Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters

Weiyi Shang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, Ontario, Canada
{swy, ahmed}@cs.queensu.ca

Mohamed Nasser, Parminder Flora
BlackBerry
Waterloo, Ontario, Canada

ABSTRACT

Performance testing is conducted before deploying system updates in order to ensure that the performance of large software systems did not degrade (i.e., no performance regressions). During such testing, thousands of performance counters are collected. However, comparing thousands of performance counters across versions of a software system is very time consuming and error-prone. In an effort to automate such analysis, model-based performance regression detection approaches build a limited number (i.e., one or two) of models for a limited number of target performance counters (e.g., CPU or memory) and leverage the models to detect performance regressions. Such model-based approaches still have their limitations since selecting the target performance counters is often based on experience or gut feeling. In this paper, we propose an automated approach to detect performance regressions by analyzing all collected counters instead of focusing on a limited number of target counters. We first group performance counters into clusters to determine the number of performance counters needed to truly represent the performance of a system. We then perform statistical tests to select the target performance counters, for which we build regression models. We apply the regression models on new version of the system to detect performance regressions.

We perform two case studies on two large systems: one open-source system and one enterprise system. The results of our case studies show that our approach can group a large number of performance counters into a small number of clusters. Our approach can successfully detect both injected and real-life performance regressions in the case studies. In addition, our case studies show that our approach outperforms traditional approaches for analyzing performance counters. Our approach has been adopted in industrial settings to detect performance regressions on a daily basis.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software Quality Assurance (SQA)*

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.

http://dx.doi.org/10.1145/2668930.2688052 .

1. INTRODUCTION

Performance assurance activities are an essential step in the release cycle of large software systems [10, 22, 37]. Such activities aim to identify and eliminate performance regressions in each newly released version. Examples of performance regressions are response time degradation, higher than expected resource utilization and memory leaks. Such regressions may compromise the user experience, increase the operating cost of the system, and cause field failures. The slow response time of the United States' newly rolled-out healthcare.gov [4] illustrates the importance of performance assurance activities before releasing a system.

Performance regression detection is an important task in performance assurance activities. The main purpose of performance regression detection is to identify performance regressions, such as a higher CPU utilization relative to the existing version of a system, before releasing a new version of the system. To detect performance regressions, performance analysts conduct performance testing to compare the performance of the existing and new version of a system under the same workload.

However, identifying performance regressions remains a challenging task for performance analysts. One approach is to compare every performance counter across the existing and new versions of the system. However, large software systems often generate thousands of performance counters during performance testing [24, 27]. Comparing thousands of performance counters generated by large software systems is a very time consuming and error-prone task.

Performance engineering research proposes model-based performance regression detection approaches [8]. Such approaches build a limited number of models for a set of target performance counters (e.g., CPU and memory) and leverage the models to detect performance regressions. By examining the results of a small number of models instead of all performance counters, model-based approaches reduce the efforts needed to uncover performance regressions. However, there are major limitations of such model-based approach since performance analysts often select the target performance counters based on their experiences and gut feeling – focusing on a small set of well known counters (e.g., response time). Such ad hoc selection of target counters may lead to the failure to observe performance regressions. For example, selecting CPU as a target counter may miss observing a performance regression for I/O.

In this paper, we propose an automated approach to detect performance regressions by automatically selecting the target performance counters. We first group performance

counters into clusters. We use the clusters to determine the number of target performance counters needed to represent the performance of the system. We then leverage statistical tests to select a target performance counter for each cluster. We build one regression model for each target performance counter. The performance models capture the relationships between the performance counters within each cluster. We apply the regression models on data from the new version of the system and measure the modelling error. If the new version of the system does not have any performance regressions, the regression models should model the performance counters in the new version of the system with low modelling error. Larger than usual modelling errors are considered as signs of performance regressions.

To evaluate our approach, we perform two case studies on two large systems: one open-source system and one enterprise system. We find that our approach can cluster performance counters into a small number of clusters. Our approach successfully detects both injected and real-life performance regression in the case studies. In addition, we apply traditional approaches of performance regression detection: comparing every performance counter across both versions and building a model for a single target performance counter. We find that our approach outperforms both traditional approaches in our case studies.

This paper makes three contributions:

1. We develop an automated approach to detect performance regressions by automatically selecting the target performance counters.
2. Our evaluation results show that our approach successfully detects both injected and real-life performance regressions.
3. Our evaluation results show that our approach outperforms traditional approaches for detecting performance regressions.

Our approach is already adopted in an industrial environment and is integrated into a continuous performance testing environment. The environment leverages our approach to flag performance regressions on a daily basis.

The rest of this paper is organized as follows: Section 2 presents prior research for detecting performance regressions and discuses the challenge of current practice. Section 3 presents an example to motivate this paper. Section 4 presents our approach of detecting performance regressions. Section 5 and Section 6 presents the design and results of our case study. Section 7 compares our approach with traditional approaches for detecting performance regressions. Section 8 discusses the threats to validity of our study. Finally, Section 9 concludes the paper.

2. BACKGROUNDS AND RELATED WORK

We now describe prior research that is related to this paper. We focus on performance regression and faults detection approaches that make use of performance counters. There are two dimensions for analyzing performance counters to detect performance regressions: amount of analysis and complexity of analysis (see Figure 1). Performance analysts conduct performance counter analysis either on all counters or on a limited set of counters. On the other dimension, performance analysts can select either simple analysis, such as comparing the mean value of a counter, or complex analysis, such as building regression models. The choice of the two dimensions makes four types of performance counter based regression detection, as shown in Figure 1. In this section, we discuss prior research based on these four types.

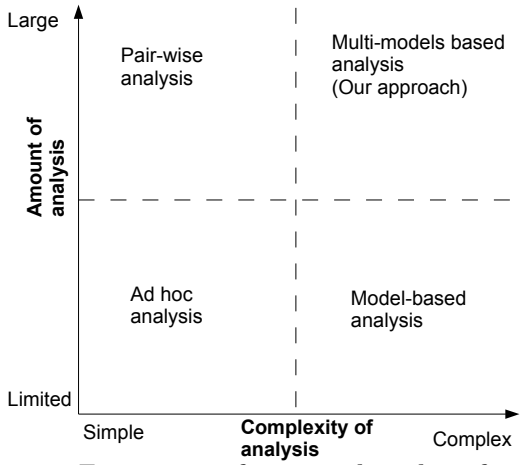

Figure 1: Four types of counter-based performance regression detection.

2.1 Ad hoc analysis

Ad hoc analysis selects a limited number of target performance counters (e.g., CPU and memory) and performs simple analysis to compare the target counters. Heger *et al.* [18] present an approach that uses software development history and unit tests to diagnose the root cause of performance regressions. In the first step of their approach, they leverage Analysis of Variance (ANOVA) to compare the response time of the system to detect performance regressions. The ad hoc approach may fail to detect performance regressions if the target counters do not capture the performance regressions. Moreover, such ad hoc analysis does not detect the change of relationships between counters, such as the relationship between I/O and CPU.

2.2 Pair-wise analysis

Pair-wise analysis leverages simple analysis to compare every performance counter across two versions of a system. Nguyen *et al.* [27–29] conduct a series of studies on performance regressions. They propose to leverage statistical process control techniques, such as control chart [33], to detect performance regressions. In particular, they build a control chart for every performance counter and examine the violation ratio of the same performance counter for a new test. Malik *et al.* [23, 24] propose approaches that cluster performance counters using Principal Component Analysis (PCA). Each component generated by PCA is mapped to performance counters by a weight value. The weight value measures how much a counter contributes to the component. For every counter, a pair-wise comparison is performed on the weight value of each component to detect performance regressions.

There are two major limitations of pair-wise analysis. The first limitation is the large number of performance counters. Nguyen *et al.* and Malik *et al.* state in their research that large software systems have thousands of performance counters [24, 27]. Comparing each performance counter across two versions of a software system is very time consuming and error-prone. Moreover, pair-wise analysis does not cap-

ture the complex interplays between counters. A follow-up of Nguyen *et al.*'s research leverages the historical test results to build machine learning models [29]. The machine learning models capture the relationship between counters, as well as their relationship with regression causes. Such models are then used to predict performance regression causes in new tests. However, their approach requires a historical repository of performance tests to build models.

2.3 Model-based analysis

Model-based analysis builds a limited number of models for a set of target performance counters (e.g., CPU and memory) and leverages the models to detect performance regressions. The model-based approach helps us deal with the large number of performance counters and helps compare the relationships between the various counters.

Recent research by Xiong *et al.* [38] proposes a model-driven framework to assist in performance diagnosis in a cloud environment. Their framework builds models between workload counters and a target performance counter, such as CPU. The models can be used to detect workload changes and assist in identifying performance bottlenecks.

Cohen *et al.* [8] propose an approach that builds probabilistic models, such as Tree-Augmented Bayesian Networks, to correlate system level counters and systems' response time. The approach is used to understand the cause to changes on systems' response time. Cohen *et al.* [9] propose that performance counters can be used to build statistical models for system faults. Bodik *et al.* [5] use logistic regression models to improve Cohen *et al.*'s work [8, 9].

Jiang *et al.* [21] propose an approach that calculates the relationship between performance counters by improving the Ordinary Least Squares regression models and using the model to detect faults in a system.

Current model-based approaches still have their limitations. Performance analysts often select the target performance counters based on their experience and gut feeling. They often focus on a small set of well-known counters (e.g., CPU and memory). Such ad hoc selection of target counters may lead to the failure to observe performance regressions (see Section 7).

2.4 Multi-models based analysis

Multi-models based analysis builds multiple models from performance counters and uses the models to detect performance regressions.

Foo *et al.* [12] propose to detect performance regression using association rules. Association rules group historically correlated performance counters together and generate rules based on the results of prior performance tests. Their approach extracts association rules from performance counters generated during performance tests. They use the change to the association rules to detect performance anomalies. The association rules make use of thresholds derived from the analyst's experience (i.e., determining low, medium and high values of counters). The approach requires a historical repository of performance tests to build association rules.

Jiang *et al.* [20] use normalized mutual information as a similarity measure to cluster correlated performance counters. Since counters in one cluster are highly correlated, the uncertainty among counters in the cluster should be lower than the uncertainty of the same number of uncorrelated counters. Jiang *et al.* leverage information theory to monitor entropy of clusters. A significant change in the in-cluster entropy is considered as a sign of a performance fault. During the evaluation of the approach, the authors were able to detect 77% of the injected faults and the faulty subsystems, without having any false positives.

In this paper, we propose an automated approach to detect performance regressions by automatically selecting the target performance counters. Our approach aims to address the limitation of current model-based analysis (see Section 2.3), i.e., the ad hoc selection of target counters. We present our approach in Section 4.

3. A MOTIVATING EXAMPLE

Ian is a performance engineer for a large-scale distributed software system. The system serves millions of users worldwide. Ensuring the efficient performance of the system is a critical job. Therefore, Ian needs to conduct performance testing whenever there is an update to the system, such as adding new features and/or fixing bugs.

A typical performance test consists of the following steps. First, Ian deploys the old version of the system into a testing environment and applies a test workload on the system. The workload is often pre-defined to exercise most of the system's features and is typically similar to the system's workload in the field. While applying the workload, the system is monitored and thousands of performance counters are collected. Second, Ian deploys the new version of the system into the same testing environment and applies the same workload. The new version of the system is monitored in the same manner as the old version. Finally, Ian examines the collected performance counters from both versions of the system in order to uncover any performance regressions in the new version.

To determine whether there exists any performance regressions, Ian adopts a model-based approach. Ian first selects a target performance counter, such as CPU, as dependent variable. The choice of dependent variable is based on Ian's experience (e.g., customer's priorities and prior experience with field problems). In his practice, CPU is by default the dependent variable. Then Ian uses the rest of the performance counters to build a regression model for the CPU counter. Ian applies the model on a new version of the system in order to predict CPU. If the prediction error is high, Ian would report the existence of a performance regression.

However, this model-based approach may not detect all instances of performance regressions. For example, Ian has two sets of performance counters from two versions of the system, shown in Table 1. Ian leverages the model-based approach and the prediction error is less than 7%. Therefore, Ian reports that the new version does not have any performance regressions. However, after the system is updated, users complain that the system is slower than before. To resolve the users' complaints, Ian starts to review the performance counters and finds that there is a big difference between the I/O read counters in the old versus new version. The model-based approach has not captured the I/O read counters, since the model considered that the I/O read counters have low correlation with CPU.

From this example, we observe the following limitations of current model-based approach for detecting performance regressions. First, all too often, one performance counter cannot represent the performance of a system. In Ian's example, he misses the information of I/O read by focusing only on

Table 1: Examples of performance counters from performance testing.

Old version								
Time Stamp	1	2	3	4	5	6	7	8
CPU Privileged	29.17	27.29	29.90	33.23	31.43	30.91	31.15	30.21
CPU User	33.02	29.48	28.23	26.25	26.95	26.22	26.04	29.38
IO read byte/sec	0	0	0	0	0	0	0	0
IO read op/sec	0	0	0	0	0	0	0	0
IO write byte/sec	7,808.09	4,481.75	7,787.79	4,715.79	7,349.50	4,499.50	4,641.17	8,319.15
IO write op/sec	180.36	163.29	174.36	178.87	188.47	192.43	187.91	178.80
Memory Working set	144,867 KB	144,888 KB	146,522 KB	145,920 KB	145,822 KB	145,822 KB	144,364 KB	144,499 KB
Memory Private byte	146,203 KB	146,625 KB	147,763 KB	147,681 KB	147,583 KB	147,587 KB	146,153 KB	146,305 KB

New version								
Time Stamp	1	2	3	4	5	6	7	8
CPU Privileged	29.38	30.52	30.21	33.02	31.77	28.23	29.48	28.02
CPU User	26.98	27.29	31.04	27.92	27.08	28.23	32.29	33.54
IO read byte/sec	175,008.30	176,262.16	177,867.55	178,745.03	181,573.41	174,242.61	165,634.03	163,400.87
IO read op/sec	1,611.37	1,628.83	1,655.19	1,649.85	1,654.89	1,615.44	1,514.00	1,526.93
IO write byte/sec	4,364.58	7,908.74	4,514.13	7,588.77	4,887.06	7,767.46	4,262.85	3,961.54
IO write op/sec	190.99	183.79	188.82	189.32	189.58	186.40	179.88	167.89
Memory Working set	144,499 KB	144,499 KB	144,499 KB	144,753 KB	144,753 KB	145,056 KB	146,874 KB	144,503 KB
Memory Private byte	146,326 KB	146,350 KB	146,379 KB	146,649 KB	146,682 KB	146,981 KB	148,759 KB	146,383 KB

CPU. Second, choosing dependent performance counters are often biased by performance analysts' experiences and gut feelings. In the example, Ian selects CPU based on his experience, while choosing an I/O related counter may have helped him uncover the I/O related regression.

To overcome such limitations, Ian designs an approach that automatically groups performance counters into clusters. He leverages the clusters to determine the number of models that he needs to build. He then leverages statistical tests to determine the dependent variable (i.e., target performance counter) for each cluster and he builds a model to capture the relationships between the counters in each cluster.

In the next section, we present this new approach for detecting performance regression by grouping counters into clusters.

4. APPROACH

In this section, we present our approach for detecting performance regressions. Each subsection corresponds to a step in our approach, as shown in Figure 2. Table 1 shows an example of performance testing results. The performance counters are recorded during the performance testing of the old and new versions a software system. The performance counters are recorded 8 times for each test run. The values of the performance counters at each time stamp are called an observation of the performance counters. To ease the illustration of our approach, we show a running example with only 8 performance counters and 8 observations. However, the number of performance counters and observations is much larger in real-life performance testing. The goal of our approach is to detect whether there are performance regressions in the new version of a system.

4.1 Reducing counters

In the first step, we clean up the performance counters by removing redundant counters or counters with low variance in both new and old tests. We first remove performance counters that have zero variance in both versions of the performance tests. We then perform redundancy analysis [17] on the performance counters in each cluster. The redundancy analysis would consider a performance counter redundant if it can be predicted from a combination of other variables. We use each performance counter as a dependent variable and use the rest of the counters as independent variables to build a regression model. We calculate the R^2 of each model and if the R^2 is larger than a threshold, the current dependent variable (i.e., performance counter) is considered redundant. We then remove the performance counter with the highest R^2 and repeat the process until no performance counters can be predicted with R^2 higher than the threshold. In this step, *I/O read op/sec* and *Memory Working set* are removed, since they can be predicted by *I/O read byte/sec*, and *Memory Private byte*, respectively.

4.2 Clustering performance counters

The second phase of our approach is to group performance counters into clusters based on their similarities. The number of clusters would show the number of models needed to represent the performance of the system, and performance counters in each cluster are used to build a model. This phase in our approach consists of three steps. First, we calculate the dissimilarity (i.e., distance) between every pair of performance counters. Second, we use a hierarchical clustering procedure to cluster the counters. Third, we convert the hierarchical clustering into k clusters (i.e., where each counter is a member of only one cluster).

4.2.1 Distance calculation

Each performance counter is represented by one point in an n-dimensional space (where n is the number of observations of the performance counter). For example, if a performance test runs for an hour and performance counters are recorded every minute, there would be 60 observations of each performance counter for this performance test. To perform clustering, we need to measure the distance between each point in this 60-dimensional space. A larger distance implies a greater dissimilarity between a pair of performance counters. We calculate the distance between every pair of performance counters to produce a distance matrix.

We use the Pearson distance (a transform of the Pearson correlation [15]). While there are many other distance mea-

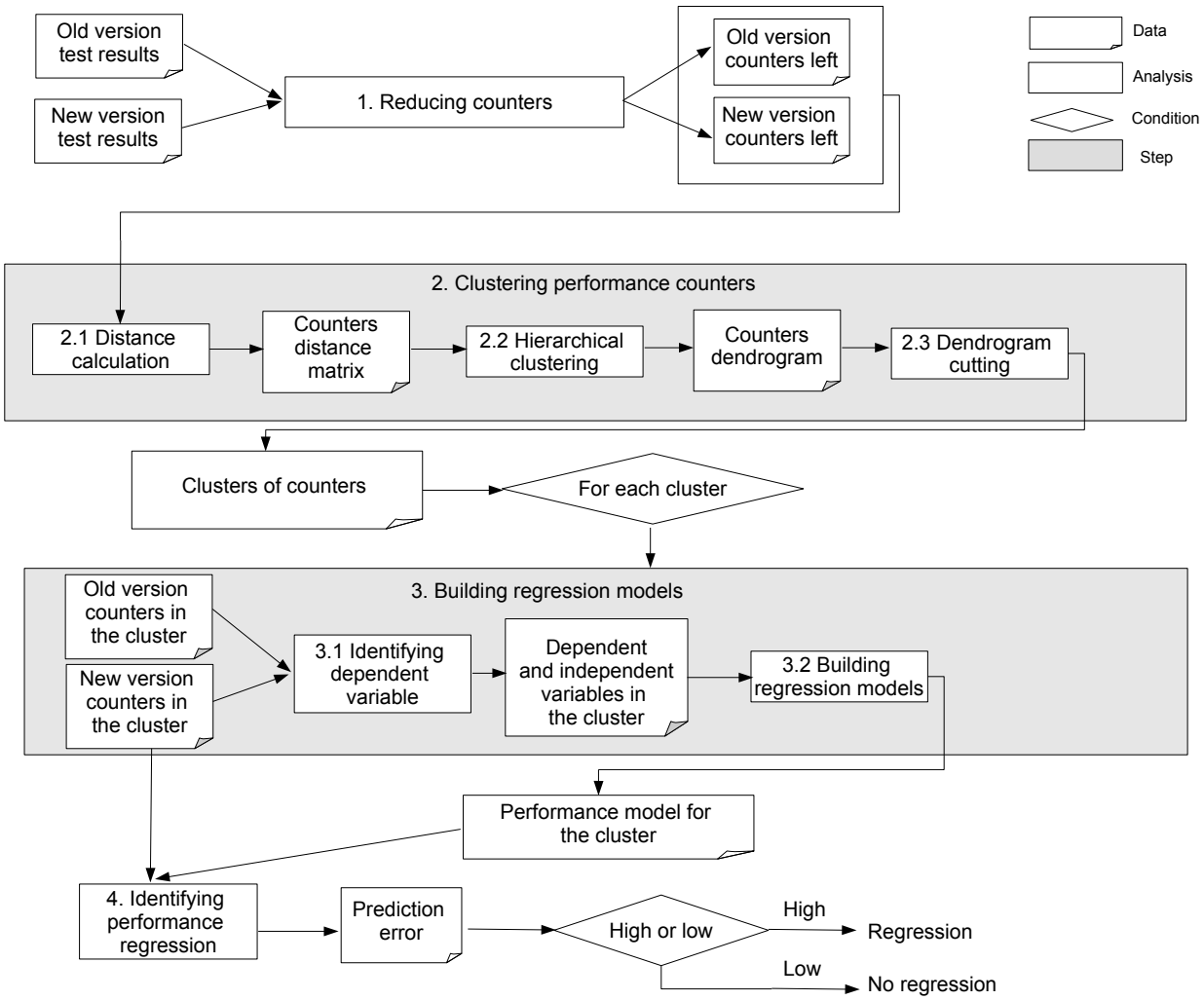

Figure 2: An Overview of Our Approach.

sures [7, 13, 15, 30], we choose Pearson distance since prior research shows that it produces a clustering that is similar to manual clustering [19,32]. Pearson distance also performs well when clustering counters in prior performance engineering research [35].

We first calculate the Pearson correlation (ρ) between two performance counters. ρ ranges from -1 to +1, where a value of 1 indicates that two performance counters are identical, a value of 0 indicates that there is no relationship between the performance counters and a value of -1 indicates an inverse relationship between the two performance counters (i.e., as the values of one performance counter increase, the values of the other counter decrease).

We then transform the Pearson correlation (ρ) to the Pearson distance (d_ρ) using Equation 1.

$$d_\rho = \begin{cases} 1-\rho & \text{for } \rho \geq 0 \\ |\rho| & \text{for } \rho < 0 \end{cases} \quad (1)$$

Table 2 shows the distance matrix of our example.

4.2.2 Hierarchical clustering

We leverage a hierarchical clustering procedure to cluster the performance counters using the distance matrix calculated in the previous step. We choose to use hierarchical clustering in our approach because we do not need to specify the number of clusters before performing the clustering and hierarchical clustering is adopted in prior performance engineering research [35]. The clustering procedure starts with each performance counter in its own cluster and proceeds to find and merge the closest pair of clusters (using the distance matrix), until only one cluster (containing everything) is left. The distance matrix is updated when the two clusters are merged.

Hierarchical clustering updates the distance matrix based on a specified linkage criteria. We use the average linkage, which has been leveraged successfully in prior performance engineering research [35]. When two clusters are merged, the distance matrix is updated in two steps. First, the merged clusters are removed from the distance matrix. Second, a

Table 2: Distance matrix of our example

	CPU Privileged	CPU User	I/O read byte/sec	I/O write byte/sec	I/O write op/sec
CPU User	0.58				
I/O read byte/sec	0.08	0.80			
I/O write byte/sec	0.90	0.07	0.15		
I/O write op/sec	0.44	0.52	0.73	0.93	
Memory Private/byte	0.84	0.06	0.14	0.12	0.03

new cluster (containing the merged clusters) is added to the distance matrix by calculating the distance between the new cluster and all existing clusters. The distance between two clusters is the average distance (Pearson distance) between the performance counters of the first cluster and the second cluster [13, 36].

Figure 3 shows the dendrogram produced by hierarchically clustering the performance counters using the distance matrix from our running example from Table 2.

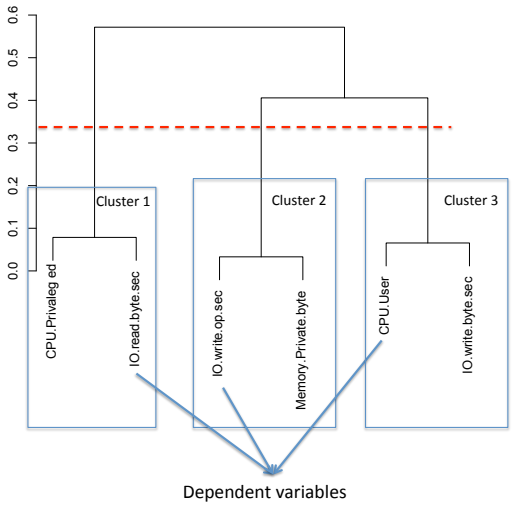

Figure 3: A dendrogram generated by performing hierarchical clustering on our example. The read dashed line in the figure shows where the dendrogram is cut into clusters using the Calinski-Harabasz stopping rule.

4.2.3 Dendrogram cutting

The result of a hierarchical clustering procedure is a hierarchy of clusters, visualized using dendrograms. Figure 3 shows an example of a dendrogram. A dendrogram is a diagram that uses a binary tree to show each stage of the clustering procedure as nested clusters [36].

Dendrogram needs to be cut in order to complete the clustering procedure. The hight of the cut line of a dendrogram represents the maximum accepted amount of intra-cluster dissimilarity within a cluster. After cutting the dendrogram, each performance counter is assigned to only one cluster. Either manual inspection or statistical tests (also called stopping rules) are used to cut dendrograms. Although a manual inspection of the dendrogram is flexible and fast, it is subject to human bias and may not be reliable. We use the Calinski-Harabasz stopping rule [6] to perform our dendrogram cutting. Although there are many other stopping rules [6, 11, 25, 26, 31] available, prior research finds that the Calinski-Harabasz stopping rule performs well when cutting dendrograms produced when clustering performance counters [35].

The Calinski-Harabasz stopping rule measures the similarity within-clusters and the dissimilarity between-clusters. The optimal clustering will have high within-cluster similarity (i.e., the performance counters within a cluster are as similar as possible) and a high between-cluster dissimilarity (i.e., the performance counter from two different clusters are as dissimilar as possible). Using the Calinski-Harabasz stopping rule, we do not need to pre-specify the number of clusters, instead the number of clusters is determined by the similarity between clusters in the dendrogram. If the distances between performance counters are large, the dendrogram cutting process would generate a large number of clusters. However, in our experiments, we find that the number of clusters generated by our approach is typically small (see Section 6).

We mark a red horizontal dashed line in Figure 3 to show where we cut the example hierarchical cluster dendrogram using the Calinski-Harabasz stopping rule. By cutting the dendrogram, we create three clusters of performance counters. The first cluster contains *CPU Privileged* and *I/O read byte/sec*. the second cluster contains *I/O write op/sec* and *Memory Private byte*. The rest of the performance counters (*CPU User* and *I/O write byte/sec*) are in the third cluster.

4.3 Building models

The third phase of our approach is to build models for each cluster of performance counters. This phase of our approach consists of two steps. First, we identify a dependent counter (i.e., target counter) in each cluster for model building. Second, for each cluster we build a regression models [14] using the performance counters in that cluster.

4.3.1 Identifying dependent variable

In model-based performance regression detection (see Section 2), performance analysts often select the target performance counters based on their experience and gut feeling. They often focus on a small set of well known metrics (e.g., CPU and memory). Such ad hoc selection of target counters may lead to the failure to observe performance regressions (see Section 7).

To address the limitation, we propose an approach to automatically select dependent variables (i.e., target counters). We select the performance counter that has the largest difference between the two versions of the system. We select such a counter as dependent variable since our approach aims to measure the largest difference between two runs of a system. To measure the difference, we use a Kolmogorov-Smirnov test [34] on the distribution of each performance counter across the two versions. The smaller the p-value computed using Kolmogorov-Smirnov test, the more likely the performance counter is non-uniformly distributed across the two versions. We select the performance counter with the smallest p-value computed by Kolmogorov-Smirnov test. We choose to use the Kolmogorov-Smirnov test because it does not have any assumptions on the distribution of the performance counters. *I/O read byte/sec*, *I/O write op/sec* and *CPU User* are chosen to be the three dependent variables for each cluster in our example.

4.3.2 Building regression models

We build a regression model, where the independent variables are the remaining performance counters in a cluster. We model the dependent variable as a linear function of independent variables. We choose a linear model since it is easier to interpret the model when developers need to identify the root cause of a detected regression.

4.4 Identifying performance regressions

The final phase of our approach is to identify performance regressions using the regression models built in the last phase. We use the average prediction error as a measure of the deviation between two test results. If the average prediction

error of a cluster in a new version is larger than a threshold (e.g., 30%), we consider that the new version of the system has a performance regression in the particular cluster. In practice, developers need to learn the best threshold for a particular system based on their experiences of the system. In our example, the average prediction errors are 100%, 4% and 2% in the three clusters. Since 100% prediction error in the first cluster is a large prediction error, we consider that there is performance regression in the new version. Developers should focus on the counters in the first cluster to identify the root cause of the performance regression.

5. CASE STUDY

To study the effectiveness of our approach of detecting performance regressions, we perform case studies with two different large systems with injected performance issues. Both systems were used in prior studies [24, 27]. In this section, we present the subject systems, the workload applied on the systems, the experimental environment and the injected performance issues.

5.1 Subject Systems

5.1.1 Dell DVD Store

The Dell DVD store (*DS2*) is an open-source three-tier web application [2]. *DS2* simulates an electronic commerce system to benchmark new hardware system installations. Performance regressions are injected into the *DS2* code to produce versions of *DS2* with performance issues. We reuse the test results generated by Nguyen *et al.* [27]. The lab setup includes three Pentium III servers running Windows XP and Windows 2008 with 512MB of RAM. The first machine is a MySQL 5.5 database server [3], the second machine is an Apache Tomcat web application server [1], and the third machine is used to run the load driver. During each run, all performance counters associated with the DS2 application server are recorded.

5.1.2 Enterprise Application

The enterprise application (*EA*) in our study is a large-scale, communication application that is deployed in thousands of enterprises worldwide and used by millions of users. Due to a Non-Disclosure Agreement (*NDA*), we cannot reveal additional details about the application. We do note that it is considerably larger than *DS2* and has a much larger user base and longer history. Performance analysts of *EA* conduct performance tests to ensure that the *EA* continuously meets its performance requirements.

5.2 Subject performance tests

We evaluate our approach by running it against performance tests with and without performance regressions.

5.2.1 Dell DVD Store

To have performance tests with and without regressions, the following performance regressions were injected into *DS2*:

- **A: Increasing Memory Usage.** Adding a field to an object increases the memory usage. Because the object is created many times by *DS2*, such a change would cause a large increase of memory usage.
- **B: Increasing CPU Usage.** Additional calculation is added to the source code of *DS2*. The source code with additional calculation is frequently executed during the performance test.
- **C: Removing column index.** Column index are used for frequently queried columns. Such regression can only be identified during a large-scale performance test since only the workload that exercises the corresponding query would suffer from the performance regression. A column index in the database is removed to cause slower database requests.
- **D: Removing text index.** Similar to **C**, a text index is often used for searching text data in the database. A database text index is removed to cause slower database requests.
- **E: Increasing I/O access time.** Accessing I/O storage devices, such as hard drives, are usually slower than accessing memory. Adding additional I/O access may cause performance regressions. For example, adding unnecessary log statements is one of the causes of performance regressions [16]. Logging statements are added to the source code that is frequently executed in order to introduce this performance regression.

Table 3: Summary of the 10 runs of our approach for the *DS2* system.

Run name	Old Version	New Version	Performance regression symptoms
No regression 1	Good run 1,2,3,4	Good run 5	NA
No regression 2	Good run 1,2,3,5	Good run 4	NA
No regression 3	Good run 1,2,4,5	Good run 3	NA
No regression 4	Good run 1,3,4,5	Good run 2	NA
No regression 5	Good run 2,3,4,5	Good run 1	NA
Regression 1	Good run 1,2,3,4,5	A	Memory usage increase
Regression 2	Good run 1,2,3,4,5	B	CPU usage increase
Regression 3	Good run 1,2,3,4,5	C	Heavier DB requests
Regression 4	Good run 1,2,3,4,5	D	Heavier DB requests
Regression 5	Good run 1,2,3,4,5	E	IO increase

A performance test for *DS2* without any injected performance regressions is run five times. We name these five runs as *Good Run* 1 to 5.

We create ten sets of data. Five sets are with performance regressions and five sets without performance regressions. Each data set has two parts: one part for the new version of the system and another part for the old version of the system. We use *Good Run* 1 to 5 as the old version of the system. We use the five performance tests with injected regressions (A to E) as the new versions of the system. Each set of data without performance regression consists of data from one run from the *Good Run* 1 to 5 as a new version of the system and the rest four runs from the *Good Run* 1 to 5 as old version of the system. We run our approach against the new and old versions of the system. In total, we have five runs of our approach between two versions of *DS2* without performance regression and five runs of our approach between two versions of *DS2* with performance regression. The summary of the ten runs is shown in Table 3.

5.2.2 Enterprise Application

We pick a performance test as a baseline in the test repository of the *EA*. We also pick one performance test without regression and five performance tests with identified regressions. We run our approach between the baseline test and each of the other five tests. Due to the *NDA*, we cannot mention the detailed information of the identified regressions in the performance tests of *EA*. We do note that the identified regressions include CPU, memory and I/O overheads.

6. CASE STUDY RESULTS

In this section, we evaluate our approach through two research questions. For each research question, we present the motivation of the question, the used approach to address the question, and the results.

RQ1: How many models does our approach need to build?

Motivation. Performance regression testing often generates a large number of performance counters, examining every performance counter is time consuming and error prone. On the other hand, model-based performance regression detection approaches often select one performance counter as dependent variable and build one model for the chosen performance counter. However, all too often, one model cannot represent the performance of a software system. Our approach groups performance counters into clusters to determine how many models are needed to represent the performance of a software system. We want to examine the number of clusters that our approach uses to group performance counters. We also want to examine whether the counters used to build regression models in each run of our approach are consistent. If the counters are consistent across different runs, performance analysts can select to use those counters without using our approach.

Approach. We measure the total number of counters in *DS2* and the number of clusters generated from each test. For *EA*, we do not report the total number of performance counters due to *NDA*, but only report the number of clusters generated. However, we do note that the total number of performance counters of *EA* is much larger than *DS2*. We then remove redundant counters in each cluster and identify a target counter for each cluster. We examine whether the counters (target counters and non-redundant independent counters) in each run is consistent.

Results. **Our approach can group performance counters into a small number of clusters**. For the ten runs of our approach on *DS2*, nine runs only have two clusters and one run has four clusters. Such results show that in the nine runs, instead of examining 28 performance counters one-by-one for every performance test, performance analysts only need to build two models and examine two values from the models for each performance test. For the six runs of our approach on *EA*, two to five clusters are generated by our approach. Since *EA* has a much larger number of performance counters, these results show that our approach can group the performance counters into a small number of clusters even when the total number of counters is considerably large.

The counters (target counters and independent counters) used to build regression models in each run of our approach are not consistent. We compare all the counters that are not removed by our variance and redundancy analysis. We find that 5 to 13 counters are used to build models across runs of our approach (i.e., 15 to 23 counters are removed). The removed counters are not the same across runs of our approach. We find that even though nine out of ten runs for *DS2* have two clusters, the clustering results are not the same. Moreover, the clustering results are not the same within the runs with a regression and runs without a regression, respectively. In addition, there is no performance counter that is always selected to be the target counters across runs of our approach. For example, CPU, one of the mostly used performance counters, is only selected twice as a target counter in the ten runs of our approach.

Counters measuring the same type of resource may not result in the same cluster. For example, CPU, memory and I/O are three typical resources in performance testing results. However, our results show that CPU related counters are not clustered together, neither do memory or I/O related counters. For example, in the clustering results of *No regression 1*, the first cluster consists of *ID Process*, *IO Data Bytes/sec* and *IO Read Operations/sec*; while the second cluster consists of *Elapsed Time*, *Page Faults/sec* and *Pool Nonpaged Bytes*.

> *Our approach can group performance counters into a small number of clusters even though the total number of performance counters is large. Results of clustering and removing counters are different across runs of our approach. Performance analysts cannot select a limited number of counters based on experience or based on a small number of runs of our approach, since the results of clustering and removing counters are different across runs of our approach.*

RQ2: Can our approach detect performance regressions?

Motivation. Detecting performance regressions is an important performance assurance task. Deploying an updated a system into with performance regressions may bring significant financial and reputational repercussions. On the other hand, incorrectly detected performance regressions may cause the waste of large amounts of resources. Therefore, we want to evaluate whether our approach can accurately detect performance regressions.

Approach. Our approach is based on a threshold and choosing a different threshold may impact the precision and recall. Hence, we do not report the precision and recall of our approach. Instead, we report the results of our approach, i.e., we build models from an old version and apply the models on a new version of the system and calculate the prediction errors of the models. If the new version of the system does not have any performance regressions, the regression models should model the performance counters in the new version of the system with low prediction errors. On the other hand, if the new version has regressions, applying the models on the new version should generate high prediction errors. Therefore, larger than usual modelling errors are considered as signs of performance regressions. We want to see whether the tests with and without performance regressions have a difference in prediction errors. Since each run of our approach has multiple models and every model has a prediction error, we focus on the largest prediction error in each run of our approach.

Table 4: Prediction errors in each cluster when building regression models for *DS2*. Each model is built using data from an old version and is applied on data from a new version to calculate prediction errors. The largest prediction error in each run is in bold font.

Run Name	Prediction Errors	
No regression 1	3%	**9%**
No regression 2	3%	**6%**
No regression 3	**11%**	4%
No regression 4	**4%**	**4%**
No regression 5	**6%**	5%

Run Name	Prediction Error			
Regression 1	44%		**1622%**	
Regression 2	**916%**		2%	
Regression 3	17%	**46%**	5%	17%
Regression 4	101%		**161%**	
Regression 5	**24%**		6%	

***Results.* Our approach can detect both the injected performance regressions in *DS2* and the real-life performance regressions in *EA*.** Table 4 shows the prediction errors of our approach on *DS2*. The largest prediction errors in each run without regression is between 4% to 11%, whereas the runs with injected performance regressions have at least one model with a much higher prediction error. The largest prediction errors in each runs with performance regressions is from 24% to 1622%. Table 5 shows that the largest prediction error in the *EA* run without regression is only 3%, while the largest prediction errors in the *EA* runs with real-life regression is 16% to 386%.

Our approach is not heavily impacted by the choice of threshold value. Table 4 and 5 show that the prediction errors generated by our approach have a large difference between the runs with and without performance regressions. For all the largest prediction errors in the *DS2* runs without regression, the *maximum* value is 11%, whereas for all the largest prediction errors in the runs with regressions, the *minimum* value is 24%. For all the largest prediction errors in the *EA* runs, the difference between runs with and without regression is also large. Even though our approach is based on a threshold to detect performance regressions, such large difference in prediction errors (shown in Table 4 and 5) indicate that our approach can successfully detect performance regressions and that the threshold value does not impact the accuracy of our approach. Nevertheless, our experiments highlight the need to calibrate the thresholds based on practitioners' experience since each of the two systems exhibits a different range of prediction errors.

Our approach can successfully detect both injected and real-life performance regressions. The threshold value of prediction errors should be calibrated using a good test run (i.e., one with no regressions).

7. COMPARISON AGAINST TRADITIONAL APPROACHES

In this section, we compare our approach against two traditional approaches: 1) building models against specific performance counters and 2) using statistical tests.

Table 5: Prediction errors in each cluster when building regression models for *EA*. Each model is built using data from an old version and is applied on data from a new version to calculate prediction errors. The largest prediction error in each run is in bold font. The rows with empty cells are the ones with less than five clusters.

Run Name	Prediction Error			
No Regression 1	2%	2%	2%	**3%**

Run Name	Prediction Error				
Regression 1	22%	38%	**94%**		
Regression 2	6%	**16%**			
Regression 3	8%	**386%**	2%	5%	
Regression 4	**65%**	22%	18%		
Regression 5	**55%**	9%	4%	4%	15%

7.1 Comparing our approach with building models for specific performance counters

Prior research often uses performance counters to build models for a specific performance counter (i.e., a target counter) (See Section 2). For example, Xiong *et al.* [38] build regression models for CPU, memory and I/O. Cohen *et al.* [8] build statistical models for a performance counter that measures the response time of the system. However, performance regressions may not have direct impact on the specific target counter. In addition, performance analysts may not have enough good or extensive knowledge to select an appropriate target counter to build a model. On the other hand, our approach automatically identifies the target counters without requiring in-depth knowledge of the system. In this subsection, we compare our approach with building performance models for specific performance counters.

We build two regression models for *DS2* with CPU and memory as target counters, respectively. The *EA* has a performance counter that measures the general load of the system. We build three regression models for the *EA*, using CPU, memory and the load counter of *EA* as target counters, respectively. We measure the prediction error in each model. The results are shown in Table 6 and 7.

Building regression models against specific performance counters may not detect performance regressions. Table 6 and 7 show that even though the prediction errors in the runs without regression are low, some runs with a regression also have low prediction errors. For example, run *Regression 1* in *DS2* has a 0% prediction error when building a model against memory. We examine the counters and find that even though *Regression 1* is injected to introduce a memory overhead, the version with regression uses only 3% more memory than the version without regression. However, the correlations between memory and other counters are different. For example, the correlation between memory and IO Other Bytes/sec is 0.46 in the old version while the correlation is only 0.26 in the new version. In the case study of *EA*, using specific performance counters is even less accurate with real-life regressions. When using CPU and memory, there are cases where prediction errors in the runs with regressions are lower than the runs without regressions. The load counter is typically used in practice for measuring the system performance of *EA*. The prediction error without regressions is 5%, while the two runs with regressions are only

Table 6: Prediction errors when building regression models for specific performance counters in *DS2*. The model is built using data from an old version and is applied on a new version to measure prediction errors.

Run Name	Prediction Error	
	CPU	Memory
No regression 1	1%	0%
No regression 2	1%	0%
No regression 3	1%	0%
No regression 4	1%	0%
No regression 5	1%	0%
Run Name	CPU	Memory
Regression 1	22%	0%
Regression 2	993%	832%
Regression 3	15%	15%
Regression 4	250%	57%
Regression 5	5%	17%

Table 7: Prediction errors when building regression models for specific performance counters in *EA*. The model is built using data from an old version and is applied on a new version to measure prediction errors.

Run Name	Prediction error		
	CPU	Memory	Load counter
Regression 1	27%	11%	45%
Regression 2	16%	3%	7%
Regression 3	0%	6%	12%
Regression 4	19%	9%	147%
Regression 5	641%	3%	7%
Run Name	CPU	Memory	Load counter
No Regression 1	6%	3%	5%

slightly higher (7%). In such cases, it is difficult for performance analysts to determine whether there are any performance regressions. Although one counter in some runs with regression (e.g., CPU in regression 2, DS2) may have high prediction error, developers can only ensure to capture such regression if they build a model for all counters. Otherwise, there is always possibility that we might miss detecting a regression because we selected a wrong target counter.

Our approach outperforms the traditional approach of building models for specific counters to detect performance regressions. Using specific counter may miss detecting performance regressions.

7.2 Comparing our approach with statistical tests

Statistical tests, such as Student T-test, are often used to compare performance counters across performance testing. We use independent two-sample unpaired two-tailed T tests to determine whether the average value of a performance counter in the old and new versions of the system is different. Our null hypothesis assumes that the average values of a performance counter are similar in two versions of the system. Our alternate hypothesis is that the average value of a performance counter is statistically different across the two versions. We reject the null hypothesis when p-values are smaller than 0.05.

Table 8 presents the results of using the T-test when comparing two versions of the system. We find that for *DS2*, the runs with regressions have more performance counters with p-values smaller than 0.05 than the runs without performance regressions. However, we notice that the runs in *DS2* without regressions still have 3 to 6 performance counters with significant differences in the two versions. In such case, the performance analysts would need to examine each performance counters to find out that such runs have no regressions.

Table 8: Number of performance counters that have significant differences in the two versions. We consider the difference is significant when the p-value of the T-test is smaller than 0.05.

of DS2	
Run Name	# significantly differenced counters
No regression 1	6
No regression 2	3
No regression 3	6
No regression 4	3
No regression 5	6
Regression 1	8
Regression 2	17
Regression 3	21
Regression 4	23
Regression 5	11

of EA	
Run Name	% significantly differenced counters
Regression 1	35%
Regression 2	30%
Regression 3	40%
Regression 4	39%
Regression 5	24%
No regression 1	32%

T-test does not perform well with *EA*. Table 8 shows that the run without regression has 32% of the performance counters with significant difference across both versions, while the runs with regression have 24% to 40% of the performance counters with significant difference. The runs *Regression 2* and *Regression 5* on *EA* have less performance counters with significant differences than the run without regressions (*No regression 1*). Moreover, in the run without regressions, examining 32% of the entire performance counters is still a very time consuming task. Such finding shows that the T-test approach does not work well in practice for identifying performance regressions.

T-test does not perform well in practice to detect performance regressions. There are a large number of performance counters with significant differences in the T-test results even though no regressions exist.

8. THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

External validity

Our study is performed on *DS2* and *EA*. Both subject systems have years of history and there are prior performance engineering research [22, 27, 35] studying both systems. Nevertheless more case studies on other software in other domains are needed to evaluate our approach.

Internal validity

Our approach is based on the recorded performance counters. The quality of recorded counters can impact the in-

ternal validity of our study. For example, if none of the recorded performance counters can track the syndrome of a performance regression, our approach would not assist in detecting the regressions. Our approach also depends on building regression models. Therefore, our approach may not perform well where there are a small number of observations of performance counters, since one cannot build a regression model based on a small data set. Our model requires periodically recorded performance counters as input counters. Some event-based performance counters (like response time), need to be transformed (like average response time during the past minute), to be leveraged by our approach.

Although our approach builds regression models using performance counters, we do not claim any causal relationship between the dependent variable and independent variables in the models. The only purpose of building regression models is to capture the relation between performance counters.

Construct validity

Our approach uses the Pearson distance to calculate the distance matrix, uses the average distance to link clusters and uses the Calinski-Harabasz stopping rule to determine the total number of clusters. There are other rules to performance those calculations. Although experiments show that the performance of other rules does not outperform Calinski-Harabasz in performance engineering research [35], choosing other rules may have better results for other systems. We use the p-value from a Kolmogorov-Smirnov test to determine target counters. In practice, multiple counters can all have minimal p-values. To address this threat, we plan to use other criteria, such as effect size, to select the target counter.

Our evaluation is based on comparing the largest prediction errors of the runs with and without performance regression. Our case studies show the large difference of prediction errors between runs with and without performance regression. However, choosing an appropriate prediction error as a threshold is still crucial to achieve high accuracy for our approach. Due to the dissimilarity between large software systems, performance analysts need to choose the best threshold for their systems. Automated identification of a threshold for a system is in our future plan.

We compare our approach with using T-test to compare every performance counter. Although the T-test is widely used to compare two distributions, other statistical tests, such as Mann-Whitney U test, may also be used in practice to compare performance counters. We plan to compare our approach with other statistical tests in future work. We also compare our approach with building regression models against CPU, memory and the load counters for the two subject systems. We plan to compare our approach with building other types of models and using other performance counters to further evaluate our approach.

9. CONCLUSION

Performance regression detection is an important task in performance assurance activities. Detecting performance regressions is still a challenging task for performance analysts. We propose an approach to automatically detect such regressions. Our approach first groups performance counters into clusters. We use the clusters to determine the number of models that we need to build. For each cluster, we leverage statistical tests to select a performance counter (i.e., a target counter), against which we build a regression model. We use the prediction error to measure the difference between two versions of a system in each cluster. A higher than threshold prediction error is considered a sign of a performance regression. Our approach addresses the challenges of detecting performance regressions in two folds: 1) our approach groups performance counters into clusters to determine how many models are needed to represent the performance of a system, 2) we do not require in-depth system experiences from performance analysts.

The highlights of this paper are:

- We propose an approach to automatically detect performance regressions by building regression models on clustered performance counters.
- Our approach can successfully detect both injected and real-life performance regressions. The accuracy of our approach is not heavily impacted by threshold.
- Our approach outperforms using statistical tests, such as T-test, and building models against one performance counter, such CPU, to detect performance regressions.

Acknowledgement

We would like to thank BlackBerry for providing access to the enterprise system 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 BlackBerry and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of BlackBerry's products.

10. REFERENCES

[1] Apache tomcat. http://tomcat.apache.org/.

[2] Dell dvd store. http://linux.dell.com/dvdstore/.

[3] Mysql. http://www.mysql.com/.

[4] J. Bataille. Operational Progress Report. `http://www.hhs.gov/digitalstrategy/blog/2013/12/operational-progress-report.html`, 2013. Last Accessed: 01-Jun-2014.

[5] P. Bodík, M. Goldszmidt, and A. Fox. Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems. In *SysML 08: Proceedings of the Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*. USENIX Association, 2008.

[6] T. Calinski and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, Jan 1974.

[7] S.-H. Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, Nov 2007.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 16–16, 2004.

[9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP '05:*

Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, pages 105–118, 2005.

[10] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb 2013.

[11] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons Inc, 1st edition, 1973.

[12] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41, 2010.

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

[14] D. Freedman. *Statistical models: theory and practice*. Cambridge University Press, 2009.

[15] M. H. Fulekar. *Bioinformatics: Applications in Life and Environmental Sciences*. Springer, 1st edition, 2008.

[16] H. W. Gunther. *WebSphere Application Server Development Best Practices for Performance and Scalability*, volume 1.1.0. IBM WebSphere White Paper, 2000.

[17] F. E. Harrell. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer, 2001.

[18] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *ICPE '13: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38, 2013.

[19] A. Huang. Similarity measures for text document clustering. In *Proceedings of the New Zealand Computer Science Research Student Conference*, pages 44–56, Apr 2008.

[20] M. Jiang, M. Munawar, T. Reidemeister, and P. Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *DSN '09: Proceedings of 2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 285–294, June 2009.

[21] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward. System monitoring with metric-correlation models: Problems and solutions. In *ICAC '09: Proceedings of the 6th International Conference on Autonomic Computing*, pages 13–22, 2009.

[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM '09: 25th IEEE International Conference on Software Maintenance*, pages 125–134, 2009.

[23] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021, 2013.

[24] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *CSMR '10: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, pages 222–231, 2010.

[25] G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, Jun 1985.

[26] R. Mojena. Hierarchical grouping methods and stopping rules: An evaluation. *The Computer Journal*, 20(4):353–363, Nov 1977.

[27] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[28] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated verification of load tests using control charts. In *APSEC '11: Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, pages 282–289, 2011.

[29] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. An industrial case study of automatically identifying performance regression-causes. In *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 232–241, 2014.

[30] T. V. Prasad. Gene Expression Data Analysis Suite: Distance measures. `http://gedas.bizhat.com/dist.htm`, 2006. Last Accessed: 11-Jun-2014.

[31] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, Nov 1987.

[32] N. Sandhya and A. Govardhan. Analysis of similarity measures with wordnet based text document clustering. In *Proceedings of the International Conference on Information Systems Design and Intelligent Applications*, pages 703–714, Jan 2012.

[33] W. A. Shewhart. *Economic control of quality of manufactured product*, volume 509. ASQ Quality Press, 1931.

[34] J. H. Stapleton. *Models for Probability and Statistical Inference: Theory and Applications*. WILEY, 2008.

[35] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Continuous validation of load test suites. In *Proceedings of the International Conference on Performance Engineering*, pages 259–270, Mar 2014.

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

[37] E. Weyuker and F. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, 26(12):1147–1156, Dec 2000.

[38] P. Xiong, C. Pu, X. Zhu, and R. Griffith. vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ICPE '13: sProceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 271–282, 2013.