Pinpointing the Subsystems Responsible for the Performance Deviations in a Load Test

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

***Abstract*—Large scale systems (LSS) contain multiple subsystems that interact across multiple nodes in sometimes unforeseen and complicated ways. As a result, pinpointing the subsystems that are the source of performance degradation for a load test in LSS can be frustrating, and might take several hours or even days. This is due to the large volume of performance counter data collected such as *CPU utilization, Disk I/O, memory consumption and network traffic,* the limited operational knowledge of analysts about all subsystems of an LSS and the unavailability of up-to-date documentation in a LSS. We have developed a methodology that automatically ranks the subsystems according to the deviation of their performance in a load test. Our methodology uses performance counter data of a load test to craft performance signatures for the LSS subsystems. Pair-wise correlations among the performance signatures of subsystems within a load test are compared with the corresponding correlations in a baseline test to pinpoint the subsystems responsible for the performance violations. Case studies on load test data obtained from a large telecom system an open source benchmark application show that our approach provides an accuracy of *79%* and our approach don't require any instrumentation or domain knowledge to operate.**

Keywords-Pinpointing; Performance Counters; Load Testing;

I. INTRODUCTION

Large scale systems (LSS) such as Google, Facebook, Hotmail and eBay keep on growing in size[1] and in complexity. They provide composite services, support a large user base and handle complex business demands. In line with Lehman's laws of continuing change and increasing complexity [1], the periodic maintenance of such LSS has become more critical and challenging than before since processing is spread across thousands of subsystems and millions of hardware nodes (and users).

Performance analysts and developers of LSS spend considerable time dealing with functional and performance bugs. Deadlocks and memory management bugs are examples of functional problems under load. Performance problems include an application not responding fast enough, crashing or hanging under heavy load or not meeting the desired service level agreements (SLA). Problems after the release of an application are seldom due to feature error, but rather due to systems not scaling to field workloads [2] [9, 34]. Companies like 'AT&T' and 'Research In Motion' also report their concerns about performance degradation and resource saturation as fundamental post-release problems [3, 4]. As a result of functional and performance bugs, it becomes increasingly likely that a given subsystem of LSS might fail, potentially propagating to the entire system and resulting in large monetary losses. For example, an hour-long PayPal outage due to periodic maintenance may have prevented up to $7.2 million in customer transactions [5].

Load testing is an important weapon in LSS development to uncover functional and performance problems of a system under load [6]. One or more load generators are used to simulate committing thousands of concurrent transactions to an application under test [7]. During the course of a load test (which may span over many days), the application is closely monitored and a huge volume of performance counter data is logged. The performance counter log captures the run-time system's performance properties such as CPU utilization, disk I/O, queues and network traffic. Such information is of vital interest to performance analysts, as it helps them to observe the system's behavior under load by comparing against documented behavior of an application/system or with an expected behavior.

LSS are vulnerable to propagation of failures due to inherit coupling between their subsystems and unexpected propagation of faults because of hidden dependencies. Hence, failures manifestation propagates to non-faulty subsystems affecting their performance. These non-faulty subsystems prove to be false-positive recommendations to an analyst. In practice, for LSS, it is impossible for an analyst to skim through the huge volume of performance counters to identify the subsystems that are the cause of starting performance degradation in a load test. Some post-deployment approaches to identify the problems and failures in the distributed systems exist [8-10][11-14], but these approaches do not explicitly pinpoint the subsystems in LSS that are the cause of problem or failure. Moreover, they are not fully automatic, since they do not sufficiently capture the dynamic complexity of LSS and require analysts to input extensive knowledge of the system [8-10].

[1] No of nodes: Facebook 10000+[36], Google 30000+[37], Hotmail 7000+[38]

Our previous work aimed to help performance analysts to automatically identify the performance deviated subsystems in a load test by crafting performance signatures [16]. Such a signature consists of the essential performance counters for a particular load test. However, our technique reported all subsystems with deviating performance counters, without ranking them according to criticality. An analyst is only interested in those subsystems that are the real cause of performance deviations among all the performance deviated subsystem reported. Without support, troubleshooting the cause of the performance deviation in a load test is both expert-intensive and time-sensitive activity.

In this paper, we improve our previous technique to rank the subsystems with performance deviations according to the extent of deviation from the baseline. The paper makes the following contributions:

C1. We empirically validate our proposed methodology through a case study on a real-world industrial software system and an open-source benchmark application.

C2. We show how simple pair-wise correlation between the performance signatures of subsystems can rank the performance deviated subsystems in order of their likelihood of causing the performance deviation.

Organization of the Paper

The rest of the paper is organized as follows: The paper first presents a motivating scenario in section II, followed by the current practice of load testing and its limitations in section III. We present our methodology in section IV. Section V presents case study setup along with case study findings in section VI. The related work is presented in section VII. The conclusion and future work are reported in section VIII.

II. MOTIVATING SCENARIO

It was not a surprise for John to see a pile of testing documents being shoved-up on his desk, with a sticky note of his boss saying: "we want to go live with the BigMail project this Friday. We have a new automatic load test analysis system to help you. I am sure you can do it John." John has been load testing the features of the BigMail in the past. But this time there is a twist. The lab (mail servers supporting 30,000 users) is physically at another location due to all testing resources being fully occupied at such busy time. John has been assigned the remote lab from 5 pm to 5 am. John gets busy like a bee setting up the test environment, installing some domain specific application and configuring load testing tools (load generators, performance monitors, emulators etc). John is done setting up the environment by 4:30. He again verifies all the steps with his check-list to ensure everything is setup correctly. John knows that a slight negligence can cost him a whole day's effort. He starts the first test at 5:00pm and leaves to home. Next day, John finds the load test analysis report in his mailbox generated by an automated load test support tool. John was disappointed to see the load test failed with high deviation from the baseline test. However, he was surprised to see 1) a neatly organized report with the list of performance deviated subsystems sorted on the basis of importance, 2) a visualization for every deviated subsystem, comparing their important counters with the base- load test and 3) a correlation table that provides statistical analysis result between the two tests.

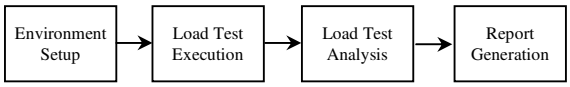


Figure 1. Performance load testing process

John pressed the 'Diagnostic button' in the html report and was presented with the list of 20 subsystems that deviated from baseline test. All the deviated subsystems belong to the remote lab. The report also pinpointed the subsystem (mail server-A) as the cause of the performance deviation. The visualization clearly showed high deviation of the mail server-A's counter 'disk Read/sec' from the base-line load test. Such high deviations could only have been possible under extreme stress conditions. John calls the mail server-A's administrator at remote location and finds out that heavy disk utilization noticed for the server 'A' was the result of scheduled maintenance activity that runs every sunday night. No doubt, John's location was 24 hours ahead of the remote lab location. John is happy; the automated system helped him to identify the problem subsystem. He did not have to spend time skimming through the large volume of performance counter data, piecing together the cause of test failure. He hurries to start the test again.

III. CURRENT PRACTICE OF LOAD TESTING

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 [15, 16]. The environment setup includes installing the application and the load testing tools, possibly on different operating platforms. Load generators, which emulate the user's interaction with the system, need to be carefully configured to match the real workload in the 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 a baseline, such as another load test's result, or against pre-defined thresholds. Unlike functional and unit testing, which result in a pass or failure classification for each test, load testing involves 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 of collected to compare them with the baseline. Based on his experience and the domain knowledge, the 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.

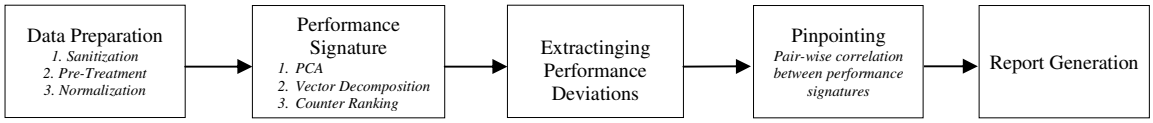


Figure 2: The steps involved in the proposed methodology

4. ***Report generation*** includes reporting 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 such as database, application, web system etc., the respective subsystem is then assigned to a relevant team for rectification.

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 [17]. Analysis of such large counter logs to identify the subsystem that is the cause of performance deviation in a load test is still a huge challenge. 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 the past practices, performance experts and domain trends as 'rules of thumb' [18]. Applying such 'rules of thumb' on load tests can provide misleading information about performance issues [18], thereby leading to pinpoint incorrect performance deviated subsystem(s).
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.

TABLE I: 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: 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

3. ***Limited global knowledge:*** Load test analysis is error-prone because of the manual process involved in analyzing performance counter data in current practice. There is no single person with complete knowledge of end to end geographically distributed system activities in an LSS [19]. 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.

Due to the above challenges, we believe that the current practice to perform load test analysis in LSS is neither efficient nor sufficient to pinpoint performance deviated subsystems accurately and in limited time.

IV. THE METHODOLOGY

In this section, we present and discuss our methodology to help analysts in load test analysis by pinpointing the subsystems that are likely the actual cause of performance deviations in limited time. Figure 2 shows the steps of our methodology.

A. Data Preparation

The performance logs obtained from a load test need to be prepared to make them suitable for the statistical techniques employed by our methodology. The three steps involved in data preparation are *1) Data sanitization*: Missing performance counter variables i.e., partially or completely are removed from the analysis. For example, the performance counter 'S' in TABLE I belongs to missing variable category. The performance counter 'T' in TABLE I belongs to incomplete variable category. Statistical techniques such as *list wise deletion* are employed to handle incomplete variables [21]. *2) Pre-treatment:* The data is converted into a format that is understood by our data reduction technique, i.e., Principal Component Analysis (PCA) [22]. Therefore, all performance counter variables are scaled to unit variance (mean of 0 and standard deviation of 1), as shown in TABLE II. *3) Normalization*: Different applications and systems may publish the same performance counter with different names. Normalization ensures the portability of our approach across different systems/platforms.

B. Performance Signatures

Performance signatures basically correspond to a minimal set of performance counters that describe the essential characteristics of a particular load test. We use a robust and scalable statistical technique for this, i.e., Principal Component Analysis (PCA) [22]. PCA transforms all load test counters into a smaller number of synthesized variables called *Principal Components* (PCs). Every PC is independent and uncorrelated with other PCs. TABLE III shows the PCA for a real world performance counter log consisting of 18 performance counter variables reduced into 12 PCs.

TABLE III: 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

TABLE IV: A SIGNATURE

Rank	PC	Counter	Imp
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

The first component PC1 has an eigen-value of 11.431, which means that it explains 11.431 times as much variance as a single counter variable. This accounts for 63.560% of the variability of the entire performance counter data set. To further trim the performance counter data, we only select the first *N* PCs that explain 90% of the '% Cumulative Variability'. 90% is adequate to explain most of the data with minimal loss in information [22].

We then decompose the selected principal components using the eigenvectors technique to map the PCs back to concrete counter variables [22]. Each performance counter is given a weight (importance) in accordance to its association with a PC. The larger the weight of a performance counter, the more it contributes to a PC. We applied the threshold to decide on the important performance counter variables and discard the rest [20]. TABLE IV shows seven performance counter variables out of 18, ranked according to their importance. Our methodology achieved a 61% data reduction. These 7 performance counters of a system forms its performance signature.

C. Extracting Performance Deviations

The goal of our methodology is to help performance analysts 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 that of the corresponding signature of a baseline test. Prior research has shown that usually stable relationships between metrics exist in a well-behaved system [23-25]. The relationships are often disturbed when error occurs [25]. We illustrate this with an example.

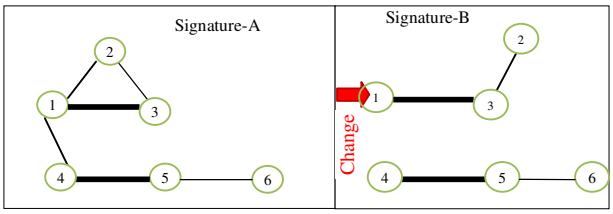


Figure 3: The sensitivity of performance signature

TABLE V: AVERAGE CORRELATIONS

	Base	*Deviated-Sys*	*Dev*	*%*
A	0.87	0.72	0.15	17.1
B	0.88	0.82	0.05	6.46
C	0.89	0.83	0.06	7.03
D	0.78	0.73	0.05	6.92

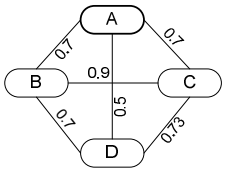


Figure 4: Pair-wise Correlations

Figure **3** shows a performance signature-A consisting of 6 important performance counters for a load test.
The thickness of the edges between the performance counters shows the strength of the association between them. Running the load test under the same workload and environment will produce the same signature. When a change (CPU stress) is introduced during the load test, disturbing the existing relationship between performance counters, a different signature-B is generated.

When a performance signature is crafted at subsystem level, it acts as a finger-print to the respective subsystem of LSS and helps to identify and compare the performance with baseline subsystems. The importance of the performance counters in a signature only holds 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 and background antivirus scan, the importance of performance counters for a subsystem shifts, causing a change in performance signature of the subsystem of LSS. This change in performance counter signature enables us to detect performance deviations at subsystem level.

We use Spearman's rank correlation to find the extent of deviation between performance signatures [26]. A value of +1 confirms that two performance signatures are identical and that 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.

D. Pinpointing

Whereas the previous step of our methodology identifies all subsystems that deviate from the baseline performance, the pinpointing step of our methodology ranks the subsystem(s) that is/are likely the cause of performance violation among all performance deviated subsystems in the load test.

To rank the subsystems, we calculate the average pair-wise correlation ($\Delta\rho$) that exists between the performance signatures of a performance deviated subsystem with that of the other misbehaved subsystems in a load test. The average pair-wise correlation of the subsystem is compared with the average pair-wise correlation of the respective subsystem in a baseline load test to determine the extent of performance deviation. Among the performance deviated subsystems, the subsystem having the highest deviation is likely to be the culprit subsystem.

We explain the pinpointing step of our methodology with the help of the following example. For a load test *'Test-I'*, the methodology extracts four performance deviated subsystems A, B, C and D. Among the pool of performance deviated subsystems, the methodology computes pair-wise correlations between the performance signatures of each subsystem with that of the rest in the pool, i.e., it calculates the pair-wise correlation (ρ) of subsystem A with that of B, C and D (AB=0.7, AC=0.7, AD=0.9) as shown in Figure 3.

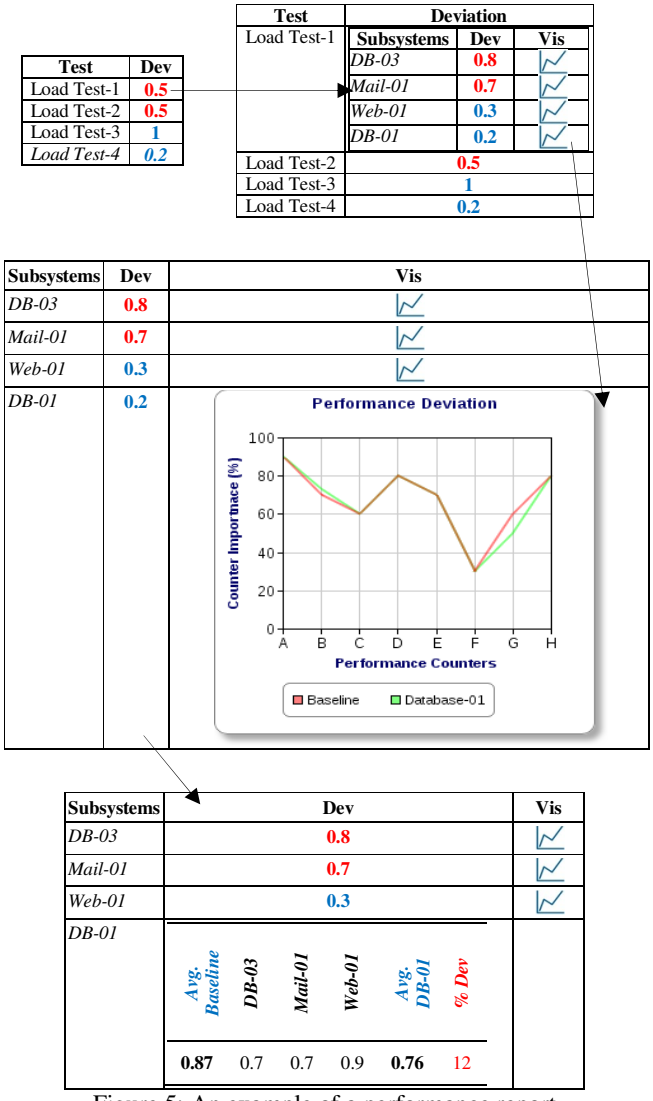


Figure 5: An example of a performance report

Then, for each subsystem the average of all pair-wise correlations that exists for the particular subsystem's signature with other performance deviated subsystem is computed. For performance deviated subsystem A, the *average correlation* of its performance signature is $\Delta\rho = (0.7 + 0.7 + 0.9 + 1.0)/4 = 0.72$, as shown in TABLE V. The average pair-wise correlation of each performance deviated subsystem is compared with that of the corresponding subsystem in a base-line test. If the signature of a different system contains a different number of performance counters, the common performance counters between two signatures are considered. The subsystem with the higher deviation among the pool of performance deviated subsystems i.e., 'A' is pinpointed to be likely the source of performance deviation in load test '*Test-I*' as shown in TABLE V. As the pinpointing step of our methodology calculates the average pair-wise correlation between the signatures of multiple subsystems, the probability of a Type I error is higher. Thus, Bonferroni's correction is made to the p-values of the Spearman correlations to correct for multiple testing. In essence, each p-value is multiplied by the number of comparisons and the adjusted p-value is compared to the standard significance level (0.05) to determine significance. Bonferroni's correction is chosen because it is a rather conservative test.

E. Report Generation

To help a load tester examine the performance deviations, we generate performance deviation reports. The report is generated in dynamic HTML so that the 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, as shown in Figure 5. It also includes a list of performance deviated subsystems. The subsystem having the highest average performance deviation is pinpointed as the cause of performance deviation in the load tests.

V. Case Study Setup

To evaluate the performance and reliability of our methodology, we conducted a case study on two different applications, i.e., the open source Dell DVD store and a large, proprietary telecom system. The goal of our case study is to thoroughly examine the following research hypothesis:

> *Our methodology accurately pinpoints the subsystems that are the cause of performance deviations in a load test*

A. The DELL DVD Store

The Dell DVD Store (DS2) application is an open source enterprise software application developed by Dell as a benchmarking workload for white papers and demonstrations of Dell's hardware solutions [2]. DS2 has a typical three-tier architecture, which consists of an application server component, database server components and a load generator engine (client emulator). The source code for the load generator is publicly available and runs on various platforms. The load generator can generate load on the application server, or directly generate load on the database server, skipping the application server altogether. The load generator emulates website users by sending HTTP requests to the application front-end. The application encodes the various business rules, such as ordering new titles or declining an order in case of insufficient inventory. All customers, titles and transactional data are stored in the database server tier.

We chose DS2 over other applications for many reasons. First, it is an open-source application, allowing us to debug and fix many problems with the application. Second, it is simple to use, through a command line interface. We created the load test environment for Dell DVD store similar to Figure 6, with three Tomcat containers running the JSP version of DS2, and a MySQL server as database server.

B. An Enterprise Application

Figure 6 shows the load test environment of the commercial system that we used as our second subject system. The four

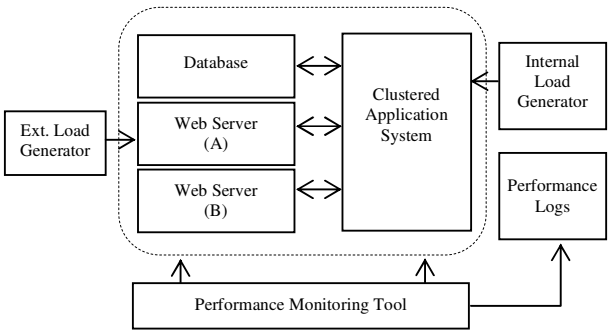


Figure 6. Components of test environment

TABLE VI. BASELINE CONFIGURATION FOR DS2

Parameter	Value
Duration	7 hours
Number of driver threads	50
Startup request rate	5
Think time (time it takes a customer to complete an order)	30 sec
Database size	Large
Percentage of new customers	15%
Average number of searches per order	3
Average number of items returned in each search	5
Average number of items per order	5

subsystems are enclosed within a dotted line. The enterprise application runs on a cluster and utilizes a database to store data. The enterprise application uses two web systems to allow users to share documents, schedule meetings and access intranet resources between geographically separated locations. An internal load generator mimics user interaction within an enterprise application by performing simultaneous concurrent transactions, placing load on the database. The external load generators emulate a large volume of traffic from the outside of the intranet to stress web systems. A customized performance monitoring tool monitors the system's performance counters.

VI. CASE STUDY FINDINGS

We conducted an experiment to find out how accurately our approach helps a performance analyst to pinpoint in limited time the subsystems that are the actual cause of performance deviation for a load test. The goal of the experiment was to validate ranking procedure of our technique under a number of representative tests. The experiment consisted of a baseline-test along with four more load tests conducted on Dell DVD store application and one load test conducted for proprietary system.

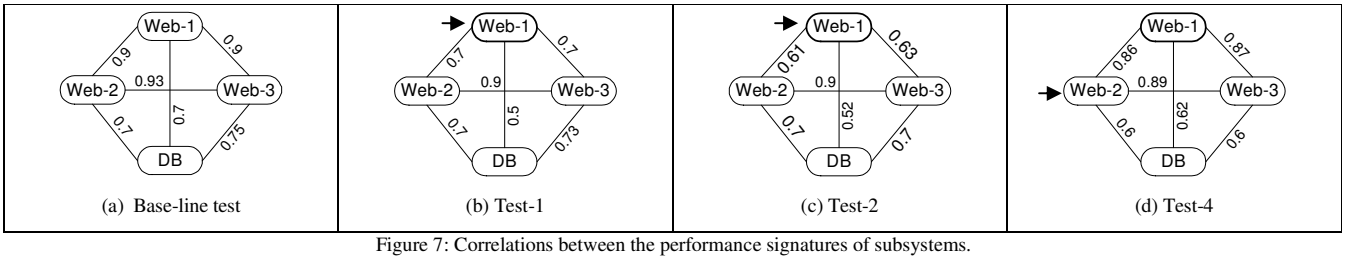
The workload configuration for the base-load test for DS2 is listed in TABLE VI. In our *first load test*, we stressed the system by pushing 4 times (4-X) more load than that of the base-line load. The reason to conduct such type of load test is based on the fact that most of the performance violations in load tests occur due to miss-configurations of load generators and during setup of complex test environments [7]. Also, unexpected intervention from other applications such as antivirus, disk scrubs, RAID reconstructions and data replication add to the change in normal workload [27]. In our *second and third test*, to simulate resource saturation, we stressed the CPUs of different subsystems. The reason to conduct such test was based on the fact that most of the post-release problems in LSS are related to performance degradation and resource saturation instead of feature bugs. This is because many feature bugs get resolved at earlier stages of testing by unit and functional testing. In the *fourth test*, we injected a memory bug in the database subsystem of DS2 , which is another way to stress the system.

Finally, the *fifth test* is based on the performance logs extracted from the load test repository of a large telecom enterprise. The motivation behind the fifth load test was to analyze how our methodology scales up to a large volume of performance logs.

We used the framework of Thakkar et al. to automate the DS2 load tests and to ensure that the environment remains constant throughout the experiments [4]. All load tests for DS-2 were repeated a minimum of five times to ensure the consistency among the results. The ramp-up and ramp-down periods of load tests were excluded from the analysis as the system usually is not stabilized at these periods. The results reported in TABLE VII and Figure 7 are the average of the correlations between the performance signatures of the respective subsystems of the load tests. A custom monitoring tool was used to record the *60* performance counters across the four subsystems of the Dell DVD Store. The tool collected the performance data periodically after every *15 sec* (sampling interval). Over a period of *8 hours*, the tool collected *1920* instances (numeric readings). The performance counter logs obtained from the repository of large enterprise contained *1400* performance counters. We are going to explain each test in more detail.

Base-line load test: Three load generators are configured to push the load on the three web servers (Tomcat running JSP application). The system under test is closely monitored and performance counter logs are collected from all the subsystems. We applied our methodology on the performance counter logs and extracted the performance signatures for all the subsystems of the base-line load test. The performance signatures of the web servers consist of 6 performance counters each. The performance signature for the database server contained 8 important performance counters. The Spearman rank correlation is used to calculate the extent of association between the subsystem's performance signatures as shown in Figure 7(a).

Test 1: To mimic the problems resulting from mis-configuration of the load generators, we conducted a test by pushing 4-X more load on webserver-1 than in the base-line load. The workload mix was kept the same as that of the baseline, except that we increased its intensity 4 times.



(a) Base-line test (b) Test-1 (c) Test-2 (d) Test-4

Figure 7: Correlations between the performance signatures of subsystems.

TABLE VII: AVERAGE PERFORMANCE DEVIATIONS OF SUBSYTEMS COMPARED TO BASE-LINE TEST

	Base	*4-X*	*Dev*	*%*
Web-1	0.87	0.72	0.15	17.1
Web-2	0.88	0.82	0.05	6.46
Web-3	0.89	0.83	0.06	7.03
DB	0.78	0.73	0.05	6.92

(a) Test-1

	Base	*CPU*	*Dev*	*%*
Web-1	0.87	0.69	0.18	21.1
Web-2	0.88	0.80	0.08	9.29
Web-3	0.89	0.80	0.09	10.6
DB	0.78	0.73	0.05	7.24

(b) Test-2

	Base	*CPU*	*Dev*	*%*
Web-1	0.87	0.83	0.03	4.28
Web-2	0.88	0.83	0.04	5.04
Web-3	0.89	0.84	0.05	6.14
DB	0.78	0.78	0.08	10.4

(c) Test-3

	Base	*MEM*	*Dev*	*%*
Web-1	0.87	0.81	0.06	7.42
Web-2	0.88	0.75	0.13	14.9
Web-3	0.89	0.81	0.08	9.49
DB	0.78	0.7	0.087	11.0

Test-4

We illustrate what we mean by workload mix and varying 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, we varied test's intensity (rate). The load pushed on the other web servers was the same as that of the base-line load test.

Test 2: Often the load test deviates from the base-line test either due to hardware failures or due to resource saturation. For example, the hard disk may fill up because the tester forgot to clean up the data from an older run. Once the disk is full, the application under test may turnoff specific features causing the test to deviate from its base-line performance. Resource saturation arising due to the intervention from other applications, i.e., from unknown background loads such as unplanned replication, virus scanner and disk-scrubs affects the performance of the load test. We slowed down the CPU of webserver-1 using a CPU stress tool known as *winThrottle* [28], webserver-1 is the weakest machine in our load test environment. We choose *winThrottle* over other CPU stress tools because it is an open source tool and can exploit a feature in system hardware that directly modifies the CPU clock speed, rather than using software "delay loops" or HLT instructions to slow down the machine. This method provides very smooth slowdowns without any incompatibilities with software. For an 8 hour load test, we stressed the CPU of the webserver-1 for 30 minutes.

Test 3: Test 3 is analogous to test 2, spanning over 8 hours, except that we stressed the database server, which runs on a more powerful machine than webserver-1. Since the database is used by all three different web servers, stressing the CPU of the database will affect all three web-servers, mimicking the propagation of performance problems from one subsystem to others. Since test 2 and test 3 are of similar nature, they enable us to validate the consistency of our methodology in pinpointing the same type of performance problem, i.e., CPU saturation, among different subsystems (webserver-1 and a database server).

TABLE VIII: CORRELATION BETWEEN THE LOAD TESTS

	Test-A	*Test-B*	*Test-C*	*Test-D*	*Test-E*
Test-A	1	0.9603	0.9732	0.37418	0.1790
Test-B		1	0.9778	0.45823	0.1359
Test-C			1	0.42581	0.1392
Test-D				1	0.2305
Test-E					1

Test 4: We conducted a load test with the same workload as the base-line load test, but injected a memory bug into webserver-2 using a customized open-source memory stress tool called EatMem [29]. The tool allocates a random amount of available memory at recurring intervals. To mimic a memory leak, webserver-2 was stressed for 30 minutes using EatMem.

Test5: The last test was conducted on the performance logs obtained from the test repository of a large telecom enterprise. The performance analysts gave us the performance logs of 5 load tests A, B, C, D and E. *Test A* was marked as a base-line test by the performance analysts and its performance counters were thoroughly analyzed by them for any performance discrepancy. Analysts told us that among the four tests, 2 tests were good and 2 of them violated performance requirements.

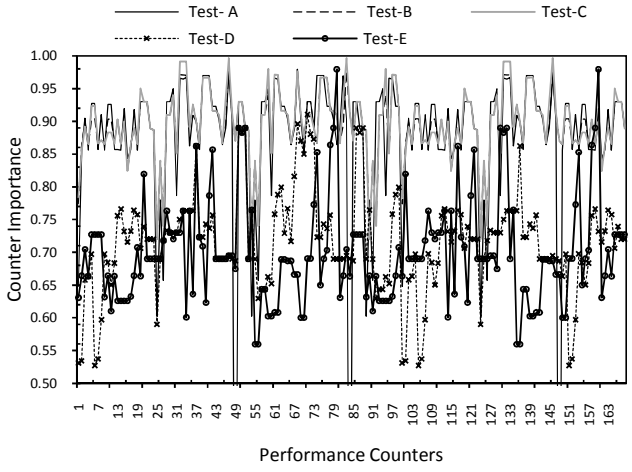


Figure 8: Comparison with the base-line performance signature

TABLE IX: PERFORMANCE OF OUR METHODOLOGY

Load Test		Performance Measure	
No.	*Sampling Interval*	*Precision*	*Accuracy*
Load Test-1	15 Sec	82%	79%
Load Test-2	30 Sec	82%	79%
Load Test-3	1 Min	77%	70%

The analysts were keen on finding out if our methodology could pinpoint the subsystems that were the cause of performance deviation between the baseline and the two failed tests. We applied our methodology on base-line *test A* to extract the performance signatures. The signatures consisted of 169 important performances counters out of a total of 1440 counters. We used the signatures as the base-line performance counters and compared them across the other tests.

Findings: Figure 7 shows the correlation between performance signatures of the DS2 subsystems. The ' ➔ ' marks the subsystem where a performance bug is injected. The '*' in TABLE VII indicates the subsystem that is pinpointed by our methodology as likely the cause of performance deviation in a load test. For test-1, our methodology correctly identified web-1 as the source of performance deviation in the load test. Web-1 has the largest deviation from the base-line based on the average pair-wise correlation of performance signatures shown in Figure 7 (b) and TABLE VII (a). Among CPU stress tests, i.e., test 2 and test 3, webserver-1 and the database server are pinpointed as the culprit subsystems. Hence, our methodology is consistent in pinpointing the accurate subsystems under same stress load (i.e. CPU stress). Although the same CPU stress was applied on webserver-1 during load test 2 and on the database server during load test 3, however, the extent of '%' average performance deviation is different for both. The last test conducted on Dell DVD store application shows that webserver-2 is r the source of performance deviations, which is correct, as a memory-bug was injected into webserver-2 for 30 minutes.

Finally, we report the findings of test 5 conducted on the performance counter logs of a large telecom enterprise. The performance analysts agreed to the authenticity of the reduced but important set of performance counters recommended by our methodology (performance signature). They agreed that all 169 performance counters are needed and the crafted performance signature is a true representative of the system under test. Figure 8 is a line plot that shows the comparison between the important performance counters of the base-line load test with that of the other 4 tests. Load tests B and C are found very similar to that of the baseline test-A. The test D & E are found deviated from the baseline, which confirms the findings of the performance analysts. TABLE VIII provides the statistical evidence of the findings. Furthermore, our methodology pinpointed the web server to be the source of performance deviation in both the test D and E and performance analysts acknowledged the findings of our methodology. Once our methodology pinpointed the culprit subsystem i.e., the web server, the performance counters representing the deviated resource were further identified by comparing the signatures crafted by our methodology for the web server of load test D&E with that of baseline load test. *The Packet Outbound Discarded*, *Packets Sent/sec* and *Message Queue length* performance counters were notably deviated from the base-line counters. It was found that the web server was unable to get a connection to the remote database server due to some network constraints, hence was unable to write messages to the store. This caused an enormous message queue build-up at web server. Once the queue got full, the web server silently started to drop the messages without any notifications. This is a common behavior of an application during resource saturation or under stress. By pinpointing the web server as the source of performance deviation in load tests along with an indication of its deviated performance counters, performance analysts had no problem in piecing together the cause of the performance deviation of the load test from its base-line test.

> Our methodology can help performance analysts to automatically pinpoint the subsystem that is the source of performance deviation in a load test with an accuracy of 79%.

A. *The effect of sampling interval on our methodology*

There is no universal sampling rate set for logging performance counter data during load testing. If the load test tends to span over multiple days, an analyst will prefer a large sampling interval, i.e., either every minute or every five minutes, to keep the growth of the performance log under control. However, if the load test takes only a couple of hours, a performance analyst may tend to sample the system under test more aggressively, i.e., every 5 sec. We wanted to find out if the sampling interval has any effect on the performance of our methodology.

Approach: To evaluate the effectiveness of our methodology we used two metrics: precision and accuracy. Precision is the ratio between the correctly identified faults and all the predicted faults. For example, predicting the set {A,B,C,D,E} when only A and B are faulty subsystems gives a precision of *40%*. Accuracy is the second metric we use in our evaluation. A result is accurate when all subsystems that are the root cause of performance deviations are pinpointed correctly. If 5 faults are injected into a subsystem {A,B,C,D,E} and our methodology only identifies 4 subsystems to be faulty then it achieved *80%* of accuracy.

We performed three load tests on the dell DVD application using the base-line workload. Each test was 1 hour long. For the first test, the performance counters were sampled at the rate of 15 sec. For the second test sampling interval was set to 30 sec and finally for the third test the sampling interval was 1 minute. We randomly choose a subsystem and manually injected a fault after every 5 minute and kept a note of the name of the subsystem and time when the fault was injected. The fault was injected by stressing the CPU of the subsystem for 30 seconds. Once the load test is finished, we divided the performance counter logs into five minutes intervals [22]. We had in total 24 performance counter logs for each type of sampling interval. We applied our methodology on each of the performance counter logs to find out how effective our methodology was in pinpointing the subsystems that are the cause of performance deviations in the load tests.

Findings: TABLE IX shows that our methodology is able to achieve 82% precision and 79% accuracy with its suggestions. The sampling interval of performance counter data does affect the accuracy of our methodology. The methodology performs better at smaller sampling intervals.

> The performance of our methodology increases when the performance counters are sampled at smaller intervals

VII. RELATED WORK

Most of the work in literature is post-deployment centric, focusing on automatic problem diagnosis techniques for enterprise systems. The main aim of these post-deployment centric approaches is on alleviating the prohibitive cost of downtime by continuously monitoring important software systems and diagnosing the root-cause of failure when they occur. Our work is pre-deployment centric and aims to uncover performance problems in a load test. In particular, it pinpoints the subsystems that are the cause of the performance problem.

The closest work to ours is that of Jiang et al. [7] to automate the performance analysis of a load test. Unlike our work, they rely on executions logs. Though the execution logs capture the detailed event information and provide finer granularities than that of the performance counter logs, however, they are vendor and application specific. This means, that the different subsystems in an LSS (web servers, databases and mail servers) produce a variety of execution logs, each with different levels of information and formats. There is no single person of an LSS that has knowledge of all its subsystems. Analogous to performance counter logs, which provide a greater level of unification across subsystems in an LSS; it is impractical for an analyst to skim through wide verity of detailed information of executions logs with limited knowledge. Jiang et al. work is the only work that has been incorporated into the load testing domain to detect automatic performance problems in a load test. However their work does not explicitly pinpoint the subsystems that are the cause of performance problems in a load test.

Sandeep et al. work is second closest to ours [27]. They used 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 into a smaller set to describe the workload. In addition, their work is partially automated and requires continuous training to produce accurate results. Huck and Malony proposed a performance data mining framework for large-scale parallel computing. The framework tries to manage data complexity by using techniques such as clustering and dimensionality reduction [30]. This data mining framework utilizes random linear projections and PCA to reduce performance data. The framework does not transform the PCs back to individual performance counters. Cohen [27] developed an application signature based on the various system metrics (like CPU and memory).

Few researchers have exploited static dependency models to capture the dynamic complexity of large systems [8, 31-33]. They use these dependency models to describe 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 limitation of such a dependency model is the difficulty of generating and maintaining an accurate model of a constantly evolving large system. The second limitation is that they typically only model a logical system, and do not distinguish among replicated components but in a large enterprise system, there will be many replicated components.

Pinpoint and *Magpie* track communication dependencies with the aim of isolating the root-cause of misbehavior; they require instrumentation of the application to tab client requests [11, 34]. Our methodology does not require any instrumentation of the system. *Magpie* characterizes transaction resource footprints in fine detail but requires that the application logic be meticulously encoded in an "event Schema". Unlike Magpie, our methodology does not require any system knowledge. *Pip* aims to infer paths and requires an explicit specification of the expected behavior of a system [12]. Our methodology does not require such explicit specifications of the expected behavior. It relies heavily on statistical methods to automatically extract the expected behavior for baseline tests. *X-trace* uses application-level instrumentation to determine paths in network protocols [35]. Application level instrumentation obscures the performance of an application during load test and may cause the system performance to deviate from the base-line test. Most of the work described cannot be directly plugged into performance analysis of load testing with little or no modification.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented our methodology to pinpoint the subsystems that are likely the real cause of performance deviations in a load test. Our methodology uses Principal Component Analysis, to reduce the large volume of performance counter data. Furthermore, our methodology crafts the performance signatures for each LSSs subsystem by ranking performance counters based on their relevance for the load test. The performance signature changes for respective subsystems of a system if any error or deviation from the normal behavior occurs. Further, the methodology helps performance analyst to pinpoints the subsystems that are the likely cause of performance deviation by ranking them by their extent of performance deviations criticality. A case study on a real-world industrial software system and on an open source bench-mark application provides empirical evidence on the ability of our methodology to pinpoint the subsystems in a load test that are the likely cause of performance degradation without implying any domain knowledge.

Our technique cannot be generalized to other domains such as network traffic and security monitoring. This is due to the fact that there is no guarantee that the directions of maximum variance will contain good variables for discrimination. A large anomaly may inadvertently pollute the normal subspace, thereby skewing the assumption that large variances always have important dynamics.

As future work, we also plan to compare the performance of our methodology with that of other techniques such as

Naïve-bayes classifier and factor analysis to yield further improvement in constructing effective performance signatures. Currently, our methodology cannot distinguish between the two independent problem causes for set of subsystems that are tightly coupled and are always operated together. We are currently incorporating techniques that can differentiate between the tightly coupled subsystems.

REFERENCES

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," Proceedings of the IEEE, vol. 68, pp. 1060-1076, 1980.

[2] M. Woodside, G. Franks and D. C. Petriu, "The future of software performance engineering," in FOSE '07: 2007 Future of Software Engineering, 2007, pp. 171-187.

[3] E. J. Weyuker and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study," IEEE Trans.Softw.Eng., vol. 26, pp. 1147-1156, 2000.

[4] D. Thakkar, A. E. Hassan, G. Hamann and P. Flora, "A framework for measurement based performance modeling," in WOSP '08: Proceedings of the 7th International Workshop on Software and Performance, Princeton, NJ, USA, 2008, pp. 55-66.

[5] S. Stephen, "PayPal hit by global outage," in 2009, . Web:http://www.zdnet.co.uk/news/it-strategy/2009/08/04/paypal-hit-by-global-outage-39705017/. Downloaded August, 2010

[6] B. Beizer, Software System Testing and Quality Assurance. 1984.

[7] Zhen Ming Jiang, A. E. Hassan, G. Hamann and P. Flora, "Automated performance analysis of load tests," in Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, 2009, pp. 125-134.

[8] Jaesung Choi, Myungwhan Choi and Sang-Hyuk Lee, "An alarm correlation and fault identification scheme based on OSI managed object classes," in Communications, 1999. ICC '99. 1999 IEEE International Conference on, 1999, pp. 1547-1551 vol.3.

[9] I. Rouvellou and G. W. Hart, "Automatic alarm correlation for fault identification," in INFOCOM '95. Fourteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Bringing Information to People. Proceedings. IEEE, 1995, pp. 553-561 vol.2.

[10] A. T. Bouloutas, S. Calo and A. Finkel, "Alarm correlation and fault identification in communication networks," Communications, IEEE Transactions on, vol. 42, pp. 523-533, 1994.

[11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, 2002, pp. 595-604.

[12] F. Mattosinho, "Pip: Detecting the Unexpected in Distributed Systems" Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3, San Jose, CA, 2006.

[13] S. Pertet, R. Gandhi and P. Narasimhan, "Fingerpointing correlated failures in replicated systems," in Proceedings of the Second Workshop on Tackling Computer Systems Problems with Machine Learning, 2007.

[14] S. Pertet, R. Gandhi and P. Narasimhan, "Group communication: Helping or obscuring failure diagnosis," Group Communication: Helping Or Obscuring Failure Diagnosis, 2006.

[15] Z. M. Jiang, A. E. Hassan, G. Hamann and P. Flora, "Automatic identification of load testing problems," in IEEE International Conference on Software Maintenance, 2008. ICSM 2008, 2008, pp. 307-316.

[16] Tech Report: M. Qadir, "Role of Automation in Computer-based Systems",http://www.cs.rutgers.edu/~rmartin/teaching/spring06/cs553/papers/008.pdf.

[17] M. W. Knop, P. K. Paritosh, P. A. Dinda and J. M. Schopf, "Windows performance monitoring and data reduction using WatchTower and argus (extended abstract)," in Proceedings of SHAMAN, 2001, .

[18] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly and A. Fox, "Capturing, indexing, clustering, and retrieving system history," SIGOPS Oper.Syst.Rev., vol. 39, pp. 105-118, 2005.

[19] 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 OSDI'04: Proceedings of the 6th Conference on Symposium on Operting Systems Design \& Implementation, San Francisco, CA, 2004, pp. 16-16.

[20] Malik, H., Jiang, Z. M., Adams, B, Hassan, A. E., " Automatic Comparison of Load Tests to Support the Performance Analysis of Large Enterprise System," in proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010). Madrid, Spain. March 16-18, 2010.

[21] Haroon Malik, Bram Adams, Ahmed E. Hassan, Parminder Flora, Gilbert Hamann, "Using load tests to automatically compare the subsystems of a large enterprise system," in proceedings of the 34th Computer Software and Applications Conference (COMPSAC 2010), Seoul, Korea, 2010.

[22] I. Jolliffe, Principal Component Analysis. Springer verlag, 2002.

[23] Z. Guo, G. Jiang, H. Chen and K. Yoshihira, "Tracking probabilistic correlation of monitoring data for fault detection in complex systems," in DSN '06: Proceedings of the International Conference on Dependable Systems and Networks, pp. 259-268, 2006.

[24] G. Jiang, H. Chen and K. Yoshihira, "Modeling and Tracking of Transaction Flow Dynamics for Fault Detection in Complex Systems," IEEE Trans.Dependable Secur.Comput., vol. 3, pp. 312-326, 2006.

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

[26] B. A. Rosner, Fundamentals of Biostatistics. Duxbury Resource Center, 2006.

[27] S. R. Sandeep, M. Swapna, T. Niranjan, S. Susarla and S. Nandi, "CLUEBOX: A Performance Log Analyzer for Automated Troubleshooting," WASL, San Diego, CA, 2008.

[28] Leyda, M. and Geiss, R., "WinThrottle," Downloaded 2010.

[29] J. McCaffrey, "Test Run: Stress Testing," .

[30] K. A. Huck and A. D. Malony, "PerfExplorer: A performance data mining framework for large-scale parallel computing," in Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, 2005, pp. 41-41.

[31] A. Brown, G. Kar and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment," in Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium , pp. 377-390, 2001.

[32] B. Gruschke, "A new approach for event correlation based on dependency graphs," in 5th Workshop of the OpenView University Association, 1998 .

[33] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini and D. Ohsie, "High speed and robust event correlation," Communications Magazine, IEEE, vol. 34, pp. 82-90, 1996.

[34] P. Barham, A. Donnelly, R. Isaacs and R. Mortier, "Using magpie for request extraction and workload modelling," in Symposium on Operating Systems Design and Implementation, pp. 259–272, 2004.

[35] R. Fonseca, G. Porter, R. H. Katz, S. Shenker and I. Stoica, "X-trace: A pervasive network tracing framework," in Networked Systems Design and Implementation, 2007, .

[36] M. Kryczka, R. Cuevas, C. Guerrero, E. Yoneki and A. Azcorra, "A First Step Towards User Assisted Online Social Networks," 2010.

[37] J. Gray, "Dependability in the internet era," in Keynote Presentation at the 2nd HDCC Workshop, 2001, .

[38] J. L. Hennessy, D. A. Patterson, D. Goldberg and K. Asanovic, Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2003.