An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments

King Chun Foo[1], Zhen Ming (Jack) Jiang[2], Bram Adams[3], Ahmed E. Hassan[4], Ying Zou[5], Parminder Flora[6]
BlackBerry[1,6], York University[2], Polytechnique Montréal[3], Queen's University[4,5], Canada
zmjiang@cse.yorku.ca[2], bram@polymtl.ca[3], ahmed@cs.queensu.ca[4], ying.zou@ece.queensu.ca[5]

***Abstract*—A key goal of performance testing is the detection of performance degradations (i.e., regressions) compared to previous releases. Prior research has proposed the automation of such analysis through the mining of historical performance data (e.g., CPU and memory usage) from prior test runs. Nevertheless, such research has had limited adoption in practice. Working with a large industrial performance testing lab, we noted that a major hurdle in the adoption of prior work (including our own work) is the incorrect assumption that prior tests are always executed in the same environment (i.e., labs). All too often, tests are performed in heterogenous environments with each test being run in a possibly different lab with different hardware and software configurations. To make automated performance regression analysis techniques work in industry, we propose to model the global expected behaviour of a system as an ensemble (combination) of individual models, one for each successful previous test run (and hence configuration). The ensemble of models of prior test runs are used to flag performance deviations (e.g., CPU counters showing higher usage) in new tests. The deviations are then aggregated using simple voting or more advanced weighting to determine whether the counters really deviate from the expected behaviour or whether it was simply due to an environment-specific variation. Case studies on two open-source systems and a very large scale industrial application show that our weighting approach outperforms a state-of-the-art environment-agnostic approach. Feedback from practitioners who used our approach over a 4 year period (across several major versions) has been very positive.**

I. Introduction

Performance regressions are regressions (defects) caused by the degradation of system performance compared to prior releases. Many large-scale systems like Facebook, Google and the Blackberry infrastructure, which are used by millions of people worldwide, cannot afford any downtime. A slight performance degradation can lead to a very large increase in operating costs [28]. In fact, many field problems in such systems are performance-related [4], [35]. Hence, to ensure the quality of these systems, performance regression testing is a required testing procedure in addition to regular, conventional functional regression testing.

Performance regression testing detects performance regressions by measuring the system's performance in field-like settings [2], [3]. A performance regression test usually runs from hours to days. During the course of the test run, execution logs and hundreds of performance counters for the running system are recorded. The logs describe the high-level actions performed during each user session, whereas the counters measure metrics like CPU and memory utilization per time interval. After the test is completed, performance analysts need to compare counters against pre-defined thresholds to flag counters that are at alarming levels. The analysts then selectively examine other counters in an attempt to uncover all performance regressions. Analysts commonly use the logs to investigate the rationale for performance regressions.

Such manual detection of performance regressions is inefficient and error-prone due to the large volume of data that is analyzed, the limited knowledge of testers about the tested system, and the time pressure associated with fast release cycles [23]. Hence, in prior research [17], [23], [25], we proposed the automation of such analysis through the mining of historical performance data (e.g., CPU and memory usage) from prior test runs. By building models based on prior successful test runs (either based on the performance counter data or on the execution logs), we are able to classify new test runs as either a pass or a failure with high accuracy. Furthermore, one can also build models to pinpoint the component or node causing the performance regression.

However, such research has had limited adoption in practice. A major hurdle is the incorrect assumption that new tests will always be run in the same environment (e.g., labs with same configurations) as the prior tests. Since labs are a significant investment and typically bought through bids, it is impossible to ensure that all prior tests are run in the same environment. Furthermore, the different hardware and software upgrade cycles of test labs lead to heterogenous lab environments at any moment in time with tests being scheduled in labs based primarily on lab availability. Finally, for long lived systems, prior tests are most likely to have been conducted on different labs than the current ones.

Heterogeneity is a show-stopper from the perspective of industrial adoption of automated performance regression analysis. For example, in an e-commerce system, the system bottleneck may shift from the database to the application server if the database is upgraded with a faster disk. Hence, when trying to adopt one of our previous approaches [17] in an industrial setting, we discovered that our approach performed significantly worse than during our evaluation, which assumed that all tests were conducted in similar environments. Analysis techniques that cannot differentiate between actual

performance regressions and performance differences caused by different environments will lead to incorrect conclusions being drawn about the quality of a system.

In this paper, we make automated performance regression analysis work in practice by building ensembles (i.e., groups) of models, with each model deriving performance rules from a specific prior test (and, hence, configuration). For example, one model might produce a rule for a specific version that indicates that low CPU utilization and high database transaction counts are associated with high throughput, whereas another model for a different version indicates that high database transactions by itself is associated with high throughput. By combining all models (and their associated rules), then trying to assign more weights to those models that were run in the environment most similar to the new test run, we aim to make the detection of performance regressions resilient to heterogenous environments. The contributions of this paper are as follows:

- We use ensemble learning techniques to detect performance regressions in heterogeneous environments.
- We empirically validate our ensemble-based techniques on two major open source and one large enterprise system. We provide the data used in our open source studies online to encourage other researchers to replicate our studies [32].
- We show that ensemble techniques achieve higher precision and recall than our state-of-the-art environment-agnostic approach.

II. Current Practice and Research

State of Practice. Today, organizations rely on manual (time-consuming and error-prone) approaches for analyzing performance regression tests. Once a performance regression test is completed, performance analysts use domain knowledge and the results of prior test runs to manually look for large deviations of counter values (e.g., higher CPU utilization). If the analysts conclude that the observed deviations represent performance regressions, a defect report is filed.

Organizations currently maintain different lab environments to execute performance tests in parallel. These labs may contain varying hardware configurations, such as different CPUs and disks, and software configurations, such as different operating system architectures (e.g., 32-bit and 64-bit) and database versions. This heterogeneity is partly because of inconsistent upgrade cycles, but mostly in order to simulate the heterogeneous environments in which the system will be deployed. Since performance tests executed with different configurations may exhibit different performance behaviour, not being able to distinguish performance differences caused by heterogeneous configurations from those caused by performance regressions, will cause many false alarms, either delaying software release by several weeks or (in the worst case) reducing the confidence practitioners have in the performance testing process.

Today, CPU centric-benchmarks (e.g., SPEC and RPE2 [33]) are often used to map and compare CPU utilizations between different configurations. However, such an approach provides no support for practitioners to understand the rationale for such regressions and cannot be used to compare other performance counters than CPU (e.g., I/O, threading and memory-related performance counters). Such counters are often more important for enterprise applications than CPU utilization, which is more relevant for high-performance computing applications.

State of Research. Unlike functional regression testing, which has been studied extensively [5], [18], [30], performance regression testing has received very little research interest due to the lack of infrastructure and availability of data. Existing research can be divided into two classes depending on whether the approaches analyze execution logs or performance counters. We limit our discussion to performance counters and refer to our previous work [34] for a discussion of related works on execution logs. Approaches for analyzing or monitoring performance counters can be categorized as either supervised or unsupervised, depending on whether or not the performance counter data is labeled with the test outcome.

Supervised Approaches: Cohen et al. apply supervised machine learning techniques to train classifiers on performance data that is labeled with violations of Service Level Objectives (SLO), as defined by stakeholders in the system's requirements [13], [14]. Bodík et al. improve Cohen's work by using logistic regression [8], whereas Zhang et al. extend Cohen's work to maintain an ensemble of classifiers [37]. When an SLO violation is detected, the most relevant classifier is selected from the ensemble to report the counters that correlate with the particular SLO violation. In [26], Malik et al. presented a wrapper-based supervised approach, which detects performance problems.

These supervised techniques require counters to explicitly exceed certain thresholds (e.g., SLOs). In practice, mapping threshold values to a concrete set of counters is not straightforward. Furthermore, threshold violations represent the worst case scenarios of performance regression. By the time such violations surface, the actual cause of the violations is hard to track back, significantly delaying the next software release.

Unsupervised Approaches: Jiang et al. use pair-wise correlations to detect performance problems [21]. Yilmaz et al. use experimental design techniques to minimize the number of configurations that a system needs to be tested on [36]. Malik et al. use Principal Component Analysis to recover correlations among performance counters in order to identify the subsystems causing performance deviations in a new performance test run [25]. Similarly, Jiang et al. identify correlating counters with Normalized Mutual Information [22]. Bulej et al. cluster the response time counters with the k-means clustering algorithm such that the performance between two test runs is detected by comparing their clusters [12]. Breitgand et al. use statistical modeling to flag performance problems (by flagging when counters exceed manually set thresholds), however their approach does not provide support to investigate the rationale for such problems [10]. In [17], we presented an approach that uses association rule mining to automatically flag counters that expose symptoms of performance regressions.

The above studies use some type of supervised or unsupervised model to automatically detect fine-grained performance degradations in large test data sets. However, they all assume that tests are run on the same hardware and software environ-

ments. A single model derived from prior test runs, conducted with multiple environment configurations, will only be able to capture the correlations that are strong enough to persist across all test runs, and ignore other, weaker, but possibly important correlations. The inability to identify which prior test runs are the most related to a new run risks producing incorrect conclusions about the performance of the system under test.

We explore ensemble-based techniques as a way to deal with heterogeneous environments. We enhance our state-of-the-art association rule mining approach [17] with ensemble based techniques and compare the performance of the resulting model to that of the original model on two major open source systems and one large enterprise system. The next section discusses the intuition and overview of our techniques, whereas Section IV presents our case study results.

III. Our Approach

Ensemble-based models consist of multiple individual classification models. To classify a new instance, the classifications of all the sub-models are combined through voting (bagging [9]) or more elaborate composition (stacking [16]). The key insight here is that each individual model specializes in one particular area, with the global ensemble model taking into account the decision of all areas.

In the context of performance regression, the ensemble model for a particular release consists of individual models, one for each passed prior test run. Such models summarize for a given test run, and hence for its particular test environment, the major patterns of performance counter values. When a new test run's performance counters are available, they are matched to the patterns in the individual models to determine deviations between the new test run and the prior test runs. The final decision about whether the new test run has passed or failed then needs to aggregate (in decreasing order of importance) deviations with prior tests having a test environment: identical/similar/different from the new test run.

A naive bagging approach would blindly pick the counters deviating in the majority of the individual models, without distinguishing explicitly between the three groups of models. If, coincidentally, the majority consists of models from groups 1 and 2, the approach will work well. If group 3 dominates, the ensemble model will likely yield the wrong result, unless the new test run violates common behaviour across all test runs. To make the analysis more clever, we also experimented with a stacking approach that gives models of groups 1 and 2 a higher weight than those in group 3. Hence, if a new test run deviates from prior runs with an identical or similar test environment, those deviations will more likely determine the fate of the new test run.

To evaluate the performance of these two ensemble techniques, we integrated them into the state-of-the-art approach for performance regression analysis that we proposed before [17]. Our resulting ensemble approach has 5 phases, as shown in Figure 1. We now discuss each phase of our approach through a running example with four prior tests T_1, T_2, T_3, T_4 and one new test, T_5.

Phase 1 – Counter Normalization. First, we must eliminate irregularities in the collected performance data. Irregularities can come from the following sources: (1) *Clock Skew:* Counters captured by different machines tend to have slightly different timestamps due to the large number of machines. (2) *Delay:* There may be a delay in the start or end of counter collection between the machines.

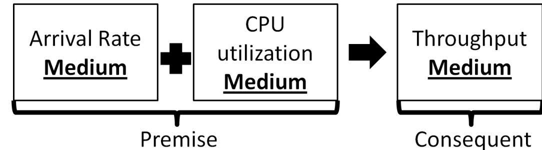

Fig. 2: Example association rule.

TABLE I: Counters flagged in T_5 by multiple rule sets.

Models	Counters of T_5 flagged as violation
R_1	CPU utilization, throughput
R_2	Memory utilization, throughput
R_3	Memory utilization, throughput
R_4	# Database transactions/s

To overcome these irregularities, we extract the portion of counter data that corresponds to the expected duration of a test. Then, we divide the time into equal intervals (e.g., every five seconds). Each interval is represented by a vector containing the medians of all counters in that interval. The size of the interval can be adjusted by the analysts depending on how often the counters are captured. Less interesting execution phases could be aggregated into longer intervals.

Phase 2 – Counter Discretization. Our association mining technique (see phase 3) requires categorical data. Therefore, we must discretize the continuous performance counters. We choose to use the Equal Width interval binning (EW) algorithm for our discretization task, because it is relatively easy to implement and performs well in conjunction with machine learning techniques [15]. The EW algorithm first sorts the observed values of a counter, then divides the value range into k equally sized bins. Each counter value is discretized to the bin to which it belongs. Note that EW typically does not lead to uniformly distributed bins.

The width of each bin and the number of bins are determined by $bin\ width = \frac{x_{max} - x_{min}}{k}$, where $k = max(1, 2 \times log(u))$ and u is the number of unique values of a counter. To determine the bin width, we apply the EW algorithm on all values of a particular counter observed in the entire performance testing repository. For example, the tests in the training set T_1, T_2, T_3, T_4 are first combined to form an aggregated dataset T_A to determine the bin boundaries.

Phase 3 – Association Rule Derivation and Violation Detection. We apply association rule mining to extract a set of association rules for each historical test run in the training set. Each rule set is a model describing the performance behaviour of a prior test. An association rule consists of a premise and a consequent. The rule states that if the premise holds in a new test run, then the consequent will also hold with high probability. Figure 2 shows an example of an association rule that states "if both the arrival rate and the CPU utilization are observed to be at the medium level, throughput should also be at the medium level."

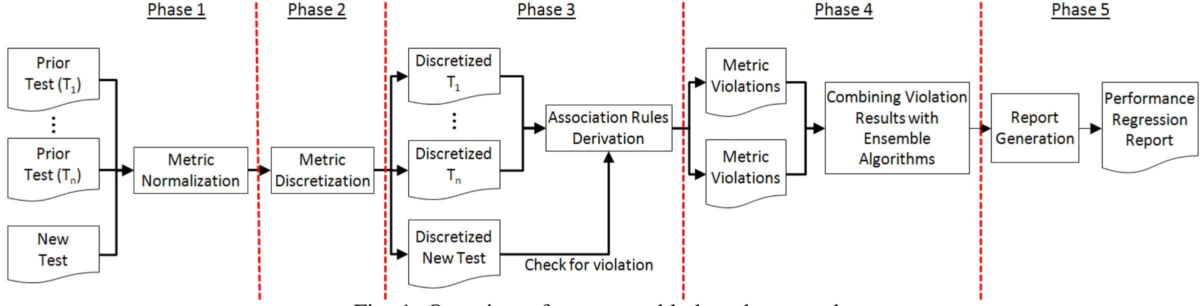

Fig. 1: Overview of our ensemble-based approach.

The probability with which an association rule holds can be characterized by its support and confidence measures. *Support* measures the ratio of times the rule holds, i.e., the counters in the premise and consequent are observed together with the specified values. Low support means that the association rule may have been found simply due to chance. *Confidence* measures the probability that the rule's premise leads to the consequent, i.e., how often the consequent holds when the premise does. For example, if the rule in Figure 2 has a confidence value close to 1, it means that when arrival rate and CPU utilization are both medium, one will almost always observe medium throughput.

To improve the quality of the association rule set, candidate rules that do not reach the minimum support (0.3) and confidence (0.9) thresholds are pruned. These values are the default values defined in the data mining package that we use [11] (yielding good results in our case studies). Lower thresholds will accept more rules, making it harder for new test runs to satisfy all the rules (higher chance for detecting deviations). Higher thresholds yield less rules and make the rule set easier to match with. Different projects can experiment with different thresholds for optimal results.

We use the confidence of a rule on a new test run's counter values as an indicator of performance regressions. For example, in Figure 2 a drop in confidence from 0.9 to 0.2 on the new test run's data would indicate that medium arrival rate and CPU utilization no longer associate with medium throughput in most of the cases. As a result, the throughput counter should be investigated. We measure such confidence changes using the cosine distance between two test runs: $\Delta_{confidence} = 1 - cosine_distance(\vec{V}_{history}, \vec{V}_{new})$, where $\vec{V}_{history} = (Conf_{history}, 1 - Conf_{history})$, $\vec{V}_{new} = (Conf_{new}, 1 - Conf_{new})$, where $\vec{V}_{history}$ and $\vec{V}_{new}$ are the vector form of the confidence values $Conf_{history}$ and $Conf_{new}$ in the historical dataset and the new test run, respectively. Since cosine distance measures the angle between two vectors, we had to convert the scalar confidence values into vector form.

The confidence change values $\Delta_{confidence}$ range between 0 and 1. A value of 1 means that the confidence of a rule is completely different in the new test run, i.e., the premise holds, but the consequent does not. If the confidence change $\Delta_{confidence}$ for a rule is higher than a specified threshold, we can conclude that the behaviour described by the rule has changed significantly in the new test run. The counters appearing in the rule's consequent can then be flagged as a regression. The threshold for confidence change again is customizable by the performance analyst to control the number of flagged counters, based on the available analysis time.

TABLE II: # of models R_1 to R_4 that flagged each counter.

Counters flagged as violation	# of times flagged
Throughput	3
Memory utilization	2
CPU utilization	1
Database transactions / second	1

TABLE III: Test configurations for stacking.

	Performance Test Repository		New Test
	T_1	T_2	T_5
CPU	2 GHz 2 cores	2 GHz 2 cores	2 GHz 2 cores
Memory	2 GB	1 GB	2 GB
Database Version	2	1	1
Architecture	32 bit	64 bit	64 bit

Each model contains performance characteristics that are common across prior test runs as well as behaviours that are specific to the test run and environment used as training data. Table I shows an example of counters flagged in T_5 as performance regressions by models R_1 to R_4. Counters that are flagged by only a few models may be due to differences in environments or other specifics of a particular test run.

Phase 4 – Combining Violations with Ensemble Algorithms. We now combine the counters flagged by each individual model in Phase 3 using one of the following two ensemble methods: bagging and stacking. We briefly review these algorithms and present our application of them.

Bagging is one of the earliest and simplest ensemble-based algorithms [9], yet in some cases it has been shown to outperform more complex approaches [6], [31]. In bagging, the prediction of each model is combined by a simple majority vote to form the final prediction. Hence, in order for a performance counter to be flagged in our performance report, we require the counter to be flagged by at least half of the models. For example, Table II shows the number of times a counter is flagged for T_5 by the 4 models (R_1, R_2, R_3, R_4) in Table I. Bagging will report both throughput and memory utilization as performance regressions as they are flagged by at least 2 models.

Stacking is a more general ensemble technique that can use any selection process to form a final set of predictions. A simple and effective way to combine the results of individual rule sets is to create a Breiman's stacked regression [13], which uses

a linear stacking function s of the form: $s(\vec{x}) = \sum_i w_i R_i(\vec{x})$, where $w_i \in [0, 1]$ represents the weight of the violations reported by rule set R_i (generated from the ith test run in the repository), and $\vec{x}$ represents the vector of performance counter values of the new test run. $R_i(\vec{x})$ yields a violation vector of 1s and 0s, where a 1 corresponds to a particular counter showing a performance regression in the new test run based on R_i. Hence, $s(\vec{x})$ basically is a weighted count of the number of times each counter is flagged across all prior test runs. Bagging would consider all weights w_i to be 1.

Although performance analysts may specify custom weights best suited for their test repositories, we define the weight of each R_i model based on the similarity between the environments used for the earlier test runs and the new test run. A prior test run with a very similar environment to the new test run should receive a larger weight. To compare the environments between two test runs, we generate a similarity vector of 1s and 0s to indicate if two test runs share common components in their environment. Which components exactly constitute the environment depends on the concrete software system or deployment, and the degree of detail in which the environment is specified can be chosen as well.

Since it is difficult to determine the relations between the system performance and individual components of the environment, we prefer a simple binary approach to compare environment configurations. For example, Table III shows three environments for T_1, T_2, and T_5 (the components shown here are just examples). The similarity vectors for the (T_1, T_5) pair would be (1, 1, 1, 0), because both T_1 and T_5 share the same versions of CPU, memory and database, and differ only in the operating system version. Likewise, for the (T_2, T_5) pair, the vector would be (1, 0, 0, 1). More elaborate similarity vectors could be used to better differentiate between critical and trivial differences in environments, for example tripling the total amount of memory versus adding a small hard disk.

To measure the degree of similarity between the new and a past test run, we use the cartesian length of the similarity vector ($\sqrt{N}$ with N the number of 1s in the vector). For example, the lengths of (T_1, T_5) and (T_2, T_5) equal 1.7 and 1.4 respectively. The longer the length of a similarity vector, the higher the similarity between prior and new test runs. We can then calculate the weights as $w_i = \frac{l_i}{\sum_j l_j}$, with l_j the length of the similarity vector of the j-th model. As such, all weights sum up to 1. For example, the weight w_1, for T_1 is calculated as follows: $w_1 = \frac{|(T_1,T_5)|}{|(T_1,T_5)|+|(T_2,T_5)|} = \frac{1.7}{1.7+1.4} = 0.55$.

The weight w_i essentially is a value that describes the relative importance of a model for analyzing a specific, new test run, with the idea that tests having identical or similar environments as the new test run have the largest weights (hence weights need to be re-calculated for each new test run). As such, the weights can be used to produce the set of counters that most commonly violate the expected behaviour from prior test runs. Since the w_i sum up to 1 and the $R_i(x_j)$ are either 0 or 1, each element of the $s(\vec{x})$ vector will have a value between 0 and 1. Performance counters with $s(\vec{x})$ values greater than 0.5 will be included in the resulting set of counters as performance regressions, since they are reported by most of the highly weighted models.

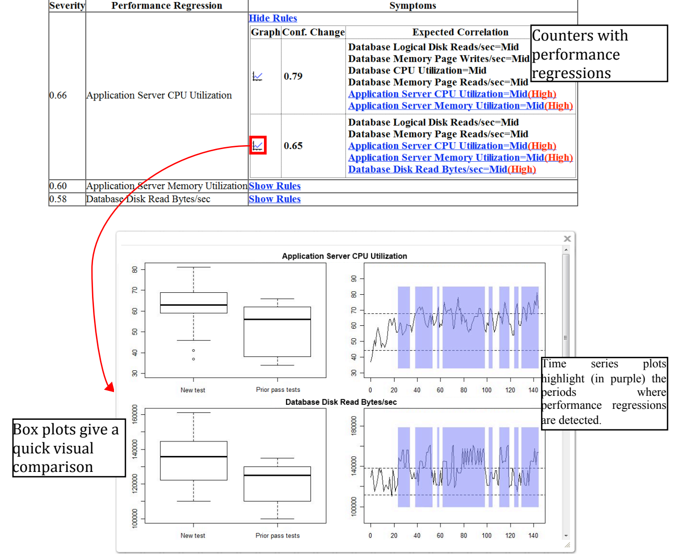

Fig. 3: An example of our performance regression report.

Phase 5 – Report Generation. To help performance analysts diagnose performance problems, our approach generates a report like Figure 3. The reports contain important information for developers, such as a list of problematic counters, correlated counters (Figure 3, top), as well as an overview of the periods in which the performance regressions occur (Figure 3, bottom). Time intervals coloured in purple indicate periods during which metrics are part of a violated rule. Performance analysts can further investigate the periods in which the performance regressions are detected (Figure 3, bottom) by clicking on the "Graph" column.

To generate the report, the flagged counters are first ranked by either the number of models flagging the counters (bagging) or the aggregated weights $s(\vec{x})$ (stacking). If two counters are ranked the same, they will be further sorted by the level of severity as defined in

$$Severity = \frac{\#\ time\ intervals\ with\ flagged\ counter}{total\ \#\ time\ intervals}$$

Severity represents the fraction of time intervals (cf. phase 1) that contain the violating counter occurrences in the new test. Severity ranges between 0 and 1. If there are only a few intervals where the counter is observed to be problematic, the severity will be close to 0. On the other hand, if the counters are violated many times, severity will have a value close to 1. Each counter is linked to the list of association rules that the counter violates (Figure 3). The rules help the analysts in investigating the performance regressions.

IV. Case Study

We conducted a series of case studies on two major open source e-commerce applications (Dell DVD Store and JPetStore) and a large enterprise system to compare the two ensemble-based techniques to our state-of-the-art association rule mining model [17]. The open source case studies serve two purposes: 1) they provide a replication package for others to replicate our work as we cannot publicly release the industrial data, and 2) they enable us to study our approach in a controlled setting with known injected problems, similar to [36].

In particular, in the Dell DVD Store and JPetStore case studies, we explore whether our ensemble approaches can handle tests with different *hardware* (Dell DVD Store) and *software* (JPetStore) environments, respectively. In our industrial case study, we examine whether our ensemble approaches can detect performance regressions in performance tests conducted both in different hardware *and* software environments, based on a large number of long-running tests.

For the two open source systems, we manually injected faults into our case studies. For this, we followed the popular "Siemens-Programs" approach of seeding bugs [20], which are based on common performance problems. Since we know which faults were injected, we can assess our approach using both precision and recall [21]:

$$Precision = \begin{cases} 1 - \frac{m_{false}}{m_{total}} & \text{if } m_{total} > 0 \\ 1 & \text{if } m_{total} = 0 \end{cases}$$

$$Recall = \begin{cases} \frac{m_{total} - m_{false}}{m_{expected}} & \text{if } m_{expected} > 0 \\ 1 & \text{if } m_{expected} = 0 \end{cases}$$

where m_{false} is the number of counters that are incorrectly flagged, m_{total} is the total number of counters flagged, and $m_{expected}$ is the number of counters expected to show performance regressions (including counters that are a side-effect of the injected fault). A high precision means that few of the flagged performance regressions are false alarms. A high recall means that our approach can discover most of the performance regressions in a new test.

For the large enterprise system, we use the existing performance counters collected by the company's performance analysts for historical test runs as the input of our approach. We compare the results of our approach against the analysts' reports filed at the time. Since an important motivator of our work is the tendency of analysts to miss problems due to the vast amount of data produced by large industrial systems, we do not blindly use the analysts' reports as the "gold standard" for precision and recall. Instead, we carefully re-verified the existing test reports with the analysts, who noted any missed problems. For example, using this process, we found 13 problematic counters in the analysis of the first of three tests. The differences with the original report of the analysts strengthens our claim that current manual analysis practices are not sufficient.

Although we do not know the real number of performance problems in the enterprise system, we calculate the relative recall in terms of the union of true performance regressions identified by the three approaches (original approach and two ensemble versions).

Finally, we use the F-measure to rank the accuracy of the three approaches in all three case studies. The F-measure is the harmonic mean of precision and recall, and outputs a value between 0 and 1. A high F-measure value means that a technique has high precision and high recall. Table IV summarizes the performance of all three approaches in the case studies. Bold entries highlight the best technique for a particular test run.

For each case study, we give a description of the system, methods used for data collection, and analysis of each approach. The replication package for the open source systems can be found online [32].

TABLE IV: **P**recision, **R**ecall and **F**-measure of the studied approaches. The highest **F**-measures across the three approaches for a particular case study are shown in **bold**.

		Single Model			Bagging			Stacking		
		P	R	F	P	R	F	P	R	F
DS2	D_4	0	1	0	1	1	**1**	1	1	**1**
	$D_5^{(1)}$	1	0.57	**0.73**	1	0.57	**0.73**	1	0.57	**0.73**
	$D_5^{(3)}$	1	0.14	0.25	1	0.57	**0.73**	1	0.57	**0.73**
JPS	J_3	1	0.5	**0.66**	1	0.5	**0.66**	1	0.5	**0.66**
Enterprise	E_1	1	0.46	0.63	0.72	1	0.84	0.85	0.85	**0.92**
	E_2	0.86	0.4	0.55	0.75	1	0.86	0.93	0.87	**0.90**
	E_3	1	1	**1**	0	1	0	1	1	**1**
	Avg.	0.84	0.58	0.55	0.78	0.81	0.69	0.97	0.77	**0.85**

A. Dell DVD Store

The Dell DVD Store (DS2) application [16] is an open source implementation of an online movie rental website. DS2 consists of a back-end database component, a web application component and load drivers (simulating user traffic). In this case study, we have chosen to use DS2's JSP distribution with a MySQL database and a Tomcat container.

Data collection: We ran five one-hour performance test runs (D_1 to D_5) with the same standard workload. We varied the CPU and memory capacity of the machine that hosts Tomcat and MySQL to simulate different hardware environments. Table V summarizes the hardware setups and expected problematic counters for this case study. Test run D_1 represents a test run in which the hardware is running at its full capacity. In test run D_2, we throttle the CPU to 50% of the full capacity to emulate a slower machine. In test run D_3, we reduce the memory from 3.5GB to about 1.5GB. Test runs D_1, D_2, and D_3 will be used as the test repository for the three approaches.

Test run D_4 is a replication of D_1 and is used to show that our approach produces few false positives. In D_5, we use an injected fault from [24] to cause a performance regression. The injected excessive-database-queries bug simulates the "n+1" pattern [19]. Prior to each case study, we manually derived a list of counters that we expect to exhibit performance regressions because of the injected bug. We use these counters to calculate the recall of our approach.

Analysis of Test Run D_4: Since D_4 is run without any injected bugs, ideally, no counter should be flagged. We use D_4 as a sanity check for the ensemble techniques.

When using our original approach, 4 counters (database CPU utilization, # database I/O writes/s, # orders/minute and the response time counter) were flagged. Upon investigation, we found that the rules of test run D_2 flagged the first 3 counters: because less processing power was available in D_2, each request would take longer to complete, resulting in an increased CPU utilization. Also, less requests could be completed, leading to a decrease in # database I/O writes/s and # orders/minute. The violations of the D_2 model seem to indicate that the behaviour of D_4 (and hence D_1) differs the most from that of D_2. Since no actual fault was injected into test run D_4, we concluded that the four flagged counters were

TABLE V: Summary of test setup for DS2.

Run	Hardware Setup	Fault Description	Expected Problematic Metrics
D_1	CPU = 100%, Memory = 3.5 GB	No fault	No problem should be observed.
D_2	CPU = 50%, Memory = 3.5 GB	No fault	No problem should be observed.
D_3	CPU = 100%, Memory = 1.5 GB	No fault	No problem should be observed.
D_4	Same as Test D_1	No fault	No problem should be observed.
D_5	Same as Test D_1	Busy loop in browsing logic	1. Increase in CPU utilization, and # disk reads/s in database. 2. Increase in # threads, # private bytes, and CPU util. in Tomcat. 3. Increase in response time and # requests/min in application.

false positives, leading to a precision of 0. Because we do not expect any counter to be flagged, recall is equal to 1, as defined by Equation 1. The F-measure of our original approach is 0. Note that the new test run (D_4) actually performs better than the old test runs, i.e., the rules did not flag a regression, but rather a speed-up. Heuristics could be added to our approach to only flag true regressions.

No counter was flagged by either the bagging or stacking approach. This can be explained by the fact that separate models are learned from test runs D_1, D_2, and D_3. In order for a counter to be considered problematic, the counter must be flagged by at least two models (to achieve an aggregated weight of 0.5). Even though three counters (database CPU utilization, database I/O writes, and # orders/minute) were flagged by the model generated from test run D_2, none of these counters could be confirmed by the models generated from test runs D_1 or D_3. The precision, recall and F-measure of our ensemble-based approaches are 1.

Analysis of Test Run $\mathbf{D_5}$: In test run D_5, we injected a database-related bug that affects the product browsing logic of the system. 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 (MySQL) and application server (Tomcat).

Test Run D_1 as Training Data. In this scenario ($D_5^{(1)}$ in Table IV), we want to evaluate the performance of our ensemble techniques in cases where the new test run and the prior test run share exactly the same environment.

All three approaches flagged two database counters (# disk reads/s and CPU utilization), one Tomcat counter (CPU utilization), and one application-level counter (response time). The precision and recall of all approaches is 1 and 0.57, respectively. The F-measure is 0.73. Basically, when only one test is used as the training data, our ensemble-based approaches reduce to our original approach.

Test Runs D_1, D_2 and D_3 as Training Data. In this scenario ($D_5^{(3)}$ in Table IV), we want to evaluate the ability of our approaches to detect performance problems when the training data is produced from heterogeneous environments.

Our original approach flags the response time counter, which agrees with the nature of the injected fault. Since DS2 must process additional queries for each request, the overall response time suffered as a result. The precision and recall of our original approach are 1 and 0.14 (1/7), respectively, and the F-measure is 0.25.

Both the bagging and stacking approaches flagged the same counters: two database counters (# disk reads/s and CPU utilization), one Tomcat server related counter (# private bytes), and one application level counter (response time). Upon inspection, we found that the model generated from test run D_3 also flagged Tomcat's # threads counter. However, in the model generated from test run D_2, the rules that contained # threads as the consequent had premises that were never satisfied in test run D_5. As a result, even though the # threads counter behaved differently in test runs D_2 and D_5, the # threads counter was only flagged by the D_3 model. Since 4 out of 7 counters were flagged, our bagging and stacking approaches achieved a precision of 1 and a recall of 0.57. The F-measure of both ensemble-based approaches is 0.73.

TABLE VI: Summary of test setup for JPetStore.

Run	Hardware Setup	Fault Description	Expected Problematic Metrics
J_1	MySQL 5.1.45	No fault	No problem should be observed
J_2	MySQL 5.0.1	No fault	No problem should be observed
J_3	MySQL 5.0.1	Cache disabled	Increase of database CPU utilization, # threads, # context switches, # private bytes, and # disk reads and writes in bytes/s.

B. JPetStore

JPetStore [7] is a re-implementation of Oracle's original J2EE Pet Store. Since JPetStore does not ship with a load generator, we use an external web testing tool to record and replay a typical scenario of a user browsing and purchasing items on the site [27].

Data collection: In this case study, we conducted three one-hour performance test runs (J_1, J_2, and J_3), all of which share the same hardware environments and workload. J_2 and J_3 use an older version of MySQL (ver. 5.0.1) than test run J_1 (ver. 5.1.45). J_1 and J_2 are used as training data, whereas J_3 is injected with an environment bug in which all caching capacities of MySQL are turned off. Such a bug simulates a typical operator error [29]. Table VI summarizes the environments used in the three test runs and the 6 counters expected to show performance regressions in J_3.

Analysis of Test Run $\mathbf{J_3}$: Our original approach detected a decrease in memory usage (# private bytes), and an increase in CPU utilization and # threads in the database. These observations align with the injected fault, as caching is turned off in the database, less (caching) memory is used during the execution of the test. Because of the extra workload of accessing the disk, the database in turn must create more threads to handle the otherwise unchanged workload, increasing CPU

TABLE VII: Summary of test setup for the enterprise system.

Run	Performance Analyst's Report	Our Findings
E_1	No performance problem found.	Our approach identified abnormal behaviours in system arrival rate and throughput counters.
E_2	Arrival rates from two load generators differ significantly.	Our approach flagged the same counters as the performance analyst.
	Abnormally high Database transaction rate	
	High spikes in job queue.	
E_3	Slight elevation of # database transactions/s.	No counter flagged.

utilization in the process. Our original approach has a precision of 1 and recall of 0.5 (3/6). The F-measure of our original approach is 0.66.

Our ensemble approaches flagged the following three counters: # private bytes, # IO reads/s, and # threads in the database. Hence, our ensemble approaches achieve the same performance as our original approach.

C. A Large Enterprise System

Our third case study is conducted on a large scale distributed enterprise system from our industrial partner. This system is designed to support millions of concurrent requests across several hundreds of machines. For each build of the software, a series of performance regression tests are done to uncover performance regressions.

Data collection: In this case study, we selected thirteen comprehensive (i.e., executing the core functionalities) 8-hour performance regression test runs from the organization's performance regression test repository. Most of these tests were run in labs with varying hardware specifications, and were conducted for a maintenance release of the system. For each run, over 2,000 counters were collected.

Out of the pool of 13 test runs, 10 test runs historically had received a pass status from the performance analysts (independent of our case study). We use those runs to derive association rule models. We evaluated the performance of the 3 newest test runs (E_1, E_2 and E_3) in the pool and compared our findings with the performance analysts' assessment at the time (Table VII).

Analysis of Test Run $\mathbf{E_1}$: By analyzing the counters flagged by the union of the three approaches for test run E_1, we found that 13 counters (out of 2,000!) show true performance regressions. These 13 counters will be used to calculate the relative recall of our approaches for E_1.

Our original approach flagged 6 counters, including 2 throughput counters, 2 arrival rate counters, the # private bytes counter of the server process and the # database transactions/s counter. The rules that flagged the counters imply that all throughput and arrival rate counters should be the same under normal circumstances. However, upon investigation, we found that for E_1 half of the arrival rates and throughput counters are high while the other half is low, suggesting that the performance regression might be due to a mismatch in the load created by the load generators of Test E_1. Our original approach achieves a precision of 1, a recall of 0.46 (6/13) and an F-measure of 0.63.

Our bagging approach flagged 18 counters. The counters flagged included the 4 throughput and arrival rate counters that were flagged by our original approach. Most of the flagged counters are side-effects of the mismatch of the arrival rate counters. For example, the CPU utilization of the system decreased because fewer requests were made due to a drop in one of the arrival rate counters. We verified our findings with a performance analyst, and found that 5 flagged counters were false positives, bringing the precision and relative recall of our bagging approach to 0.72 (13/18) and 1, respectively. The F-measure of our bagging technique is 0.84.

Our stacking approach flagged 13 counters, including the ones flagged by our original approach. Out of the 13 counters flagged, the performance analyst and we identified 2 false positives, bringing both the precision and relative recall of our stacking approach to 0.85 (11 out 13). The F-measure of our stacking approach is 0.92.

Analysis of Test Run $\mathbf{E_2}$: Our three approaches detected 15 unique counters with performance regressions in test run E_2. These 15 counters will be used to evaluate the relative recall of each of our approaches.

Our original approach flagged 7 counters in total, 1 of which was a false positive. The correct performance regressions flagged included 2 arrival rate and 2 job queue counters, and the # private bytes and # virtual bytes counters of the application process. The resulting precision, recall and F-measure are 0.86, 0.4, and 0.55, respectively.

Our bagging approach flagged 20 counters, 5 of which were false positives. The remaining 15 counters included the 7 counters that were flagged by our original approach. The additional counters reported by our bagging approach were mainly the side-effects of the regression detected in the arrival rate counters. For example, as one of the load generators pushed a higher than the expected load, the extra requests caused the system to read from the disk more often, leading to an increase in the # disk reads/s. The results of these extra requests were written to the disk, causing an increase in the # disk writes/s. Although these side-effects are the result of the higher testing load, they can help analysts investigate the ripple effect of the fault. Hence, side-effects should be considered as true positives. The precision and relative recall of our bagging approach are 0.75 (15/20) and 1, respectively. The F-measure of our original approach is 0.86.

Our stacking approach flagged 14 counters, 1 of which was a false positive. All counters flagged by our original approach were also flagged by our stacking approach. The precision and relative recall of our stacking approach are 0.93 (13/14) and 0.87 (13/15). The F-measure for our stacking approach is 0.90.

Analysis of Test Run $\mathbf{E_3}$: There are no true positives that were detected by any of our three approaches.

Our original approach did not flag any rule violation for this test run. Upon inspection of the historical values for the counters reported by the performance analyst, we noticed that the increase of # database transactions/s observed in test run E_3 actually fell within the counter's historical value range. Upon discussing with the Performance Engineering team, we concluded that the increase did not represent a performance

problem, contradictory to the results of the team's earlier (manual) analysis. In this test run, our original approach of using a historical dataset of prior tests is resistant to fluctuations of counter values. Since no counter was flagged, the precision, relative recall and the F-measure of our original approach are 1.

Our bagging approach flagged 8 counters, all of which were false positives. Two counters flagged by our bagging approach indicated that one of the load generators output service requests at a higher rate (11%) than the other one. The extra service requests led to a slight increase in the # disk reads/s counter. Upon investigation, we do not believe that these two counters represent significant performance regressions. Hence, these counters are considered as false positives, leading to a precision and relative recall of our bagging approach of 0 and 1. The F-measure of our bagging approach is 0.

Since our stacking approach did not flag any counter, the precision and the relative recall of our stacking approach are 1. The F-measure of our stacking approach is 1.

V. Discussion and Limitations

As can be seen in Table IV, our stacking approach improves over the performance of our original approach and performs slightly better than our bagging approach. The precision of the three approaches is similar (the averages are a bit skewed because of the zero outliers of our original approach and bagging). Recall-wise, bagging performs the best, since it aggregates the results of individual models by voting, without incorporating the environment information of performance test runs. This recall goes at the expense of a slightly lower precision. Stacking has the same recall values, except for test runs E_1 and E_2. Our original approach behaves sub-par in significantly heterogeneous environments, i.e., runs $D_5^{(3)}$, E_1 and E_2. Overall, the combination of high precision and recall makes stacking the best approach (highest F-values), followed by bagging, then by our original approach.

Feedback from Practitioners. Over the past 4 years, the presented approach has been used on a daily basis for the performance analysis of a very large scale enterprise system. Analysts (other than the authors) liked that the ensemble approaches could detect problems that are often missed by manual analysis (e.g., E_1 and E_3). A key benefit of our approach was its support for investigation of the observed regressions by providing information about correlating counters and by marking periods where the regression occurred (see Figure 3). Using the marked time periods and the metrics, an analyst can refer to the logs in these time periods and work with developers to investigate the noted regressions.

The analysts also liked the upkeep cost of the approach. When a new test lab is configured, the similarity between the different environments is measured once using our simple similarity metric. From there on, no additional intervention is needed. Interestingly, the practitioners found that the approach works even better for tests deployed on virtual machines and cloud systems, since (1) those environments suffer even more from heterogeneity (since it is very easy to switch environments), while (2) the similarity in test environment can be automatically calculated based on the configuration files describing the virtual machines [1].

Furthermore, for the past 4 years, no changes have been needed to any of the used thresholds, even though the approach has been used across several major versions. That said, others interested in adopting this approach in practice might consider investigating the thresholds used by us. While we did use the default thresholds for the association rule mining tool since they were quite successful for industrial adoption (and in the two open source case studies), better tuned (i.e., non-default) thresholds are likely to improve the results, potentially at the expense of making the approach too specialized for the particular system. However, our industrial partners noted that *psychologically* thresholds lower than 50% are much harder to sell, e.g., "this passes even though we saw it in just 20% of the runs". This is important, since an analyst needs to get manager and developer buy-in for the results.

Finally, the analysts felt that some type of human intervention will always be needed. In particular, as a system evolves its performance signature might change considerably and hence the analysts should be given the option to remove old tests from the repository. While an ensemble of models can reduce the impact of outdated performance behaviours by spreading the weights across models, the performance of our approach will suffer if we allow the size of the ensemble to grow unbounded. We have developed a sliding window approach that discards tests when they no longer reflect the current system performance. An analyst can also manually remove old tests, if he felt that a test is not representative.

Limitations. We evaluated our work on three systems. These numbers seem low and ideally we would like to perform much larger experiments, yet there are a number of factors to take into account. First, realistic performance tests are complex to design and execute, as they require large and distributed labs with multiple machines, each hosting a different component, and must be run for extended periods of time. For that reason, we analyzed not only open source systems, but also an existing industrial large-scale system. Second, it is hard to get access to performance counter data for large software systems. In that sense, our data is rather unique. Third, we had to compare and discuss the enterprise results with the performance engineering team, as well as explore trends and patterns in the 2,000 collected counters. Fourth, seeding realistic bugs in the open source systems is time-consuming since we chose to implement both programmatic and configuration faults to simulate common mistakes observed in industry [21], [24].

We have made available a replication package to allow other researchers to build and expand on our work [32]. For example, given the many choices for thresholds and algorithms (e.g., similarity of environments or weight of a model), more research is needed to explore the configuration state space. Furthermore, if the behaviour of a new test is radically different from prior test runs, our ensemble approaches will not be accurate. Hence, in the future, other ways to automatically pick the most suitable ensemble of test models from a test repository should be investigated.

VI. Conclusion

The heterogeneity of the test environments has limited the widespread adoption in practice of performance regression analysis techniques. In this paper, we propose to use ensemble models (bagging and stacking) to compose individual models of the expected behaviour of prior test runs. The composition tries to take into account those prior test runs with the most closely related test environment. Case studies on two major open source systems and one large enterprise system show that (1) the ensemble techniques outperform our state-of-the-art environment-agnostic approach, and that (2) stacking consistently performs better than bagging in terms of precision and F-measure. Feedback from practitioners has been very positive.

Acknowledgments

We are grateful to BlackBerry for providing access to the used enterprise system. 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 in any way reflect the quality of BlackBerry's products.

References

[1] Puppet labs: It automation software for system administrators. http://puppetlabs.com/.

[2] A. Avritzer and B. Larson. Load testing software using deterministic state testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1993.

[3] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 1995.

[4] A. Avritzer and E. J. Weyuker. The role of modeling in the performance testing of e-commerce applications. *IEEE Transactions on Software Engineering*, 2004.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[6] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 1999.

[7] C. Begin. ibatis jpetstore. http://sourceforge.net/projects/ibatisjpetstore/.

[8] P. Bodík, M. Goldszmidt, and A. Fox. Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems. In *Proceedings of the SYSML workshop*, 2008.

[9] L. Breiman. Bagging predictors. *Machine Learning*, 1996.

[10] D. Breitgand, M. Goldstein, E. Henis, and O. Shehory. Performance management via adaptive thresholds with separate control of false positive and false negative errors. In *Integrated Network Management*, 2009.

[11] C. Buchta and M. Hahsler. arules: Mining association rules and frequent itemsets. http://r-forge.r-project.org/projects/arules/.

[12] L. Bulej, T. Kalibera, and P. Tma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 2005.

[13] 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 *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[14] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[15] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of ICML*, 1995.

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

[17] 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 10th International Conference on Quality Software*, 2010.

[18] M. J. Harrold. Reduce, reuse, recycle, recover: Techniques for improved regression testing. In *Proceedings of the IEEE International Conference on Software Maintenance*, 2009.

[19] J. Herrington. Five common php database problems. http://www.ibm.com/developerworks/library/os-php-dbmistake/index.html, 2006.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, 1994.

[21] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward. System monitoring with metric-correlation models: problems and solutions. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC)*, 2009.

[22] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2009.

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

[24] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2009.

[25] H. Malik, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. Using load tests to automatically compare the subsystems of a large enterprise system. In *Proceedings of the Annual Computer Software and Applications Conference*, 2010.

[26] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[27] Neoload. http://www.neotys.com.

[28] L. Northrop. Ultra-large-scale systems – the software challenge of the future. Technical report, SEI, Carnegie Mellon, June 2006.

[29] D. Oppenheimer and D. A. Patterson. Architecture and dependability of large-scale internet services. *IEEE Internet Computing*, 2002.

[30] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[31] R. Quinlan. Bagging, boosting, and c4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, volume 5, 1996.

[32] Replication package. https://dl.dropboxusercontent.com/u/4803154/FooRep.zip.

[33] Rpe2 relative server performance. http://www.ideasinternational.com/IT-Buyers/Server-Performance.

[34] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[35] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Proceedings of the Future of Software Engineering track (FOSE)*, 2007.

[36] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

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