# Automated Performance Analysis of Load Tests

Zhen Ming Jiang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, ON, Canada
{zmjiang, ahmed}@cs.queensu.ca

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

## Abstract

*The goal of a load test is to uncover functional and performance problems of a system under load. Performance problems refer to the situations where a system suffers from unexpectedly high response time or low throughput. It is difficult to detect performance problems in a load test due to the absence of formally-defined performance objectives and the large amount of data that must be examined.*

*In this paper, we present an approach which automatically analyzes the execution logs of a load test for performance problems. We first derive the system's performance baseline from previous runs. Then we perform an in-depth performance comparison against the derived performance baseline. Case studies show that our approach produces few false alarms (with a precision of* 77% *) and scales well to large industrial systems.*

## 1. Introduction

Many systems ranging from e-commerce websites to telecommunication infrastructures must support concurrent access by hundreds or thousands of users. Many of the field problems are related to systems not scaling to field workloads instead of feature bugs [9, 34]. To assure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, such as unit and integration testing.

Load testing, in general, refers to the practice of assessing a system's behavior under *load* [18]. A load is typically based on an operational profile, which describes the expected workload of the system once it is operational in the field [13, 15]. A load consists of the types of executed scenarios and the rate of these scenarios. 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.

A load test usually lasts for several hours or even a few days. Load testing requires one or more load generators which mimic clients sending thousands or millions of concurrent requests to the application under test. During the course of a load test, the application is monitored and performance data along with execution logs are recorded. Performance data store resource usage information such as CPU utilization, memory, disk I/O and network traffic. Execution logs store the run time behavior of the application under test.

The goal of a load test is to uncover functional and performance problems under load. Functional problems are often bugs which do not surface during the functional testing process. Deadlocks and memory management bugs are examples of functional problems under load. Performance problems often refer to performance issues like high response time or low throughput under load.

Existing load testing research focuses on the automatic generation of load test suites [13, 14, 15, 17, 20, 25, 35]. There is limited work, which proposes the systematic analysis of the results of a load test to uncover potential problems. Unfortunately, looking for problems in a load test is a time-consuming and difficult task. Our previous work [28] flags possible functional problems by mining the execution logs of a load test to uncover dominant execution patterns and to automatically flag functional deviations from this pattern within a test.

In this paper, we introduce an approach that automatically flags possible performance problems in a load test. We cannot derive the dominant performance behavior from just one load test as we did in our previous work, since the load is not constant. A typical workload usually consists of periods simulating peak usage and periods simulating off-hours usage. The same workload is usually applied across load tests, so that the results of prior load tests are used as an informal baseline and compared against the current run. If the current run has scenarios which follow a different response time distribution than the baseline, this run is probably troublesome and worth investigating. The main contributions of our work are as follows:

1. To the best of our knowledge, our approach is the first

work to automatically detect performance problems in the load testing results.
2. Our approach makes use of readily available execution logs, which avoids the need of performance impacting instrumentation [16, 22] .
3. Our approach automatically reports scenarios with performance problems and pin-points the performance bottlenecks within these scenarios. Case studies show that our approach scales well to large systems and produces few false alarms (with a precision of 77%).

### Organization of the Paper

The paper is organized as follows: Section 2 describes the current practice of performance analysis and its limitations. Section 3 shows an example of how a load tester can use our performance analysis report to analyze the performance of a load test. Section 4 describes our performance analysis approach. Section 5 presents three case studies on our performance analysis approach: two on open source applications and one on a large enterprise application. Section 6 presents some discussions and future work. Section 7 describes related work. Section 8 concludes the paper.

## 2 Current Practice of Performance Analysis

Most large enterprise applications must be load tested to ensure satisfactory performance under load. Unfortunately, looking for performance problems in a load test is a time-consuming and difficult task, due to many of the following challenges:

**Lack of Documented Performance Baselines:** Correct and up-to-date system requirements rarely exist [32].

**Time Pressure:** Load testing is usually the last step in an already delayed release schedule which managers must speed up. Hence the time allocated to run a test, which could be days, and analyze the results is usually limited.

**Monitoring Overhead:** Approaches which monitor or profile an application affect the validity of performance measurements and are not practical for load testing.

**Large Volume of Data:** A load test, running for a few hours, generates performance data and logs that are several hundred megabytes in size. This data must be analyzed to uncover any problems in the load test.

Due to the above challenges, a load tester often uses the results of a prior load test as an informal baseline and performs high level performance checks. The following two checks are commonly used in practice:

**Metrics check:** A load tester examines performance metrics for large fluctuations. For example, an up-trend for the memory usage throughout a load test is a good indicator of a memory leak.

**Response Time Check:** Using domain knowledge, a load tester checks the response time of a few key scenarios and compares them against an informal baseline.

We believe this current practice is not efficient since it takes hours of manual analysis, nor is it sufficient since it might miss problems showing up in other scenarios.

## 3 Performance Analysis Report

The previous section discussed the challenges and limitations of the current performance analysis practices. In response, we have developed an automatically generated report, which can uncover potential performance problems. The report extracts information from the readily available execution logs and uses a previous run as an informal performance baseline to compare against.

Consider the following example. Jack, a load tester, is asked to load test a new version of an online application. He needs to determine whether the application can scale to hundreds of concurrent client requests just like the earlier version did. The application is an online pet store which supports operations like login/logout, browse, purchase and registration for new users. We now demonstrate how Jack can use our performance analysis report to uncover performance problems based on the run he did on an early release.

### Overall Performance Summary

Since extra instrumentation or profiling of the application may slow it down and affect the performance measurement, Jack has to work with readily available execution logs and performance metrics.

As shown in Figure 1(a), from millions of log lines in the execution logs, our report flags 3 scenarios whose performance is statistically different than the previous run. Each row in the table corresponds to one performance deviated scenario. Among these three scenarios, there are two scenarios which are worse than the previous run (shown in red) and one scenario which is better (shown in blue). Scenarios are sorted by the degree of performance deviations from the previous run so that Jack can make best use of his time by working from the top.

Looking at the first row, Jack discovers that the first scenario has around the same throughput ($5,000$ versus $5,002$ requests per second) from both runs. However, he notices that the current scenario performance is worse than the previous run, indicated by the red colour under the deviation value 0.8. The event sequence triggered by this scenario is also displayed. Each event is abbreviated using its abstracted event id ($E_1$, $E_2$, $E_3$, and $E_4$).

### Visual Performance Comparison

After gaining an overall impression about the system performance, Jack clicks the hyperlink under the deviation value to dig deeper into the performance of the first scenario. As shown in Figure 1(b), the report expands and shows a visual comparison of this scenario's performance

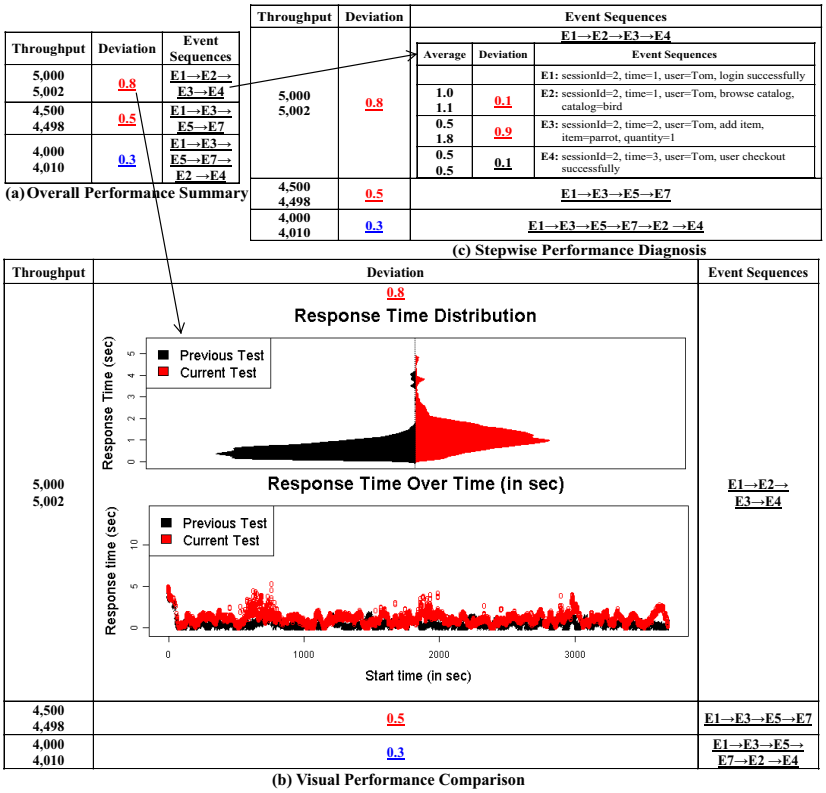


**Figure 1. An example performance analysis report**

between the two runs. He examines the two graphs seeking to answer the following two questions:

**1. How does the scenario performance differ?**

The top graph in Figure 1(b) is a beanplot [29], which visualizes the response time distributions of the same scenario in the previous and current runs side-by-side. The left side shows the response time distribution from the previous run and the right side shows the current run. The width of the plot indicates the frequency. For example, most of the instances in the current run take about 1 second and around 0.5 second in the previous run.

After examining the beanplot, Jack now has a better idea of why this scenario is flagged: First, the majority of the cases from the current run are slower than the previous run (1 second versus 0.5 second). Second, the maximum response time from the current run is 5 seconds compared with 4 seconds from the previous run.

**2. How does the performance evolve over time?**

The bottom graph in Figure 1(b) shows whether the system performance has degraded over the course of the load test. The horizontal axis indicates the start time of a scenario instance. The vertical axis indicates the response time of the scenario instances triggered at this moment. If more than one instances from this scenario are trigged at the same moment, the average response time from these instances is used.

For the first scenario, there are no performance degradations even though the response time fluctuates over time. Most of the time, the current run (in red) is above the previous run (in black), which again indicates that the current run is worse (i.e. slower).

## Stepwise Performance Diagnosis

Once Jack gets a visual comparison of the performance differences for the first scenario, Jack clicks the link below the event sequences to find out the exact cause for the performance deviations (Figure 1(c)). Sample log lines from the execution logs are shown along with the average durations between the adjacent steps. In addition, deviations from the adjacent event pairs between the previous and current runs are also shown.

For the investigated scenario, Jack concludes that the scenario performance degradation is mostly caused by a slow down between the browsing ($E_2$) and purchasing ($E_3$) events. The average duration of this event pair jumps from 0.5 second in the previous run to 1.8 seconds in the current run. The deviation between these two events, flagged (in red), is the highest deviation (0.9) among all the event pairs

in this scenario. Jack then clicks the hyperlink below the 0.9 deviation to get a visual comparison of the performance differences in the $E_2 \rightarrow E_3$ pair. The report is again expanded to display similar graphs as in Figure 1(b) but for specific event pairs.

### Reporting

Based on the analysis of the first scenario, Jack concludes that there is a performance problem between the browsing and purchasing events. He then performs similar analysis on the other two scenarios. Jack can now compose an email, which explains the performance problems. The email is sent to the appropriate developers with the performance analysis report attached.

## 4. Our Approach to Create the Performance Analysis Report

We now present our approach to generate the performance analysis report discussed in Section 3. As shown in Figure 2, our approach consists of five phases. For both tests, we conduct log abstraction, sequence performance recovery and performance summarization. Then we flag the performance deviating scenarios by statistically analyzing the recovered performance data. Finally, a performance analysis report, ranked by the degree of performance deviation, is generated.

We explain our approach using the running example shown in Table 1.

### Phase 1. Log Abstraction

Since additional instrumentation or profiling of the system slows down its execution, these techniques are not feasible for load testing analysis. As a result, our approach analyzes the readily available execution logs. These logs are commonly available to enable remote support or to cope with legal acts such as the "Sarbanes-Oxley Act of 2002" [5]. Such logs record software activities (e.g. "User authentication successful") and errors (e.g. "Failed to retrieve customer profile").

Table 1(a) shows the first 8 log lines from an execution log file. These logs are hard to parse and analyze automatically since they are in free-form with no strict format and use a large vocabulary of words. Logs contain a great deal of redundancy. For example, even though the first and second log lines are triggered by the same event in the code, they are different yet have similar structure to each other. We aim to remove the redundancy in the logs by abstracting log lines into execution events. We have developed a technique [27] which exploits the common structure among the log lines: each log line is a mixture of static and dynamic information. The static information describes the execution event (i.e., the context), whereas the varying parts (i.e., dynamic) are parameter values generated at runtime. Dynamic information, specific to the particular occurrence of an event, causes the same execution event to result in different log lines.

The log lines in Table 1(a) are abstracted into 6 execution events shown in Table 1(b). The "$v" sign indicates a dynamic value.

### Phase 2. Sequence Performance Recovery

As the system handles concurrent client requests, log lines from different scenarios are intermixed with each other in the execution logs. As shown in Table 1(a), scenarios related to users Tom, John and Mike are mangled together. Furthermore, some log lines (e.g., the 8th log line), which only output the system status information, are not related to any customer scenarios. These log lines should be filtered from our performance analysis.

In order to recover the performance from all the customer scenarios, we need to first recover the event sequences. We recover the sequences by linking the appropriate parameter values.

The first phase abstracts each log line into an execution event. Here, we use the dynamic values ($v) as well as their parameter names in the execution event. The parameter name-value pairs are used to link the related log lines into sequences. For example, the first line contains 3 parameter name-value pairs: sessionId with value 1, time with value 1, and user with value $John$.

Table 1(c) shows the results after extracting and linking information from the log lines using the sessionId values of the log lines in Table 1(a). For example, the second ($E_1$), third ($E_2$), fourth ($E_3$), and sixth ($E_4$) log lines all share the same sessionId (2), so we group them together. The 8th log line in Table 1(a) corresponds to a periodic health check event, which has no sessionId and is used to determine if the system is alive. This log line is filtered out.

The performance of the recovered sequence is calculated by taking the time difference between the first and the last events. In our running example, the overall response time of the sequence ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$) with $sessionId = 2$ is 2 seconds. Since there is only one log line with sessionId 3, there is no duration information for this session.

The durations between every adjacent event pair are also calculated. The pairwise duration information is used later for stepwise performance diagnosis. The time duration between the first two events ($E_1$ from the 2nd log line and $E_2$ from the 3rd log line) is 0 seconds, and so on.

### Phase 3. Performance Summarization

The previous phase has recovered the performance from the individual sequences. Identical event sequences correspond to the same scenario. As a load test repeatedly executes scenarios over and over, the sequence $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$ can appear multiple times. In the third phase, we summarize the performance of each scenario.

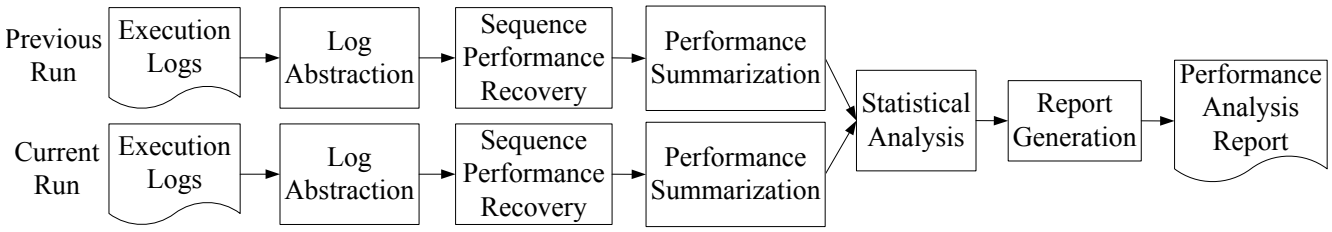


**Figure 2. Our Approach to Generate the Performance Analysis Report**

Tables 1(a), 1(b) and 1(c) only process the first 8 log lines from the logs. Execution logs usually contain millions of lines. Table 1(d) shows an example of the summarized scenario performance using the entire execution logs. In total, we have 2 scenarios. The first scenario ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$) occurs 300 times. Among them, 100 times occurs with a duration of 2 seconds and 200 times with 3 seconds. The performance of the adjacent event pairs from each scenario is also summarized. For example, the event pair $E_2 \rightarrow E_3$ from the first scenario takes on average 1.7 seconds. Among the total 300 occurrences of this event pair in this scenario, it takes 1 second in 100 times and 2 seconds in the other 200 times.

### Phase 4. Statistical Analysis

Phrase 3 uncovers the performance of all the scenarios from the previous and current runs. The fourth phase evaluates the overall performance of each scenario and pin-points the performance deviating event-pairs.

We compare each scenario's performance in the previous and current runs using a statistical test. We use the unpaired student-t test to compare the scenario response time distributions from the previous run against the current run. Furthermore, we use a modified student-t test [26], which outputs a confidence interval. Compared with hypothesis testing, whose output only answers whether the two distributions are statistically the same, a confidence interval also provides possible ranges. Depending on the relative position of the confidence interval to zero, we can tell which run has better performance (i.e. timing) for this scenario. We only show the scenarios, whose performance is statistically different between the two runs, in our report.

Once the scenarios with deviated performance are flagged, we need to pin-point the event pairs which cause the performance deviations. We achieve this by applying the same statistical test on all the adjacent event pairs. For example, for the flagged scenario ($E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$), we compare the performance of $E_1 \rightarrow E_2$, $E_2 \rightarrow E_3$, and $E_3 \rightarrow E_4$ between the previous and the current runs.

### Phase 5. Report Generation

As the load tester has limited time, we rank the potentially troublesome scenarios to help him prioritize his time. We rank each scenario by the degree of the performance deviation between the previous and current runs.

Cosine distance, which measures the degree of similarity between two distributions, outputs a value between 0 and 1. If the two distributions are very similar, the cosine distance is close to 1. If they are very different, the cosine distance is close to 0. As deviation is the opposite of similarity, we use the following formula to calculate the deviation:

$$deviation(P, C) = 1 - cosine(P, C) \quad (1)$$

$$cosine(P, C) = \frac{\sum_x P(x)C(x)}{\sqrt{\sum_x P(x)^2}\sqrt{\sum_x C(x)^2}} \quad (2)$$

Note that P(x) and C(x) correspond to the number of instances in the previous and current runs which have response time x for a particular scenario/ For example, if the cosine distance of the first scenario in Table 1(d) is $0.2$, their deviation value is $0.8$.

## 5. Case Studies

We conducted 3 case studies on 3 different systems (2 open source applications and 1 large enterprise application). We seek to verify whether our performance analysis report can help load testers in their usual tasks. As our approach only flags the performance deviated scenarios, it is up to his knowledge to decide whether the deviated performance leads to performance problems.

We use precision to evaluate the performance of our approach. The precision is defined as follows:

$$precision(\%) = (1 - \frac{\#\ of\ false\ alarm\ scenarios}{Total\ \#of\ flagged\ scenarios}) \times 100\%$$

Multiple scenarios with deviating performance can be caused by the same performance problem, which can be an indication of the impact of this problem. If a flagged scenario does not lead to a performance problem, then this scenario is considered as a false alarm.

A load tester conducts and analyzes a load test of an evolving system seeking to accomplish many of the following tasks: to recommend optimal system settings, to certify

| # | Log Lines |
|---|---|
| 1. | sessionId=1, time=1, user=John, login successfully |
| 2. | sessionId=2, time=1, user=Tom, login successfully |
| 3. | sessionId=2, time=1, user=Tom, browse catalog, catalog=bird |
| 4. | sessionId=2, time=2, user=Tom, add item, item=parrot, quantity=1 |
| 5. | sessionId=1, time=3, user=John, browse catalog, catalog=dog |
| 6. | sessionId=2, time=3, user=Tom, user checkout successfully |
| 7. | sessionId=3, time=4, user=Mike, user search item, item=rattlesnake |
| 8. | time=4, health check |

**(a) Example Execution Log Lines**

| Event ID | Event Template | # |
|---|---|---|
| E1 | sessionId=$v, time=$v, user=$v, login successfully | 1, 2 |
| E2 | sessionId=$v, time=$v, user=$v, browse catalog, catalog=$v | 3, 5 |
| E3 | sessionId=$v, time=$v, user=$v, add item, item=$v, quantity=$v | 4 |
| E4 | sessionId=$v, time=$v, user=$v, user checkout successfully | 6 |
| E5 | sessionId=$v, time=$v, user=$v, user search item, item=$v | 7 |
| E6 | time=$v, health check | 8 |

**(b) Abstracted Execution Events and Corresponding Log Lines**

| sessionId | Event Sequences | Durations |
|---|---|---|
| 1 | E1→E2 | 2 |
| | ⇨ E1→E2 | 2 |
| 2 | E1 →E2→E3→E4 | 2 |
| | ⇨ E1→E2 | 0 |
| | ⇨ E2→E3 | 1 |
| | ⇨ E3→E4 | 1 |
| 3 | E5 | |

**(c) Recovered Sequences with Durations Calculated**

| Event Sequences | Durations |
|---|---|
| E1 →E2→E3→E4 | 2 (100), 3 (200) |
| ⇨ E1→E2 (0) | 0 (300) |
| ⇨ E2→E3 (1.7) | 1 (100), 2 (200) |
| ⇨ E3→E4 (1) | 0 (100), 1 (100), 2 (100) |
| E1 →E3→E5→E7→E4 | 3 (200) |
| ⇨ E1→E3 (0.5) | 0 (100), 1 (100) |
| ⇨ E3→E5 (1) | 1 (200) |
| ⇨ E5→E7 (1) | 1 (200) |
| ⇨ E7→E9 (0.5) | 0 (100), 1 (100) |

**(d) Summarized Response Time for Each Scenario**

**Table 1. The Running Example**

software/hardware platforms, and to study the impact of design changes. We use this to structure our case study. For each study, we first give a brief description of the studied system and an overview of the task which the load tester needs to complete. Then we explain the steps to conduct the experiments and data collection procedure. Finally, we present our results and conclusions.

Our performance analysis prototype is written in Perl and uses R [7] to generate the graphs.

## Task 1: Recommending Optimal Configurations

Enterprise systems interact often with other software components like databases or mail servers. Sub-optimal configurations of these components can impact the performance of the system. A load tester needs to explore these configurations and provide recommendations for deployment. In this task, we use the Dell DVD Store (DS2) as our case study system.

### Studied System: the Dell DVD Store

The Dell DVD Store (DS2) application is an open source online application used for benchmarking Dell hardware. [6]. DS2 provides basic e-commerce functionality, including: user registration, user login, product search, and item purchase.

DS2 contains a database, a load generator and a web application. It comes in different distribution packages which support various web platforms (e.g. Apache Tomcat or JBoss) and database vendors (MySQL, Microsoft SQL Server, and Oracle). The web application consists of four JSP pages which interact with the database and display dynamic contents. The DS2 load generator supports a range of configuration parameters to specify the workload. In this task, we use MySQL as the backend database and Apache Tomcat as our web server engine.

### Goal: Recommending Optimal MySQL Configuration

We seek to evaluate the performance impact due to the following two software configuration options for the MySQL database:

**Caching:** Whether to cache the query results or not;

**Storage Engines:** Whether to use MyISAM or InnoDB as the storage engine.

### Experiments and Data Collections

DS2 has no logs, thus we manually instrumented its four JSP pages. We run 4 one-hour load tests: InnoDB with and without query caching, and MyISAM with and without caching. All 4 runs are exercised under the same workload and are all conducted on a single core CPU machine with 1 GB of RAM and a $5,400$ rpm hard disk. Each test generated over $120,000$ log lines.

### Result Analysis and Conclusions

We conduct the performance analyses on the results of these $4$ runs.

To determine the performance impact of caching, we perform two comparisons. First, we compare the results from InnoDB with/without caching, then MyISAM with/without caching. We use the session ids to recover the sequences. Both of our performance analysis reports have flagged $18$ performance deviated scenarios. Examining the stepwise performance diagnosis sub-tables across all the event pairs, we find events, which invoke database operations, perform better with caching enabled.

To determine the performance impact of different storage engines, we compare the results from MySQL configured with MyISAM engine against those with the InnoDB engine. Since our previous analysis shows that enabling caching yields better performance, we only compare the two runs with caching enabled. Again, 18 scenarios are flagged. By investigating stepwise performance diagnosis sub-tables, we find database-related events perform better with the InnoDB engine. Our finding agrees with prior benchmark studies [4].

Using our performance analysis reports, we conclude that DS2 should be deployed with the following MySQL configurations to achieve optimal performance: InnoDB as the storage engine and cache-enabled. The precision is 100% among all three performance analysis reports.

## Task 2. Certifying Software/Hardware Platforms

Nowadays, systems support various software platforms (e.g., operating systems) as well as hardware platforms. A load tester needs to check whether a system under test can make optimal use of the available services and resources. In this task, we use the JPetStore as our case study system.

### Studied System: JPetStore

JPetStore [2] is a larger and more complex open source web application relative to DS2. Unlike Sun's original version of Pet Store which is more focused on demonstrating the capability of the J2EE platform, JPetStore is a re-implementation with a more efficient design [3] and is targeted on benchmarking the J2EE platform against other web platforms such as .Net. Unlike DS2 which embeds all the application logic into the JSP code, JPetStore uses the "Model-View-Controller" framework and XML files for object/relational mappings. In this task, we have deployed the JPetStore application on Apache Tomcat and used MySQL as the database backend.

### Goal: Certifying a Multicore Server

In this case study, we seek to certify the hardware platform by checking whether there is a performance gain by migrating JPetStore from a single core machine to a more powerful multi-core server.

### Experiments and Data Collection

As JPetStore does not come with a load generator, we use Webload [8], an open source web load testing tool, to load test the application. Using Webload, we have recorded a single customer scenario for replay during load testing. In addition, we configure the Webload so that it incrementally increases the workload as the load test progresses.

On each hardware platform, we conduct one one-hour run. The first run uses one machine which has a single CPU with 1G of memory and one $5,400$ rpm hard-disk. The second run uses another machine which has a Quad-Core CPU with 8G of memory and one $7,200$ rpm hard-disk. Both runs generate over $770,000$ log lines.

### Result Analysis and Conclusions

Figure 3 shows the performance analysis report for JPetStore. The report has flagged our only scenario. The colour of deviation (blue) shows that the run on the Quad-core machine statistically out-performs the run on the single CPU machine. However, the visual comparison of the two runs reveals that:

1. The contrast of the maximum and minimum response time from both runs is very large, especially for the run on the multi-core server. As shown in the beanplot of Figure 3, the minimum response time for the Quad-core machine is around 30 seconds and maximum is around 200 seconds.
2. As shown in the lower graph of Figure 3, both runs slow down as the load test progresses due to the steady increase in the workload. However, the response time on the Quad-core machine during the last few seconds is about 3 times longer than the single core machine during heavy load (around 200 seconds on the Quad-core machine versus around 80 seconds on the single core machine).

We repeat both runs and still obtain the same patterns. Through close examination of the stepwise performance diagnosis sub-table in our report, we reveal a MySQL performance bug. The MySQL InnoDB storage engine has trouble scaling to multi-core machines. Our finding is confirmed by MySQL developer postings [1].

Using our performance analysis approach, we conclude that the JPetStore performance improves on a more powerful machine. However, there is a MySQL performance bug which prohibits the efficient use of the multi-core hardware architecture under heavy load. The precision of this report is 100%.

## Task 3. Studying the Impact of System Changes

Systems are constantly undergoing maintenance changes to fix bugs and to cope with new standards or new interfaces. Changes can cause suboptimal system performance. In this task, we use a large enterprise application as our case study system.

### Studied System: A Large Enterprise Application

The system we studied is a large distributed enterprise application. This application supports a large number of scenarios which are used by thousands of users simultaneously.

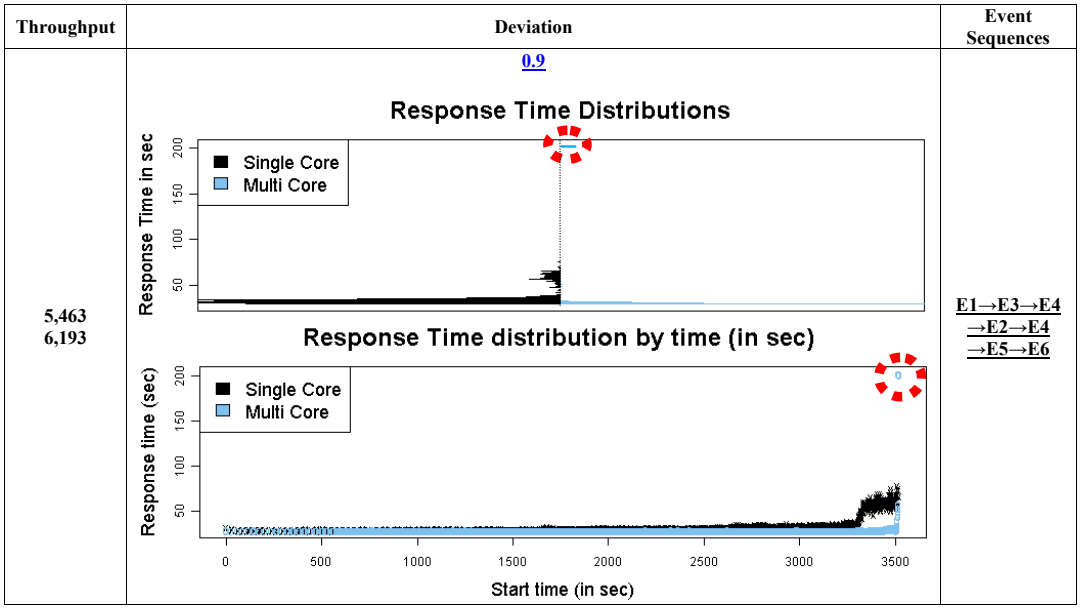


**Figure 3. The performance analysis report on JPetStore**

### Goal: Evaluating Different Software Designs

There is a new software build which uses a different communication mechanism. We want to determine the performance impact of the new design.

### Experiments and Data Collection

A load tester has conducted the experiments. Two 8-hour load tests are run on different software builds under the same load. Both runs generate over 2 million log lines.

### Result Analysis and Conclusions

This is a large enterprise application with readily available execution logs. After briefly going through the log lines, we find three main parameters which can be used to link log lines to form scenarios.

Our report flags around 30 performance deviating scenarios. Most of the scenarios are executed more than $4,000$ times. Our report reveals that the current build performs worse than the previous build. Examining the step-wise performance diagnosis sub-tables, this load tester is able to pinpoint the performance deviating steps and to subsequently identify the performance problems of the new design.

Our performance analysis report has a precision of $77\%$. A few flagged sequences are not scenarios. Thus, false alarms are caused by the random time intervals between the adjacent event pairs.

We conclude that our performance analysis report has helped the load tester evaluate the performance impact of different designs. The precision of our performance analysis report is $77\%$.

## 6. Discussion and Future Work

In this section, we discuss our results and future work.

### Performance Comparisons

Our approach uses results from a prior run as an informal performance baseline. The performance baseline can also be derived from the data in one or more runs. For example, we can form the performance baseline by combining the results from 10 runs conducted last year. The large resulting data is more desirable to infer the past common performance behavior.

We believe both the performance improving and degrading scenarios are worth investigating for two reasons. First, a load tester needs to verify the performance bug fixes by comparing the current run against a test which suffers from performance problems. In this case, the performance improvement cases are also of interest to him. Second, we do not want to miss potential performance problems. Take our JPetStore case study for example. Performance on a powerful server is generally better but suffers from severe slow down under stress conditions.

### Scenario Recovery

Our approach requires some manual work at the beginning as we need to find what are the identifying parameters in the logs. This is a one-time effort and requires minimal effort. These parameters can either be obtained by asking a domain expert or by skimming through the logs.

**Statistical Analysis**

Our approach uses a student-t test to compare the response time distributions. A student t-test is a parametric statistical test which assumes that the response time is normally distributed. Non-parametric statistical tests, like the Kolmogorov-Smirnov test [19], can also be used in our analysis. These tests hold no assumptions about the distribution of the data. Parametric tests are less strict than non-parametric tests. In other words, non-parametric tests tend to say there are no statistical differences between two data sets whereas the parametric tests would indicate a statistical significant difference. We choose to use the less strict student t-test, since we want to find all the possible performance problems. Furthermore, we have also tried using the Kolmogorov-Smirnov test to verify the findings in our case studies. We find that Kolmogorov-Smirnov test results agree with the student-t test. This finding confirms with the results reported by Bulej et al. [21].

**Analyzing System Performance Using Execution Logs**

Our approach uses readily available execution logs and infers system performance based on the time differences between log lines. Therefore, our approach is limited by the granularity of the log lines. Furthermore, we assume that logs are relatively stable over time. This assumption holds true for many industrial systems. Since the logs are already processed by many other tools, log changes are usually minimized. In the future, we plan to apply approximate matching on different sequences between runs.

**New Scenarios**

Since there are no formal performance objectives or data from previous runs for new scenarios, our current approach will not analyze them. In the future, we plan to infer the expected scenario performance based on existing scenarios.

## 7 Related Work

We discuss two areas of related work.

**Load Testing**

Most existing load testing research focuses on the automatic generation of load test suites [13, 14, 15, 17, 20, 25, 35]. Our previous work [28] focuses on automatically uncovering functional problems in a load test. Bulej et al. [21] propose the concept of regression benchmarking as a variant of regression testing for performance regression. Regression benchmarking compares the performance across different versions of the systems using the same benchmarking suite. Our work is an improvement over the regression benchmarking in four aspects. First, our approach recovers the scenarios automatically without specification. Second, our approach is more fine-grained, as we analyze the performance of the scenarios as well as the individual steps in each scenario. Third, our approach minimizes the amount of manual processing. Fourth, the reported problems in our performance analysis report are ranked so that a load tester can prioritize his efforts and make optimal use of his time.

**Automated Performance Monitoring and Analysis of Production Systems**

Automated performance monitoring and analysis of production systems can be divided into two subcategories: analyzing the performance metrics and analyzing the logs.

The following work analyzes the performance metrics: Avritzer et al. [12, 11] propose various algorithms to detect the need for software rejuvenation by monitoring the changing values of various system metrics. Mi et al. [23, 31] and Cohen et al. [24, 36] develop application signatures based on the various system metrics (like CPU, memory). The application metrics are further used for efficient capacity planning and anomaly detection. The main difference between these approaches and ours is that we use execution logs for our analysis. Compared with system metrics, execution logs provide more in-depth domain specific information.

The following work analyzes the readily logs with no additional instrumentations: Aguilera et al. [10, 33] developed various algorithms to perform black-box performance debugging on distributed systems. They use the header information on the TCP packet traces (source, destination and time) to infer the dominant causal paths through a distributed system. Unfortunately, the accuracy of the inferred causal paths decreases as the degree of parallelism increases. This is not ideal for load testing analysis as there are many message exchanges occurring simultaneously. Marwede et al. [30] use the timing anomaly to automatically uncover functional problems.

## 8. Conclusions

It is difficult to conduct a performance analysis of load testing results due to the absence of a documented performance baseline, time pressure, monitoring overhead and large volume of data. In this paper, we propose an approach which automatically flags possible problems by adopting a previous run as a performance baseline and comparing against it. Our approach is easy to adopt and scales well to large systems with high precision (77%).

## Acknowledgement

We are grateful to Research In Motion (RIM) for providing access to the enterprise application 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] Heikki Tuuri Innodb answers - Part I. http://tiny.cc/N8qVq.

[2] iBATIS JPetStore. http://sourceforge.net/projects/ibatisjpetstore/.

[3] Implementing the Microsoft .NET Pet Shop using Java. http://www.clintonbegin.com/JPetStore-1-2-0.pdf.

[4] InnoDB vs MyISAM vs Falcon benchmarks. http://tiny.cc/lPDFp.

[5] Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/.

[6] The Dell DVD Store. http://linux.dell.com/dvdstore/.

[7] The R Project for Statistical Computing. http://www.r-project.org/.

[8] WebLoad. http://www.webload.org/.

[9] Applied Performance Management Survey, Oct 2007.

[10] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[11] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker. Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In *the International Conference on Dependable Systems and Networks*, 2006.

[12] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring stable performance for systems that degrade. In *the 5th international workshop on Software and performance*, 2005.

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

[14] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, 1994.

[15] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.*, 21(9), 1995.

[16] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[17] M. S. Bayan and J. W. Cangussu. Automatic stress and load testing for embedded systems. In *30th Annual International Computer Software and Applications Conference*, 2006.

[18] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, March 1984.

[19] S. Boslaugh and D. P. A. Watters. *Statistics in a Nutshell A Desktop Quick Reference*. O'Reilly, 2008.

[20] L. C. Briand, Y. Labiche, and M. Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2), 2006.

[21] L. Bulej, T. Kalibera, and P. Tůma. Repeated results analysis for middleware regression benchmarking. *Perform. Eval.*, 60(1-4):345–358, 2005.

[22] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[23] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *IEEE International Conference on Dependable Systems and Networks*, 2008.

[24] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[25] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on uml models. In *Proceedings of the 28th international conference on Software engineering*, 2006.

[26] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal on Software Maintenance and Evolution: Research and Practice*, 20(4), 2008.

[28] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *In Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008.

[29] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1), 2008.

[30] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In *the 13th European Conference on Software Maintenance and Reengineering*, 2009.

[31] N. Mi, L. Cherkasova, K. M. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *Network Operations and Management Symposium*, 2008.

[32] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Los Alamitos, CA, USA, 1994.

[33] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, 2006.

[34] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12), 2000.

[35] J. Zhang and S. C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Softw. Pract. Exper.*, 32(15), 2002.

[36] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.