Using Load Tests To Automatically Compare the Subsystems of a Large Enterprise System

Haroon Malik, Bram Adams, Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Canada
{malik, bram, ahmed}@cs.queensu.ca

Parminder Flora and Gilbert Hamann
Performance Engineering
Research In Motion (RIM)
Waterloo, ON, Canada

***Abstract*—Enterprise systems are load tested for every added feature, software updates and periodic maintenance to ensure that the performance demands on system quality, availability and responsiveness are met. In current practice, performance analysts manually analyze load test data to identify the components that are responsible for performance deviations. This process is time consuming and error prone due to the large volume of performance counter data collected during monitoring, the limited operational knowledge of analyst about all the subsystem involved and their complex interactions and the unavailability of up-to-date documentation in the rapidly evolving enterprise. In this paper, we present an automated approach based on a robust statistical technique, Principal Component Analysis (PCA) to identify subsystems that show performance deviations in load tests. A case study on load test data of a large enterprise application shows that our approach do not require any instrumentation or domain knowledge to operate, scales well to large industrial system, generate few false positives (89% average precision) and detects performance deviations among subsystems in limited time.**

Keywords-Load test; Signatures; PCA; performance

I. INTRODUCTION

Large scale systems (LSS), such as Google, Facebook, Amazon and eBay are complex systems composed of many underlying components. These systems grow rapidly in size to handle growing traffic, complex services, and business-critical functionality. This exponential growth increases the individual component's complexity and hence, the integration between the geographically distributed components. The performance of LSS is periodically measured to satisfy the high business demands on system quality, availability and responsiveness.

Load testing remains the most integral part of testing the performance of the Large Scale Systems (LSS). Load testing uncovers residual functional and performance problems that slipped through the conventional functional testing, such as unit and integration testing. A functional problem results in processing happening at the wrong place in the wrong order [4]. A performance problem results in processing taking too much or too little of important resource. A request that takes too long may indicate a bottleneck, while a request that finishes too quickly may indicate truncated processing or some other performance bug.

Load testing assesses how a system performs under a given load [1]. Load is defined as the rate at which transactions are submitted to a system [2]. Load generators are used to induce load on the system under test [3], i.e., imitating thousands of users committing concurrent transactions to a system, resembling the harsh field environment. During the course of a load test, the system is strictly monitored and important sources of data exposed by the system i.e., performance metrics are collected. These metrics include numerical measurements related to the system's state and performance (e.g., CPU, memory utilization, network usage etc). These are called performance counters. After the completion of the load test, performance analyst select the important performance counters based on their domain knowledge and experience gained from previous tests and compare them against the base-line test [3]. They use various plotting tools manually, to compare performance counters against a base-line test. If a performance analyst finds large deviations from base-line load test, a defect/bug report is filed. The defect/bug report is assigned to software engineers and experts in the corresponding subsystem area, i.e., database, web servers, mail, application, network etc to further investigate and rectify the performance issue/bug.

Unfortunately, the current practice to analyze load test is costly, time consuming and error prone [5][3]. The load test analysis practices have not kept pace with the rapid growth in size and complexity of the large enterprise systems. In practice, the dominant tools and techniques to analyze large distributed systems have remained unchanged for over twenty years [4]. Performance analysts have to face the challenge of isolating the required performance information distributed across thousands of correlated performance counters along multiple subsystems in limited time. Furthermore, they have to leverage extensive knowledge about such large scale systems to identify subsystems responsible for performance deviations in load tests, as the formal performance baseline in large and dynamic enterprise system rarely exist [3] . The paper presents an automated and effective methodology to help performance analysts to pinpoint performance deviations among subsystems in LSS and to facilitate performance analysts to analyze load test.

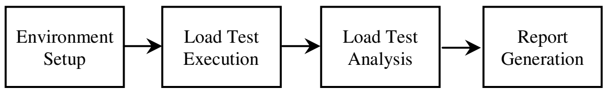

Figure 1. Performance load testing process

In particular, our methodology helps them to finger-point the subsystems deviating from their normal performance behavior (i.e., baseline) in limited time. We make the following contributions:

C1. We apply statistical methods to reduce the dimensionality of the observed performance counter set of every subsystem in ULSS.

C2. We show a nontrivial way to generate a performance signature for each subsystem in LSS by automating the ranking of performance counters according to their importance for load test. Furthermore, we show how simple pair-wise correlation is used to compute the divergence between the performance signatures of two subsystems, eliminating the need of employing sophisticated mathematical models and domain intensive knowledge.

C3. We empirically validate our proposed approach through a large case study on a real-world industrial software system.

The rest of the paper is organized as follows: Section II presents the current load test practice and its limitations. We present our methodology in section III. The section IV present the performance measures of our methodology followed by a case study in section V. the section VI presents the related work. Finally, section VII presents conclusion and future work.

II. CURRENT PRACTICE

The typical process of load testing involves four phases, as shown in Figure 1 :

1. ***Environment setup*** is the most important phase of load testing. Most common load test failures occur due to improper environment setup for a load test. The environment setup includes installing the application and the load testing tools possibly on different operating platforms. Load generators, which emulate user's interaction with the system, need to be carefully configured to match the real workload in field.
2. ***Load test execution*** involves starting the components of the systems under load test, i.e., starting the required services, hardware resources and tools (load generators and performance monitors). During the execution of a load test, the application/system under load is strictly monitored and performance counters are recorded in performance logs.
3. ***Load test analysis*** involves comparing the results of a load test against another load test's result or against pre-defined thresholds as baselines. Unlike functional and unit testing, which result in a pass or failure classification for each test; load testing requires additional quantitative metrics like response time, throughput and hardware resource utilizations to summarize results. The performance analyst selects few of the important performance counters among the thousands collected. Based on his experience and the domain knowledge, performance analyst manually compares the selected performance counters with those of past runs to look for evidence of performance deviation, for example using plots and correlation tests.
4. ***Report generation*** includes filing the performance deviations, if found, based on the personal judgment of the analyst. In most cases the results filed in a performance report are verified by an experienced analyst. Based on the extent of performance deviation and its relevance to a team responsible for handling the subsystems i.e., (database, application, web system etc.).

Many challenges and limitations associated with the current practice of load test analysis remain unsolved:

1. ***Large number of performance counters***: Load tests last from a couple of hours to several days. They generate performance logs that can be of several terabytes in size. Even logging all counters on a typical machine at 1Hz generates about 86.4 million values in a single week, A cluster of 12 machines over one week would generate 13 TB of performance counter data per week, assuming a 64 bit representation for each counter value [20]. Analysis of such large counter logs is still a big challenge in load tests.
2. ***Limited time:*** Performance analysts in LSS have only limited time to reach and complete diagnostics on performance counter logs and to make necessary configuration changes. Load testing is usually the last step in an already tight and usually delayed release schedule. Hence, managers are always eager to reduce the time allocated for performance testing.
3. ***The risk of error:*** Load test analysis is error-prone because of the manual process involved in analyzing performance counter data in current practice. In practice, it is impossible for analysts to skim through the huge volume of performance counters to find the required information in LSS. Instead, analysts use few key performance counters known to them from past practices, performance experts and domain trends as 'rules of thumb' [22]. There is no single person with complete knowledge of end to end geographically distributed system activities in LSS [21]. An analyst with good knowledge of the database server will quickly uncover important database counters from performance counter logs however; he may overlook some important web counters. Applying same the 'rules of thumb' on load tests can mislead performance issues [22].

Due to the above challenges, we believe that the current practice to perform load test analysis is neither efficient nor sufficient to uncover performance deviation accurately and in limited time. Our methodology automatically identifies the important counters across every subsystem of an LSA to create robust performance signatures. By mean of simple correlations between performance signatures, our methodology can pinpoint performance deviations among LSS subsystems from a baseline.

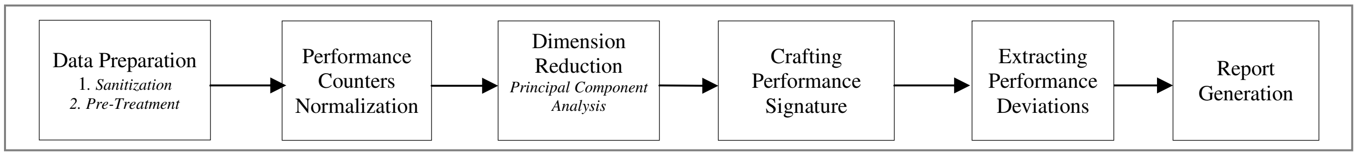
Data Preparation
1. Sanitization
2. Pre-Treatment
Performance Counters Normalization
Dimension Reduction
Principal Component Analysis
Crafting Performance Signature
Extracting Performance Deviations
Report Generation

Figure2: The steps involved in proposed methodology

TABLE I. SAMPLE OF OBSERVATIONS BEFORE DATA PREPARATION

	Observations						
Var	Tot	Mis	Avail	Mini	Max	Mean	Std. Dev
Q	599	0	599	246.18	1946.11	754.654	292.00
R	599	0	599	009.59	0063.46	023.427	011.14
S	0	0	0	000.00	0000.00	000.000	000.00
T	599	2	597	001.00	0117.11	0030.90	018.99

TABLE II. SAMPLE OF OBSERVATIONS AFTER DATA PREPARATION

	Observations						
Var	Tot	Mis	Avail	Mini	Max	Mean	Std. Dev
Q	599	0	597	-13.37	000.07	0.00	1.00
R	599	0	597	-00.71	006.52	0.00	1.00
T	597	0	597	-1.694	001.46	0.00	1.00

III. THE METHODOLOGY

In this section, we present and discuss our methodology to help analysts in load test analysis. To address the limitations listed in section III, our approach eliminates the need to invest the domain extensive knowledge by identifying important performance counters for each subsystem in a load test. Apart from previous effort to automatically compare load tests [14], the methodology is refined further to help an analyst to pinpoint performance deviations in LSS at subsystem level in limited time. Figure2 shows the steps required in our methodology.

A. Data Preparation

The performance logs obtained from a load test do not suffice for direct analysis by our methodology. The logs need to be prepared to make them suitable for the statistical techniques employed by our methodology. The two steps involved in data preparation are:

1) Data sanitization:

Performance logs need to be filtered from noise i.e., missing counter data or empty counter variables. Counter data is missing when performance monitors fail to record an instance of a performance counter variable. A counter variable is completely empty when a resource cannot start the service. Table 1 shows a sample of our real world performance counters for a load test before the data preparation step is applied. Counter variable 'T' belongs to the missing counter data category whereas 'S' is an empty counter variable. A Total of 599 observations were required for each performance counter variable. Monitoring tool recorded only 597 observations for performance counter 'T'. To deal with this kind of problem (incomplete data) we employed *list wise deletion*. If the i^{th} observation for counter 'T' is missing, list wise deletion will delete the corresponding i^{th} observation of all the counter variables. Partial empty counter variables such as 'S' and counter variables that have more than 2% of the data missing are removed during sanitization process. TABLE II. shows the performance counters after data sanitization.

2) Pre-treatment:

Pre-treatment converts the data into a format that is understood by our data reduction technique, i.e., Principal Component Analysis (PCA). PCA is a maximum variance projection method [12]. Performance counters have different ranges of numerical values; they have different variance. PCA identifies those variables that have a large data spread (variance), ignoring variables with low variance [19]. To eliminate PCA bias towards those variables with a larger variance, we standardized the performance counters via Unit Variance scaling (UV scaling), i.e., by dividing the observations of each counter variable by the variable's standard deviation. Each scaled variable then has equal (unit) variance. TABLE II. shows the variables after pre-treatment. Each variable has a mean of *0* and Standard deviation of *1*. Scaled performance counter data is then further mean centered to reduce the risk of collinearity. With mean-centering, the average value of each performance counter variable is calculated then subtracted from its respective counter data.

B. Performance Counters Normalization

Modern applications provide user to publish custom performance counters, it is not uncommon to find a same counter with different names on different systems. Performance counter data is collected from various subsystems of LSS during load test. Performance counters normalization ensures consistency among the performance counter data for our methodology. E.g. \\Server-1\app1\service/sec and \\Server-2\application1\service/sec are two same performance counters, with different counter name i.e., app1 and application1 and same instance name i.e., service/sec. Normalization ensures the portability of our approach across different platform and eliminated the false performance deviations between the subsystems of two load tests.

C. Dimension Reduction

We used the robust and scalable statistical technique, Principal Component Analysis (PCA) to reduce the sheer volume of performance counters [12]. What PCA does is to synthesize new variables called 'Principal Components' (PC). Every PC is independent and uncorrelated with other PCs. PCA is maximum variation projection method. To overcome this, we standardize the performance counters so that variance of each counter variable =1.0. TABLE III. shows the PCA for a real world performance counter log consisting of 18 performance counter variables. PCA groups the data of the 18 counter variables into principal components, each of which explains a particular amount of variance of the original data. This means the total variance of our counter data is 18. The first component PC1 has an eigen-value = 11.431, which means it explains more variance than a single counter variable, indeed 11.431 times as much and it accounts for 63.60% of the variability of entire performance counter data set. The second and third components have eigen-values 2.74 and 1.720 respectively. The rest of the components explain less variance than a single counter variable. To further trim the performance counter data we use '% Cumulative Variability' in selecting the number of top_k components. Using '% Cumulative Variability' of 90% is adequate to explain most of the data with minimal loss in information [12]. This means, we only need to take into account the first four PCs as shown in TABLE III.

TABLE III. PRINCIPAL COMPONENT ANALYSIS (PCA)

PC	*Eigen-Value*	*Variability (%)*	*Cumulative Variability %)*
PC1	11.43	63.506	63.506
PC2	2.47	15.260	78.765
PC3	1.720	9.554	88.319
PC4	0.926	5.143	93.463
⋮	⋮	⋮	⋮
PC12	0.001	0.003	100.00

D. *Crafting Performance Signatures*

Performance analysts are interested in performance counters, not in principal components. Table 1 shows that we have four principal components that can explain over 90% of the original counter data variability. We now decompose principal components of a subsystem using the eigenvectors technique to map the PCs back to counter variables [19]. Each performance counter is given a weight in accordance to its association with a PC. The larger the weight of a performance counter, the more it contributes to a PC.

TABLE IV. THE PERFORMANCE SIGNATURE OF A SUBSYSTEM

Rank	PC	Counter	Importance
1	PC1	N	0.974
2	PC1	M	0.972
3	PC1	R	0.966
4	PC1	Q	0.946
5	PC1	P	0.944
6	PC1	E	0.933
7	PC2	I	0.912

We applied the threshold to decide on the important performance counter variables and discard the rest [14].

The TABLE IV. shows seven performance counter variables out of 18 performance counters, ranked according to their importance. Our methodology achieved a 61% data reduction. These 7 performance counters form the performance signature of LSS subsystem. The performance signature acts as a finger-print to the subsystems of LSS and helps to identify and compare them with other subsystems. The importance of the performance counters in a signature hold for the same environment and workload of the test. However, when an error or unknown change occurs in the load test environment or in the workload, such as server replications, background antivirus scan etc, importance of performance counters for a subsystem shifts, causing the change in performance signature of the subsystem of LSS. This change in performance counter signature enables us to detect performance deviations.

E. *Extracting Performance Deviations*

One of the goals of our methodology is to help performance analyst in load test analysis by automatically identifying the subsystems of LSS that deviate from the baseline. This step of our methodology measures the correlation between the performance signatures of a subsystem in the load test with the corresponding subsystem's signature of a baseline test. The prior research has shown that stable relationship between metrics exists in a well-behaved system [6][7][8]. The relationships are often disturbed when error occurs. We use spearman's rank correlation to find the extent of deviations between performance signatures [23]. A value of +1 confirms that two performance signatures are identical and there is no performance deviation between the respective subsystems .We choose spearman's rank correlation over other correlation coefficients such as Pearson product-momentum, Kendall's tau and gamma because spearman's rank correlation does not require any assumptions about the frequency distribution of the variables. This is necessary because load test data contains traces that do not follow normal distribution of data.

F. *Report Generation*

To help a load tester examine the performance deviations, we generate performance deviation report. The report is generated in dynamic- HTML so testers can easily attach it to emails that are sent out while investigating a particular subsystem's performance deviations. The report contains visualizations and correlation tables to point out the divergence between two subsystems.

IV. MEASURING THE PERFORMANCE OF OUR METHODOLOGY

To evaluate the effectiveness of our approach, we used two metrics: *Precision* and *Recall.* Precision is the ratio between correctly identified performance deviations and predicted performance deviations between the same subsystems of two load tests. Recall is defined as the ratio between the correctly identified performance deviation and actual performance deviations in a subsystem. We use Figure 3 as an example to explain how we can measure the precision and recall of our methodology. Figure 3 show the performance counter logs of the database subsystem for the two load tests. These two load tests are conducted under constant environment and with the same workload. For both the load tests, performance counter logs are divided into equal time intervals, i.e., from t_1 to t_5. For the load test-2, an unexpected network link failure occurred between database and load generator during time interval t_3, t_4 *and* t_5.

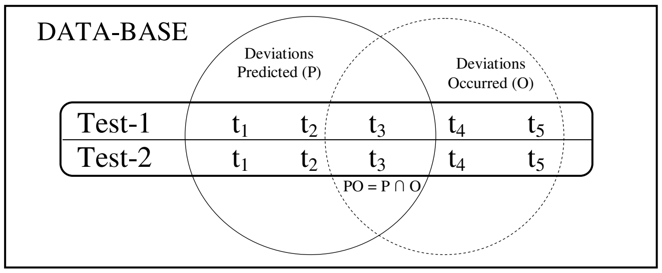

Figure 3. Performance measure of our methodology.

On careful analysis by performance analysts, both subsystems are found to be identical in performance except at time interval t_3, t_4, t_5. An ideal methodology should only report the time interval at which the deviations occurred. We define the number of *time intervals* when actual performance *deviations occurred* as $O = 3$. We applied our methodology on the database subsystem of both tests and it predicted performance deviations between two test at time intervals t_1, t_2 and t_3. We define the predicted number of time intervals by our methodology as $P = 3$. An ideal methodology will predict the same deviations P as that of actual deviations occurred, i.e., O in a subsystem. Based on these definitions we define: *Recall* = *PO/O* and *Precision* = *PO/P*. We have defined the prediction and recall for a single subsystem. To measure the performance of a methodology for all the subsystems of LSS, i.e., '*C*' involved in a load test, we use the following definitions:

$$\text{Average Recall} = \frac{1}{C} \times \sum_{I=1}^{C} (\text{Recall}_i) \qquad (3)$$

$$\text{Average Precision} = \frac{1}{C} \times \sum_{I=1}^{C} (\text{Precision}_i) \qquad (4)$$

V. CASE STUDY

To evaluate the performance and reliability of our approach, we conducted a case study on the load test logs of a large enterprise application. The goal of our case study is to thoroughly examine the following research questions:

Q1 *How accurate is our methodology to identify the subsystems of an LSS, which have performance deviations relative to prior tests?*

Q2. *Can we save time on the unnecessary load test completion by early identifying the performance deviations along different subsystems of a LSS?*

Q3. *How is the performance of our methodology affected by different sampling intervals?*

A. Environment Setup

The Figure 4 shows our load test environment. The four subsystems are enclosed within a dotted line. An enterprise application runs on a cluster and utilizes a database to store data. The enterprise application uses two web servers to allow users to share documents, schedule meetings and access intranet resources between two geographically separated locations. An internal load generator mimics the user's interaction with an enterprise application by performing simultaneous concurrent transactions, thereby places load on the database. The external load generators emulate large volume of traffic that results from the outside of the intranet to stress web servers. A customized performance monitoring tool monitors the numeric measurements related to the system's state and performance and records them as performance counter logs.

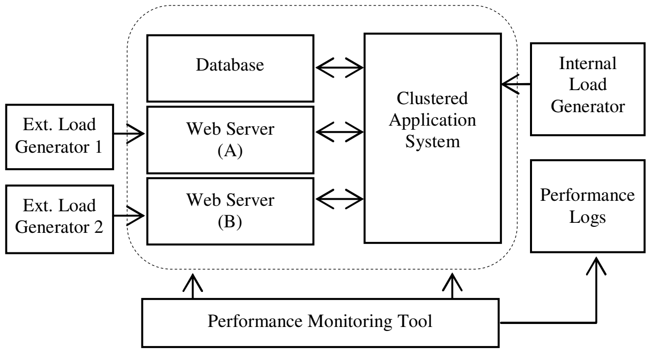

Figure 4. Components of test environment

Q1 *How accurate is our methodology to identify the subsystems of an LSS, which have performance deviations relative to prior tests?*

Motivation: During the course of the load test, an LSS is strictly monitored and thousands of performance counters are recorded. Performance analysts select few of the important counters based on their domain knowledge and expertise and compare them with a baseline test or predefined thresholds [3]. If the performance analyst finds significant deviations in a load test from the baseline test, analyst further investigates to find the sub-systems that are the cause of performance deviations. This is not an easy task, as the performance engineer needs to drill down thousands of performance counters distributed across several subsystems to find the required information. The process of identifying the performance deviated subsystems in a LSS can take many days.

Approach: We conducted an experiment to find out how accurately our approach can help performance analysts to pinpoint the performance deviated sub-systems in limited time, with accuracy and without implying domain knowledge. This experiment uses one baseline load test and three other load tests. The first load test was a re-run of the baseline load test. In the second test, we synthetically induced faults in to the baseline load test's performance counter log and in third test, we stressed the system by pushing 8-times (8X) more load than its expectation. The baseline load test was conducted by the performance engineers of a large enterprise. They carefully analyzed the baseline test to find any performance issues and deviations that raise the concerns of the LSS stakeholders.

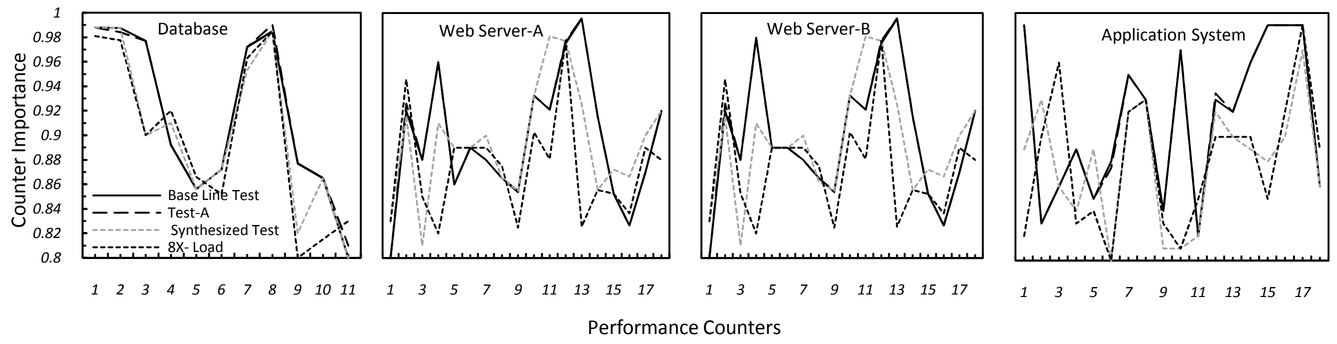

Figure 5. Comparison with baseline load test

TABLE V. CORRELATION BETWEEN PERFORMANCE SIGNATURES.

Subsystems	Load Test		
	Test-A	*Synthesized*	*8-X load*
Data Base	*0.997*	*0.732*	*0.826*
Web Server-A	*1.000*	*0.701*	*0.795*
Web Server-B	*1.000*	*0.700*	*0.790*
Application	*1.000*	*0.623*	*0.681*

The performance counter log of the baseline load test contained seven hundred (700) performance counters belonging to four subsystems. The load test duration for each test was 8 hours. The performance monitoring tool collected the performance counter data periodically after ever *15 sec* (sampling interval) and there were *1922* instance *(numeric readings)* in total, recorded for each performance counter in the performance counter log. We applied our methodology on the performance counter log of the baseline load test and extracted the performance signatures for the sub-systems. Our methodology recommended *11* important performance counters for database subsystem, ranked on the basis of their importance for the load test. Furthermore, our methodology recommended 18 important performance counters each, as performance signatures for the remaining three subsystems. Thereby, achieving 85% of reductions in the performance counter log data.

Test 1: We used the framework of Thakkar et al. to automate the load test and to ensure the environment remains constant [10]. We ran the load test under same environment, workload and duration as baseline load test and call it as 'Test-A'. The intuition for conducting the load test similar to that of baseline- load test is to validate our methodology. Our methodology should craft the same performance signature for every subsystem as that of the baseline load test.

Test 2: We synthetically injected faults into the performance counter log of our base-line load test. First we select *50 %* the performance counters of each sub-system. Second, among the 50% of performance counters selected, we ensured that *50%* of the important performance counters that contribute toward constructing the signature of a subsystem are present. Third, we mutated the values of *40%* of the selected performance counter data using out of bag (OOB) approach [11]. Using OOB, reduce the chance of a performance counter instance being mutated twice. We did not generate totally random load test data, because there are important correlations and associations between counters that need to be satisfied to obtain realistic load test results. For example, if the CPU utilization is high for a load, corresponding performance counter instances for disk IOPS, memory consumption and queue levels will also have high instance values reflecting the same observed load.

Test 3: We conducted the load test with same workload mix as of baseline load test but increased its intensity 8 times. We illustrate what we mean by workload mix and increasing its intensity. For example, the load of an e-commerce website would contain information such as: browsing (40%) with a min/average/max rate of 5/10/20 requests/sec, and purchasing (40%) with a min/average/max rate of 2/3/5 requests/sec. In our experiment we keep the workload mix (browsing (40%) and purchasing (40%)) constant, however varied its intensity i.e., rate (request/sec). our institution was to deviate the behavior of load test from baseline load test by stressing its sub-subsystem; subjecting them to load beyond their design constraints.

Findings: our methodology achieved 85% performance counter data reduction. The Figure 5 shows and compares the signature plots between baseline test and other three tests. The TABLE V. provides rank correlations to calculate the performance deviations between the signatures. The 'Test-A', which is a replica of the baseline test, has a near perfect signature match for every subsystem with the baseline- test, as shown in Figure 5. The subsystems of 'Test-A' have high correlation with the baseline test. The database performance signatures of 'Test-A' differs from the baseline- test by spearman's rank correlation coefficient of 0.003. This difference is so small that it can be attributed to experimental measurement error. Our methodology did not suggested any performance deviations between 'Test-A' and baseline test at the sub-system level, yielding a precision and recall of 100%. For test-2, where we synthetically injected faults and for test-3, where we stressed the system by pushing 8 times more load then its expectation, our methodology suggested performance deviations from the baseline test for all four-subsystems.

TABLE VI. CORRELATIONS AT VARIOUS TIME INTERVALS

Time-(Observations)		Database
30-mins	*(120)*	*1*
15-mins	*(60)*	*1*
10-mins	*(40)*	*0.9893*
5-mins	*(20)*	*0.8255*

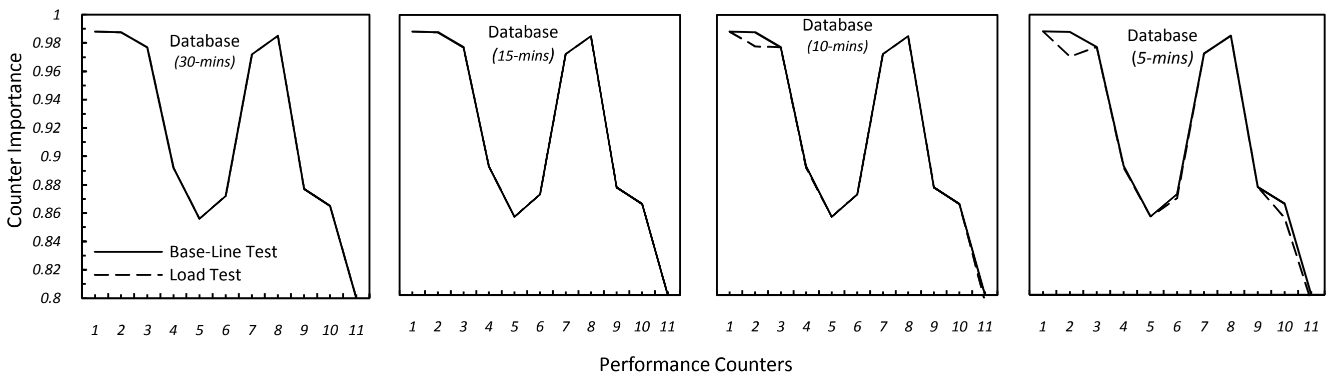

Figure 6. Effect of time interval on our methodology

The subsystem signature plots of test 2 and test 3 load tests in Figure 5 clearly indicate them different form the signatures of respective subsystems of baseline load test. The TABLE V. also indicates significant deviations between the sub-systems of both tests from baseline. This shows the consistency of our methodology to identify sub-systems that deviate from their performance baselines.

Our methodology helps performance analysts to identify sub-systems with performance deviations relative to prior tests in limited time. Our methodology can achieve *85%* reduction in performance counter data.

Q2. *Can we save time on the unnecessary load test completion by early identifying the performance deviations along different subsystems of a LSS?*

Motivation: There are two motivations for this research question. First, performance testing is usually done as the last step in an already tight and usually delayed release schedule. Hence, managers are always eager to reduce the time allocated for performance testing. Software analysts are required to conduct many load tests, which typically span over few hours to days. In is not uncommon for analyst to find out after the completion of the test that it failed due to unexpected circumstances. This includes mis-configurations and various unpredicted background activities, such as disk scrubs, RAID reconstructions, anti-virus scans, and data replication [15]. Once problems are addressed, the test needs to start from scratch. A lot of precious time is wasted, in many cases the performance analysts have to compromise on the number of tests they need to run to gain statistical confidence in the results in limited time. The idea of test reduction has been researched thoroughly in the functional testing area [16][17]. However, this idea has not been explored much for load testing. A detailed discussion on how to reduce the testing time by improving experimental design technique and careful planning of test to avoid mistakes is presented in [18]. However, to the best of our knowledge, when the load test is set in motion, there exists no automated and proactive technique that can guide and report analyst in real-time about the performance deviations that occurs during the load test run. So that, performance analyst can stop the test at that point of time where performance deviation occurred, fix the issue if it can be fixed and run the load test again, rather waiting for days to let the load test finish and get surprises at cost of time.

Approach: we want our methodology to report the deviations of subsystem's performance from baseline as soon as it occurs in a running load test. One way to overcome the situation is to frequently compare the performance signatures obtained from the periodic samples of the performance counter logs as they become available from a running load test and compare them with the baseline. To detect a performance deviations among subsystems such as caused by resource failure, e.g., ethernet link failure, disk failure etc, sample of performance counter logs are required as soon as they become available from running load test. However, if performance analysts are interested to detect performance deviations resulting from resource saturations that usually evolve over time, e.g., cache pollution and memory leaks, aggressively sampling at smaller time interval of performance counter log is not required. We don't want performance analyst to apply our methodology very generously by sampling the performance counter log at large interval thereby, missing performance bugs. We now explain what we mean by this. We used a statistical technique, PCA towards constructing signatures from the performance counter logs. Unlike statistical techniques, PCA is not highly sensitive to minute changes in data and is a maximum variance projection method [19]. This means PCA takes into account the counter's majority data value to calculate its importance for the load test, giving less weight to counter data in minority. For example, one thousand observations are recorded for a performance counter (CPU Utilization) of a subsystem by a monitoring tool in a load test. All the performance counter values are stabilized at *40%* to *50%* of the CPU utilization. Among thousand observations recorded, only one observation has *80%* of the CPU utilization. For such a small difference, our methodology will mark the subsystems of both the load and baseline test analogous. However, if there is one observation among 10 observations collected, our methodology will report the performance deviation between the subsystems. To find out how much our methodology can be relaxed in constructing performance signatures such that it never misses to pinpoint the performance deviations among LSS sub-systems,

TABLE VII. PERFORMANCE OF OUR METHODOLOGY

Test Run			Database		Web Server -1		Web Server- 2		Application System		Average	
Min	*Obs*	*Samples*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
30	120	4	0.50	1.00	0.50	1.00	0.30	1.00	0.25	1.00	0.325	1.000
15	60	8	0.62	1.00	0.62	1.0	0.62	1.0	0.50	1.0	0.590	1.000
10	40	12	1.00	0.90	1.00	0.9	1.00	0.9	0.9	0.69	0.975	0.847
5	20	24	1.00	0.70	1.00	0.7	1.00	0.8	1.00	0.66	1.000	0.715
All		**-**	**0.78**	**0.90**	**0.78**	**0.90**	**0.73**	**0.92**	**0.66**	**0.83**	**0.738**	**0.890**

we conducted an experiment based on the two load tests using the same environment as shown in Figure 4. Each load test spanned over two hours and performance monitoring tool was set to capture performance counter data after every 15 seconds. Total of 480 observations were recorded for each performance counter. We marked the first load test as our baseline load test. During the mid of the second load test, i.e., after 60 minutes, we slowed down the CPU of the database server for 15 seconds by using a CPU stress tool [31]. We stressed the database CPU for such short period of time as we were interested to find out the maximum time period within which minimum performance deviations can detect by our methodology. The slowing down of the CPU increased the CPU utilization of database server to *100%.* After the completion of the load tests, we used their performance counter logs and applied our methodology to find out the performance deviation among subsystems. Our methodology reported both test to be analogous.

We started to reduce the performance counter data of both test by roughly 12%, i.e., 10 minutes. We removed the 5 minutes, roughly (6%) of performance counter data from the start and from the end of the performance counters log. We kept reducing the performance counter log by 12% until out methodology reported the performance deviation between the subsystems of two load tests.

Findings: The Figure 6 shows that our methodology can identify deviation between two tests for sample collected within 10 minutes span. According to Figure 6, sampling interval larger than 10 minutes or 40 observations may lead to miss minute deviation between the subsystems. The values reported in TABLE VI. are the spearman's rank correlations between the performance signatures of database and corresponding subsystem in baseline load test. The values show that smaller the sampling time interval for performance counter log, better the extent of deviation measure. We conducted experiments on other subsystems and found the results to be consistent. We can conclude that frequently sampling the performance log of a running load test by our methodology helps performance analyst to detect early performance deviations among LSS subsystems. Thereby, saving time that is required to finish the load test, before they can be repeated.

Our methodology helps to reduce the unnecessary load test completion time by detecting early performance deviations in subsystems and reporting them in real-time to performance analyst.

Q3. How is the performance of our methodology affected by different sampling intervals?

Motivations: A methodology with low recall is not useful because it fails to identify the real faults. A methodology that produces results with high recall and low precision is not useful either because it floods performance analysts with too many false positives. An ideal methodology should predict a minimal and correct set of faults/deviations in a system. In practice, however, there is a tension between having high recall and high precision [32][33]. Maximizing the precision of a methodology often means that potential faults are being thrown out, which decreases its recall. We want to evaluate the performance of our methodology using the definitions of precession and recall in section VI.

Approach: We conducted and experiment that consisted of two load test with identical environment and workload. Both load tests spanned over two hours. The monitoring was set to capture the performance counters every *15 seconds*. The first load test, 'Load test-1' is taken as a baseline test. During the course of 'Load-test-2', we stopped the load generators ten times, repeatedly after every ten minutes for the duration of 15 seconds. After the completion of the load tests, we took the performance counter logs and applied our methodology on them for various time intervals, i.e., 30, 15, 10 and 5 minutes as shown in TABLE VII. The time interval is the duration, during which performance counter data is collected for a running test. For 2 hours of test, a 30 minutes time interval means that our methodology can construct 4 performance counter signatures based on the performance counter data collected for every 30 minutes of the load test.

Findings: The TABLE VII. shows the performance of the four subsystems in terms of precision and recall, as shown in Figure 4. The *Average* column in TABLE VII. shows the average performance of the complete load test. The *Test Run* column shows the length of the time interval during which the performance of our methodology is measured. The *samples* column indicates the number of samples that can be obtained in 2 hours of test based on the length of time interval. As we have injected the faults in the `Load test -2` after every 9 minutes by, the first sample of 30 minutes contains three faults. The second and third sample also contains three faults. The fourth segment has one only 1 fault bringing the total faults to ten. We applied our methodology on all four samples of the database subsystem of 'Load Test-1' to construct the performance signatures and compare them against the respective baseline subsystems. Our methodology only found the deviations between the performance signatures of sample 1 and sample 4 from the baseline. We expected our methodology to find the performance deviation among all the four segments of performance counter logs. According to the definitions of precision and recall in section IV, *PO=2, O=4 and P= 2*. Therefore, the recall is 0.50 and precision is 1.0. Our methodology has a high precession for 30 minutes time interval. However, the average recall is 0.325, which is very low. Applying our methodology at smaller time interval or smaller set of observation improves its performance.

We get a perfect recall of 1.0 at 5 minutes time interval but our methodology does report some false positive deviations. The reason for reduction in precision at such small time interval is due to fact that with little data (small sample size), PCA sensitivity towards minute changes increase. Our methodology performs well at a 10 minutes time interval, with nice balance of both recall and precision.

The performance of our methodology is time dependent. Over large time intervals, our methodology achieves high precision. Applying our methodology at small time interval yields high recall.

VI. RELATED WORK

The related work falls in to two areas:

Dimension Reduction:

We direct the readers to the detailed related work on dimension reduction [14].

Automatic Performance Monitoring and Analysis of Enterprise Systems:

How to automate performance monitoring and analysis of an enterprise system is not a new problem. Since past five years, several tools and techniques have been proposed by various researchers. Huck and Malony proposed a performance data mining frame work for large-scale parallel computing. The framework tries to manage data complexity by using techniques such as clustering and dimensionality reduction [24]. This data mining framework utilizes random liner projection and PCA to reduce performance data. The framework only reduces the performance data to Principal component but doesn't achieve fine grain analysis like our proposed methodology by decomposing the PCs to reveal the performance counters. Sandeep, et al. work is closest to ours [15]. They employed principal feature analysis (PFA) to achieve data reduction. The main difference between their approach and ours is that they utilized machine learning to distil the large counter set in to smaller set to describe the workload. Also, their work is partially automated and requires continuous training to produce accurate results.

Cohen et al.[22] develop application signatures based on the various system metrics (like CPU, memory). Jiang, et al automated the performance analysis of load test [3]. Unlike our work they relied on execution logs. Few researches have exploited static dependency models to capture the dynamic complexity of large systems [25][26][27][28]. They use these dependency models describing the relationship among the hardware and software components in the systems. These dependency models are used to determine which components might be responsible for the symptoms of the given problem. The first major limitations of traditional dependency model is the difficulty of generating and maintain and accurate model of a constantly evolving large system. Their second limitation is that they typically only model a logical system, and do not distinguish among replicated components. Whereas, in a large enterprise system, there will be many replicated components.

Pinpoint and *Magpie* track communication dependencies with aim of isolating the root cause of misbehavior; they require instrumentation of the application to tab client requests [29][30]. Whereas, our methodology, do not require any instrumentation of the system. *Magpie* characterizes transaction resource footprints in fine details but requires that the application logic be meticulously encoded in "event Schema". Unlike Magpie, our methodology does not require any system knowledge. *Pip* aims to infer casual paths and require and explicit specification of the expected behavior of a system [4]. Whereas, our methodology, do not require such explicit specifications of the expected behavior. It relies heavily on statistical methods to automatically extract the expected behavior for baseline tests.

VII. CONCLUSION AND FUTURE WORK

In this paper we presented our methodology to identify performance deviations between the components of LSS. Our methodology uses Principal Component Analysis, a statistical technique to reduce large volume of performance counter data. Furthermore, our methodology identifies and ranks performance counters for each component based on its importance for the load test. The importance of performance counters hold for the normal behavior of system under load. The importance of the performance counters changes for respective components of a system if any error or deviation for normal behavior occurs. A large case study on a real-world industrial software system provides empirical evidence on the ability of our methodology to uncover the performance deviation between two load tests in limited time, without implying any domain knowledge.

Currently, our methodology identifies the time interval in which performance deviations occurs during the load test. In future we plan to employ sliding windows to identify the deviations in subsystems as they occur during the load test. In the future, we plan to compare the performance of our methodology with that of other techniques such as Naïve-bays classifier and factor analysis to yield further improvement in constructing effective performance signatures. We also want to strengthen our methodology by taking in to account the effect of deviations that a subsystem has one other as a "carry over" effect. This will help performance analyst to perform root-cause analysis on the performance issues in LSS

ACKNOWLEDGMENTS

We are grateful to Research In Motion (RIM) for providing access to the enterprise applications 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 RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's software products.

REFERENCES

[1] Beizer. B., "Software System Testing and Quality Assurance"' Van Nostrand Reinhold, March 1984.

[2] Avritzer. A., Larson. B., "Load testing software using deterministic state testing", In Proceedings of the ACM SIGSOFT international symposium on Software, 1993.

[3] Jiang, Z. M., Hassan, A. E., Hamann, G., Flora, P., Automated Performance Analysis of Load Tests. In Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM) 2009, Edmonton, Canada, September 20-26, 2009.

[4] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. 2006. Pip: detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (San Jose, CA, May 08 - 10, 2006). USENIX Association, Berkeley, CA, 9-9

[5] M, Jiang., Munawar, M.A., Reidemeister, T.; Ward, P.A.S., "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on* , vol., no., pp.285-294, June 29 2009-July 2 2009

[6] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In DSN, pages 259–268, 2006.

[7] G. Jiang, H. Chen, and K. Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Trans. on Dependable and Secure Computing*, 3(4):312–326, 2006.

[8] M. A. Munawar and P. A. Ward. Adaptive monitoring in enterprise software systems. In SysML, June 2006.

[9] Jiang, M., Munawar, M. A., Reidemeister, T., and Ward, P. A. 2008. Information-theoretic modeling for tracking the health of complex software systems. In *Proceedings of the 2008 Conference of the Center For Advanced Studies on Collaborative Research: Meeting of Minds* (Ontario, Canada, October 27 - 30, 2008). CASCON '08. ACM, New York, NY, 236-247

[10] Thakkar, D., Hassan, A. E., Hamann, G., and Flora, P. 2008. A framework for measurement based performance modeling. In Proceedings of the 7th international Workshop on Software and Performance (Princeton, NJ, USA, June 23 - 26, 2008). WOSP '08. ACM, New York, NY, 55-66.

[11] Malik, H.; Chowdhury, I.; Hsiao-Ming Tsou; Zhen Ming Jiang; Hassan, A.E., "Understanding the rationale for updating a function's comment," *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* , vol., no., pp.167-176, Sept. 28 2008-Oct. 4 2008

[12] Jolliffe IT., "Principal Component Analysis", Second Edition. New York, Springer-Verlag; (Springer Series in Statistics), 2002.

[13] Ringberg, H., Soule, A., Rexford, J., Diot, C., "Sensitivity of PCA for traffic anomaly detection", In ACM SIGMETRICS,San Diego, CA, USA, 2007.

[14] Malik, H., Jiang, Z. M., Bram, A, Hassan, A. E., "Automatic Comparison of Load Tests to Support the Performance Analysis of Large Enterpirse Systems", *The European Conference on Software Maintenance and Reengineering,* CSMR 2010, Mar 15-18, In press.

[15] Sandeep, S. Ratna; Swapna, M.; Niranjan, Thirumale; Susarla, Sai & Nandi, Siddhartha: CLUEBOX: A Performance Log Analyzer for Automated Troubleshooting. USENIX Association (2008) .

[16] Xie, T., Marinov, D., and Notkin, D. 2004. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *Proceedings of the 19th IEEE international Conference on Automated Software Engineering* (September 20 - 24, 2004). Automated Software Engineering. IEEE Computer Society, Washington, DC, 196-205.

[17] Rothermel, G. and Harrold, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (Apr. 1997), 173-210.

[18] Jain R. 1992. The art of computer systems performance analysis. John Wiley.

[19] Jolliffe IT., "Principal Component Analysis", Second Edition. New York, Springer-Verlag; (Springer Series in Statistics), 2002.

[20] M.W. Knop, J.M. Schopf and P.A. Dinda, "Windows performance monitoring and data reduction using watch tower", *Workshop on Self-Healing, Adaptive and self-MANaged Systems* (SHAMAN), 2002.

[21] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. *In Proc. 6th USENIX OSDI*,San Francisco, CA, Dec. 2004.

[22] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. 2005. Capturing, indexing, clustering, and retrieving system history. *In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 23 - 26, 2005). SOSP '05. ACM, New York, NY, 105-118

[23] Rosner B., Fundamentals of Biostatistics, 4th Edition, Duxbury Press, Belmont, California, USA, 1995.

[24] Huck, K.A. and Malony, A.D., "PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing," *Supercomputing, 2005. Proceedings of the* ACM/IEEE SC 2005 Conference, vol., no., pp. 41-41, 2005.

[25] A. Brown. and D. Patterson., An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. *In Seventh IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[26] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. *In IEEE International Conference on Communications*, Vancouver, BC, Canada, 1999.

[27] B. Gruschke. A New Approach for Event Correlation basedon Dependency Graphs. *In 5th Workshop of the OpenView University Association:* OVUA'98, Rennes, France, April 1998.

[28] A. Yemini and S. Kliger. High Speed and Robust Event Correlation. *IEEE Communication Magazine*, 34(5):82–90, May 1996.

[29] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In OSDI, 2004.

[30] Chen, M.Y.; Kiciman, E.; Fratkin, E.; Fox, A.; Brewer, E., "Pinpoint: problem determination in large, dynamic Internet services," *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* , vol., no., pp. 595-604, 2002

[31] Stress Tool: http://www.oldskool.org/pc/throttle, Downloaded: December 2009.

[32] Malik, H. Hassan, A.E., "Supporting software evolution using adaptive change propagation heuristics," *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* , vol., no., pp.177-186, Sept. 28 2008-Oct. 4 2008

[33] Hassan, A. E. and Holt, R. C. 2004. Predicting Change Propagation in Software Systems. In Proceedings of the 20th IEEE international Conference on Software Maintenance (September 11 - 14, 2004). ICSM. IEEE Computer Society, Washington, DC, 284-293.