

# Mining Performance Regression Testing Repositories for Automated Performance Analysis

King Chun Foo<sup>1</sup>, Zhen Ming Jiang<sup>2</sup>, Bram Adams<sup>2</sup>, Ahmed E. Hassan<sup>2</sup>, Ying Zou<sup>1</sup>, Parminder Flora<sup>3</sup>

Department of Electrical and Computer Engineering<sup>1</sup>  
Queen's University  
Kingston, ON, Canada  
{k.foo, ying.zou}@queensu.ca

School of Computing<sup>2</sup>  
Queen's University  
Kingston, ON, Canada  
{zmjiang, bram, ahmed}@cs.queensu.ca

Performance Engineering<sup>3</sup>  
Research In Motion  
Waterloo, ON, Canada

**Abstract**— Performance regression testing detects performance regressions in a system under load. Such regressions refer to situations where software performance degrades compared to previous releases, although the new version behaves correctly. In current practice, performance analysts must manually analyze performance regression testing data to uncover performance regressions. This process is both time-consuming and error-prone due to the large volume of metrics collected, the absence of formal performance objectives and the subjectivity of individual performance analysts. In this paper, we present an automated approach to detect potential performance regressions in a performance regression test. Our approach compares new test results against correlations pre-computed performance metrics extracted from performance regression testing repositories. Case studies show that our approach scales well to large industrial systems, and detects performance problems that are often overlooked by performance analysts.

**Keywords:** Mining software repositories, performance regression testing, performance analysis

## I. INTRODUCTION

Performance regression testing is integrated into traditional regression testing to reveal performance bottleneck and design problems early on [9]. Traditional regression testing focuses on verifying the functional correctness of a change [17]. However, research on large industrial projects shows that the primary problems observed in the field are often performance related [19]. Examples of such problems are response time degradation, or higher than expected resource utilization. This phenomenon of performance degradation is known as performance regression, which refers to situations where software performance degrades compared to previous releases.

Performance regression testing is the process of putting load on a system to test whether the system is able to support a specific demand that resembles the field usage intensity [6, 7]. In performance regression testing, a load consists of a mix of scenarios to be executed and the rates at which each scenario appears [13]. For example, the MMB3 [2] benchmark, which is used to measure the performance of computers running Microsoft Exchange Server, specifies that a typical user will send 8 emails per day on average, 15% of which have high priority, and another 15% have low priority. A performance regression test usually spans from a few hours to a few days.

During the course of the test, various performance data about the running system is recorded (e.g., CPU and memory utilizations). After each test, performance analysts would use domain knowledge and prior tests to manually look for large deviations of metric values between the past test and the new test. A defect report will be filed if the performance analyst concludes that the observed deviations represent performance regressions. In a large enterprise system, an analysis of a performance regression test can take up to a few days.

Unfortunately, the current practice of performance regression test analysis is both time consuming and error-prone. Because of the number of metrics available, it is difficult for performance analysts to manually look for metric correlations and instances when the correlations are violated. As a result, performance analysts often overlook potential performance problems that may exist in a test. Furthermore, a formal performance baseline rarely exists. A performance analyst must often exert her own judgment to decide whether an observation constitutes a performance problem.

Performance regression testing repositories are used to archive performance regression testing results for bookkeeping purposes, but are rarely used in performance analysis. In this paper, we introduce an automatic approach to derive performance signatures by capturing the correlations among metrics in performance regression testing repositories. Violations of these performance signatures are flagged as potential performance problems. In an e-commerce application for example, as visitors make purchases on the site, transaction records are stored in the database. As a result, a correlation between the visitor arrival rate, application server's CPU utilization, and database disk writes/sec can be extracted as a performance signature. In a new version of the software with the same visitor arrival rate, a significant drop in the number of database disk writes would lead to a violation of the extracted performance signature, signifying a potential performance problem (e.g., deadlock in database).

The performance regression reports generated by our approach signal potential problematic metrics that violate the extracted performance signatures. Performance analysts can leverage our report to ensure better coverage in their assessments of performance regression tests. The main contributions of our work are as follows:

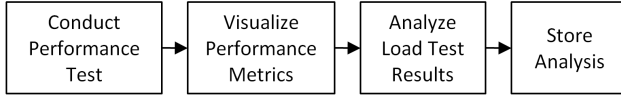


Figure 1. Performance Regression Testing Process

1. Our approach is the first work to leverage performance regression testing repositories to automatically detect occurrences of performance regression in a performance regression test.
2. Performance analysts can leverage information such as the expected metric correlations and visualizations in our performance regression report to derive the cause of a performance problem.

The paper is organized as follows. Section 2 provides an overview of the current practice and limitations of performance regression testing. Section 3 gives an example of a report generated by our research prototype and how performance analysts can benefit from it. Section 4 provides the details of our performance regression approach. Section 5 presents three case studies on two open source systems and one commercial system to demonstrate the effectiveness of our approach. Section 6 discusses our approach and future research directions. Section 7 discusses related research and Section 8 concludes the paper.

## II. CURRENT PRACTICE AND LIMITATIONS

As shown in Figure 1, the typical process of conducting and analyzing a performance regression test involves 4 phases:

1. Performance analysts start a performance regression test. During the course of the test, various performance metrics are recorded.
2. After the test has completed, performance analysts use tools to perform simple comparisons of the averages of metrics against a pre-defined threshold.
3. Performance analysts visually compare the metrics between past runs and the new run to look for evidence of performance regressions or divergences of supposedly correlated metrics. If a metric in the new run exhibits deviations from past runs, this run is probably troublesome and worth further investigations. Depending on individual judgment, a performance analyst would decide whether the changes are significant and file defect reports accordingly.
4. All performance data is archived in a central repository for bookkeeping purposes.

There are three major challenges associated with the current practice of performance regression test analysis.

First, during the course of the test, a large number of metrics is collected. It is difficult for performance analysts to compare multiple metrics at the same time. Although correlated metrics could be plotted on the same graph based on heuristics defined by domain experts, these heuristics only give a limited view of the correlations in order not to overload the plots. For example, in a performance regression test for an e-commerce website, one heuristic would be to group arrival rate and

throughput in one graph while ignoring other related metrics such as request queue length. Although performance analysts can use these composite graphs to spot metrics that deviate from the correlation, the analysts must manually look for and analyze other metrics that are related to the observed anomalies in order to determine the cause.

Second, a correct and up-to-date performance baseline rarely exists. Performance analysts usually base the analysis of a new test on a recently passed test [13]. However, it is rarely the case that a performance regression test is problem free. Using just one prior test as baseline typically ignores problems that are common in both the baseline test and the new test.

Third, subjectivity of performance analysts may influence their judgment in identifying performance regressions. Performance analysts usually compare the metrics' averages between two tests, ignoring the fluctuations that might exist. This would result in inconsistent conclusions among performance analysts. For example, one analyst notes in her analysis a 5% increase of the number of database transactions per second to be worrying while another analyst would ignore the increase because it can be attributed to experimental measurement error.

Due to the above challenges, we believe that the current practice of performance regression testing analysis is neither efficient nor sufficient to uncover performance problems. There is a high chance that performance analysts would bypass or overlook abnormal metric values due to the volume of the data and individual subjectivity. Our approach focuses on reducing analysis effort (phase 3) by automating the detection of performance regressions in a performance regression test.



## III. ILLUSTRATION OF OUR APPROACH

To overcome the challenges presented in the previous section, we have developed an approach to automatically compare the result of a new test to a set of expected metric correlations extracted from prior tests. Our approach generates a report with those metrics that violate the expected metric correlations. Our report can shorten the time required for locating performance regressions in the new test and ensure better coverage of potential problematic metrics.

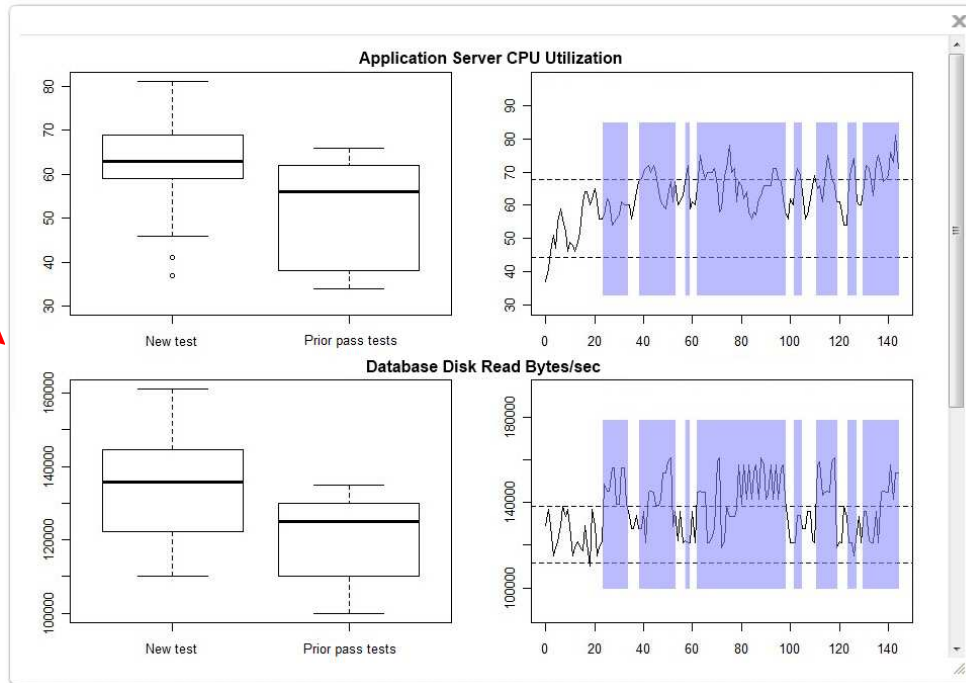
In this section, we present an example of how a performance analyst, Derek, can leverage our report to spot performance problems. Derek is given the task to assess the performance of a new version of an e-commerce application. After conducting a performance regression test on the new version of the software, Derek decides to examine the two metrics that he deems the most important: CPU utilization and the number of disk writes per second in the database. He finds that there is a 5% increase in average CPU utilization between a recently passed test and the new test, but the CPU utilization is still below the pre-defined threshold of 75%. Because of a tight deadline, Derek runs our research prototype to check whether the increase in CPU usage represents a performance regression and whether there are any other performance problems in the new test. Our prototype generates a performance regression report such as the one shown in Figure 2. The sections below explain how Derek can use the report to uncover performance problems.

Severity	Performance Regression	Symptoms
0.66	Application Server CPU Utilization	<a href="#">Show Rules</a>
0.60	Application Server Memory Utilization	<a href="#">Show Rules</a>
0.58	Database Disk Read Bytes/sec	<a href="#">Show Rules</a>

(a) Overview of Problematic Metrics

Severity	Performance Regression	Symptoms		
0.66	Application Server CPU Utilization	<a href="#">Hide Rules</a>		
		Graph	Conf. Change	Expected Correlation
			0.79	Database Logical Disk Reads/sec=Mid Database Memory Page Writes/sec=Mid Database CPU Utilization=Mid Database Memory Page Reads/sec=Mid <a href="#">Application Server CPU Utilization=Mid(High)</a> <a href="#">Application Server Memory Utilization=Mid(High)</a>
			0.65	Database Logical Disk Reads/sec=Mid Database Memory Page Reads/sec=Mid <a href="#">Application Server CPU Utilization=Mid(High)</a> <a href="#">Application Server Memory Utilization=Mid(High)</a> <a href="#">Database Disk Read Bytes/sec=Mid(High)</a>
0.60	Application Server Memory Utilization	<a href="#">Show Rules</a>		
0.58	Database Disk Read Bytes/sec	<a href="#">Show Rules</a>		

(b) Details of Performance Regressions



(c) Performance Comparison

Figure 2. An Example Performance Regression Report

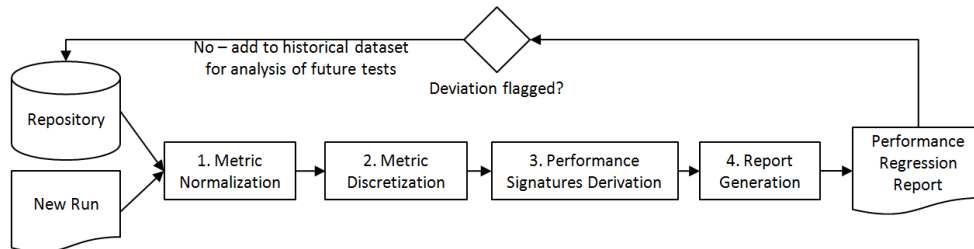


Figure 3. Overview of Performance Regression Analysis Approach

**Report Summary:** The table shown in Figure 2(a) provides a summary of the metrics that are flagged by our approach as deviating from the expected behavior. The metrics are sorted by the level of severity. Severity is the fraction of time intervals in which a metric exhibits problems. By looking at this summary, Derek discovers that 3 metrics (CPU and memory utilizations in the application server, and # of disk read bytes/sec in the backend database) are flagged with severity greater than 0.5, meaning that these three metrics deviate from the expected behavior for over half of the test duration.

**Details of Performance Regression:** Derek clicks on the “Show Rules” hyperlink to reveal a list of metric correlations that are violated by the top flagged metric. The list is ordered by the degree of deviation between the correlation confidence of the new test and prior tests. Correlation confidence measures how well a correlation holds for a data set.

By looking at Figure 2(b), Derek realizes that historically the application server’s CPU utilization, memory usage, and various database metrics are always observed to be in the medium range together. However, in the new run, all flagged metrics shift from medium to high (highlighted in red and blue in Figure 2b). Derek can conclude that all 3 flagged metrics represent performance degradation. Instead of requiring Derek to examine each metric manually, our report automatically identifies metrics in the new test that show significant deviations from prior tests. It is up to Derek to study these metrics to conclude if they represent performance problems.

**Performance Comparison and In-depth Analysis:** Derek can conveniently compare the metric values in prior tests and the new tests by opening a series of charts such as Figure 2(c). The charts on the left are the box-plots of the violated metrics (e.g. the Application server’s CPU utilization) for the new test and prior tests. A box-plot shows the five-number summaries about a metric: the minimum, first, second, and third quartile, and maximum value observed. By placing the box-plots side-by-side, Derek can visually compare the value ranges in the new test and prior tests. In this example, Derek can easily see that half of the observed values of the Application server’s CPU utilization in the new test exceed the historical range.

To streamline Derek’s analysis, our report places the time-series plots of the flagged metrics next to the box-plots. The two dotted lines define the boundaries of the high, medium, and low levels extracted from metric ranges in prior tests. The shaded areas show the time instances where the correlation of performance metrics is violated.

Using our performance regression report, Derek was able to verify his initial analysis and discover new performance problems that he would have missed with only a manual analysis. Furthermore, our report allows Derek to reason about the detected problem by complementing the flagged metrics with the metric correlations that are violated.

#### IV. OUR APPROACH

Our approach to detect performance regressions in a performance regression test has 4 phases as shown in Figure 3. The input of our approach consists of the result of the new test and the performance regression testing repository from which

we distill a historical dataset consisting of the collection of prior passed tests. We apply data-mining techniques to extract performance signatures by capturing metric correlations that are frequently observed in the historical data. In the new test, metrics that violate the extracted performance signatures are flagged. Based on the flagged metrics, a performance regression report is generated. We now discuss each phase of our approach.

##### 1. Metric Normalization

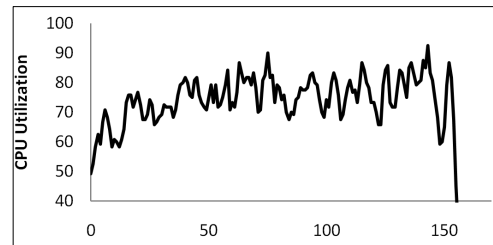
Before we can carry out our analysis, we must eliminate irregularities in the collected data, such as:

**Clock Skew:** Components of a large enterprise system are usually deployed across multiple machines, each of which is responsible for gathering performance data. Since the clock on each machine might be out-of-sync, metrics recorded by different machines will have slightly different timestamps. Moreover, metrics can be recorded at different rates. For example, the CPU utilization can be recorded every 10 seconds while disk I/O is recorded every half minute.

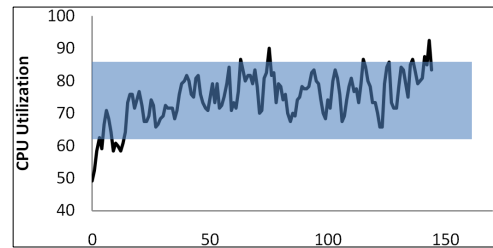
**Extended Test:** After the load generator has stopped, there may be unprocessed requests queued in the system. Performance analysts usually allow the test to continue until all requests are processed. As a result, metrics may be recorded for a prolonged period of time.

**Delay:** There may be a delay in the start of metric collection between the machines that are used in a test.

To overcome these irregularities, we extract the portion of metric data that corresponds to the expected duration of a test. For example, Figure 4(a) shows the CPU utilization of a system in a performance regression test. We filter out the last 20 seconds, which correspond to the period where the load generator has stopped (Figure 4(b)). Then, we resample all metrics at a consistent rate to obtain a normalized data set.



(a) Original Metric Data



(b) Metric Discretization

(Shaded area corresponds to the medium Discretization level)

Figure 4. Metric normalization and Discretization

For each metric,  
**High** = All values above the medium level  
**Medium** = Median +/- 1 standard deviation  
**Low** = All values below the medium level

Figure 5. Definition of Metric Discretization Levels

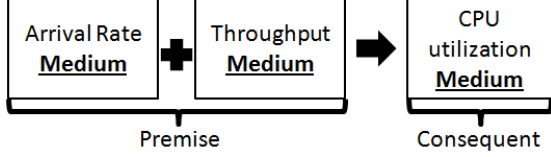


Figure 6. Example of an Association Rule

## 2. Metric Discretization

Since the machine learning techniques we use only take categorical data, we need to discretize metric values into levels (e.g., high/medium/low). Figure 5 shows how the Discretization levels are calculated from the historical dataset.

To discretize the historical data and the new test, we divide time into equal intervals (e.g., every five seconds). Each interval is represented by a vector of metric medians. Each element in the vector is put into one of the 3 levels defined above. For example, assuming the median and standard deviation in Figure 4(b) are 74 and 14 respectively. The medium level will span from 60 to 88. As a result, the CPU utilization around the 50<sup>th</sup> second will be mapped to medium. We have experimented with using arithmetic mean instead of median, but found that arithmetic mean suffered from the effect of outliers and failed to group similar values into the same level.

## 3. Derivation of Performance Signatures

The third phase extracts performance signatures by capturing frequently observed correlations among metrics from the historical dataset. Many metrics will exhibit strong correlations under normal operation. For example, medium request arrival rate would lead to medium usage of system processing power, and medium throughput. Thus, one signature of frequently observed correlation could be {Arrival Rate = Medium, CPU utilization = Medium, Throughput = Medium}.

To extract metric correlations, we use two data mining concepts: frequent item set and association rule. A frequent item set describes a set of metrics that appear together frequently. In this paper, we use the Apriori algorithm [5] to discover the frequent item sets. Association rules can be derived from a frequent item set. For example, Figure 6 shows one of the three association rules that can be derived from the frequent item set in the previous example. An association rule has a premise and a consequent. The rule predicts the occurrence of the consequent based on occurrences of the premise. The Apriori uses support and confidence to reduce the number of candidate rules generated:

- Support is defined as the frequency at which all items in an association rule are observed together. Low support means that the rule occurs simply due to chance and should not be included in our analysis.

- Confidence measures the probability that the rule's premise leads to the consequent. For example, if the rule in Figure 6 has a confidence value close to 1, it means that when arrival rate and throughput are both medium, there is a high tendency that medium CPU utilization will be observed.

We apply the association rules extracted from the historical dataset to the new test and flag metrics in the rules that have significant change in confidence, as defined in eqn. (1).

$$\text{Confidence change} = 1 - \frac{\vec{V}_b \cdot \vec{V}_n}{|\vec{V}_b| |\vec{V}_n|} \quad (1)$$

$$= 1 - \frac{\text{Conf}_b \times \text{Conf}_n + (1 - \text{Conf}_b) \times (1 - \text{Conf}_n)}{\sqrt{\text{Conf}_b^2 + (1 - \text{Conf}_b)^2} \sqrt{\text{Conf}_n^2 + (1 - \text{Conf}_n)^2}}$$

$$\vec{V}_b = (\text{Conf}_b, 1 - \text{Conf}_b) \quad (2)$$

$$\vec{V}_n = (\text{Conf}_n, 1 - \text{Conf}_n) \quad (3)$$

$\text{Conf}_b$  and  $\text{Conf}_n$  represent the confidence of a rule in the historical dataset and the new test respectively. Confidence change is defined in terms of cosine distance, which measures the similarity between two vectors. Since  $\text{Conf}_b$  and  $\text{Conf}_n$  are scalar values, we must convert them into vector form with eqn. 2 and 3 in order for the cosine distance to be calculated.

The confidence change for a rule will have a value between 0 and 1. A value of 0 means the confidence for a particular rule has not changed in the new test; value of 1 means the confidence of a rule is completely different in the new test. If the confidence change for a rule is higher than a specified threshold, we can conclude that the behavior described by the rule has changed significantly in the new test and the metrics in the rule's consequent are flagged. For example, if the rule in Figure 6 drops in confidence from 0.9 to 0.2 in the new test, it indicates that medium arrival rate and throughput would no longer be associated with medium CPU utilization for the majority of the time. As a result, CPU utilization exhibits a significant change of behavior and should be investigated.

## 4. Report generation

In the last phase, we generate a report of the flagged metrics that highlights the association rules that the metrics violate. To further help a performance analyst to prioritize her time, we rank the metrics by level of severity (eqn. 3). For each metric, the report lists the violated rules ordered by confidence change (eqn. 1) as shown in the inner table in Figure 2b.

$$\text{Severity} = \frac{\text{\# of time instances containing the flagged metric}}{\text{total \# of time instances}} \quad (3)$$

Severity represents the fraction of time in the new test that contains the flagged metric. Severity ranges between 0 and 1. If there are only a few instances where the metric is observed to be problematic, the severity will have a value close to 0. On the other hand, if the metrics are violated many times, severity will have a value close to 1 (Figure 1a). Finally, if no metric is flagged in the report, we can conclude that the new test

TABLE I. AVERAGE PRECISION AND RECALL

	# of Test Scenarios	Duration per Test (hours)	Size of Data per Test	Avg. Precision	Avg. Recall
DS2	4	1	360 KB	100%	52%
JPetStore	2	0.5	92 KB	75%	67%
Enterprise System	13	8	4.5 MB	93%	N/A

performance has no performance regression and can be included in the historical dataset for analysis of future tests.

## V. CASE STUDY

We conducted three case studies on two open source e-commerce applications and a large enterprise system. In each case study, we wanted to verify that our approach can reduce the amount of data a performance analyst must analyze and the subjectivity involved, by automatically reporting a list of potential problematic metrics.

We manually injected faults into the test scenarios of the two open source e-commerce systems. This allows us to assess our approach using the precision (eqn. 4) and recall (eqn. 5) evaluation metrics.

$$Precision = \left(1 - \frac{\# \text{ of false positives}}{\text{total} \# \text{ of metrics flagged}}\right) \quad (4)$$

$$Recall = \frac{\# \text{ of problematic metrics detected}}{\# \text{ of metrics that should exhibit problems}} \quad (5)$$

High precision and recall mean that our approach can accurately detect most performance problems. Performance analysts can reduce the effort required for an analysis by investigating the flagged metrics. Note that false positives are metrics that are incorrectly flagged (they do not lead to a performance regression).

For the large enterprise system, we use the existing performance metrics collected by the Performance Engineering team as the input of our technique. We seek to compare the results generated by our approach against the performance analysts' observations. In cases where our approach flagged more metrics than the performance analysts noted, we verify the additional problematic metrics with the organization's Performance Engineering team to determine if the metrics truly represent performance regressions. Since we do not know the actual number of performance problems, we can only provide the precision of our approach.

We use the average precision and recall to show the overall performance of our approach across all test scenarios for each system. Average precision and recall combine the precision (eqn. 6) and recall (eqn. 7) for all  $k$  different test scenarios ( $t_1, t_2, \dots, t_k$ ) conducted for a system. Table 1 summarizes the performance of our approach in each case study.

$$Average \text{ Precision} = \frac{1}{k} \times \sum_{k=1}^k Precision_{t_k} \quad (6)$$

$$Average \text{ Recall} = \frac{1}{k} \times \sum_{k=1}^k Recall_{t_k} \quad (7)$$

**Research Prototype:** Our research prototype is implemented in Java and uses the Weka package [20] to perform various data-mining operations. The graphs in the performance analysis reports are generated with R [4].

### A. Studied System: Dell DVD Store

**System description:** The Dell DVD Store (DS2) application [3] is an open source simulation of an online e-commerce website. It is designed for benchmarking Dell hardware. DS2 includes basic e-commerce functionalities such as user registrations, user login, product search and purchase.

DS2 consists of a back-end database component, a Web application component, and driver programs. DS2 has multiple distributions to support different languages such as PHP, JSP, or ASP and databases such as MySQL, Microsoft SQL server, and Oracle. The load driver can be configured to deliver different mixes of workload. For example, we can specify the average number of searches and items per purchase.

In this case study, we have chosen to use the JSP distribution and a MySQL database. The JSP code runs in a Tomcat container. Our load consists of a mix of use cases, including user registration, product search, and purchases.

**Data collection:** We collected 19 metrics as summarized in table 2. The data is discretized into 2-minute intervals. We ran 4 one-hour performance regression tests. The same load is used in tests A, B, and C. Our performance signatures are derived from Test A during which normal performance is assumed. For tests C and D, we manually inject faults into either the JSP code or the load driver settings to simulate implementation defects and performance analyst's mistakes. The types of faults we injected are commonly used in other studies [13]. Prior to the case study, we derive a list of metrics that are expected to show performance problems, as summarized in Table 3. The Recall of our approach is calculated based on the metrics listed in Table 3.

TABLE II. SUMMARY OF METRICS COLLECTED FOR DS2

<b>Load Generator</b>	% Processor Time # Orders/minute # Network Bytes Sent/sec # Network Bytes Received/Sec
<b>Tomcat</b>	% Processor Time # Threads # Virtual Bytes # Private Bytes
<b>MySQL</b>	% Processor Time # Private Bytes # Bytes written to disk/sec # Context Switches/sec # Page Reads/sec # Page Writes/sec % Committed Bytes In Use # Disk Reads/sec # Disk Writes/sec # I/O Reads Bytes/sec # I/O Writes Bytes/sec



TABLE III. SUMMARY OF INJECTED FAULTS FOR DS2

Test	Fault Injected	Expected Problematic metric
A	No fault	N/A
B	No fault	No problem should be observed.
C	Busy loop injected in the code responsible for displaying item search results	Increase in # I/O reads bytes /sec, and # disk read/sec in database Increase in # threads, # private and virtual bytes, and CPU utilization in the Tomcat server.
D	Heavier load applied to simulate error in load test configuration	Increase in CPU utilization, # threads, # private and virtual bytes in the Tomcat server. Increase in database CPU utilization, # disk reads, writes and I/O read bytes per second, and # context switches. Increase in # orders/minute and network activities in the load generator.

Severity	Metric Violation	Symptoms
1	tomcat5Process(tomcat5)\$Thread Count	<a href="#">Show Rules</a>
1	tomcat5Process(tomcat5)\$Virtual Bytes	<a href="#">Show Rules</a>
0.21	DB\$Process(mysql)\$IO Write Bytes/sec	<a href="#">Show Rules</a>
0.21	DB\$LogicalDisk(Total)\$Disk Writes/sec	<a href="#">Show Rules</a>
0.17	DB\$Process(mysql)\$% Processor Time	<a href="#">Show Rules</a>
0.07	DB\$Process(mysql)\$IO Read Bytes/sec	<a href="#">Show Rules</a>
0.03	DB\$System\$Context Switches/sec	<a href="#">Show Rules</a>

Figure 7. Performance Regression Report for DS2 Test 4 (Increased Load)

**Analysis of Test B:** The goal of this experiment is to show that the rules generated by our approach are stable under normal system operation. Since Test B shares the same configuration and same load as Test A, ideally our approach should not flag any metric.

Our prototype did not report any problematic metric in Test B. The output is as expected, since Test B uses the same configuration as Test A and no performance bug was injected.

**Analysis of Test C:** In test C, we injected a database-related bug to simulate the effect of an implementation error. This bug affects the product browsing logic in DS2. Every time a customer performs a search on the website, the same query will be repeated numerous times, causing extra workload for the backend database and Tomcat server.

Our approach flagged a database related metric (# Disk Reads/sec) and two Tomcat server related metrics (# Threads and # private bytes). All three metrics have severity of 1, signaling that the metrics are violated during the whole test. The result agrees with the nature of the injected fault: each browsing action generates additional queries to the database. As a result, an increase in database transaction leads to an increase of # Disk Reads/sec. When the result of the query returns, the application server uses additional memory to extract the results. Furthermore, since each request would take longer to complete due to the extra queries, more threads are created in the Tomcat server to handle the otherwise normal workload. Since 3 out of 6 expected problematic metrics are detected, the precision and recall of our approach in Test C are 100% and 50% respectively.

**Analysis of Test D:** We injected a configuration bug into the load driver to simulate that a wrongly configured workload

is delivered to the system. This type of fault can either be caused by a malfunctioning load generator or by a performance analyst when preparing for a performance regression test [14]. In the case where a faulty load is used to test a new version of the system, the assessment derived by the performance analyst may not depict the actual performance of the system under test.

In Test D, we double the visitor arrival rate in the load driver. Furthermore, each visitor is set to perform additional browsing for each purchase. Figure 7 below shows the violated metrics reported by our prototype. The result is consistent with the nature of the fault. Additional threads and memory are required in the Tomcat server to handle the increased demand. Furthermore, the additional browsing and purchases lead to an increase in the number of database reads and writes. The extra demand on the database leads to additional CPU utilization.

Because of the extra connections made to the database caused by the increased number of visitors, we would expect the “# context switch” metric in the database to be high throughout the test. To investigate the reason for the low severity of a database’s context switch rate (0.03), we examined the rules flagged the “# context switch” metric. We found that the premises of most rules that flagged the “# context switch” metric also contain other metrics that were flagged with high severity. Consequently, the premises of the rules that flagged “# context switch” are seldom satisfied, resulting in the low detection rates of the “# context switch” metrics. Since 7 out of 13 expected metrics are detected, the precision and recall of our approach in this test are 100% and 54% respectively.

#### B. Studied System: JPetStore

**System description:** JPetStore [1] is a larger and more complex e-commerce application than DS2. JPetStore is a re-implementation of Sun’s original J2EE Pet Store and shares the same functionality as DS2. Since JPetStore does not ship with a load generator, we use a web testing tool to record and replay a scenario of a user logging in and browsing items on the site.

**Data collection:** In this case study, we have conducted two one-hour performance regression tests (A and B). Our performance signatures are extracted from Test A during which caches are enabled. Test B is injected with a configuration bug in MySQL. Unlike the DS2 case study where the configuration bug is injected in the load generator, the bug used in Test B simulates a performance analyst’s mistake to accidentally disable all caching features in the MySQL database. Because of the nature of the fault, we expect the following metrics of the database machine to be affected: CPU utilization, # threads, # context switches, # private bytes, and # I/O read and write bytes/sec.

**Analysis of Test B:** Our approach detected a decrease in memory footprint (# private bytes) and “# I/O writes bytes / sec” in the database, and increase in “# disk reads/sec” and “# threads” in the database. The I/O metrics include reading and writing data to network, file, and device. These observations align with the injected fault: Since the caching feature is turned off in the database, less memory is used during the execution of the test. In exchange, the database needs to read from the disk for every query submitted. The extra workload in the database

TABLE IV. SUMMARY OF ANALYSIS FOR THE ENTERPRISE SYSTEM

Test	Performance Analyst's Report	Our Findings
A	No performance problem found.	Our approach identified abnormal behaviors in system arrival rate and throughput metrics.
B	Arrival rates from two load generators differ significantly. Abnormally high database transaction rate. High spikes in job queue.	Our approach flagged the same metrics as the performance analyst's analysis with one false positive.
C	Slight elevation of database transactions/sec.	No metric flagged.

translates to a delay between when a query is received and the result is sent back, leading to a decrease in “# IO write bytes/sec” to the network.

Instead of an increase, an unexpected drop of the # threads was detected in the database. Upon verifying with the raw data for both tests, we found that the “thread count” in Test A (with cache) and Test B (without cache) consistently remains at 22 and 21 respectively. Upon inspecting the data manually, we do not find that the decrease of one in thread count constitutes a performance problem and this is therefore a false positive. Finally, throughout the test, there is no significant degradation in the average response time. Since 4 out of 6 expected problems are detected, our performance regression report has a precision of 75% and recall of 67%.

### C. Studied System: A Large Enterprise System

**System description:** Our third case study is conducted on a large distributed enterprise system. This system is designed to support thousands of concurrent requests. Thus, performance of this system is a top priority for the organization. For each build of the software, performance analysts must conduct a series of performance regression tests to uncover performance regressions and to file bug reports accordingly. Each test is run with the same workload, and usually spans from a few hours to a few days. After the test, a performance analyst will upload the metric data to an internal website to generate a time series plot for each metric. This internal site also serves the purpose of storing the test data for future reference. Performance analysts then manually evaluate each plot to uncover performance issues. To ensure correctness, a reviewer must sign off the performance analyst's analysis before the test can be concluded. Unfortunately, we are bounded by a Non-Disclosure Agreement and cannot give more details about the commercial system.

**Data collection:** In this case study, we selected thirteen 8-hour performance regression tests from the organization's performance regression testing repository. These tests were conducted for a minor maintenance release of the software. The same workload was applied to all tests. In each test, over 2000 metrics were collected.

Out of the pool of 13 tests, 10 tests have received a pass status from the performance analysts and are used to derive performance signatures. We evaluated the performance of the 3 remaining tests (A, B and C) and compared our findings with the performance analysts' assessment (summarized in table 4).

In the following sections, we will discuss our analysis on each target test (A, B and C) separately.

**Analysis of Test A:** Using the history of 10 tests, our approach flagged all throughput and arrival rate metrics in the system. The rules produced in the report imply that throughputs and arrival rates should fall under the same range. For example, component A and B should have similar request rate and throughput. However, our report indicates that half of the arrival rates and throughput metrics are high, while the other half is low. Our approach has successfully uncovered problems associated with the arrival rate and throughput in Test A that were not mentioned in the performance analyst's report. We have verified our finding with a performance analyst. Our performance regression report has a precision of 100%.

**Analysis of Test B:** Our approach flagged two arrival rate metrics, two job queue metrics (each represents one sub-process), and the “# database scans/sec” metric. Upon consulting with the time-series plots for each flagged metric as well as the historic range, we found that the “# database scans/sec” metric has three spikes during the test. These spikes are likely the cause of the rule violations. Upon discussing with a performance analyst, we find that the spikes are caused by the system's periodic maintenance and do not constitute a performance problem. Therefore, the “# database scans/sec” metric is a false positive. Our performance analysis report has a precision of 80%.

**Analysis of Test C:** Our approach did not flag any rule violation for this test. Upon inspection of the historical value for the metrics noted by the performance analyst, we notice that the increase of “# database transactions/sec” observed in Test C actually falls within the metric historical value range. Upon discussing with the Performance Engineering team, we conclude that the increase does not represent a performance problem. In this test, we show that our approach of using a historical dataset of prior tests is more resistant to fluctuations of metric values. Our approach achieves a precision of 100%.

The case studies show that our approach is able to detect problems in metrics when the faults are present in the systems. Our approach detects problematic metrics with high precisions in all three case studies. In our case studies with the two open source systems, our approach is able to cover 50% and 67% of the expected problematic metrics.

## VI. DISCUSSION AND FUTURE WORK

### A. Quantitative Techniques

Although there are existing techniques [10, 11] to correlate anomalies with performance metrics by mining the raw performance data without discretization, these techniques usually assume the presence of Service Level Objectives (SLO) that can be used to determine precisely when an anomaly occurs. As a result, classifiers that predict the state of SLO can be induced from the raw performance data augmented with the SLO state information. Unfortunately, SLOs rarely exist during development. Furthermore, automated assignment of SLO states by analyzing metric deviations is also challenging as there could be phase shifts in the performance tests, e.g., the spikes do not align. These limitations prevent us from using classifier based techniques to detect performance regression.



### B. Sampling period and Metric Discretization

We choose the size of time interval for metric discretization based on how often the original data is sampled. For example, an interval of 200 seconds is used to discretize data of the enterprise system, which was originally sampled approximately every 3 minutes. The extra 20 second gap is used because there was a mismatch in sampling frequencies for some metrics. We also experimented with different interval lengths. We found that less metrics are flagged as the length of the interval increases, while precision is not affected.

In our case studies, we found that the false negatives (metrics that were expected to show performance regressions but were not detected by our approach) were due to the fact that no rule containing the problematic metrics was extracted by the Apriori algorithm. This was caused by our discretization technique sometimes putting all values of a metric that had large standard deviation into a single level. Candidate rules containing those metrics would exhibit low confidence and were thus pruned. In the future, we will experiment on other discretization techniques, such as Equal Width Interval Binning.

### C. Performance Regression Testing

Our approach is limited to detecting performance regressions. Functional failures that do not have noticeable effect on the performance of the system will not be detected. Furthermore, problems that span across the historical dataset and the new test will not be detected by our approach. For example, no problem will be detected if both the historical dataset and the new test show the same memory leak. Our approach will only register when the memory leak worsens or improves.

### D. Passed Tests

The historical dataset from which the association rules are generated should contain tests that have the same workload, configuration, preferably same hardware, and exhibit correct behavior. Using tests that contain performance problems will decrease the number of frequent item sets extracted, making our approach less effective in detecting problems in the new test. In our case study with the enterprise system, we applied the following measure to avoid adding problematic tests to our historical dataset:

- We selected a list of tests from the repository that have received a pass status from the performance analyst.
- We manually examined the performance metrics that are normally used by a performance analyst in each test from the list of past test to ensure no abnormal behavior was found.

### E. System Evolution and Size of Training Data

The system is often updated to support new environments or requirements. These updates may lead to changes in performance. A large variability in metric values will negatively affect the confidence of association rules generated in our approach. Therefore, it is necessary to update the set of tests included in the historical dataset. We are currently studying the effect of using a sliding window to select prior tests to include in the historical dataset. A sliding window

allows us to automatically discard outdated tests that no longer reflect the current system's performance. However, the optimal size of the sliding window will likely be project-dependent, since each project has different release frequency.

Alternatively, the historical dataset can also be derived from within the run. For example, the first hour of the current test can be used to derive performance signatures. Assuming that the system runs correctly during the first hour, the performance signature generated from this historical dataset will be useful to assess the stability of the system.

### F. Hardware Differences

In practice, performance regression tests of a system can be carried out on different hardware. Furthermore, third party components may change in between tests. In the future, we plan to improve our learning algorithm so that, given a new test, our tool will automatically select the tests from the repository with similar configurations.

### G. Automated Diagnosis

Our approach automatically flags metrics by using association rules that show high deviations in confidence between the new tests and the historical dataset. These deviations represent possible performance regressions or improvements and are valuable to performance analysts in assessing the system under test. Performance analysts can adjust the deviation threshold to restrict the number of rules used and, thus, limit the number of metrics flagged. Alongside with the flagged metrics, our tool also displays the list of rules that the metric violated. Performance analysts can inspect these rules to understand the relations among metrics. From our case study, we notice that some of the rules produced are highly similar. In the future, we will research for ways to merge similar rules to further condense information for performance analysts to analyze.

The association rules presented in our performance regression report represent metric correlations rather than causality. Performance analysts can make use of these correlations to manually derive the cause of a given problem.

## VII. RELATED WORK

Our goal in this work is to detect performance problems in a new test using historical data. Existing approaches monitor or analyze a system through one of two sources of historical data: execution logs and performance metrics.

### A. Analyzing Execution Logs:

Reynolds et al. [18] and Aguilera et al. [22] developed various algorithms for performance debugging on distributed systems. Their approach analyzes message trace of system components to infer the dominant causal paths and identify the components that account for a significant fraction of the system's latency. Unfortunately, the accuracy of the inferred paths decreases as the degree of parallelism increases, leading to low precision in identifying problematic components. Our approach is different from Reynolds's and Aguilera's in that we pinpoint performance issues on the metric level rather than locating the system components that contribute significantly to system latency. Jiang et al. introduce a technique [14] to

identify functional problems in a load test from execution logs. The authors extended this approach to analyze performance in scenarios as well as in the steps of each scenario [13]. Chen et al. proposed Pinpoint [21] to locate the subset of system components that are likely to be the cause of failures. Our work is different from Pinpoint in that Pinpoint focuses on identifying system fault rather than performance regression, which can occur even when the system functions correctly.

In contrast to the above studies, which analyze execution logs, our approach analyzes performance metrics to identify performance problems.

#### B. Analyzing Performance Metrics:

Bondi [9] presented a technique to automatically identify warm-up and cool-down transients from measurements of a load test. While Bondi's technique can be used to determine if a system ever reaches a stable state in the test, our approach can detect performance problems at the metric level.

Cohen et al. [11, 12] applied supervised machine learning techniques to induce models on performance metrics that are likely to correlate with observed faults. Bodik et al. improved Cohen's work [8] by using logistic regression. Our approach is different from the above work as we do not require knowledge of violations of Service Level Objectives.

Jiang et al. proposed an approach [16] for fault detection using correlations of two system metrics. A fault is suspected when the portion of all derived models that report outliers exceeds a predefined threshold. Our approach is based on frequent item sets that can output correlations of more than two metrics. Performance analysts can leverage these metric correlations to better understand the cause of a fault. Jiang et al. [15] proposed an approach to identify clusters of correlated metrics with Normalized Mutual Information as similarity measure. The authors were able to detect 77% of the injected faults and the faulty subsystems, without any false positives. While the approach in [15] can output only the faulty subsystems, our approach can detect and report details about performance problems, including metrics that deviate from the expected behaviors.

### VIII. CONCLUSIONS

It is difficult for performance analysts to manually analyze performance regression testing results due to time pressure, large volumes of data, and undocumented baselines. Furthermore, subjectivity of individual analysts may lead to incorrect performance regressions being filed. In this paper, we explored the use of performance regression testing repositories to support performance regression analysis. Our approach automatically compares new performance regression tests to a set of association rules extracted from past tests. Potential performance regressions of system metrics are presented in a performance regression report ordered by severity. Our case studies shows that our approach is easy to adopt and can scale well to large enterprise system high precision.

#### ACKNOWLEDGMENT

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 products.

#### REFERENCES

- [1] iBATIS JPetStore, <http://sourceforge.net/projects/ibatisjpetstore/>
- [2] MMB3, <http://technet.microsoft.com/en-us/library/cc164328%28EXCHG.65%29.aspx>
- [3] The Dell DVD Store, <http://linux.dell.com/dvdstore/>
- [4] The R Project for Statistical Computing, <http://www.r-project.org>
- [5] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," Proc. of 20<sup>th</sup> Int'l Conf. Very Large Data Bases, 1994.
- [6] A. Avritzer and B. Larson, "Load testing software using deterministic state testing," Proc. of Int'l Symp. on Software Testing and Analysis, 1993.
- [7] A. Avritzer, E. J. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," IEEE Trans. Softw. Eng., 21(9), 1995.
- [8] P. Bodik, M. Goldszmidt, A. Fox, "HiLighter: Automatically Building Robust Signatures of Performance Behavior for Small- and Large-Scale Systems", Proc. of the 3<sup>rd</sup> SysML, Dec 2007.
- [9] A. B. Bondi, "Automating the Analysis of Load Test Results to Assess the Scalability and Stability of a Component," Proc. of 33rd Int'l CMG Conf., San Diego, CA, USA, Dec. 2-7, 2007.
- [10] L. Bulej, T. Kalibera, P. Tuma, "Regression Benchmarking with Simple Middleware Benchmarks," Proc. of the 2004 IPCCC, 2004.
- [11] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," Proc. of 6th OSDI, Dec. 2004.
- [12] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox, "Capturing, indexing, clustering, and retrieving system history" Proc. of the 20th ACM Symp. on Operating Systems principles, 2005.
- [13] Z. M. Jiang, A. E. Hassan, G. Hamann, P. Flora, "Automated Performance Analysis of Load Tests," Proc. of the 25th ICSM, Sept 09.
- [14] Z. M. Jiang, A. E. Hassan, P. Flora, G. Hamann, "Automatic Identification of Load Testing Problems," Proc. of the 24th Int'l Conf. on Softw. Maintenance, Sept 2008.
- [15] M. Jiang, M. A. Munawar, T. Reidemeister, P. A. S. Ward, "Automatic Fault Detection and Diagnosis in Complex Software Systems by Information-Theoretic Monitoring," Proc. DSN, Jun 2009.
- [16] M. Jiang, M. A. Munawar, T. Reidemeister, P. A. S. Ward "System Monitoring with Metric-Correlation Models: Problems and Solutions," Proc. of the 6th Int'l Conf. on Autonomic Computing, 2009.
- [17] T. Kalibera, L. Bulej, P. Tuma, "Automated Detection of Performance Regressions: The Mono Experience," 13<sup>th</sup> MASCOTS, 2005.
- [18] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, A. Vahdat, "WAP5: Black-box Performance Debugging for Wide-Area Systems," Proc. of the 15th Int'l World Wide Web Conf.s, 2006.
- [19] E. J. Weyuker, F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," IEEE Trans. Softw. Eng., 26(12), pp. 1147 – 1156, 2000.
- [20] I. H. Witten, E. Frank, "Data Mining: Practical Machine Learning Tools and Techniques," Morgan Kaufmann, June 2005.
- [21] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks, June 2002.
- [22] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," Proc. of the 19th ACM Symp. on Operating systems principles, Oct 2003.