high-quality performance data is not straightforward.

The proposed methodology was evaluated using a fixed selection of resource usage metrics. Our results may not generalize to other performance metrics. This paper does not explore the limit on the number of resource usage metrics required to perform our analysis.

Our methodology was evaluated against a small subset of performance deviation symptoms (i.e., CPU spikes, memory leaks and IO leaks) using a small pool of "good" and "bad" threads. Our results may not generalize to other performance deviation symptoms, such as disk IO spikes or declining network throughput. In addition, our methodology was evaluated using a small, fixed pool of "good" and "bad" threads.

VIII. RELATED WORK

This paper presents a methodology for understanding the resource usage of the threads in a thread pool. Although our methodology is a kind of dynamic program analysis, most of the existing work on dynamic analysis focuses on the functional behaviour of a system instead of on its performance, except for some work on visualizing threads [26], [27]. An excellent survey can be found in [28]. In practice, the closest area of research to our work is the area of load test analysis.

Most of the work in the area of load testing has focused on the automatic generation of load test cases [29]–[31]. Recently, load test researchers have noted the difficulties of detecting performance problems in load tests and have proposed the use of execution logs of load tests to automatically identify functional and performance problems [2], [5]. Jiang et al. mine these logs to determine the dominant (expected) behaviour of the system and to flag anomalies from the dominant behaviour. Their technique is able to flag $<0.01\%$ of the execution log lines for closer analysis [30]. Our paper uses only the resource usage metrics and not the execution logs, to determine which threads have deviating behaviour.

Other work in load test analysis has focused on automatically identifying the most important resource usage metrics and comparing those across multiple load tests to identify changes to a system's performance [3], [9]. Using their technique, Malik et al. are able to reduce the size of the performance data by 88%. Our methodology uses all of the resource usage metrics, but applies a top-down, lighter-weight technique (covariance matrices) than the variable reduction technique used by Malik et al.

Other work in automated performance monitoring has developed system signatures based on resource usage metrics that can be used to detect changes to the performance of a system as it evolves over time [32], [33]. These techniques require a baseline model of the system's performance to characterize changes resulting from software evolution. Our methodology is able to identify and rank deviating behaviour without the use of a baseline.

IX. CONCLUSIONS

This paper has presented a methodology for automatically identifying deviating behaviour in ULS systems. In particular, we focused on identifying and ranking the most deviating thread behaviour of a thread-pool based system using resource usage metrics. Our methodology is an iterative, top-down process that identifies deviating behaviour based on the level of dissimilarity between threads or groups of threads, then ranks the most dissimilar threads or thread groups. Qualitative and

quantitative evaluation of our methodology on an industrial software system shows that we are able to accurately identify and rank the most deviating thread behaviour with high precision and recall. We are also able to identify the effects of ramp-up and ramp-down that company experts have also acknowledged.

Our methodology performs well when ranking long-lived deviations, such as memory leaks. However, to address our methodology's inability to detect short-lived deviations (such as CPU spikes), we are working on adapting our methodology to a small, moving window across the data set. We believe that considering smaller portions of the data set will allow identification of these short-lived deviations. We are also interested in applying our methodology to performance problems outside thread pools.

ACKNOWLEDGEMENT

We are grateful to SAP for providing the load test data for 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 SAP and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of SAP software products.

We would like to thank Dr. Hongyu Zhang and Dr. Yasutaka Kamei for their comments and implementation of Forstner and Moonen's metric for covariance matrices. We would like to thank Mr. Brent Cromarty for his assistance and discussions on our case study.

REFERENCES

[1] 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.

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

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

[4] W. Visser. Willem visser's research. [Online]. Available: http://www.visserhome.com/willem

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

[6] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the International Conference on Software Engineering (ICSE)*, Jun. 2000, pp. 449–458.

[7] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, Jun. 2006, pp. 181–190.

[8] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj, "An approach for detecting execution phases of a system for the purpose of program comprehension," in *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA)*, May 2010, pp. 207–214.

[9] 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 (CSMR)*, Mar. 2010, pp. 222–231.

[10] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler, "A design framework for highly concurrent systems," Comput. Sci. Division UC Berkeley, Berkeley, CA, Tech. Rep. UCB/CSD-00-1108.

[11] D. Lea, *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1999.

[12] P. Hyde, *Java Thread Programming*. Sams, 1999.

[13] P. Donald, S. Singh, and S. Ghosh, "Whirlwind: Overload protection, fault-tolerance and self-tuning in an internet services platform," in *Proceedings of the Malaysia International Conference on Communications (MICC)*, Dec. 2009, pp. 397 –402.

[14] P. J. Denning, "Thrashing, its causes and prevention," in *Proceedings of the Fall Joint Computer Conference*, 1968, pp. 915–922.

[15] W. Forstner and B. Moonen, "A metric for covariance matrices."

[16] Y. Pang, Y. Yuan, and X. Li, "Gabor-based region covariance matrices for face recognition," *Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 7, pp. 989 –993, Jul. 2008.

[17] M. Sharif, N. Ihaddadene, and C. Djeraba, "Crowd behaviour monitoring on the escalator exits," in *Proceedings of the International Conference on Computer and Information Technology (ICCIT)*, 2008, pp. 194 –200.

[18] O. Tuzel, F. Porikli, and P. Meer, "Region covariance: A fast descriptor for detection and classification," in *Proceedings of the European Conference on Computer Vision (ECCV)*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 3952, pp. 589–600.

[19] M. Donoser and H. Bischof, "Using covariance matrices for unsupervised texture segmentation," in *Proceedings of the International Conference on Pattern Recognition (ICPR)*, Dec. 2008, pp. 1–4.

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

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

[22] pvclust: An R package for hierarchical clustering with p-values. [Online]. Available: http://www.is.titech.ac.jp/~shimo/prog/pvclust

[23] The r project of statistical computing. [Online]. Available: http://www.r-project.org

[24] R. Suzuki and H. Shimodaira, "Pvclust: an R package for assessing the uncertainty in hierarchical clustering," *Bioinformatics*, vol. 22, pp. 1540–1542, Jun. 2006.

[25] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 1994, pp. 191–200.

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

[27] S. P. Reiss, "Controlled dynamic performance analysis," in *Proceedings of the International Workshop on Software and Performance (WOSP)*, Jun. 2008, pp. 43–54.

[28] 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.

[29] A. Avritzer and E. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *Transactions on Software Engineering*, vol. 21, no. 9, pp. 705–716, Sep. 1995.

[30] A. Avritzer and E. J. Weyuker, "Generating test suites for software load testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '94, 1994, pp. 44–57.

[31] J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Software: Practices and Experiences*, vol. 32, pp. 1411–1435, Dec. 2002.

[32] 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 (DSN)*, 2008, pp. 452–461.

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