AUTOMATED DISCOVERY OF PERFORMANCE REGRESSIONS IN ENTERPRISE APPLICATIONS

by

King Chun Foo

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

January, 2011

Author's Declaration for Electronic Submission of a Thesis

Abstract

Performance regression refers to the phenomena where the application performance degrades compared to prior releases. Performance regressions are unwanted side-effects caused by changes to application or its execution environment. Previous research shows that most problems experienced by customers in the field are related to application performance. To reduce the likelihood of performance regressions slipping into production, software vendors must verify the performance of an application before its release. The current practice of performance verification is carried out only at the implementation level through performance tests. In a performance test, service requests with intensity similar to the production environment are pushed to the applications under test; various performance counters (e.g., CPU utilization) are recorded. Analysis of the results of performance verification is both time-consuming and error-prone due to the large volume of collected data, the absence of formal objectives and the subjectivity of performance analysts. Furthermore, since performance verification is done just before release, evaluation of high impact design changes is delayed until the end of the development lifecycle. In this thesis, we seek to improve the effectiveness of performance verification. First, we propose an approach to construct layered simulation models to support performance verification at the design level. Performance analysts can leverage our layered simulation models to evaluate the impact of a proposed design change before any development effort is committed. Second, we present an automated approach to detect performance regressions from results of performance tests conducted on the implementation of an application. Our approach compares the results of new tests against counter correlations extracted from performance testing repositories. Finally, we refine our automated analysis approach with ensemble-learning algorithms to evaluate performance tests conducted in heterogeneous software and hardware environments.

Acknowledgements

I would like to thank Professors Ying Zou and Ahmed E. Hassan for their guidance and support in this research and the opportunity to visit Research In Motion (RIM) during the second year of my study.

In addition, I would like to thank Zhen Ming Jiang and Dr. Bram Adams from SAIL, and Parminder Flora from the Performance Engineering team at RIM for their fruitful suggestions and tireless reviews that greatly improved my work.

Finally, I would like to express my gratitude to my parents, Gloria and John, for their decision and sacrifice to allow me to study abroad, and my aunt and uncle, Margaret and Don, for their care and advice after I moved to Canada.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the years, software applications are continuously updated in response to new requirements or bug fixes. For example, applications may change to support new usage scenarios or protocols. As the size and complexity continue to grow, software applications may become vulnerable to performance degradation such as a slow-down of response time, or higher than expected resource utilization. Such phenomenon where the application performance degrades compared to prior releases is known as performance regression. Previous research notes that most problems experienced by the end users are related to the application performance [3] [61]. Consequently, the ability to detect performance regressions in software applications is of prime importance to organizations.

Currently, organizations make use of performance tests as the primary means to carry out performance verification on applications [11]. In contrast to traditional regression testing, which focuses on verifying the functional correctness of code changes, performance testing focuses on measuring the response of an application under load and uncovering evidence of performance regressions [3] [4]. A load typically resembles the usage patterns observed in a production environment and contains a mix of scenarios [33]. For example, the MMB3 [23] benchmark describes how typical users access the services provided by the Microsoft Exchange Server. The MMB3 benchmark specifies that a user will send 8 emails per day on average, 15% of which are high priority, and another 15% have low priority. A performance test usually runs such a load for hours or even days, during which execution logs and hundreds of performance counters (e.g., response time and CPU utilizations) about the running application are recorded. After the test, performance analysts first compare counter averages against pre-defined thresholds to flag counters that are at alarming levels. The analysts then selectively compare other counters in an attempt to uncover any performance regressions in the applications [24]. Analysis of performance tests is a manual process that would typically take up to a few days. All data collected during a performance test as well as its analysis are archived into the performance testing repositories for bookkeeping purposes.

1.1 Challenges in discovering performance regression

Performance verification is usually the last step in an already delayed schedule [34]. At Design changes that may have negative impact on the application performance are usually not evaluated until those changes are reflected in the source code. At this late stage, performance regressions introduced by the design changes are often the most difficult and expensive to fix [52] [53]. While design changes can be evaluated with performance modeling, existing analytical modeling approaches show very limited industrial adoption due to the steep learning curve involved [61]. In addition, the pressure to release on time and the high cost associated with analyzing the result of performance verification prevent organizations from thoroughly evaluating the performance of the updated applications on custom environment configurations. As a result of insufficient testing, a large number of performance regressions slips into production and are experienced by the end users [3] [4] [19].

The implementation of the application is tested through performance testing. The analysis of the results of performance tests is both time-consuming and error-prone due to the following factors [34] [45]:

1. **No documented application behavior**: Correct and up-to-date documentation of application behavior rarely exists. A performance analyst must often exert her own judgment to decide whether an observation constitutes a performance regression. As

a result, performance analysts often overlook potential performance regressions that may exist in a test, leading to the wrong conclusions being drawn.

2. **Large volume of data**: During a test, a large number of counters is collected. It is difficult for performance analysts to manually look for counter correlations and instances where the correlations are violated.
3. **Time pressure**: Performance testing is usually the last step in an already delayed schedule. In an effort to ship the product on time, managers may reduce the time allocated for performance testing and analysis.
4. **Heterogeneous environments**: Organizations maintain multiple labs to conduct performance tests in parallel. Each lab may have varying hardware and software configurations due to inconsistent upgrades and the need to certify the application under different environments. This further complicates the analysis of performance tests as the analysts must consider the environment differences between tests.

Although organizations employ automated tools to detect performance regressions, these tools are typically threshold-based tools which detect performance regressions based on violations of pre-defined thresholds. Threshold-based tools are incapable of providing information about the violations that can help analysts to diagnose the cause of the performance problems. Furthermore, performance testing repositories, which contain a rich history about the past performance behavior of an application, can be useful in diagnosing the behavior of new releases of the application. These repositories are rarely used by the analysts when reviewing performance tests.

1.2 Thesis statement

The current practice of performance verification is ineffective in discovering performance regressions at the design and implementation levels of an application. Existing analytical approaches to evaluate design changes require a steep learning curve that discourages the industry from applying these approaches. Furthermore, the current approach to analyze the results of performance tests is error-prone and requires considerable effort. We believe a systematic and automated approach can improve the effectiveness of performance verification at both levels. For example, at the design level, simulation models, which avoid the use of complex mathematical concepts, would present a lower learning curve to the performance analysts. Performance analysts can leverage the simulation model to evaluate the performance impact of a proposed change of design. While at the implementation level, the industry will benefit from an automated approach to discover performance regressions from new performance tests by mining the repositories of performance testing results.

1.3 Our approach

To uncover performance regressions in a tight release schedule, we propose to extend performance verification to not only cover the implementation but also the design of an application. For example, when the application design is updated, the new design should be compared to the old one so we can minimize the risk of introducing performance issues in the updated application. Furthermore, before the revised application is released, the application should be efficiently tested and analyzed to ensure that performance has not degraded from the previous version. Section 1.3.1 and Section 1.3.2 outline our approaches to discover performance regressions at the design and implementation levels respectively.

1.3.1 Discovery of performance regression at the design level

While performance verification can be conducted on the implemented application by pushing load onto the application binaries, evaluation of application designs can only be done through performance modeling. Although there is already a vast array of analytical modeling approaches available to construct and estimate performance from application designs, the construction and usage of analytical models demand a substantial level of expertise in the mathematical foundations on which the analytical models are based. Stakeholders such as end-users or developers, who may not have the proper training in performance modeling, may find it difficult to rationalize the analytical models and, as a result, will distrust the result derived from these models [61].

Many shortcomings of analytical models can be overcome with simulation models. A simulation model is a computer program that emulates the dynamic behavior of a software application. Simulation models can be implemented by performance analysts with the help of existing modeling frameworks. Visualization of application components and their communications patterns in a simulated model can also improve the understandability of the model. One shortcoming of simulation models is that they traditionally are constructed in an ad-hoc manner to test a specific aspect of the application for a stakeholder [3]. There is no systematic approach to construct a simulation model with appropriate level of detail that can address performance concerns of multiple stakeholders.

To address the above challenge for adopting performance verification at the design level, we propose in this thesis an approach to create layered simulation models. These layered models separate different concerns of stakeholders and can be used to evaluate the performance impact of changes to the existing design of applications.

1.3.2 Discovery of performance regression at the implementation level

In this thesis, we propose to mine the performance testing repositories to support automated performance regression verification of new application releases. Our approach captures the correlations among performance counters in the performance testing repositories in the form of performance signatures. Violations of these signatures in a new performance test are flagged as potential performance regressions.

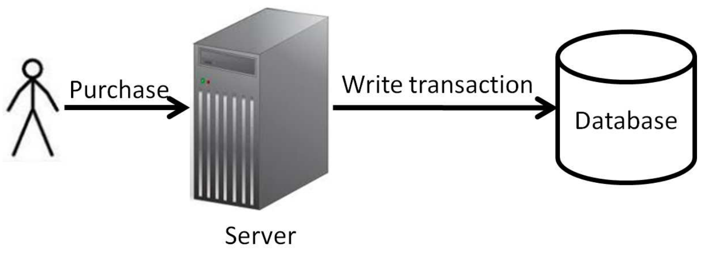


Figure 1-1 Purchase processing in an e-commerce application

In an e-commerce application deployment denoted in Figure 1-1, as visitors make purchases on the site, transaction records are stored in the database. As a result, a strong correlation between the visitor arrival rate, the application server's CPU utilization, and the database disk's # writes/second can be extracted as a performance signature. In a new version of the software with the same visitor arrival rate, a bug that leads to deadlocks in the database would result in a drop in the number of database disk writes/second counter, causing the counter to deviate from the previously extracted correlations and violate the performance signatures. As a result, a performance regression is found in the number of disk writes/second counter.

The rest of this chapter consists of the following parts: Section 1.4 briefly discusses the contributions of this thesis. Section 1.5 presents the organization of the thesis.

1.4 Thesis contribution

In this thesis, we introduce automated approaches to support performance verification at the design and implementation levels. In particular, our contributions are as follows:

1. We propose to analyze design changes through simulation modeling. An approach to create layered simulation models is introduced. Our layered simulation models can aid

stakeholders to understand the structure and performance of an application by separating different aspects of an application. Application designers can use the layered simulation model to estimate the performance impact of a proposed design change, thus reducing the risk of performance regression.

2. We introduce an automated analysis approach for discovering performance regressions from performance tests. The expected counter correlations and visualizations produced by our approach can aid performance analysts to diagnose the cause of a performance regression.
3. We further refine our automated analysis approach to analyze performance tests conducted with heterogeneous software and hardware environments. Our approach allows performance analysts to analyze new tests using the results of tests conducted across labs.

1.5 Thesis organization

The rest of the thesis is organized as follow. Chapter 2 presents background on performance verification. Chapter 3 provides a literature review of existing work and practice on the topics related to our thesis. Chapter 4 presents our approach for building a layered simulation model for discovering performance regressions at the design level. Chapter 5 presents our automated analysis approach to discover performance regressions at the implementation level. Chapter 6 extends our automated analysis approach such that prior tests conducted with heterogeneous environments can also be used in the analysis. Chapter 7 concludes this thesis and discusses future work.

Chapter 2

Performance Verification

In this chapter, we explain approaches on evaluating application designs using performance modeling. In addition, we present existing approaches to organize models for addressing the concerns of different stakeholders. Finally, we give a description of the current practice of performance verification carried out on the implementation of software applications.

2.1 Performance verification at the design level

Performance verification at the design level can be carried out by performance modeling. Performance modeling is a structured and repeatable process of modeling the performance of an application [52] that captures performance-related aspects of the designs of applications. Performance-related information, such as response time and resource utilization at various arrival rates, are obtained by solving the performance models. For example, the Layered Queueing Network (LQN) model can be used in early development to estimate the average response time of service requests. Performance modeling can provide valuable information for system architects to catch bad designs early, and for developers to make informed decisions about potential performance hotspots [21] [52].

A good performance model should enable different stakeholders to understand the performance of an application without overloading the stakeholders, who may have different background, with unnecessary information [59]. Furthermore, visualization of the model can also greatly improve the understandability of the program design. In the following sections, we will review the two classes of performance models: analytical models and simulation models.

2.1.1 Analytical models

Analytical modeling approaches model software applications with mathematical equations and statistical concepts. Popular analytical modeling approaches include the LQN models. The inputs of an analytical model are the average arrival rate of requests and a set of average values that represent the (hardware and software) resource usage of each request [21]. Performance behavior and resource utilization of software applications can be derived from analytical models by solving a set of mathematical equations. The construction and usage of analytical models, however, demand a substantial level of expertise in performance modeling [61]. Recently, an approach to automatically generate LQN models from application traces is introduced [32]. Such an approach, however, requires the message traces of the application, which may not always be available. Moreover, the verification of the generated models still demands expertise on the modeling theory used in the generation process.

2.1.2 Simulation models

Simulation models emulate the runtime behavior of applications and can be implemented rapidly with the support of existing simulation libraries, such as OMNet++ [43]. Such libraries provide the building blocks for discrete-event simulation models. Discrete-event simulation refers to modeling approaches in which the application changes its state at discrete points in time [46]. The operation of the application is represented as a chronological sequence of events. The libraries provide mechanisms for implementing and visualizing simulation models, and support the parallel executions of simulations.

In contrast to analytical models, actual performance data is obtained by executing the simulation model against a simulation clock. The execution of an application in an 8-hour work day can be simulated in a matter of minutes. During the execution of a simulation model, various statistics about the simulated application, such as response time for each request, are collected.

Performance analysts can use these statistics to pinpoint bottlenecks in an application. Simulation models are usually created to test specific aspects about an application, depending on stakeholders' needs, e.g., the performance impact of the Java garbage collection. Because of the size and complexity of large enterprise applications, it is difficult to create and maintain a separate simulation model for each stakeholder.

2.1.3 Maintaining performance models to address different stakeholders' concerns

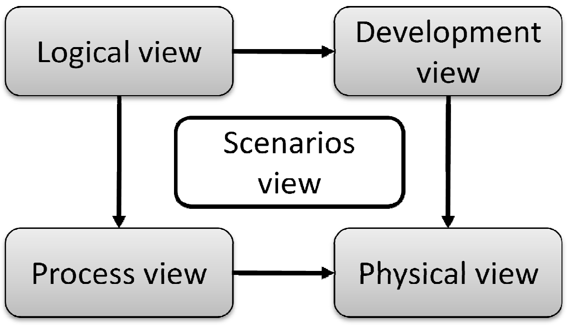


Figure 2-1: The "4+1" view model

The challenges associated with maintaining multiple models about a software application to address different stakeholders' concerns have already been faced by the software engineering community. For example, Kruchten proposed the 4+1 view model to document different aspects of a software application's architecture [39]. The 4+1 view model contains five concurrent views (Figure 2-1), each representing the viewpoint of a stakeholder:

- **Logical view:** The logical view focuses on the functional requirements of a software application and primarily targets the concerns of *end users*.
- **Process view:** The process view addresses the concerns of *system integrators*, who specialize in bringing together different components of the application. This view concerns the execution behavior of the application, e.g., concurrency, performance, and

scalability, and illustrates the execution of a set of independent processes, each made up of the components in the logical view.

- **Development view:** The development view considers the organization of software modules and mainly targets the concerns of *programmers and software managers*.
- **Physical view:** The physical view illustrates the application from a *system engineer's* perspective. This view describes how the software application is deployed and takes into account non-functional requirements such as reliability, availability and scalability.
- **The "Plus-one" view (Scenarios):** The plus-one view consists of a set of test cases and scenarios to show how the elements identified by the other 4 views work together. The plus-one view is useful for validating the software design.

Similar to software architecture, different stakeholders have different performance concerns about the same application. To avoid overloading the stakeholders with unnecessary details, a general purpose simulation model should allow stakeholders to study the application at the level appropriate to their knowledge, interest and experience.

2.2 Performance verification at the implementation level

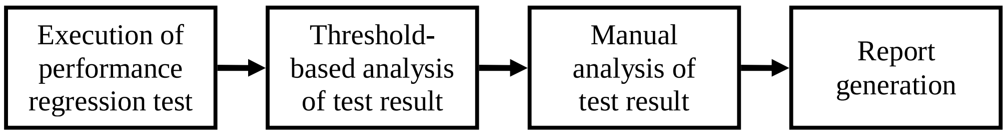

Execution of performance regression test
Threshold-based analysis of test result
Manual analysis of test result
Report generation

Figure 2-2: The process of performance verification

Currently, performance verification is performed at the implementation level to discover performance regression, ensuring that application updates would not degrade the performance of the application [61]. This section explains the current procedure of performance verification.

As shown in Figure 2-2, the typical process for the verification of performance regressions has 4 phases [24]:

1. Performance analysts start a performance test. During the course of the test, various performance counters are recorded. Performance counters include hardware counters (e.g., CPU utilization and # of disk writes/second) and software counters (e.g., response time, which measures how long the application takes to complete a request, and throughput, which measure how many requests an application can process in a given time).
2. After the completion of the test, performance analysts use tools to perform simple comparisons of the averages of counters against pre-defined thresholds.
3. Performance analysts visually compare the counters values of the new run with those of the past runs to look for evidence of performance regressions or divergences of expected counter correlations. If a counter in the new run exhibits deviations from past runs, this run is probably troublesome and requires further investigations. Depending on individual judgment, a performance analyst would decide whether the changes are significant and file defect reports accordingly.
4. Performance analysts would note any observed abnormal behaviors in a test report. Finally, all data and the test report are archived in a central repository for bookkeeping purposes.

Performance verification at the implementation level has two analysis phases to uncover performance regressions. During the threshold-based analysis, analysis tools automatically flag counters where the averages of the counters exceed the pre-defined thresholds. Threshold violations typically represent application instability and must be investigated. However, the threshold-based analysis is not effective for discovering performance regressions, because performance regressions may not be significant enough to violate the thresholds. To complement the threshold-based analysis, performance analysts would manually examine the counters to look for divergence of expected counter correlations. These counter correlations are defined by domain experts.

There are three major challenges associated with the manual analysis of counters.

1. **During the course of the test, a large number of counters is collected.** It is difficult for performance analysts to compare multiple counters at the same time. Although correlating counters specified by domain experts could be plotted on the same graph, these graphs only give a limited view of the correlations in order to avoid overloading the analysts. For example, in a performance test for an e-commerce website, one heuristic would be to plot the arrival rate and throughput counters in one graph and leave out other correlating counters like request queue length. These graphs may aid analysts to spot obvious deviations of expected correlations. However, the cause of the deviations may lie in other correlating counters that are not included in the graph.
2. **An up-to-date performance baseline rarely exists.** Performance analysts usually base the analysis of a new test on a recently passed test [13]. However, it is rarely the case that a performance test is problem-free. Using just one prior test as baseline typically ignores problems that might be common to both the baseline and the new test.
3. **The subjectivity of performance analysts may influence their judgment in performance verification.** Performance analysts usually compare the counters' averages between two tests, ignoring the fluctuations that might exist. Simply comparing averages may lead to inconsistent conclusions among performance analysts. For example, one analyst notes in her analysis a 5% increase of the number of database transactions per second to be worrisome while another analyst would ignore the increase because he feels that the 5% increase can be attributed to experimental measurement error.

Due to the above challenges, we believe that the current practice of the analysis phase of performance verification at the implementation level is neither effective nor sufficient. There is a high chance that performance analysts would overlook abnormal performance problems. In this thesis, we aim to reduce the analysis effort (phase 3 in Figure 2-2) by automating the detection of performance regressions in a performance test.

2.3 Summary

Current practice of performance verification focuses solely on discovering performance regressions at the implementation level. Often, because of the high cost associated with the analysis of performance tests and tight deadlines, design changes are not evaluated for performance regressions until those changes are already implemented. In this chapter, we provide the background on evaluating application designs with performance models. In addition, we give a description of the current practice of performance verification at the implementation level, and identified the major challenges that the analysts face when analyzing the results of performance tests. In the next chapter, we will survey the existing work related to performance verification at the design and implementation levels.

Chapter 3

Literature Review

In the previous chapter, we discussed performance verification at the design and implementation levels and identified the challenges associated. In this chapter, we present the prior work related to testing software applications for potential performance issues at the design and implementation levels.

3.1 Performance verification at the design level

Before an application is implemented, performance models can be constructed based on the design documents. Sections 3.1.1 and 3.1.2 summarize the existing approaches to create analytical and simulation models.

3.1.1 Analytical modeling approaches

The Queueing Network (QN) model has been studied extensively by researchers to analyze the performance of software applications. A QN model represents an application by a network of resources for which a request must be obtained in order to be serviced. Due to the finite service rate of each resource, requests may need to wait in queues for the availability of resources. Extensions of QN models are proposed to analyze different types of applications. Table 3-1 summarizes different QN models.

Table 3-1: Summary of approaches based on QN model

QN models	Types of application suitable to be modeled
Open QN [6]	Applications with jobs arriving externally; these jobs will eventually depart from the applications.
Closed QN [6]	Applications with a fixed number of jobs circulating within the applications.
Mixed QN [6]	Applications with jobs that arrive externally and jobs that circulate within the applications.
SQN-HQN [41]	
SRN [60] [63]	Distributed applications with synchronous communication.
LQN [25] [49]	Distributed applications with synchronous or asynchronous communication.

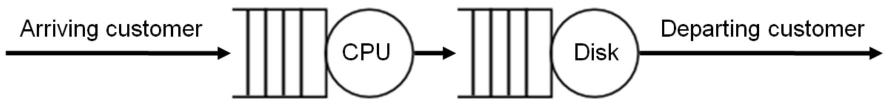


Figure 3-1: Open queueing network model

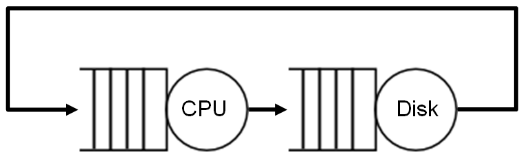


Figure 3-2: Closed queueing network model

Baskett et al. proposed algorithms to solve the open, closed and mixed QN models [6]. Open QN models are used to model applications with external arrivals and departures. An example of an open QN model is shown in Figure 3-1 where customers purchase items and depart. Closed QN models (Figure 3-2) are used to model applications that have a fixed number of jobs circulating in the applications. Mixed QN models are used to model applications that have both open and closed workloads. Menascé proposed the SQN-HQN model to incorporate software contention into QN models analysis [41]. In an SQN-HQN model, applications are represented as two queueing networks: SQN and HQN, modeling the software and hardware resources respectively. In the QN models considered above, concurrency is not taken into account and the availability of each resource is assumed to be independent. These assumptions do not hold for distributed applications where dependencies between resources exist; that is, resource entities (e.g., software resources) can request services from other resource entities (e.g., software and hardware resources).

To accurately model concurrent distributed applications, Woodside et al. developed the Stochastic Rendezvous Network (SRN) model that can be used to model distributed applications with synchronous communication [60] [63]. Rolia et al. proposed the Layered Queueing Network (LQN) model that is capable of modeling distributed applications with synchronous and asynchronous communication [49]. Franks et al. extended LQN models to handle concurrent processes [25]. Woodside et al. proposed an automatic approach to create LQN models from traces captured from the communication between application components [62].

Woodside proposed a Three-View model for performance engineering of concurrent applications [59]. The three views in the model are constructed using existing analytical approaches. The views are connected by a "core model" that passes the result of one view to the input of another view. In contrast to the 4+1 view model, which documents the software architecture, Woodside's three-view model is suitable for documenting and analyzing performance information. However, the three-view model still poses a steep learning curve due to the use of analytical modeling approaches.

Performance can be derived from analytical models by solving a set of equations. Because of the use of complex formulas, knowledge encapsulated in an analytical model can be difficult to transfer. Furthermore, in order to update analytical models, performance analysts must possess a certain level of expertise in mathematics. Such expertise is not needed with simulation modeling approaches, which we discuss next.

3.1.2 Simulation modeling approaches

Compared to analytical models, fewer approaches exist for constructing simulation models of the performance of software applications.

Xu et al. used colored Petri Nets to model the architecture of software applications [65]. Approaches based on Petri Nets analyze the behavior of an application (e.g., the time needed to recover from an error). Xu et al. analyzed both time and space performance of an application by executing the simulation model. Bause et al. extended Petri Nets with queueing networks such that shared resources can be modeled easily [8]. However, performance information such as hardware utilization cannot be derived from Petri Net models, making them unsuitable for analysis of performance regression.

Smit et al. proposed a simulation framework to support capacity planning for Service-oriented architectures (SOA) [54]. Each service in an SOA-based applications is modeled as an entity that can send and receive messages. Smit's framework focuses solely on deriving the response time of completing a request by modeling the interaction between software services. The purpose of performance verification is to compare the usage of software and hardware resources between the old release and the new release of the application. Since Smit's framework does not consider hardware resources, this framework would not be suitable for our task of supporting performance verification at the design level.

3.2 Performance verification at the function level

As developers implement bug fixes and new features, these code changes may introduce performance regressions within functions. Such regressions can be detected with a variant of unit testing that focuses on performance. Traditionally, unit testing is a method for testing individual units of source code, typically a function, to determine if they are functionally correct. Packages such as JUnitPerf [38] and NPerf [28] are extensions to existing unit testing frameworks that enable developers to measure the performance of individual code units. While unit testing focuses on the performance of individual functions, our concern in this thesis lies in the overall application performance where software components work together to handle the service requests.

3.3 Performance verification at the implementation level

The implementation of an application is tested for performance issues by pushing workload through the application. There are three active research areas for evaluating the performance of applications at the implementation level: test case generation, test reduction (shortens the time needed to test an application), and analysis of test results. In this section, we focus our discussion on the analysis of test results. Existing approaches to analyze the results of performance tests are divided into two classes, depending on whether the approaches analyze the execution logs or performance counters.

3.3.1 Analyzing performance tests using execution logs

Execution logs describe the runtime behavior of applications and are readily available in existing applications without instrumentation [5] [16]. Reynolds et al. [48] and Aguilera et al. [2] develop several approaches to reconstruct the execution paths for requests from the communication traces between the components of an application. An execution path describes the sequence of application components involved in processing a request. By analyzing the execution paths, components that account for most of the latency can be determined. However, the accuracy of the extracted execution paths decreases as the level of concurrency in the application increases.

Jiang et al. introduce an approach to identify functional problems from execution logs [33]. In Jiang's approach, each log line is considered as one execution event. Jiang's approach uncovers the dominant behavior for an application by analyzing the frequency that two adjacent events occur together. Execution anomalies can be flagged by identifying execution sequences that deviate from the dominate behavior. Jiang et al. extended this approach [34] to uncover sequences that contain more than two events. By comparing the response time distribution of each dominant behavior across two releases of the same application, we can identify the execution events that show performance regression.

Usage of hardware resources is rarely recorded in the execution logs. Approaches that analyze execution logs to detect abnormal behaviors during a performance test are not capable to identify cases where the usage of hardware resources has increased. An important task of performance verification is to detect changes in the utilization of hardware resources. Approaches that rely solely on execution logs would not be comprehensive enough to detect performance regressions in hardware resources.

3.3.2 Analyzing performance tests using performance counters

Another source of performance information, typically recorded during tests, are the performance counters. Approaches for analyzing performance counters can be further divided into two categories: Supervised and unsupervised. The application of each class of approach depends on whether or not the performance counter is labeled. A label describes the state of the application at each point in time, e.g., whether or not an application is in compliance with its Service Level Objectives (SLO) as defined in the requirements.

3.3.2.1 Supervised approaches for analyzing performance counters

Supervised approaches analyze labeled performance counters to derive classifiers for each type of SLO. A classifier describes the conditions (e.g., a subset of counters reaching particular levels) that would most likely lead to the observed SLO state. Cohen et al. apply supervised machine learning approaches to train classifiers on performance counters that are labeled with SLO violations [17] [18]. Bodik et al. improve the accuracy of Cohen's work by using logistic regression as the classifier algorithm [10].

Zhang et al. extended Cohen's work by maintaining an ensemble of classifiers [67]. When an SLO violation is detected, a classifier is selected from the ensemble based on the Brier score to report the counters that correlate the most with the particular SLO violation. The Brier score is a statistical measure that assesses how well a model fits the observed event. Similar to Zhang's approach, Jin et al. proposed an approach to derive classifiers from data coming from multiple sources [37].

Supervised approaches only work when counters exceed the thresholds defined in SLOs. For performance verification, it is difficult to use hard thresholds to quantify whether performance regression has occurred because performance may degrade without exceeding the thresholds.

3.3.2.2 Unsupervised approaches for analyzing performance counters

Jiang et al. propose an approach to use pair-wise correlations to detect performance problems [36]. Jiang's approach first uncovers pairs of counters that are highly correlated to each other. Violations of the correlations are reported as anomalies. Bulej et al. propose to compare the performance between two tests by clustering the recorded response times with the k-means clustering algorithm [15]. The accuracy of Bulej's approach depends highly on the quality of the clusters generated by the k-means algorithm. Malik et al. use Principal Component Analysis (PCA) to recover clusters of counters that are correlated to each other. These clusters are used to identify the subsystems in a new performance test that show anomalous behaviors [40]. Similarly, Jiang et al. propose an approach to identify correlating counter clusters with Normalized Mutual Information [35].

The above studies assume that different hardware and software environments used to conduct performance tests will not affect the underlying counter correlations. In practice, application performance, especially for large scale applications with many components, is highly dependent on the software and hardware environment. Organizations maintain multiple labs such that tests can be executed in parallel. Each lab may have different software and hardware environments due to inconsistent upgrades or purchases. Furthermore, organizations would also purposefully test the applications on a variety of environments to gain confidence that the application would perform as expected when they are deployed in the customers' environment. The existing unsupervised approaches do not distinguish the performance differences resulting from heterogeneous environments, and risk producing incorrect conclusions about the performance of the application.

3.4 Summary

In this chapter, we surveyed existing research related to performance verification at the design, and implementation levels. While there exists a vast array of analytical modeling approaches, these approaches require a steep learning curve, preventing wide adoption in industry. Existing approaches for performance simulation, on the other hand, do not model the usage of hardware resources, making these approaches less attractive for performance analysis. At the implementation level, approaches to analyze performance tests with performance counters do not take into account the differences of software and hardware environments, making them difficult to adopt in practice. In the next chapter, we will present our approach to construct simulation models that are suitable for performance verification at the design level.

Chapter 4

Discovering Performance Regressions at the Design Level

Application designs evolve over time. To ensure that design changes do not cause performance regressions later on in the updated applications, a good understanding of the performance impact brought by these changes is needed. While traditional modeling approaches enable performance analysts to evaluate application designs without referring to the implementation, these approaches are often not suitable for all stakeholders due to their abstract mathematical and statistical concepts. In this chapter, we present our framework for constructing layered simulation models for the purpose of analyzing the performance regressions brought by software design changes. Simulation models constructed with our approach contain multiple layers of abstraction, each addressing the performance concerns of a different group of stakeholders. As a proof-of-concept, we conducted two case studies. One study is on a new system for Really Simple Syndication (RSS). The system actively delivers notifications of newly published content to subscribers. Another study is on a performance monitor for an ultra-large-scale (ULS) application.

Table 4-1: Performance concerns of stakeholders

Stakeholder	Performance Concerns
End user	Overall system performance for various deployment scenarios
Programmer	Organization and performance of system modules
System Engineer	Hardware resource utilization of the running application
System Integrator	Performance of each high-level component in the application

4.1 Layered simulation model

Table 4-2: Mapping of our simulation models to the 4+1 view model

Stakeholder	Layer in Our Simulation Model	4+1 View Model
Architects, Managers End users Sales Representatives	World View Layer	Logical view
Programmers System Integrators	Component Layer	Development View Process View
System Engineers	Physical Layer	Physical View
All Stakeholders	Scenario	Scenario

In Chapter 2, we reviewed the 4+1 model, which uses five views to separate different stakeholders' concerns about the architecture of an application. Similar to software architecture, different stakeholders also have different performance concerns about the same application (Table 4-1). A good simulation model should separate each stakeholder's concerns in order to avoid overloading the stakeholders with unnecessary details. In this thesis, we propose a software simulation model that decomposes a software application into a three-layer hierarchy with an extra layer to describe the different usage scenarios.

As shown in Table 4-2, the layers in our model roughly correspond to the five views in the 4+1 model. Each layer addresses the concerns of a group of stakeholders. The process and development views from the 4+1 model concern the integration of individual components in an application and their performance; these views can be combined into a single layer – component layer – that captures the organization and functionalities of software entities. The runtime behavior of an application, as documented by the process view, is reflected by the execution of the simulation model. System integrators can examine the dynamic aspects of the application by monitoring the communication between the simulated components of the application.

The layers in our simulation model can be constructed incrementally from high to low level of abstraction as details about the software become available. A partially complete model, e.g., a model that only contains high-level components such as databases and servers, can be used to guide the software design at the early stages of development. The following sections discuss in detail the purpose of each layer separately.

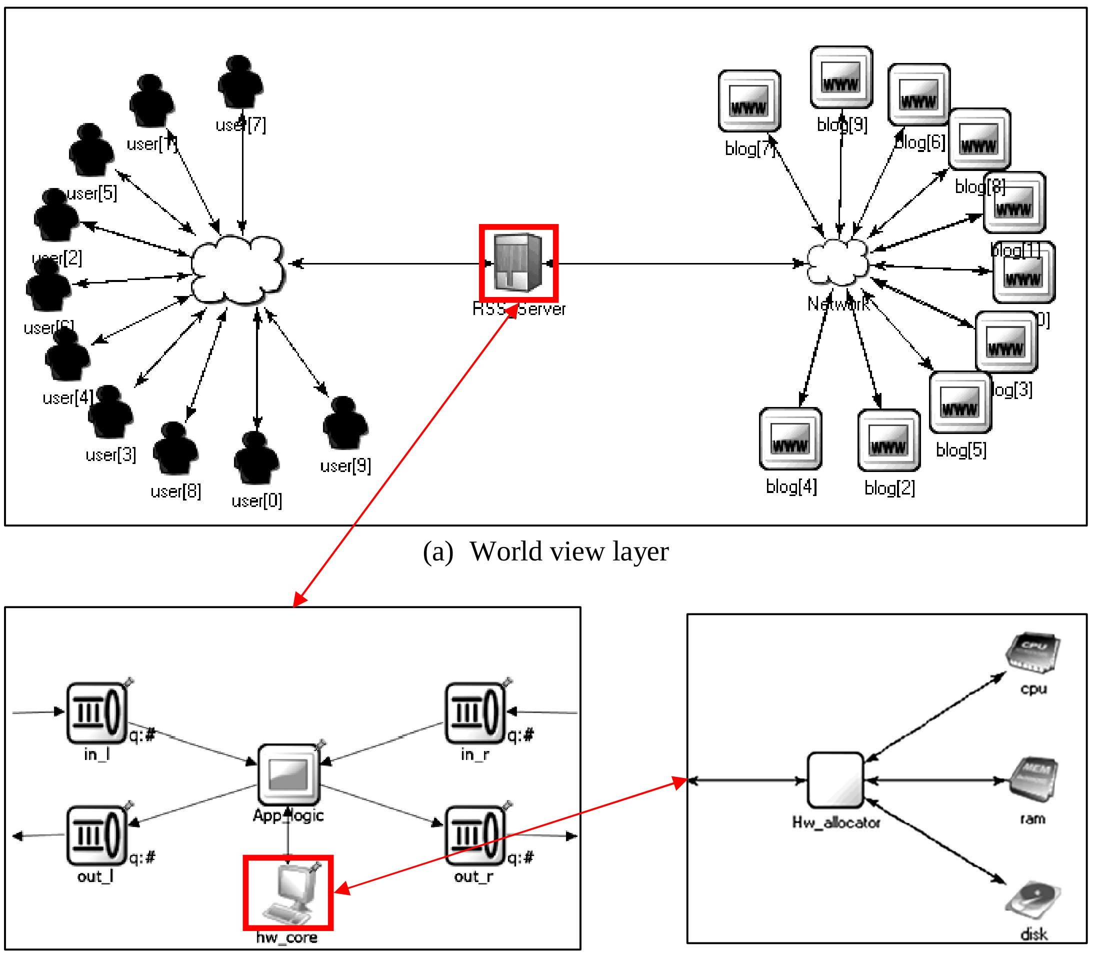
4.1.1 World view layer

The world view layer aims at addressing high-level, often business-oriented, concerns such as evaluating whether the current infrastructure of the application can support the projected growth in the customer base. This layer is the top and most abstract layer in our model. The world view layer represents the high-level application components and their relations as a network of nodes and edges.

Initially, each high-level component in the world view layer represents a place-holder for the logic that will be added by other layers. When only the world view layer exists, performance analysts can assign rough resource estimates to the place-holders in this layer for initial analysis.

In a complete model, the world view layer hides the details of the software application and can be used to measure the performance impact of adding new nodes to a distributed application or to test different deployment scenarios. Because the world view layer is built on the foundation of the lower layers in our model, when running the simulation program at the world view layer, end-users transparently take advantage of the detailed logic provided by the layers below the world view layer, if those exist.

Figure 4-1 shows an example of a layered simulation model constructed for an RSS Cloud system. RSS [50] is a format for delivering frequently-updated content to subscribers. In an RSS Cloud system, an RSS server actively sends notifications of new content to the users. Table 4-3 summarizes the different components in each layer and the connections between them. Figure 4-1a shows the world view layer of the RSS Cloud system. The simulation model consists of three high-level components, i.e., the websites that publish personal journals – blogs, the users that subscribe to the blogs, and the RSS server through which each blog connects to its users. The bidirectional arrows in Figure 4-1a depict the two-way communication between components.



(a) World view layer

(b) Component layer

(c) Physical layer

Figure 4-1: Example of layered simulation model for an RSS cloud

Table 4-3: Components and connections in Figure 4-1

Layer	*Component*	*Has connection to*
World view layer	Users, blogs	RSS server
	RSS server	Users, blogs
Component layer	In Queues, out queues	Application logic
	Application logic	Input queues, output queues, hardware
	Hardware	Application logic
Physical layer	Hardware allocator	CPU, RAM, disk
	CPU, RAM, disk	Hardware allocator

4.1.2 Component layer

The component layer further decomposes each high-level component defined in the world view layer into logical entities. Similar to the world view layer, the components and the communication between them are represented as a network of nodes and edges (Figure 4-1b).

For example, the RSS server in Figure 4-1a can be broken down into a number of components: the software component that represents the application logic, and the input and output queues that act as communication channels from the server to other high-level components defined in the world view layer. Performance analysts can define different processing requirements, for instance the time required to process each type of service request, and the capacity of various logical resources such as the thread pool and queues.

Developers can leverage the component layer to understand the communication patterns in an application and the performance ramifications of handling different mixes of service request types or to study the performance of different threading models. During the execution of a simulation model, performance analysts can temporarily stop the simulation program and examine internal information such as queue size or network bandwidth consumption.

4.1.3 Physical layer

The physical layer connects the logical components in the component layer to the underlying hardware resources. The physical layer mainly targets the concerns of system engineers. Figure 4-1c shows three hardware resources in the RSS server: CPU, Memory, and Disk. Performance analysts can specify the hardware resource requirements for each type of service request. For example, a request to submit a new blog post may consume 50 kilobytes of Memory while the request is being processed. Using the Physical Layer, system engineers can study the behavior of resource utilization at different request rates. Furthermore, system engineers can use this view to determine the bottleneck of the application at high request rate.

4.1.4 Scenario layer

The scenario layer uses a set of test case scenarios to show how the elements defined in the three other layers work together. Scenarios from the end-user’s point of view include the different deployment scenarios and the composition of different types of service requests passed to the application. For the example depicted in Figure 4-1, one scenario could specify that there are four blogs connected to the RSS server, 50% of which are located in North America with an average bandwidth of 2 Megabits/second, and the rest are located in Europe with an average bandwidth of 1 Megabit/second. The scenarios define how the software components are deployed, and what workload is used in the simulation to estimate the application performance.

4.2 Model construction

The construction of a layered simulation model is an iterative process. The three layers in our proposed model can be constructed in a top-down fashion during different stages of the software development life cycle. For example, performance analysts would start with the world view layer to model the general deployment scenario of the application. The world view layer can initially be constructed according to the application specification or to similar products in the market. Estimates of resource requirements are given to each high-level component. The partial simulation model is run with the request arrival rates that are either observed from similar applications or derived from existing benchmarking standards. Our partial simulation model that contains only the world view layer is similar to the Software Execution Model, which is used in the early stage of software development, when only limited processing requirements are available [52] [53]. Similar to the software execution model, the accuracy of our partial simulation model should reflect the resource utilization and response time within 10% and 30% accuracy respectively.

As more details become available, performance analysts can improve the simulation model by extending each high-level component with the component and physical layers, and giving better resource estimates for different request types. Such an incremental model building approach requires programming libraries that support modular development of simulation model. With such programming libraries, one can initially use “place-holders” to represent high-level components. As more details become available, each place-holder can be expanded to model the logical and hardware resources.

All layers of the model are interconnected. As a request flows through the model, the request is passed between the layers transparently. For example in Figure 4-1, a request is first generated by a blog and passed to the internal components that make up the RSS server. The left- and right-most arrows in Figure 4-1b represent the incoming and outgoing ports of the RSS server for communication with other high-level components, such as the blogs and users.

4.3 Case studies

We conducted two case studies using our layered simulation models. In the first case study, we demonstrate the construction of our layered simulation model for an RSS Cloud system and how performance data that can be extracted from the model. In the second case study, we show how our layer-based simulation model can help to evaluate different design options for a performance monitor used to detect problems in ULS applications.

Our layer-based simulation model is created using the OMNet++ libraries. OMNet++'s compound modules provides the means to implement the three hierarchical layers in our model. Each layer of an application is composed of a collection of entities. Each entity contains a set of state variables to reflect the properties of the entity at any point in time during the simulation. The collection of state variables from all entities represents the overall state of the application.

4.3.1 Case Study 1: Constructing a layered simulation model for an RSS Cloud

In this case study, we demonstrate the process of constructing a simulation model for the RSS cloud system in Figure 4-1. By determining the application bottleneck, we show how performance information can be obtained from our simulation model. An application bottleneck is a phenomenon where the performance of the entire application is limited by a single component. In an example where the server CPU is the application bottleneck, the notification request rate may overwhelm the server's CPU capacity, resulting in a continuous growth of the request buffer usage. As a result, the average response time and throughput suffer.

4.3.1.1 Application description

In order to model an RSS cloud, we need to better understand the RSS communication protocol. The RSS protocol is heavily used by blogging service providers such as Wordpress.com [64] to publish new web content. The traditional "pull" mechanism used by RSS readers periodically queries the RSS server for updates of a specific feed. The pull mechanism introduces latency between when new content is published and when the update is received by the RSS readers. To eliminate the latency introduced by pull, the RSS cloud [51] – an extension to RSS – actively delivers notifications of newly published items to feed subscribers. This mechanism is also known as "push".

Because the RSS cloud requires the hosting server to actively send notifications for each new item, the push mechanism puts a heavy resource requirement on the blogging service provider's infrastructure. For example, each time new content is published, the hosting server must initiate a separate connection for each subscriber to send a notification. For a service provider that hosts hundreds of thousands of blogs with possibly millions of subscribers, the resource requirements of sending notifications would potentially exceed the available capacity. Furthermore, the large number of notifications may overwhelm online feed aggregator services such as Google Reader [27] that automatically download feed content for hundreds of thousands of users. The sections below document our experience of constructing a layered simulation model for the RSS cloud.

4.3.1.2 World view layer of the RSS Cloud system

Figure 4-1a shows the world view layer for the simulation model of the RSS cloud. In this layer, the service provider, which supports the RSS cloud extension, connects the various blogs to the subscribing users. When new content is published, the service provider will send a fixed-size (e.g., 5 kilobytes) notification to all subscribers. To simplify our simulation, we set the number of subscribers to vary in a normally distributed fashion around the mean of 20 subscribers per feed.

To ensure reliability, the subscribers will reply with an acknowledgement to the RSS server upon receiving the notification. The RSS server uses an internal timer to monitor the delivery of the notification. If an acknowledgment is not received before the timer expires, the RSS server will assume that the message is lost and will automatically resend the notification until the maximum number of resends for a subscriber is reached. The network connections between all entities are characterized by two parameters: bandwidth and latency. We vary these two parameters to model a realistic environment where subscribers and blogs are scattered globally.

4.3.1.3 Component layer of the RSS Cloud system

Figure 4-1b shows the component layer of the RSS server component. The RSS server has four major logical components: two pairs of IN and OUT queues that buffer the communications between the subscribers and the blogs, the “app_logic” component, which abstracts away the application logic of the RSS server, and the “hw_core” component, which represents the physical hardware platform on which the RSS server resides.

Two types of resources are required to process a notification in the RSS Cloud system:

- **Logical resources:** In our simulation model, there is one logical resource – the thread pool in the RSS server. Each notification request will be processed by a thread in the pool. If all threads in the pool are busy, the request will wait in the buffer of the input queue until a thread becomes available.
- **Hardware resources:** Each notification request received from the blog will consume a specific number of units from each hardware resource. If all resources are used up, the request will wait in the buffer until the required resources become available.

The overall resident time of a notification request in the RSS server is the sum of the wait time in the RSS server’s queue for acquiring the resources and the processing time required by the RSS server.

4.3.1.4 Physical layer of the RSS Cloud system

Figure 4-1c shows the hardware platform of the RSS server. In the simulation of the RSS Cloud system, we assume that each notification request will compete for three physical resources:

CPU, disk and memory. Each of these resources has a finite number of units that can be used for processing. Each request will require a certain number of units from each resource while the request is being processed. For example, we can specify in the simulation that our RSS server has access to 1 gigabyte of memory, and that each notification will hold 50 kilobytes of memory when it is being processed. The resources are released when the request is serviced.

4.3.1.5 Model validation

To ensure that the simulation model is specified correctly, we tested our model with a simplified use case where only one blog and one user are connected to the RSS server. In the test, all requests generated by the blog component are serviced by the RSS server and subsequently received by the subscriber. Through visualization of the simulation and detailed traces, we are able to verify the timing of the requests as they propagate through the components in the different layers of our model.

Table 4-4: Processing requirement for an RSS notification

Resource	**Requirement**
CPU	2 unit
RAM	5 KB
Thread	1
Processing time	2 seconds

4.3.1.6 Experiments and data collection

Performance analysts must determine if changes in the design of an application would result in performance regressions and whether or not such regressions would lead to application bottlenecks. We manually specify the resource requirements for processing an RSS notification request, and examine the RSS server performance by varying the notification request arrival rate.

Requests sent from the blogs to the RSS server are initially stored in the request buffer. Table 4-4 summarizes the resource requirements for processing each notification. Depending on whether or not the RSS server has enough resources (e.g., CPU, RAM, and a free thread) available, a thread will pick up the request from the buffer and allocate the required resources. Each request is processed for 2 simulated seconds during which all allocated resources are blocked by the thread. When the thread has completed processing the request, all previously allocated resources are released. The thread that has just been freed will pick up the next message in the request buffer.

We ran 10 simulations each simulating one hour of operation with arrival rates ranging from 5 to 25 requests per second. We collected various statistics from different layers of the simulation model of the RSS server (Figure 4-2 to Figure 4-4). For example, throughput and response time from the world view layer, number of threads used from the component layer, and CPU and RAM utilization from the physical layer. As evident from Figure 4-2 and Figure 4-3, both the application throughput and response time degrade at 17 requests/second. In order to determine the bottlenecks of the application, we examine statistics of resource utilizations (thread, CPU, and RAM) collected at the component and physical layers. Figure 4-4 shows that the CPU reaches 100% utilization at 17 requests/second while the thread and RAM utilizations are below 50%. A request is only processed if all required resources can be allocated at once. If the CPU does not have enough capacity to serve a request, the request will wait in the buffer until the CPU becomes available, regardless of the availability of other resources. In other words, the CPU prevents other resources from being fully used and is therefore the bottleneck of the application. To ensure that the application can handle future growth of request arrival rate, system engineers should recommend a faster CPU or increase the number of CPUs.

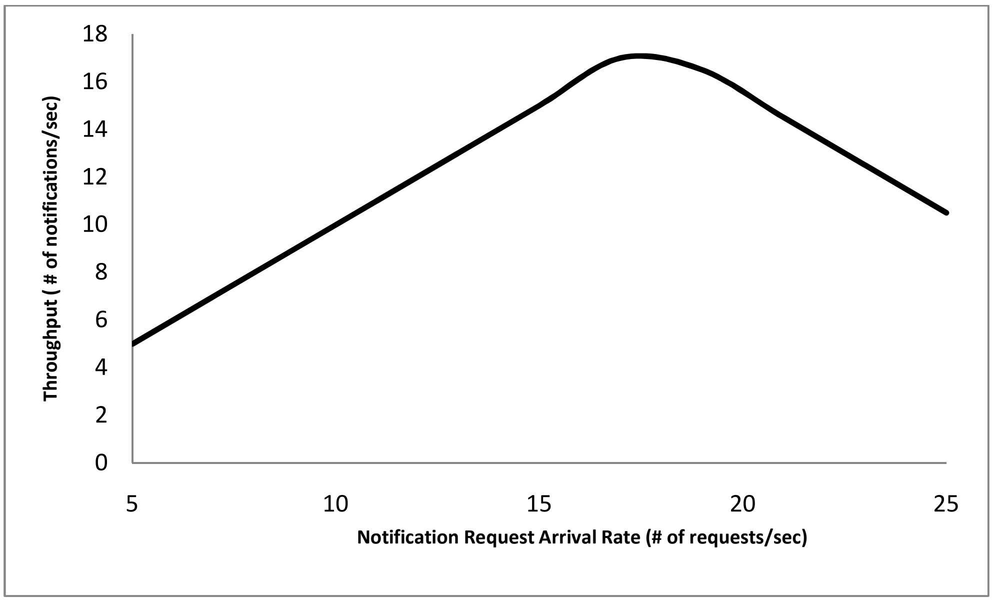


Figure 4-2: Plot of the throughput of the RSS server at various request arrival rates

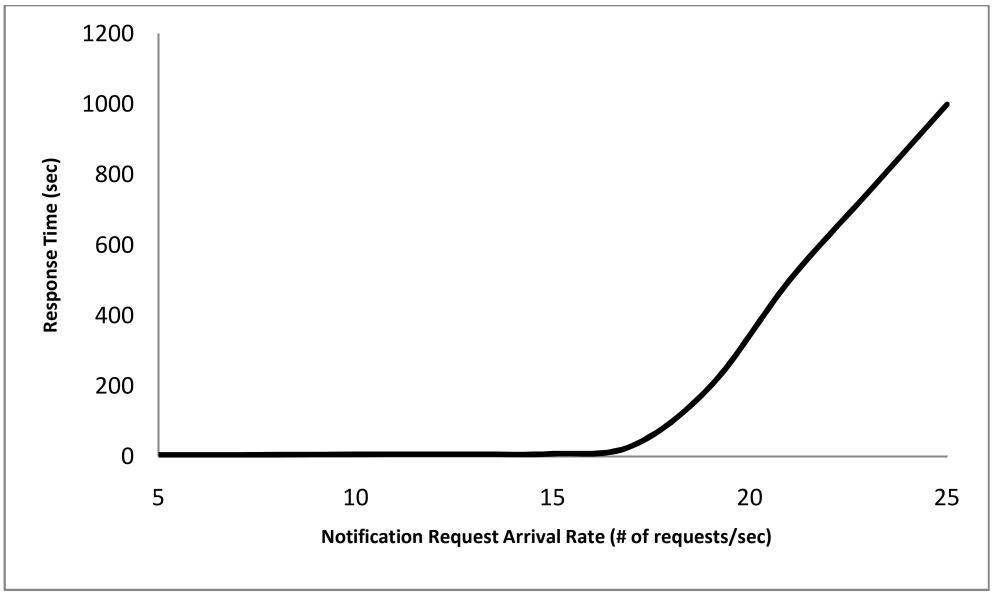


Figure 4-3: Plot of the response time of the RSS server at various request arrival rates

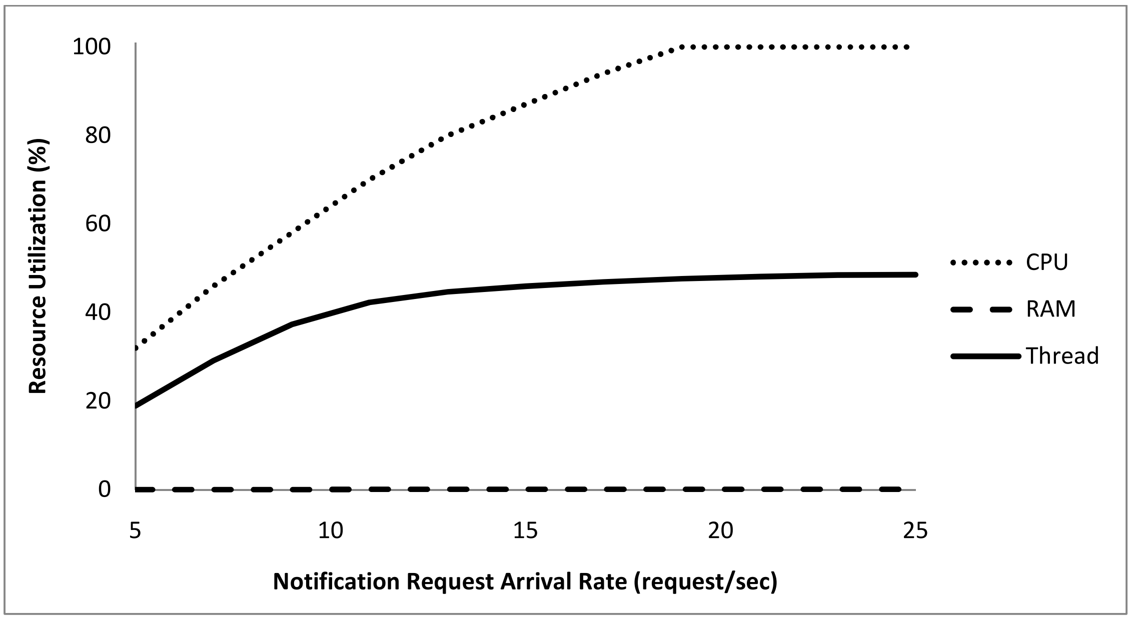


Figure 4-4: Plot of resource utilization of the RSS server at various request arrival rates

4.3.2 Case Study 2: Using simulation models to evaluate changes to application design

As the demand for service increases, organizations examine different options to increase the performance of their systems to cope with the high volumes of workload. In this case study, we use our simulation model to evaluate the performance benefit of migrating a centralized performance monitor for Ultra-Large-Scale (ULS) applications to a distributed architecture. However, the performance of distributed applications is highly dependent on the configuration. To carry out a fair evaluation of the performance gain from changing the existing design to a distributed architecture, the performance analyst should first determine which configuration of the distributed application would show the best performance.

4.3.2.1 Application description

Performance monitors are used to detect problems in the services provided by the enterprises. Enterprises have computational units around the world to keep servers geographically close to the users. In the original design, one performance monitor would periodically collect performance data from each computational unit directly. While it is easy to administer, this centralized design has heavy resource requirements. In the distributed design, each computational unit is connected to a local monitoring agent as shown in Figure 4-5. The local monitoring agents periodically collect, compress and upload the performance data to a central monitor for analysis. The central monitor may occasionally send back an updated set of monitoring policies to the local agents. There are two major challenges of monitoring ULS applications:

- **Communication latency**: ULS applications are decentralized around the world. The performance data may not reflect the current state by the time the data is received by the central monitor due to the large physical distances between the nodes and the central monitor.
- **Financial cost of data transmission**: Depending on the frequency with which the performance data is sent, the cost may be prohibitively high for ULS applications that have many nodes deployed across the globe.

The simulation model of the performance monitor in this case study has two tunable parameters:

- **Data collection frequency**: The rate at which local monitoring agents collect performance data from their respective computational units.
- **Data broadcast period**: The amount of time an agent would wait between each successive upload of performance data. Data collected by the agents is first stored locally. When the send timer fires, all data stored is uploaded to the central monitor.

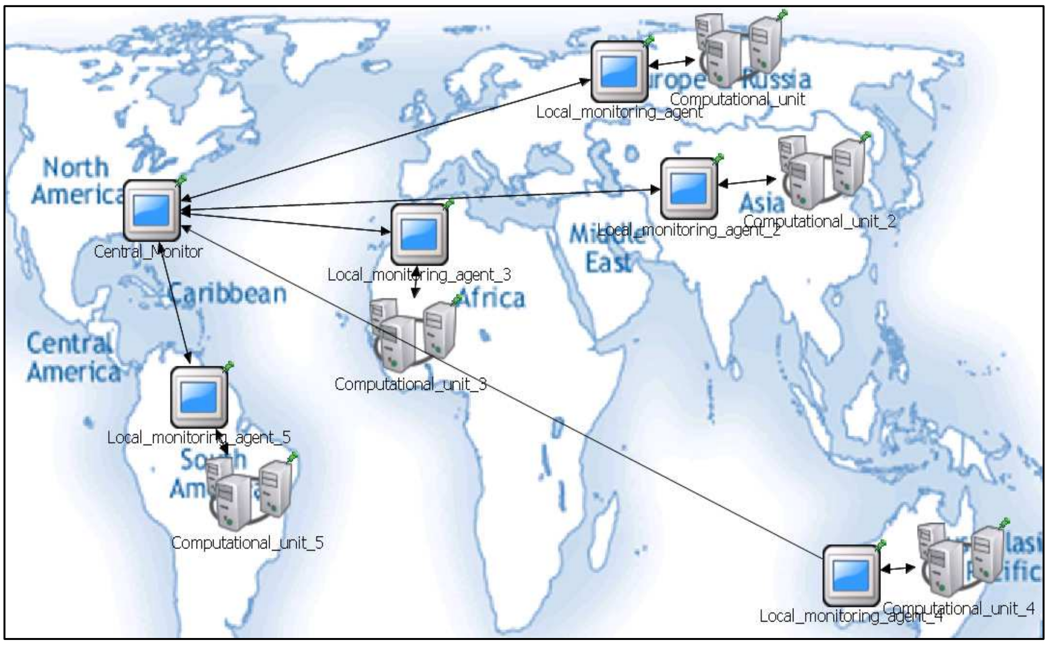


Figure 4-5: World view layer of the performance monitor for ULS applications

4.3.2.2 Experiments and data collection

Figure 4-5 shows the world view layer of the simulation model for the performance monitor. The local monitoring agents and the central monitor are modeled using the same architecture as the RSS server (Figure 4-1).

We conducted a series of simulated runs with 15 combinations of collection frequencies (0.1, 0.2, and 0.3 times per second) and broadcast periods (1, 3, 5, 7, and 9 seconds). Each of the 15 runs simulates an eight-hour workday. Each data collection consumes on average 30 megabytes of memory and 10 CPU units, and lasts for 3 simulated seconds. Table 4-5 shows the various performance data collected in each layer during a simulation run.

Table 4-5: Performance data collected per layer

Layers	***Performance Data***
World view layer	Response time, Transmission Cost
Component layer	Thread Utilization
Physical layer	CPU and RAM utilization

Table 4-6: Categorization of CPU utilization

CPU Util.	***Low***	***OK***	***High***	***Very High***
Range (s)	*< 30*	*30 – 60*	*60 – 75*	*> 75*
Discretization	*0.25*	*0.5*	*0.75*	*1*

Table 4-7: Categorization of RAM utilization

RAM Util.	***Low***	***OK***	***High***	***Very High***
Range (%)	*< 25*	*25 – 50*	*50 – 60*	*> 60*
Discretization	*0.25*	*0.5*	*0.75*	*1*

4.3.2.3 Evaluating parameter configuration

In this section, we show that, by considering the performance data from all three layers, we are able to select a configuration that leads to a balance of three important aspects: cost, performance, and resource consumptions. In the next subsections, we define a score to assist us in ranking configurations that have multiple performance objectives.

4.3.2.3.1 Configuration score

We evaluate a given configuration of performance counters collected during the simulation using the concept of configuration score. Some counters are perceived by people in a fuzzy manner. For example, a CPU utilization of 20% and 25% would have the same monitoring effect.

On the other hand, counters such as the response time and dollar cost are usually perceived by customers in a crisp way. To take into account the fuzziness of the way a counter is perceived, we discretize the counter values into levels, where each level is ranked by a number between 0 and 1 to indicate the ranking of customer experience. The boundaries of each level are selected through domain knowledge.

Table 4-5: Performance data collected per layer

Layers	*Performance Data*
World view layer	Response time, Transmission Cost
Component layer	Thread Utilization
Physical layer	CPU and RAM utilization

Table 4-6 and Table 4-7 show the categorization for CPU and RAM utilizations respectively. For counters that the customers may deem the most important (e.g., response time and cost), we use the original counter values to calculate the score that will be used to rank a set of configurations.

The configuration score is calculated as the product of the discrete or crisp counter values of a configuration. As a simplified example, if a given configuration exhibits an average response time of 5.61 seconds, consumes on average 46% of the central monitor CPU, and results in a cost of \$10, then the score of this configuration would be calculated as follows:

$$Response\ time = 5.61\ seconds$$
$$Cost = \$10$$
$$CPU\ Utilization = 46\% \rightarrow 0.5$$
$$Score = 5.61 \times 0.5 \times 10 = 28.1$$

4.3.2.3.2 Choosing the optimal configuration

Since we aim to minimize resource consumption, response time and cost, the configuration that has the lowest score gives the best overall performance. Table 4-8 shows the optimal

Table 4-8: Simulation result for the performance monitor case study

Data Collection Frequency (Hz)	***Layers***	***Data Broadcast Period* (s)**	**Response time (s)**	**Cost per transmission ($)**	**Central Monitor Thread Util. (%)**	**Central Monitor CPU Util. (%)**	**Central Monitor RAM Util. (%)**
0.1	World View	1	6.8	5.0	1.6	15.6	6.1
	Component	1	6.8	5.0	1.6	15.6	6.1
	Physical	1	6.8	5.0	1.6	15.6	6.1
0.2	World View	1	7.7	5.0	4.0	40.3	15.7
	Component	1	7.7	5.0	4.0	40.3	15.7
	Physical	**7**	**8.9**	**5.3**	**2.3**	**23.4**	**9.2**
0.3	World View	1	8.9	5.0	6.4	64.4	25.3
	Component	1	8.9	5.0	6.4	64.4	25.3
	Physical	**3**	**9.2**	**5.0**	**5.6**	**56.0**	**21.9**

configurations determined by incrementally considering more performance data collected at each layer. For each data collection frequency, we show three optimal configurations that are determined by considering the performance data visible up to the designated layer in the second column. For example, we calculate the score of a configuration at the world view layer by multiplying the response time and the cost; if we are to calculate the score at the physical layer level, we would take into account all five variables (response time, cost, thread, CPU and RAM utilizations) by taking the product of all of them.

At low data collection frequency (0.1 Hz), the optimal configuration determined using the world view layer is the same as the configuration determined with information from all three layers. This effect is explained by the fact that the application is only slightly loaded and all counters collected in the component and physical layers only exhibit small variations, e.g., RAM utilization ranges between 3% to 6%. As a result, all low level counters are discretized to the same range, diminishing the effect of these low level counters in the score calculation. Our discretization approach effectively hides the small variations of the performance counter that are insignificant in terms of customer experience.

As the data collection frequency increases and more layers are being considered, our ranking algorithm outputs configurations that balance between cost, performance and resource consumption. For example, at the collection frequency of 0.3 Hz, the configuration selected by considering data collected up to the physical layers reduces the CPU utilization from 64.4% to 56% in a tradeoff of 0.3s increase of response time while maintaining the same cost.

4.3.2.4 Evaluating the benefit of migrating the performance monitor to a distributed architecture

Table 4-9 : Simulation result for the original design of the performance monitor

Data Collection Frequency (Hz)	**Response time (s)**	**Cost per transmission ($)**	**Central Monitor Thread Util. (%)**	**Central Monitor CPU Util. (%)**	**Central Monitor RAM Util. (%)**
0.1	0.2	2.2	31.0	42.6	37.2
0.2	0.3	2.7	43.7	68.2	47.6
0.3	0.4	3.1	60.2	86.6	59.2

Table 4-9 shows the simulation result for the original design of the performance monitor at various data collection frequencies. Comparing to our distributed design, the original design, while providing better response time, consumes more hardware (e.g., CPU and RAM) and logical (e.g., thread) resources. At the collection frequency of 0.3, the CPU of the central monitor is close to running at its full capacity which will likely result in system instability. Moreover, while the original design has lower cost per transmission, the performance data in the original design is transmitted more frequently due to the lack of a local batching mechanism provided by the local monitoring agent. As a result, the overall cost for monitoring all computational units would increase in the original design compared to the distributed design.

In this case study, we demonstrated the usefulness of our layered simulation model in evaluating the different design options for the performance monitor. Furthermore, we show that configurations selected by analyzing information from just one layer are sometimes suboptimal. In an environment where a server may also provide multiple services, a configuration that consumes fewer resources can avoid jeopardizing the stability of other services. As more information from different layers is supplied, our ranking algorithm is able to select the configurations that balance between cost, performance, and resource consumptions.

4.4 Discussion

In this section, we discuss how our simulation models can be updated and what the limitations of our approach are.

4.4.1 Updating the simulation model to reflect changes to the application

In discrete-event simulations, application components react based on received messages. Therefore, performance analysts can model the application's operation by specifying the state a component should be in when a specific message is received. For example, to simulate the time required to process an RSS notification request, performance analysts can specify that the RSS server would hold each request for 2 simulated seconds before forwarding the notification to subscribers. Performance analysts do not need to know the low-level programming details when constructing the model and thus reducing the modeling effort.

A model constructed for a previous release of the software can be adapted to the new version by updating the resource requirements in the configuration. New application behavior can be introduced to the model by adding new message types and the corresponding behavior in the simulation code.

4.4.2 Capturing resource requirements

Correct resource requirements are essential in deriving useful performance conclusion from a simulation model. To ensure that the simulation model would accurately reflect the performance of the final application, resource requirements should be validated as information becomes available throughout development. Due to the lack of access to production data, we can only estimate a list of resources and processing requirements in our case studies.

4.5 Summary

A performance model should convey information about the behavior of an application to stakeholders who have different performance concerns. In this chapter, we proposed an approach to construct simulation models with four layers of abstractions: world view layer, component layer, physical layer, and usage scenarios. These layers can be built gradually as information about the software project becomes available. In the case studies, we showed that our layered model can be used to extract performance information about an application and evaluate design changes. In the following chapter, we will turn our focus on discovering performance regressions after design changes have been implemented into the application.

Chapter 5

Discovering Performance Regressions at the Implementation Level

In the previous chapter, we introduced an approach to construct layered simulation models for the purpose of discovering the performance regressions introduced by changes to application designs. Design changes and other code changes will eventually be integrated into the implementation of the application. In current practice, performance analysts must manually analyze the data collected from performance tests to uncover performance regressions. This process is both time-consuming and error-prone due to the large volume of collected counters, the absence of formal performance objectives and the subjectivity of individual performance analysts.

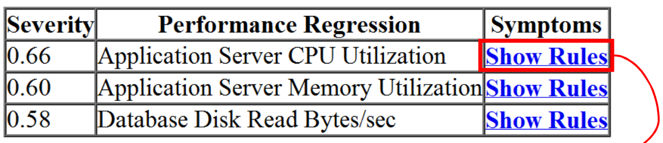
In this chapter, we present an automated approach to detect potential performance regressions in a performance test. Our approach compares the results of new tests against correlations of performance counters extracted from performance testing repositories. Case studies show that our approach scales well to large industrial applications, and detects performance problems that are often overlooked by analysts. An early version of this chapter was published at the 10th International Conference on Quality Software [24].

5.1 An illustration of our automated analysis approach

In this section, we present an example of how a performance analyst, Derek, can leverage our report to spot performance regressions. Derek is given the task to assess the performance of a new version of an e-commerce application. After conducting a performance test on the new version of the software, Derek decides to examine the two counters that he deems the most important: CPU utilization and the number of disk writes per second in the database. He finds that there is a 5% increase in average CPU utilization between a recently passed test and the new test, but the CPU utilization is still below the pre-defined threshold of 75%. Because of a tight deadline, Derek runs our research prototype to check whether the increase in CPU usage represents a performance regression and whether there are any other performance regressions in the new test. Our prototype generates a performance regression report such as the one shown in Figure 5-1.

5.1.1 Report summary

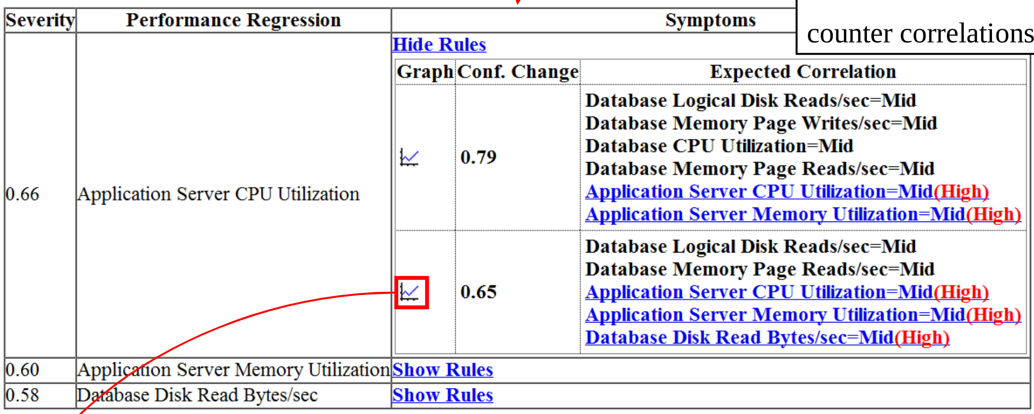
The table shown in Figure 5-1a provides a summary of the counters that are flagged by our approach as deviating from the expected counter correlations. The counters are sorted by the level of severity. Severity of a counter is the fraction of time during the performance test where the counter exhibits regressions. For example, the "application server CPU utilization" counter has a severity of 0.66, meaning that 66% of the time the "application server CPU utilization" counter violates one of the expected counter correlations. By examining this summary, Derek discovers that 3 counters (CPU and memory utilizations in the application server, and # of disk read bytes/second in the database) are flagged with severity greater than 0.5, meaning that these three counters deviate from the expected behavior for over half of the test duration.



Severity	Performance Regression	Symptoms
0.66	Application Server CPU Utilization	Show Rules
0.60	Application Server Memory Utilization	Show Rules
0.58	Database Disk Read Bytes/sec	Show Rules

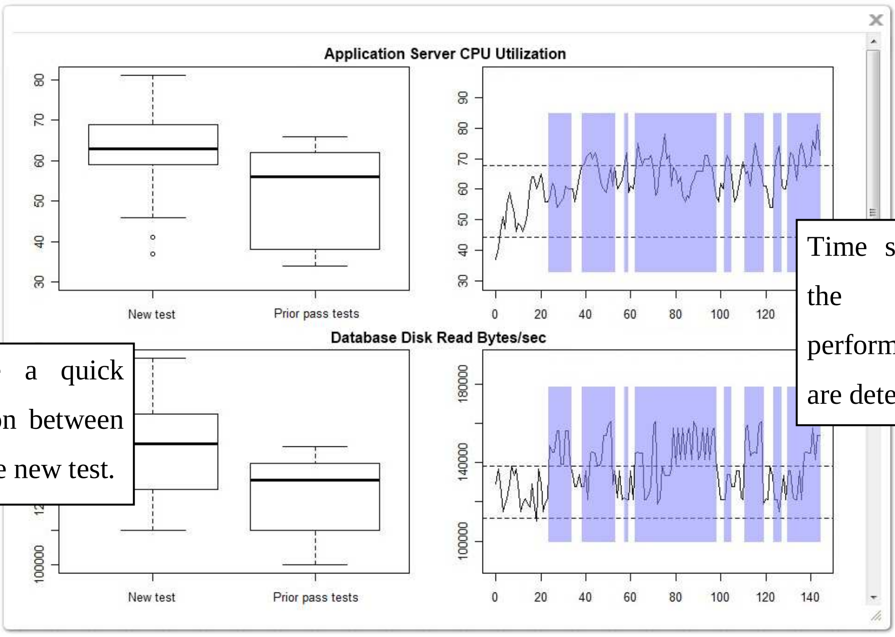
(a) Overview of problematic regressions

Counters with performance regressions (underlined) are annotated with expected counter correlations.



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

(b) Details of performance regressions



Box plots give a quick visual comparison between prior tests and the new test.

Time series plots show the periods where performance regressions are detected.

(c) Periods detected to have performance regression

Figure 5-1: An example performance regression report

5.1.2 Details of performance regression

Derek clicks on the “Show Rules” hyperlink to reveal a list of counter correlations that are violated by the top flagged counter (Figure 5-1). The list is ordered by the degree of deviation between the correlation confidence of the new test and prior tests. Correlation confidence measures how well a correlation holds for a data set. For example, a confidence of 0.9 for a correlation extracted from prior tests means that the correlation holds for 90% of the duration of the test for all prior tests. If the same correlation has a drop of confidence in the new test, this correlation will be included in the report.

By looking at Figure 5-1b, Derek realizes that historically the application server’s CPU utilization, memory usage, and various database counters are always observed to be at the medium level together. However, in the new run, all flagged counters shift from medium to high (highlighted in red and blue in Figure 5-1b). Derek concludes that all 3 flagged counters represent performance regressions. Instead of requiring Derek to examine each counter manually, our report automatically identifies counters in the new test that show significant deviations from prior tests. It is up to Derek to uncover the causes of the performance regressions.

5.1.3 Performance comparison and in-depth analysis

Derek can conveniently compare the counter values in prior tests and the new tests by opening a series of charts such as in Figure 5-1c. The charts on the left are box-plots of the violated counters (e.g., the Application server’s CPU utilization) for the new test and prior tests. A box-plot shows a five-number summary about a counter: the observed minimum, first, second, and third quartile, and maximum value. By placing the box-plots side-by-side, Derek can visually compare the value ranges in the new test and prior tests. In this example, Derek can easily see that half of the observed values of the Application server’s CPU utilization in the new test exceed the historical range of values.

To streamline Derek's analysis, our report places the time-series plots of the flagged counters next to the box-plots. The two dotted lines define the boundaries of the high, medium, and low levels extracted from counter ranges in prior tests. The shaded areas show the time instances where the expected counter correlations do not hold in the new test.

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

5.2 Our approach

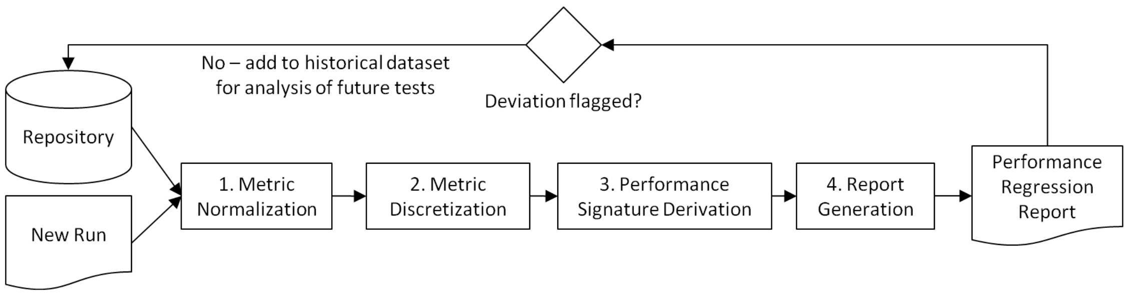

No – add to historical dataset for analysis of future tests
Deviation flagged?
Repository
New Run
1. Metric Normalization
2. Metric Discretization
3. Performance Signature Derivation
4. Report Generation
Performance Regression Report

Figure 5-2: Overview of performance regression analysis approach

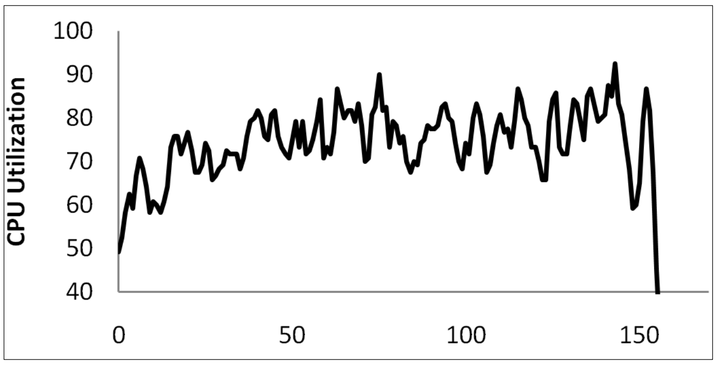
Our approach to detect performance regressions in a performance test has 4 phases as illustrated in Figure 5-2. The input of our approach consists of the result of the new test and the performance testing repository from which we distill a historical dataset consisting of the collection of prior passed tests. We apply data-mining approaches to extract performance signatures by capturing counter correlations that are frequently observed in the historical data. In the new test, counters that violate the extracted performance signatures are flagged. Based on the flagged counters, a performance regression report is generated. We now discuss each phase of our approach in the following subsections.

5.2.1 Counter normalization

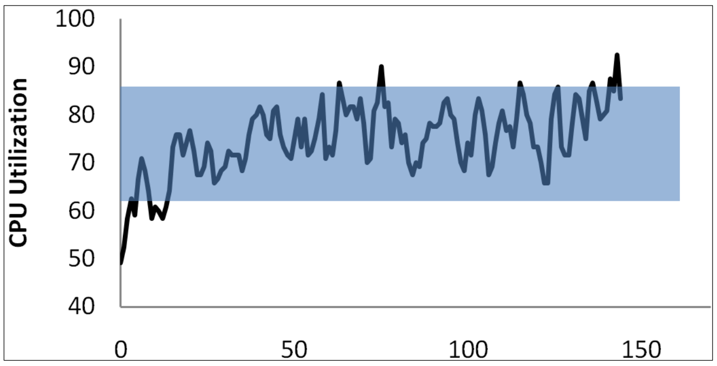
Before we can carry out our analysis, we must eliminate irregularities in the collected performance data. Irregularities can come from the following sources:

- **Clock skew:** Large enterprise applications are usually distributed across multiple machines. Since the clocks on different machines might be out-of-sync, counters captured by different machines will have slightly different timestamps. Moreover, counters can be captured at different rates.
- **Queued requests:** Even when the load generator has stopped, there may be unprocessed requests queued in the application. Performance analysts usually let the application run until all requests are processed. As a result, counters may be recorded for a prolonged period of time.
- **Delay:** There may be a delay in the start of counter collection between the machines that are used in a test.

To overcome these irregularities, we use only the portion of the counter data that corresponds to the expected duration of a test. Then, we divide time into equal time intervals (e.g., every five seconds). Each interval is represented by a vector. Each element in the vector represents the median of a specific counter in that interval. The size of the interval can be adjusted by the analysts depending on how often the counters are captured.



(a) Original counter data



(b) Counter discretization

(Shaded area corresponds to the medium Discretization level)

Figure 5-3: Counter normalization and discretization

5.2.2 Counter discretization

For each counter,

High = All values above the medium level

Medium = Median +/**-** 1 standard deviation

Low = All values below the medium level

Figure 5-4: Definition of counter discretization levels

Since the machine learning approaches we use only take categorical data, we discretize counter values into levels (e.g., high/medium/low). Figure 5-4 shows how the Discretization levels are calculated from the historical dataset. For example, assuming the median and standard deviation in Figure 5-3b are 74 and 14 respectively, the medium level will span from 60 to 88. As a result, the CPU utilization around the 50th second on Figure 5-3b will be mapped to medium. We have experimented with using arithmetic mean instead of median, but found that the arithmetic mean suffered from the effect of outliers and failed to cluster similar measurements into the same level.

5.2.3 Derivation of performance signatures

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

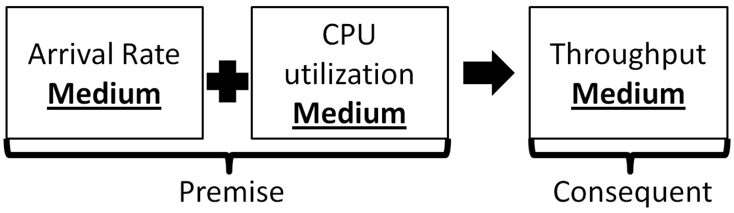


Figure 5-5: Example of an association rule

We use association rule mining to extract counter correlations from the discretized performance counters. An association rule has a premise and a consequent. The rule predicts the occurrence of the consequent based on occurrences of the premise. For example, Figure 5-5 shows one of the three association rules that can be derived from the previous example. In this paper, we use the Apriori algorithm [1] to discover association rules. The Apriori algorithm uses support and confidence to reduce the number of candidate rules generated:

- *Support* is defined as the frequency at which all items in an association rule are observed together. Low support means that the rule occurs simply due to chance and should not be included in our analysis.
- *Confidence* measures the probability that the rule's premise leads to the consequent. For example, if the rule in Figure 5-5 has a confidence value close to 1, it means that when arrival rate and CPU utilization are both medium, there is a high tendency that medium throughput will be observed.

We apply the association rules extracted from the historical dataset to the new test and flag counters in the rules that have significant change in confidence, as defined in Eq. (5-1).

$$Confidence\ change = 1 - \text{cosine\_distance}\left(\overrightarrow{V_{history}}, \overrightarrow{V_{new}}\right) \quad \textbf{Eq. (5-1)}$$

$$\overrightarrow{V_{history}} = (Conf_{history}, 1 - Conf_{history}) \quad \textbf{Eq. (5-2)}$$

$$\overrightarrow{V_{new}} = (Conf_{new}, 1 - Conf_{new}) \quad \textbf{Eq. (5-3)}$$

where $\overrightarrow{V_{history}}$ and $\overrightarrow{V_{new}}$ are the vector form of the confidence values ($Conf_{history}$ and $Conf_{new}$) in the historical dataset and the new test respectively. Since cosine distance measures the angle between two vectors, it is necessary to first convert the scalar confidence values into vector form.

The confidence change for a rule will have a value between 0 and 1. A value of 0 means the confidence for a particular rule has not changed in the new test; value of 1 means the confidence of a rule is completely different in the new test. If the confidence change for a rule is higher than a specified threshold, we can conclude that the behavior described by the rule has changed significantly in the new test and the counters in the rule's consequent are flagged. For example, if the confidence of the rule in Figure 5-5 drops from 0.9 to 0.2 in the new test, it indicates that medium arrival rate and CPU Utilization would no longer be associated with medium throughput utilization for the majority of the time. As a result, throughput exhibits a significant change of behavior and thus should be investigated. The threshold for confidence change is customizable by the performance analyst to control the number of counters returned by our approach based on time available.

5.2.4 Report generation

In the last phase, we generate a report of the flagged counters that highlights the association rules that the counters violate. To further help a performance analyst to prioritize his time, we rank the counters by their level of severity Eq. (5-4).

$$Severity = \frac{\text{\# of time instances containing the flagged values of the counter}}{\text{total \# of time instances in the new test}} \qquad \textbf{Eq. (5-4)}$$

At each time instance, we compare the level of the counter against the expectation from the association rules. A mismatch of levels means that the counter shows abnormal behavior during that time instance. Severity represents the fraction of time in the new test that contains the flagged values of the counter. Severity ranges between 0 and 1. If there are only a few instances where the counter is observed to be problematic, the severity will have a value close to 0. On the other hand, if the counters are violated many times, severity will have a value close to 1 (Figure 5-1a).

For each counter, the report lists the violated rules ordered by confidence change Eq. (5-1) as shown in the inner table in Figure 5-1b. Confidence measures how well a rule applies to a data set. A large change in the confidence of a rule between the prior tests and the new test indicates that the counter in the rule's consequence has a change of behavior.

Finally, if no counter is flagged in the report, we can conclude that the new test has no performance regression and can be included in the historical dataset for analysis of future tests.

5.3 Case studies

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

We manually injected faults into the test scenarios of the two open source e-commerce applications. We follow the popular “Siemens-Programs” approach of seeding bugs [30], which is based on common performance problems. Manual injection of faults allows us to calculate the precision (Eq. (5-5)) and recall (Eq. (5-6)) of our approaches.

$$Precision = \left(1 - \frac{\#\ of\ false\ positives}{total\ \#\ of\ counters\ flagged}\right) \qquad \textbf{Eq. (5-5)}$$

$$Recall = \frac{\#\ of\ problematic\ counters\ detected}{\#\ of\ actual\ problematic\ counters} \qquad \textbf{Eq. (5-6)}$$

High precision and recall mean that our approach can accurately detect most performance regressions. Performance analysts can reduce the effort required for an analysis by investigating the flagged counters. Note that false positives are counters that are incorrectly flagged (e.g., they do not contain any performance regression).

For the large enterprise application, we use the existing performance tests collected by the organization’s Performance Engineering team as input to our approach. We seek to compare the results generated by our approach against the performance analysts’ observations. In the cases where our approach flagged more counters than the performance analysts noted, we verify the additional problematic counters with the Performance Engineering team to determine if the counters truly represent performance regressions. Since we do not know the actual number of performance problems, we calculate the recall for our industrial case study based on the correct performance regression, flagged in the organization's original reports.

We use the average precision and recall to show the overall performance of our approach across all test scenarios for each application. Average precision and recall combine the precision (Eq. (5-7)) and recall (Eq. (5-8)) for all k different test scenarios ($t_1,t_2,\dots,t_k$) conducted for an application. Table 5-1 summarizes the performance of our approach in each case study.

$$Average\ Precision = \frac{1}{k} \times \sum_{1}^{k} Precision_{t_k} \qquad \textbf{Eq. (5-7)}$$

$$Average\ Recall = \frac{1}{k} \times \sum_{1}^{k} Recall_{t_k} \qquad \textbf{Eq. (5-8)}$$

Table 5-1: Average precision and recall

	# of test scenarios	**Duration per test (hours)**	**Average precision**	**Average recall**
DS2	4	1	100%	52%
JPetStore	2	0.5	75%	67%
Enterprise Application	13	8	93%	100% (relative to organization's original analysis)

Research prototype: Our research prototype is implemented in Java and uses the Weka package [57] to perform various data-mining operations. The graphs in the performance analysis reports are generated using the statistical analysis and modeling tool, R [56].

5.3.1 Studied application: Dell DVD Store

5.3.1.1 Application description

The Dell DVD Store (DS2) application [55] is an open source implementation of a simple e-commerce website. DS2 is designed for benchmarking Dell hardware. DS2 includes basic e-commerce functionalities such as user registration, user login, product search and purchase.

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

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

5.3.1.2 Data collection

We collected 19 counters as summarized in Table 5-2. The data is discretized into 2-minute intervals. We conducted 4 one-hour performance tests. The same load is used in tests D_1, D_2, and D_3. Our performance signatures are derived from test D_1 during which normal performance is assumed. For tests D_3 and D_4, we manually inject faults into either the JSP code or the load driver settings to simulate implementation defects and mistakes of performance analysts. The types of faults we injected are commonly used in other studies [33]. Prior to the case study, we derive a list of counters that are expected to show performance regressions, as summarized in Table 5-3. The Recall of our approach is calculated based on the counters listed in Table 5-3.

Table 5-2: Summary of counters collected for DS2

Load generator	% Processor Time # Orders/minute # Network Bytes Sent/second # Network Bytes Received/Second
Tomcat	% Processor Time # Threads # Virtual Bytes # Private Bytes
MySQL	% Processor Time # Private Bytes # Bytes written to disk/second # Context Switches/second # Page Reads/second # Page Writes/second % Committed Bytes In Use # Disk Reads/second # Disk Writes/second # I/O Reads Bytes/second # I/O Writes Bytes/second

Table 5-3: Summary of injected faults for DS2 and expected problematic regressions

Test	Description of the injected fault	Expected problematic regressions	
D_1	No fault	N/A (training data)	
D_2	No fault	No problem should be observed.	
D_3	Busy loop injected in the code responsible for displaying item search results	**Tomcat counters** (where "+" represents increase of counter)	
		# threads	+
		# private bytes	+
		#virtual bytes	+
		CPU utilization	+
		Database counters	
		# I/O reads bytes /second	+
		# disk reads/second	+
D_4	Heavier load applied to simulate error in load test configuration	**Tomcat counters**	
		# threads	+
		# private bytes	+
		#virtual bytes	+
		CPU utilization	+
		Database counters	
		# disk reads /second	+
		# disk writes/second	+
		I/O write bytes/second	+
		I/O read bytes/second	+
		CPU utilization	+
		# context switches	+
		Load generator	
		# Orders/minute	+
		# Network Bytes Sent/second	+
		# Network Bytes Received/Second	+

5.3.1.3 Analysis of test D_2

The goal of this experiment is to show that the rules generated by our approach are stable under normal operation. Since test D_2 shares the same configuration and same load as test D_1, ideally our approach should not flag any counter.

As expected, our prototype did not report any problematic counter in test D_2.

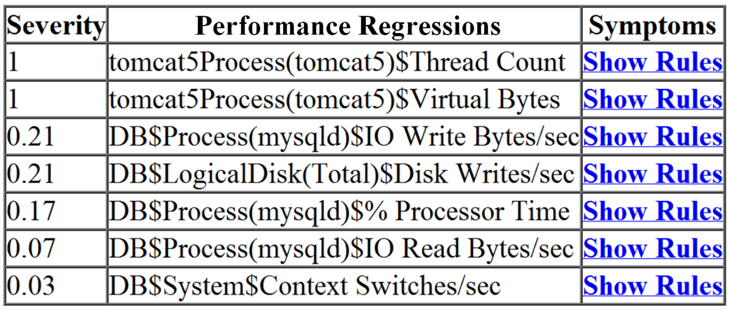
5.3.1.4 Analysis of test D_3

In test D_3, we injected a database-related bug to simulate the effect of an implementation error. This bug affects the product browsing logic in DS2. Every time a customer performs a search on the website, the same query will be repeated numerous times, causing extra workload for the backend database and Tomcat server. The excessive-database-queries bug simulates the n+1 pattern [29].

Our approach flagged a database-related counter (# Disk Reads/second) and two Tomcat server-related counters (# Threads and # private bytes). All three counters have severity of 1, signaling that the counters are violated during the whole test. The result agrees with the nature of the injected fault: each browsing action generates additional queries to the database. As a result, an increase in database transaction leads to an increase of # Disk Reads/second. When the result of the query returns, the application server uses additional memory to extract the results. Furthermore, since each request would take longer to complete due to the extra queries, more threads are created in the Tomcat server to handle the otherwise normal workload. Three other counters (Database #I/O reads bytes/second, Tomcat CPU utilization and # virtual bytes) that are expected to show regressions are not flagged by our approach. Upon investigation, we discover that the variations of these counters compared to the training data were too small and did not lead to a change of level after the discretization step. Since 3 out of 6 expected problematic counters are detected, the precision and recall of our approach in test D_3 are 100% and 50% respectively.

5.3.1.5 Analysis of test D_4

We injected a configuration bug into the load driver to simulate that a wrongly configured workload is delivered to the application. This type of fault can either be caused by a malfunctioning load generator or by a performance analyst when preparing for a performance test [34]. In the case where a faulty load is used to test a new version of the application, the assessment derived by the performance analyst may not depict the actual performance of the application under test.



Severity	Performance Regressions	Symptoms
1	tomcat5Process(tomcat5)$Thread Count	Show Rules
1	tomcat5Process(tomcat5)$Virtual Bytes	Show Rules
0.21	DB$Process(mysqld)$IO Write Bytes/sec	Show Rules
0.21	DB$LogicalDisk(Total)$Disk Writes/sec	Show Rules
0.17	DB$Process(mysqld)$% Processor Time	Show Rules
0.07	DB$Process(mysqld)$IO Read Bytes/sec	Show Rules
0.03	DB$System$Context Switches/sec	Show Rules

Figure 5-6: Performance Regression Report for DS2 test D_4 (Increased Load)

In test D_4, we double the visitor arrival rate in the load driver. Furthermore, each visitor is set to perform additional browsing for each purchase. Figure 5-6 shows the counters flagged by our prototype. The result is consistent with the nature of the fault. Additional threads and memory are required in the Tomcat server to handle the increased demand. Furthermore, the additional browsing and purchases lead to an increase in the number of database reads and writes. The extra demand on the database leads to additional CPU utilization.

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

5.3.2 Studied application: JPetStore

5.3.2.1 Application description

JPetStore is a larger and more complex e-commerce application than DS2 [9]. JPetStore is a re-implementation of Sun's original J2EE Pet Store and shares the same functionality as DS2. Since JPetStore does not ship with a load generator, we use a web testing tool, NeoLoad [42], to record and replay a scenario of a user logging in and browsing items on the site.

5.3.2.2 Data collection

In this case study, we have conducted two one-hour performance tests (J_1 and J_2). Our performance signatures are extracted from test J_1 during which caches are enabled. Test J_2 is injected with a configuration bug in MySQL. Unlike the DS2 case study where the configuration bug is injected in the load generator, the bug used in test J_2 simulates a performance analyst's mistake to accidentally disable all caching features in the MySQL database [44]. Because of the nature of the fault, we expect the following counters of the database machine to be affected: CPU utilization, # threads, # context switches, # private bytes, and # disk read and write bytes/second.

5.3.2.3 Analysis of test J_2

Our approach detected a decrease in the memory footprint (# private bytes) and "# disk writes bytes/second" in the database, and an increase in the "# disk reads bytes/second" and "# threads" in the database. The disk counters include reading and writing data to network, file, and device. These observations align with the injected fault: Since the caching feature is turned off in the database, less memory is used during the execution of the test. In exchange, the database needs to read from the disk for every query submitted. The extra workload in the database translates to a delay between when a query is received and when the results are sent back, leading to a decrease in "# disk writes bytes/second" to the network.

Instead of an increase, an unexpected drop of the # threads was detected in the database. Upon verifying with the raw data for both tests, we found that the “thread count” in test J_1 (with cache) and test J_2 (without cache) consistently remain at 22 and 21 respectively. Upon inspecting the data manually, we do not find that the decrease of one in thread count constitutes a performance regression; therefore we conclude this as a false positive. Finally, throughout the test, there is no significant degradation in the average response time. Since 4 out of 6 expected problems are detected, our performance regression report has a precision of 75% and recall of 67%.

5.3.3 Studied application: A large enterprise application

5.3.3.1 Application description

Our third case study is conducted on a large enterprise application. This application is designed to support thousands of concurrent requests. Thus, the performance of this application is a top priority for the organization. For each build of the application, performance analysts must conduct a series of performance tests to uncover performance regressions and to file bug reports accordingly. Each test is run with the same workload, and usually spans from a few hours to a few days. After the test, a performance analyst will upload the counter data to an internal website to generate a time series plot for each counter. This internal site also serves the purpose of storing the test data for future reference. Performance analysts then manually evaluate each plot to uncover performance issues. Unfortunately, we are bounded by a Non-Disclosure Agreement and cannot give more details about the commercial application.

Table 5-4: Summary of analysis for the enterprise application

Test	Summary of the report submitted by the performance analyst	Our findings
E_1	No performance problem found.	Our approach identified abnormal behaviors in system arrival rate and throughput counters.
E_2	Arrival rates from two load generators differ significantly. Abnormally high database transaction rate. High spikes in job queue.	Our approach flagged the same counters as the performance analyst's analysis with one false positive.
E_3	Slight elevation of # database transactions/second.	No counter flagged.

5.3.3.2 Data collection

In this case study, we selected thirteen 8-hour performance tests from the organization's performance testing repository. These tests were conducted during the development of a particular release. The same workload was applied to all tests. Over 2000 counters are collected in each test.

Out of the pool of 13 tests, 10 tests have received a pass status from the performance analysts and are used to derive performance signatures. We evaluated the performance of the 3 remaining tests (E_1, E_2 and E_3) and compared our findings with the performance analysts' assessment (summarized in Table 5-4). In the following sections, we will discuss our analysis on each target test (E_1, E_2 and E_3) separately.

5.3.3.3 Analysis of test E_1

Using data from these passed 10 tests, our approach flagged all throughput and arrival rate counters in the application. The rules produced in the report imply that throughput and arrival rates should fall under the same range. For example, component A and B should have similar request rate and throughput. However, our report indicates that half of the arrival rates and throughput counters are high, while the other half is low. Verification of our report by a performance analyst showed that our indications were correct, i.e., our approach has successfully uncovered problems associated with the arrival rate and throughput in test E_1 that were not mentioned in the performance analyst's report. Our performance regression report has a precision and recall of 100% relative to the original analyst's report.

5.3.3.4 Analysis of test E_2

Our approach flagged two arrival rate counters, two job queue counters (each represents one sub-process), and the "# database scans/second" counter. Upon consulting with the time-series plots for each flagged counter as well as the historic range, we found that the "# database scans/second" counter has three spikes during the test. These spikes are likely the cause of the rule violations. Upon discussing with a performance analyst, we find that the spikes are caused by the application's periodic maintenance and do not constitute a performance regression. Therefore, the "# database scans/second" counter is a false positive. Since four out of five flagged counters are valid performance regressions, our performance analysis report has a precision of 80% and a recall of 100%.

5.3.3.5 Analysis of test E_3

Our approach did not flag any rule violation for this test. Upon inspection of the historical value for the counters noted by the performance analyst, we notice that the increase of "# database transactions/second" observed in test E_3 actually falls within the counter historical value range. Upon discussing with the Performance Engineering team, we conclude that the increase does not represent a performance problem. In this test, we show that our approach of using a historical dataset of prior tests is more resistant to fluctuations of counter values. Our approach achieves a precision and recall of 100%.

The case studies show that our automated approach is able to detect similar problems as the analysts. Our approach detects problematic counters with high precision in all three case studies. In our case studies with the two open source applications, our approach is able to cover 50% to 67% of the expected problematic counters.

5.4 Discussion

In this section, we discuss our approach to analyze performance tests conducted for the implementation of an application.

5.4.1 Quantitative approaches

Although there are existing approaches [14] [18] to correlate anomalies with performance counters by mining the raw performance data without discretization, these approaches usually assume the presence of Service Level Objectives (SLO) that can be used to determine precisely when an anomaly occurs. As a result, classifiers that predict the state of SLO can be induced from the raw performance counters augmented with the SLO state information (See Chapter 3). Unfortunately, SLOs rarely exist during development. Furthermore, automated assignment of SLO states by analyzing counter deviations is also challenging as there could be phase shifts in the performance tests, e.g., the spikes do not align. These limitations prevent us from using classifier-based approaches to detect performance regression.

5.4.2 Sampling period and counter discretization

We choose the size of time interval for counter discretization based on how often the original data is sampled. For example, an interval of 200 seconds is used to discretize data of the enterprise application, which was originally sampled approximately every 3 minutes. The extra

20 second gap is used because there was a mismatch in sampling frequencies for some counters. We also experimented with different interval lengths. We found that the recall of our approach drops as the length of the interval increases, while precision is not affected.

In our case studies, we found that the false negatives (counters that were expected to show performance regressions but were not detected by our approach) were due to the fact that no rule containing the problematic counters was extracted by the Apriori algorithm. This was caused by our discretization approach sometimes putting all values of a counter that had large standard deviation into a single level. Candidate rules containing those counters would exhibit low confidence and were thus pruned.

5.4.3 Performance testing

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

5.4.4 Training data

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

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

In the future, we will explore approaches to automatically filter out problematic tests within our training set.

5.4.5 Automated diagnosis

Our approach automatically flags counters by using association rules that show high deviations in confidence between the new tests and the historical dataset. These deviations represent possible performance regressions or improvements and are valuable to performance analysts in assessing the application under test. Performance analysts can adjust the deviation threshold to restrict the number of used rules and, thus, limit the number of flagged counters. Alongside with the flagged counters, our tool also displays the list of rules that the counter violated. Performance analysts can inspect these rules to understand the relations among counters.

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

5.5 Summary

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

Due to limited resources and lab constraints, organizations would conduct performance tests on a variety of software and hardware configurations to uncover performance regressions. Difference in hardware and software may affect the performance behavior of an application. In the next chapter, we extend our automated analysis approach such that tests conducted with different hardware and software configurations can be used as training data.

Chapter 6

Detecting Performance Regression using Tests conducted in Heterogeneous Environments

Organizations maintain multiple lab environments to execute performance tests in parallel. Over time, each lab may receive inconsistent upgrades. As a result, labs may contain varying hardware configurations, such as different CPU and disk, and software configurations, such as different operating system architectures (e.g., 32-bit and 64-bit) and database versions. Performance tests executed with different configurations may exhibit different performance behavior. Performance analysis approaches that cannot differentiate between the performance differences caused by varying configurations and those caused by performance regressions, will lead to incorrect conclusions being drawn about the quality of the application.

In this chapter, we extend our previous approach from Chapter 5 with ensemble-learning algorithms to deal with performance tests that are conducted in different environments. Ensemble-learning algorithms involve building a collection of models from the performance testing repository with each model specializing in detecting the performance regressions in a specific environment, and combining the output of all the models to detect performance regressions in a new test. By considering multiple models, we reduce the overall risk of following the result of a single model that might be derived from data that is significantly different from the new test.

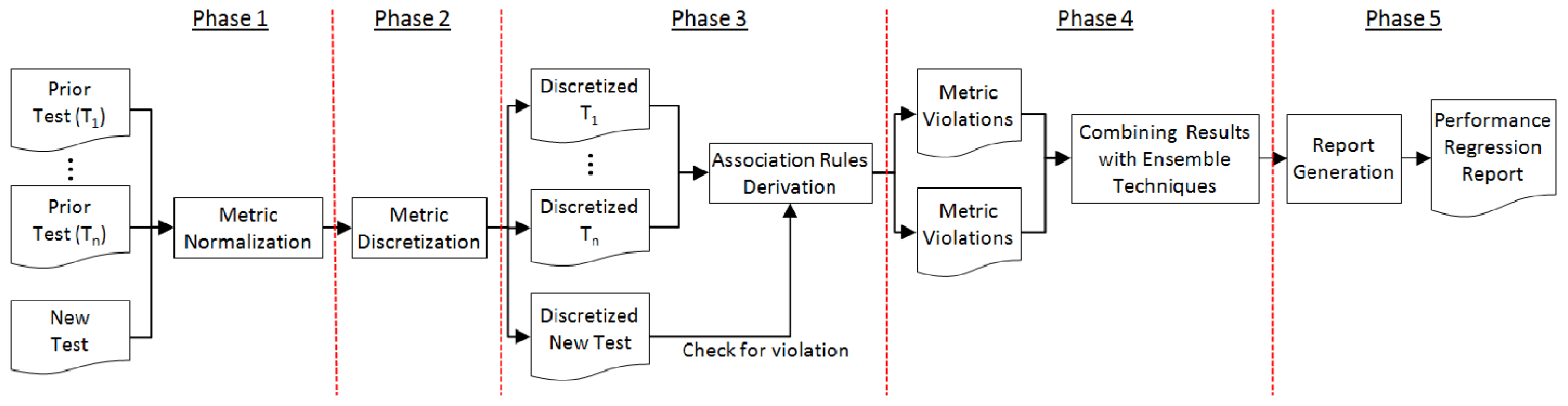


Figure 6-1: Overview of our ensemble-based approach

6.1 Our ensemble approach

Our ensemble approach has 5 phases, as shown in Figure 6-1. The input of our approach consists of a set of prior tests and a new test. For each prior test, we apply association rule mining to extract counter correlations that are frequently observed from the test. Each set of correlations is checked against the new test for counters that violate the correlations. Violation results are then combined with ensemble-learning algorithms to produce a performance regression report similar to Figure 5-1. 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.

6.1.1 Counter normalization

Counters in a performance test may contain irregularities such as clock skew, unfinished requests, and delay. We follow the procedure outlined in Section 5.2.1 to normalize these irregularities.

6.1.2 Counter discretization

Our machine learning approach (association rule mining) takes categorical data. Therefore, we must discretize the continuous performance counters. In Chapter 5, we introduced a discretization algorithm where counters are put into one of three levels (high, medium and low). However, for tests conducted in heterogeneous environments, performance counters may reside at many distinct levels due to hardware and software differences. Forcing the performance counters from performance tests conducted in heterogeneous environments into only three levels may significantly degrade the data and make our analysis approach less robust. To avoid the above problem, we choose to use the Equal Width interval binning (EW) algorithm, which determines the number of levels automatically, for our discretization task. The EW algorithm is relatively easy to implement and has similar performance as other more advanced discretization algorithms when used in conjunction with machine learning approaches [22]. The EW algorithm first sorts the observed values of a counter, then divides the value range into k equally sized bins. The width of each bin and the number of bins are determined by Eq. (6-1) and Eq. (6-2).

$$bin\ width = \frac{x_{max} - x_{min}}{k}$$ **Eq. (6-1)**

$$k = \max\{1, 2\times\log(u)\}$$ **Eq. (6-2)**

where u is the number of unique values in a counter.

The bin boundaries, used to discretize each counter in the tests, are found by applying 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. Eq. (6-1) and Eq. (6-2) are then applied to T_A to obtain a set of bin boundaries.

6.1.3 Derivation of counter correlations

Similarly to Chapter 5, we apply association rule mining to extract a set of association rules from each test in the training data. Each of these rule sets, e.g., R_1, R_2, R_3 and R_4, is a model describing the performance behavior of a prior test. Association rules that do not reach the minimum support and confidence thresholds will be pruned. As mentioned earlier in Chapter 5, counters that contain performance regression are identified by measuring the changes to the rule's confidence in a new test.

Each model contains performance behaviors that are common across prior tests as well as behaviors that are specific to the test configuration that is used as the training data. Table 6-1 shows an example of counters flagged in T_5 as performance regressions by R_1 to R_4. Counters that

Table 6-1: Counters flagged in T5 by multiple rule sets

Model	**Counters flagged as Violation**
R1	CPU utilization, throughput
R2	Memory utilization, throughput
R3	Memory utilization, throughput
R4	Database transactions/second

are flagged by a small number of models may be due to differences in environments and may not represent real performance regressions.

6.1.4 Combining results using an ensemble-learning algorithm

We combine the counters individually flagged by each model in phase 3 using one of the following two ensemble-learning algorithms: Bagging [12] and Stacking [13]. In the following subsections, we review the traditional Bagging and Stacking algorithms and present our adoption of these ensemble-learning algorithms.

6.1.4.1 Bagging

Table 6-2: Count of counters flagged as violations by individual rule set

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

Bagging is one of the earliest and simplest ensemble-learning algorithms [12]. Bagging has been shown to often outperform a single monolithic model [7] [47]. In Bagging, the prediction of each model is combined by a simple majority vote to form the final prediction. 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 available models. For example, Table 6-2 shows the number of times a counter is flagged by the 4 models (R_1, R_2, R_3, R_4) in Table 6-1. Both throughput and memory utilization will be reported as performance regressions as they are flagged by at least 2 models.

6.1.4.2 Stacking

Stacking, or stacked generalization, is a more general ensemble-learning algorithm that is not limited to a specific strategy to combine the results of individual models [58]. Unlike the Bagging algorithm, which uses majority voting to aggregate results of individual models, Stacking 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 seeks a linear stacking function *s* of the form:

$$s(x) = \sum_i w_i R_i(x) \qquad \textbf{Eq. (6-3)}$$

where the w_i represent the weight of the rule set R_i generated from the i^{th} test in the repository, and x represents the performance data of the new test. Thus, $R_i(x)$ flags counters that show performance regressions in the new test x according to R_i. For example, Table 6-1 shows the counters flagged as performance regressions in test 5 by models generated from tests 1 to 4.

We define a function to calculate the weight of each model based on the similarity between the environments used for the tests in the performance testing repository and the new test. A prior test with environment settings very similar to the new test will receive a heavier weight. To

Table 6-3: Test Configurations

	Performance Testing Repository		New Test
	T1	**T2**	**T5**
CPU	2 GHz, 2 cores	2 GHz, 2 cores	2 GHz, 2 cores
Memory	2 GB	1 GB	2 GB
Database Version	1	2	1
OS Architecture	32 bit	64 bit	64 bit

compare the environments between two tests, we generate a similarity vector of 1s and 0s to indicate if two tests share common components. For example, Table 6-3 shows three environments for T_1, T_2, and T_5. The similarity vectors for the pairs (T_1, T_5) 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. Since it is difficult to determine the relations between the application's performance and individual hardware or software components, such a binary approach provides us with the safest way to compare environment configurations.

The length of the similarity vector can be used to measure the degree of similarity between the new test and the past test. For example, the lengths of (T_1, T_5) and (T_2, T_5) equal 1.7 (square root of 3) and 1.4 (square root of 2) respectively. The longer the length of a similarity vector, the higher the similarity between prior and new tests. Based on the distance of each similarity vector, we can assign a weight to each model. We calculate the weight by dividing the distance of the similarity vector by the sum of the distances of all similarity vectors. Each weight has a value between 0 and 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 is essentially a value that describes the relative importance of a model. In practice, performance analysts may specify custom weights best suited for their repositories.

The weights are used in Eq. (6-3) to produce the final set of counters that are likely to show performance regression. Each counter that is reported by our Stacking approach will have an aggregated weight between 0 and 1. Performance counters with coefficients greater than 0.5 will be included in the resulting set of counters that show performance regressions. Note that our Stacking approach is equivalent to Bagging if previous tests in the performance testing repository have the same environment as the new test.

6.1.5 Report generation

We generate a report, similar to Figure 5-1 in Chapter 5, of the counters flagged in phase 4. The flagged counters are first ranked by either the number of models voted for the counter (for Bagging) or by the aggregated weights (for Stacking). If two counters are ranked the same, they will be further sorted by the level of severity as defined in Eq. (5-4).

6.2 Case study

We conducted a series of case studies with two open source e-commerce applications (Dell DVD Store and JPetStore) and a large enterprise application to compare the performance of our ensemble-based approaches and our previous approach (with both the 3-level discretization algorithm as described in Section 5.2.2, and the EW discretization algorithm).

In the case study with the Dell DVD Store, we show that our ensemble-based approaches can handle different hardware configurations in performance tests, while delivering an increase of accuracy over our previous approach. In the case study with JPetStore, we show that our ensemble-based approaches can be used to analyze performance tests that are conducted with different software configurations. Finally, in our industrial case study, we demonstrate that our ensemble-based approaches can be used to detect problems in performance tests that are conducted with varying hardware and software components.

We use the same faults as Chapter 5 in our experiments with the two open source e-commerce applications. This allows us to assess the precision (Eq. (6-4)) and recall (Eq. (6-5)) of our approach.

$$Precision = \begin{cases} 1 - \frac{m_{false}}{m_{total}}, if\ m_{total} > 0 \\ 1, if\ m_{total} = 0 \end{cases} \qquad \textbf{Eq. (6-4)}$$

$$Recall = \begin{cases} \frac{m_{total} - m_{false}}{m_{expected}}, if\ m_{expected} > 0 \\ 1, if\ m_{expected} = 0 \end{cases} \qquad \textbf{Eq. (6-5)}$$

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 that we expect to show performance regressions (including counters that are flagged as side-effect of the injected fault). High precision means that our approach can identify performance regressions with low false positives. High recall means that our approach can discover most performance regressions in a test.

We use the existing performance counters collected by the organization’s performance analysts as the input of our approach for the case study with the enterprise application. The data used in this case study contain tests that were executed in heterogeneous environments, and are different from the ones used in Chapter 5. We compare the results of our approaches against the analysts’ reports. A big motivator of our work is the tendency of analysts to miss problems due to the vast amount of data produced by large industrial applications. Hence, we do not use the analysts' reports as the "gold standard" for precision and recall. Instead, we carefully re-verified the test reports with the analysts, who noted any missed problems. We calculate the recall by determining the ratio of the number of true positives of a particular approach to the size of the

Table 6-4: Summary of performance of our approaches

Test		Original approach (with 3-level discretization)			Original approach (with EW discretization)			Bagging			Stacking		
		P	R	F	P	R	F	P	R	F	P	R	F
Dell DVD Store	D_4	0	1	0	0	1	0	1	1	1	1	1	1
	D_5 (D_1 as training)	1	0.4	0.6	1	0.6	0.7	1	0.6	0.7	1	0.6	0.7
	D_5 (D_1, D_2, and D_3 as training)	1	0	0	1	0.1	0.3	1	0.6	0.7	1	0.6	0.7
JPetStore	J_3	1	0.3	0.5	1	0.5	0.7	1	0.5	0.7	1	0.5	0.7
Enterprise System	E_1	1	0.3	0.5	1	0.5	0.6	0.7	1	0.8	0.8	0.8	0.8
	E_2	0.8	0.3	0.4	0.9	0.4	0.5	0.8	1	0.9	0.9	0.9	0.9
	E_3	1	1	1	1	1	1	0	1	0	1	1	1
Average		**0.8**	**0.5**	**0.4**	**0.8**	**0.6**	**0.5**	**0.8**	**0.8**	**0.7**	**1**	**0.8**	**0.8**

P represents precision, R represents recall, and F represents F-measure
(Values are rounded up to 1 significant digit)

union of all true positives of our original approach (either with the 3-level discretization algorithm, or the EW discretization algorithm), and our Bagging and Stacking approaches.

Finally, we use the F-measure to rank the accuracy of our approaches across all 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 approach has high precision and recall.

Table 6-4 summarizes the performance of our approaches in all 3 case studies. In each row, the cells corresponding to the approach that produces the highest F-measure value is shaded.

For each case study, we give a description of the application, methods used for data collection, and analysis of each approach.

6.2.1 Case study 1: Dell DVD Store

6.2.1.1 Data collection

We ran five one-hour performance tests (D_1 to D_5) with the same workload. We varied the CPU and memory capacity of the machine that hosts Tomcat and MySQL to simulate different hardware environments. Table 6-5 summarizes the hardware setups and the expected problematic counters for this case study. Test D_1 represents the test in which the hardware is running at its full capacity. In test D_2, we throttle the CPU to 50% of the full capacity to emulate a slower machine. In test D_3, we reduce the memory from 3.5 gigabyte to about 1.5 gigabyte. Tests D_1, D_2 and D_3 will be used as test repository for our approaches. Test D_4 is a replication of test D_1 and will be used to show that our approaches produce few false positives. In test D_5, we inject a fault in the browsing logic of DS2 to cause a performance regression. Prior to the case study, we manually derived a list of counters that we expected to show performance regressions. We use these counters to calculate the recall of our approach.

Table 6-5: Summary of hardware setup for DS2

Test	Hardware setup	Fault description	Expected problematic counters

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	**Tomcat counters** (+ represents increase**)** # threads | + # private bytes | + CPU utilization | + **Database counters** # disk reads /second | + CPU utilization | + **Application counters** Response time | + Orders / minutes | +

6.2.1.2 Analysis of test D_4

The goal of this experiment is to examine whether our ensemble-based approaches would produce false positives when analyzing tests that are conducted in heterogeneous environments. Since test D_4 is run without any injected bug, ideally, no counter should be flagged.

When using our original approach with the 3-level discretization algorithm, 3 counters (database CPU utilization, # database I/O writes, and # orders/minute) were flagged. Upon investigation, we found that the rules that flagged the 3 counters reflected the behavior of test D_2: because less processing power was available, each request would take longer to complete, resulting in an increase of CPU utilization. Also, less requests can be completed, leading to a decrease in # database I/O writes/second and # orders/minute. Since no actual fault was injected into test D_4, we concluded that the three counters flagged were 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 Eqn. 6-5. The F-measure of our original approach with the 3-level discretization algorithm is 0.

Our original approach with the EW discretization algorithm flagged 4 counters, which included the 3 counters flagged when the 3-level discretization algorithm was used and the response time counter. Upon investigation, we found that the EW discretization algorithm was able to more precisely capture the counter values in the tests with multiple levels. As a result, the association rules derived from test D_2 were able to pick up the response time differences between tests D_2 and D_4. However, since no actual fault was injected into test D_4, the precision and recall of our original approach with the EW discretization algorithm are 0 and 1, respectively. As a result, the F-measure is 0.

No counter was flagged by either the Bagging or Stacking approach. This can be explained by the fact that separate models are learned from tests 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 or 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 D_2, none of these counters could be confirmed by models generated from tests D_1 or D_3. The precision, recall and F-measure of our ensemble-based approaches are 1. This study shows that both ensemble approaches produce less false positives than our original approach. There is no performance difference between the Bagging and Stacking approaches.

6.2.1.3 Analysis of test D_5

In test D_5, we injected a database-related bug into DS2 that affects the product browsing logic of the application. 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 D_1 as training data: In this scenario, we want to evaluate the performance of our ensemble-based approaches and our original approach in cases where the new test and the prior test share the same environment.

Our original approach with the 3-level discretization algorithm flagged one database counter (# disk reads/second), and two tomcat counters (# threads and # private bytes). The result agrees with the nature of the injected fault: each browsing action generates additional queries to the database, leading to an increase of # disk reads/second counter on the database server. When the result of the query returns, Tomcat uses additional memory to extract the results. Furthermore, since each request would take longer to complete due to the extra queries, more threads are created in Tomcat to handle the otherwise normal workload. According to Table 6-5, 3 out of 7 expected problematic counters are detected by our original approach, leading to a precision of 1 and recall of 0.4. The F-measure is thus 0.6.

Our original approach with the EW discretization algorithm, and the Bagging and Stacking approaches flagged two database counters (# disk read/second and CPU utilization), one Tomcat counter (CPU utilization), and one application level counter (response time). We found that the EW discretization algorithm, by design, tends to create more levels than our original discretization method, which put counter values into 1 of 3 levels. Because of that, we were able to extract more association rules that reach the confidence threshold required by our mining algorithm. As more association rules are available, we were able to detect more rules that have significant change of confidence, which leads to more problematic counters being flagged. Since 4 out of 7 counters are flagged, the precision and recall of our original approach with the EW discretization algorithm as well as the Bagging and Stacking approaches are 1 and 0.57 respectively. The F-measure is 0.73.

This study shows that both our Bagging and Stacking approaches perform equally well when both the training set and the new test share the same configuration, and even experience an improvement in recall over our original approach with the 3-level discretization algorithm. Furthermore, when only 1 test is used as the training data, our ensemble-based approaches reduce to our original approach with the EW discretization algorithm, which detects performance regressions based on a single model.

Test D_1, D_2 and D_3 as training data: In this scenario, 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 with the 3-level discretization algorithm did not flag any counter. When we extract association rules from the combined dataset of test D_1, D_2, and D_3, only performance behaviors that are strong enough to persist in all three tests will be extracted. In this scenario, none of the premises of the rules generated from the combined training set were satisfied in test D_5. One such rule indicated that when the # orders/minute counter is medium, the database CPU utilization would be high. However, our original approach discretized the # orders/minute counter in test D_5 to the low level. Because of that, the premises of the rules were never satisfied and no counter was flagged. The precision of our original approach is 1 since no incorrect counter was reported. The recall and the F-measure are 0.

Our original approach with the EW discretization algorithm flagged 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 with the EW discretization algorithm are 1 and 0.1, respectively, and the F-measure is 0.3.

Both the Bagging and Stacking approaches flagged the same counters: two database counters (# disk reads/second 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 D_3 also flagged Tomcat's # threads counter. However, in the model generated from test D_2, the rules that contained "# threads" as the consequent had premises that were never satisfied in test D_5. As a result, even though the "# threads" counter behaved differently in test D_2 and D_5, the counter was not flagged in test D_5. 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.

This study shows that our ensemble approaches outperform our original approach when using a training set that contains tests with different software configurations. Our Stacking and Bagging approaches have the same performance.

6.2.2 Studied application: JPetStore

In this case study, we have conducted three one-hour performance tests (J_1, J_2, and J_3). All three tests share the same hardware environments and workload. Tests J_2 and J_3 use an older version of MySQL (ver. 5.0.1) than test J_1 (ver. 5.1.45). Tests J_1 and J_2 are used as training data. Test J_3 is injected with an environment bug in which all caching capability in MySQL is turned off. Table 6-6 summarizes the environments used in the three tests and the 6 counters expected to show performance regressions in Test J_3.

Table 6-6: Summary of Test Setup for JPetStore

Test	Software setup	Fault description	Expected problematic counters
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	**Database counters** # private bytes + # threads + CPU utilization + # context switches + # disk read bytes / second + # disk write bytes / second +

6.2.2.1 Analysis of test J_3

Our original approach with the 3-level discretization algorithm detected a performance regression in memory usage (# private bytes), and the "# threads" in the database. These observations align with the injected fault: since the caching feature is turned off in the database, less memory is used during the execution of the test. Because of the extra workload of accessing the disk, the database in turn must create more threads to handle the otherwise unchanged workload. Our original approach has a precision of 1 and recall of 0.3. The F-measure of our original approach is 0.5.

Our original approach with the EW discretization algorithm detected a decrease in the # private bytes counter in the database, as well as an increase in the CPU utilization and # thread counters. The precision of our original approach with the EW discretization algorithm is 1. The recall and F-measure are 0.5 and 0.7, respectively.

Our Bagging and Stacking approaches flagged the following three counters: # private bytes, # # disk read bytes/second, and # threads in the database. Our ensemble-based approaches achieved a precision of 1 and recall of 0.5, respectively. The F-measure is equal to 0.7.

In this case study, our original approach with the EW discretization algorithm achieved the same performance as our ensemble-based approaches.

6.2.3 Studied application: a large enterprise application

In this case study, we selected thirteen 8-hour performance tests from the organization's performance testing repository. Most of these tests were run in different labs that differ in hardware specifications, and were conducted for a minor maintenance release of the software application. In each test, over 2000 counters were collected. For each test, we removed the first and last hour, which represent the warm-up and cool-down periods.

Out of the pool of 13 tests, 10 tests have received a pass status from the performance analysts and are used to derive association rules. We evaluated the performance of the 3 newest tests (E_1, E_2 and E_3) in the pool and compared our findings with the performance analysts' assessment (Table 6-7). We now discuss our analysis for each test (E_1, E_2 and E_3) separately.

Table 6-7: Summary of analysis for the enterprise system

Test	**Performance analyst's report**	**Report status**	**Our findings**
E_1	No performance problem found.	Passed	Our approach identified abnormal behaviors in system arrival rate and throughput counters.
E_2	Arrival rates from two load generators differ significantly. Abnormally high Database transaction rate. High spikes in job queue.	Failed	Our approach flagged the counters identified by the performance analyst.
E_3	Slight elevation of # database transactions/second.	Failed	No counter flagged.

6.2.3.1 Analysis of test E_1

By analyzing the counters flagged by all our approaches for test E_1, we found that 13 counters showed true performance regressions. These 13 counters will be used to calculate the recall of our approaches for test E_1.

Using the history of 10 tests as training data, our original approach with the 3-level discretization algorithm flagged 2 throughput and 2 arrival rate counters in test E_1. The rules that flagged the counters imply that all throughput and arrival rate counters should be the same. However, upon investigation, we found that half of the arrival rates and throughput counters are high while the other half is low, suggesting that there was a mismatch in load created by the load generators. Our performance regression report has a precision of 1. Since 4 out of 13 counters are flagged, our recall and F-measure are 0.3 and 0.5 respectively.

Our original approach with the EW discretization algorithm flagged 6 counters, including 2 throughput counters, 2 arrival rate counters, the # private bytes counter of the server process and the database transactions/second counter. Our original approach with the EW discretization algorithm achieves a precision of 1, a recall of 0.46 and an F-measure of 0.6.

Our Bagging approach flagged 18 counters, including the 4 throughput and arrival rates counters 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 application decreased because fewer requests were made due to a drop in one of the arrival rate counters. We verified our finding with a performance analyst, and found that 5 out of 18 flagged counters were false positives, bringing the precision and recall of our Bagging approach to 0.7 and 1, respectively. The F-measure of our Bagging approach is 0.8.

Our Stacking approach flagged 13 counters, including the ones flagged by our original approach. Out of the 13 counters flagged, we identified 2 false positives, bringing both the precision and recall of our Stacking approach to 0.8. The F-measure of our Stacking approach is 0.8. This study shows that the performance of our Bagging and Stacking approaches are better than our original approach.

6.2.3.2 Analysis of test E_2

Our approaches together detected 15 unique counters that showed performance regressions in test E_2. These 15 counters will be used to evaluate the recall of each of our approaches in test E_2.

Our original approach with the 3-level discretization algorithm flagged 2 arrival rate counters, 2 job queue counters (each represents one sub-process), and the "# database scans/second" counter. Upon consulting with the time-series plots for each flagged counter as well as the historical range, we found that the "# database scans/second" counter had three spikes during the test. These spikes were likely the cause of the rule violations. After verifying with a performance analyst, we concluded that the spikes were caused by the application's periodic maintenance and did not constitute a performance problem. Therefore, the "# database scans/second" counter was a false positive. The precision and recall of our original approach are 0.8 and 0.3, respectively. The F-measure equals to 0.4.

Our original approach with the EW discretization algorithm 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 # private bytes and # virtual bytes counters of the application process. The precision, recall and F-measure of our original approach with the EW discretization algorithm are 0.9, 0.4 and 0.5, respectively.

Our Bagging approach flagged 20 counters, 5 of which were false positives. The remaining 15 counters included those that were flagged by our original approach. The new counters reported by our Bagging approach were mainly the side effects of the abnormality detected in the arrival rate counters. For example, as one of the load generators pushed a higher than normal load to the application, the extra requests caused the application to read from the disk more often, leading to an increase of "# disk reads/second". The results of these extra requests were written to the disk, causing an increase in the "# disk writes / second" counter. Although these side effects are the result of a higher load being pushed to the application, they can provide insight to performance analysts to investigate the ripple effect of the fault. As such, side effects should be considered as true positives. The precision and recall of our Bagging approach are 0.8 and 1 respectively. The F-measure of our Bagging approach is 0.9.

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 recall of our Stacking approach are 0.9 and 0.9. The F-measure of both our Stacking approach is 0.9.

6.2.3.3 Analysis of test E_3

There are no true positives that were detected by any of our approaches. Our original approach (with the original 3-level discretization algorithm and the EW discretization algorithm) did not flag any rule violation for this test. Upon inspection of the historical values for the counters reported by the performance analyst, we noticed that the increase of "# database transactions/second" observed in test E_3 actually fell within the range of the counter's historical values. Upon discussing with the Performance Engineering team, we concluded that the increase did not represent a performance problem. In this test, 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, 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 rate (11%) higher than the other one. The extra service requests led to a slight increase in the "# disk reads/second" counter. Upon investigation, we do not believe that these two counters represent significant performance regressions and therefore should be considered as false positives. As a result, the precision and recall of our Bagging approach are 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 recall of our Stacking approach are 1. The F-measure of our Stacking approach is 1. This study shows that our original approach shows similar performance as our Stacking approach. In the case of test E_3, our Bagging approach flags the most false positives.

From our case studies with three applications, we find that the EW discretization algorithm improves the performance of our original approach. Our ensemble approaches outperform our original approach (with either the 3-level discretization algorithm or the EW discretization algorithm), with our Stacking approach performing slightly better than our Bagging approach.

6.3 Discussion

Much of current industrial practice uses threshold-based approaches to locate performance problems. These approaches involve comparing counter averages against a set of pre-defined thresholds. When a counter average exceeds the threshold, the counter is reported. In our industrial case study, we compared the performance of our automated approaches against the organization's reports generated using such a threshold-based approach. We showed that our ensemble-based approaches could detect problems that were missed by such threshold-based approach. In addition, our ensemble-based approaches can provide details such as the correlating counters and periods where the performance regression occurred. These are useful in investigating performance regressions and locating the root cause of such regressions.

The performance tests used in our industrial case studies were conducted on a large and distributed platform with multiple machines, each hosting a different component. Changes to any component may affect the overall application. As a result, the performance of the tests used in our industrial case study is very sensitive to differences in the environments. Since Bagging aggregates the results of individual models by taking a simple majority voting and does not incorporate the environment information of performance tests, Bagging tends to produce higher recall than Stacking in exchange of precision.

Our ensemble-based approaches permit us to detect performance regressions with high accuracy in new tests that share some similarity with prior tests. However, if the new test behavior is radically different than those observed in prior tests, our ensemble-based approaches will not be accurate in detecting performance problems. In the future, we will investigate ways to automatically select tests with similar environment configurations as the new test from the repository.

6.4 Summary

The traditional approach of analyzing performance tests to detect performance regressions is an error-prone and time consuming process. In this chapter, we extended our automated approach introduced in Chapter 5 with Bagging and Stacking ensemble-learning algorithms. Our improved analysis approaches address the practical need for an automated approach for analyzing performance tests that are conducted with heterogeneous environments. In our three case studies, we showed that our ensemble approach with Stacking outperforms our original approach. In the next chapter, we will summarize the thesis and present a list of possible future work.

Chapter 7

Conclusion and Future Work

We presented approaches to address the challenges of identifying performance regressions in the software applications at the design and implementation levels. This chapter summarizes the main ideas presented in this thesis. In addition, we propose avenues for future work.

Performance verification is an essential step for organizations to prevent performance regressions from slipping into production. Currently, organizations limit performance verification at the functional level with the support of specialized testing frameworks [38] [28] and the implementation level through performance testing. Evaluation of high impact design changes is delayed until those changes are implemented, at which time performance regressions are the most costly to fix. Furthermore, the analysis of the results of performance verification is both time-consuming and error-prone due to the large volume of collected data, the absence of formal objectives, the subjectivity of performance analysts and heterogeneous environments. In this thesis, we proposed several approaches to address the above challenges of performance verification. First, we proposed a layered simulation modeling approach that can be use to uncover performance regression at the design level. Second, we introduced an automated analysis approach for uncovering performance regressions in performance tests. Finally, we extended our automated analysis approach with ensemble-learning algorithms to handle performance tests conducted in heterogeneous environments. Through our case studies, we demonstrated our success in applying our approaches to various applications.

7.1 Main topics covered

We introduced our approach of constructing layered simulation models for analyzing the application performance at the design level in Chapter 4. Our layered simulation model separates the performance concerns of different stakeholders and can be constructed incrementally. We show that our approach is easy to adopt through two case studies with the RSS cloud system and the performance monitor for ULS applications. By adjusting the workload fed into the layered simulation model, performance analysts can evaluate the performance impact of a proposed design changes.

Chapter 5 presented our automated approach to analyze results of performance tests. Our approach uses association rule mining to extract a set of expected counter correlations from the performance testing repository and check the new test for violations of the expected correlations. Our case studies with two open source e-commerce applications and a large enterprise application showed that our approach can effectively discover performance regressions and scale to large applications.

Chapter 6 extends the approach in Chapter 5 using the Bagging and Stacking ensemble-learning algorithms. Our ensemble-based approach can be used to analyze performance tests that are executed on a variety of software and hardware environments. Our case studies with three applications showed that our ensemble approaches detected more performance regressions than our original approach in Chapter 5 in the presence of performance tests that are conducted in heterogeneous environments.

7.2 Contributions

The contributions of this thesis are as followed:

1. We presented an approach to build layered simulation models for the purpose of evaluating changes to application designs. Our layered simulation model separates the different performance concerns of the applications.
2. We proposed an automated approach to automatically detect performance regressions. The approach uses prior tests to derive performance signatures, and then compares the

new test to these signatures. Our approach flags the time when counters violate the performance signature, easing root cause analysis.

3. We propose an approach to deal with performance tests conducted in heterogeneous environments, which is common in practice.
4. We are providing a replication package of our case studies with the open source projects to foster research on the automated discovery of performance regressions [31].

7.3 Future work

7.3.1 Online analysis of performance tests

Previous approaches as well as the automated analysis approaches introduced in this thesis focus on offline analysis, that is, after the performance tests are completed. A performance test can take up to a few days to execute; anomalous behavior that exhibits early on in the test will only be discovered after the test is finished. We believe early knowledge of the anomalous behavior would allow performance analysts to make the decision of stopping the test early and freeing up the lab resources. In the future, we intend to build a behavioral database by extracting the expected performance behavior from prior tests with our analysis approach. As a new test is executing, the test's performance counter will be checked against the behavioral database in real time. Performance analysts will be notified for any deviations of the expected behavior.

7.3.2 Compacting the performance regression report

Our automated analysis approaches produce performance regression reports that show the counters that contain performance regressions and the list of rules the counter violated. Performance analysts can inspect these rules to understand the relations among counters. From our case studies, we notice that some of the rules produced are highly similar. For example, the premise of one rule is the superset of another rule, while having the same consequence. Another issue is that some performance counters provide redundant information. For example, operating systems are capable of recording the utilizations of individual CPU cores as well as the total CPU utilization. Since the total CPU utilization is basically the sum of all the CPU cores, we can solely analyze the counters corresponding to the individual cores without losing information. In the future, we will research ways to automatically eliminate redundant counters and merge similar rules to condense information presented to performance analysts.

7.3.3 Maintaining the training data for our automated analysis approach

Our ensemble-based approach permits us to detect performance regressions with high accuracy in new tests that share some similarity with prior tests. However, the accuracy of our ensemble approach will decrease if the new test behavior is radically different from those observed in prior tests. In the future, we will investigate ways to automatically update the training data over time and prevent polluting the training data with tests that contain performance regressions.

7.3.4 Using performance signatures to build performance models

Many modeling approaches are proposed to predict and evaluate application performance [20] [26]. These approaches can greatly help performance analysis and capacity planning efforts. However, creation of such performance models requires detailed documentation about the application that may not be available. In the future, we plan to explore the possibility of using the performance signatures generated by our approach to specify the resource consumption relationship between different components in the performance models.

References

[1] R Agrawal and Ramakrishnan Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, San Francisco, CA, USA, 1994, pp. 487-499.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, 2003, pp. 74-89.

[3] A. Alberto and E. J. Weyuker, "The Role of Modeling in the Performance Testing of E-Commerce Applications," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 1072-1083, Dec. 2004.

[4] A. Avritzer and B. Larson, "Load Testing Software using Deterministic State Testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, Massachusetts, USA, 1993, pp. 82-88.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, Berkeley, CA, USA, 2004, pp. 259-272.

[6] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 248-260, Apr. 1975.

[7] E. Bauer and R. Kohavi, "An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants," *Machine Learning*, vol. 36, no. 1-2, pp. 105-139, July 1999.

[8] F. Bause and P. Kemper, "QPN-Tool for qualitative and quantitative analysis of queueing Petri nets," in *Proceedings of the 7th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, Vienna, Austria, 1994, pp. 321-334.

[9] C. Begin. (2011, Jan.) iBATIS JPetStore. [Online]. http://sourceforge.net/projects/ibatisjpetstore/

[10] P. Bodik, M. Goldszmidt, and A. Fox, "HiLighter: Automatically Building Robust

Signatures of Performance Behavior for Small- and Large-Scale Systems," in *Proceedings of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, San Diego, California, 2008, pp. 33-40.

[11] A. B. Bondi, "Automating the Analysis of Load Test Results to Assess the Scalability and Stability of a Component," in *Proceedings of the 33rd International Computer Measurement Group Conference (CMG)*, San Diego, CA, USA, 2007, pp. 133-146.

[12] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123-140, Aug. 1996.

[13] L. Breiman, "Stacked regressions," *Machine Learning*, vol. 24, no. 1, pp. 49-64, July 1996.

[14] L. Bulej, T. Kalibera, and P. Tuma, "Regression benchmarking with simple middleware benchmarks," in *Proceedings of the International Conference on Performance, Computing, and Communications (IPCCC)*, Phoenix, AZ, USA, 2004, pp. 771-776.

[15] L. Bulej, T. Kalibera, and P. Tuma, "Repeated results analysis for middleware regression benchmarking," *Performance Evaluation*, vol. 60, no. 1-4, pp. 345-358, May 2005.

[16] M. Y. Chen et al., "Path-based faliure and evolution management," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, Berkeley, CA, USA, 2004, pp. 63-72.

[17] 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 Symposium on Opearting Systems Design & Implementation (OSDI)*, San Francisco, CA, USA, 2004, pp. 231-244.

[18] I. Cohen et al., "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, 2005, pp. 105-118.

[19] Compuware. (2006, Oct.) Applied performance management survey. [Online]. http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/ APM_Survey_Report.pdf

[20] V. Cortellessa, P. Pierini, and D. Rossi, "Integrating software models and platform models for performance analysis," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 385-401, June 2007.

[21] M. A. Daniel, V. Almeida, and L. W. Dowdy, *Performance by design*. Upper Saddle River,

New Jersey, USA: Prentice Hall, 2004.

[31] (2011, Jan.) Data used in the case studies with open source applications. [Online]. https://qshare.queensu.ca/Users01/3kcdf/www/icse2011.zip

[55] (2011, Jan.) Dell DVD Store Database Test Suite. [Online]. http://linux.dell.com/dvdstore/

[22] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *Proceedings of the 20th International Conference on Machine Learning (ICML)*, 1995, pp. 194-202.

[23] (2011, Jan.) Exchange Server 2003 MAPI Messaging Benchmark 3. [Online]. http://technet.microsoft.com/en-us/library/cc164328%28EXCHG.65%29.aspx

[24] K. C. Foo et al., "Mining performance regression testing repositories for automated performance analysis," in *Proceedings of the 10th International Conference on Quality Software (QSIC)*, Zhangjiajie, China, 2010, pp. 32-41.

[25] G. Franks and C.M. Woodside, "Performance of multi-level client-server systems with parallel service operations," in *Proceedings of the 1st international workshop on Software and performance (WOSP)*, Santa Fe, New Mexico, USA, 1998, pp. 120-130.

[26] V. Garousi, L. C. Briand, and Y Labiche, "Traffic-aware stress testing of distributed real-time systems based on UML models," in *Proceedings of the 1st international conference on software testing, verification, and validation*, Lillehammer, 2006, pp. 92-101.

[27] (2011, Jan.) Google Reader. [Online]. http://www.google.com/reader

[28] J. Halleux. (2011, Jan.) NPerf, A Performance Benchmark Framework for.NET. [Online]. http://www.codeproject.com/KB/architecture/nperf.aspx

[29] J. Herrington. (2011, Jan.) Five common PHP database problems. [Online]. http://www.ibm.com/developerworks/library/os-php-dbmistake/index.html

[30] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, Sorrento, Italy, 1994, pp. 191-200.

[32] Tauseef A. Israr, Danny H. Lau, Greg Franks, and Murray Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th International Workshop on Software and Performance (WOSP)*, New

York, NY, USA, 2005, pp. 147-158.

[33] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, Edmonton, AB, Canada, 2009, pp. 125-134.

[34] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, Beijing, China, 2008, pp. 307-316.

[35] M. Jiang, M. A. Munawar, and T. Reidemeister, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *Proceedings of the 2009 International Conference on Dependable Systems and Networks (DSN)*, Estoril, Lisbon, Portugal, 2009, pp. 285-294.

[36] M. Jiang, M. A. Munawar, T. Reidemeister, and P. Ward, "System monitoring with metric-correlation models: problems and solutions," in *Proceedings of the 6th International Conference on Autonomic Computing (ICAC)*, Barcelona, Spain, 2009, pp. 13-22.

[37] R. Jin and H. Liu, "SWITCH: A novel approach to ensemble learning for heterogeneous," *Machine Learning: ECML 2004*, vol. 3201, pp. 560-562, 2004.

[38] (2011, Jan.) JUnitPerf. [Online]. http://www.clarkware.com/software/JUnitPerf.html

[39] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42-50, Nov. 1995.

[40] 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 34th Computer Software and Applications Conference (COMPSAC)*, Seoul, Korea, 2010, pp. 117-126.

[41] D. A. Menasce, "Two-level iterative queuing modeling of software contention," in *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, Fairfax, VA, USA, 2002, pp. 267-276.

[42] (2011, Jan.) NeoLoad. [Online]. http://www.neotys.com/

[43] (2011, Jan.) OMNeT++ Network Simulation Framework. [Online]. http://www.omnetpp.org/

[44] D. Oppenheimer and D. A. Patterson, "Architecture and dependability of large-scale internet

services," *IEEE Internet Computing*, vol. 6, no. 5, pp. 41-49, Sep. 2002.

[45] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, Los Alamitos, CA, USA, 1994, pp. 279-287.

[46] L. F. Pollacia, "A survey of discrete event simulation and state-of-the-art discrete event languages," *ACM SIGSIM Simulation Digest*, vol. 20, no. 3, pp. 8-25, Sep. 1989.

[47] J. R. Quinlan, "Bagging, boosting, and C4.5.," in *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, Portland, Oregon, 1996, pp. 725-730.

[48] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th International Conference on World Wide Web (WWW)*, Edinburgh, Scotland, 2006, pp. 347-356.

[49] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689-700 , Aug. 1995.

[50] (2011, Jan.) RSS 0.90 Specification. [Online]. http://www.rssboard.org/rss-0-9-0

[51] (2011, Jan.) RSSCloud. [Online]. http://rsscloud.org/

[52] C. U. Smith, *Performance engineering of software systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[53] C. U. Smith and L. G. Williams, *Performance solutions: a practical guide to creating responsive, scalable software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002.

[54] Michael Smit et al., "Capacity planning for service-oriented architectures," in *Proceedings of the conference of the center for advanced studies on collaborative research (CASCON)*, Toronto, Canada, 2008, pp. 144-156.

[56] (2011, Jan.) The R Project for Statistical Computing. [Online]. http://www.r-project.org

[57] I. H. Witten and E. Frank, *Data mining: practical machine learning tools and techniques*. San Francisco, CA, USA: Morgan Kaufmann, 2005.

[58] D. H. Wolpert, "Stacked Generalization," *Neural Networks*, vol. 5, no. 2, pp. 241-259, 1992.

[59] C. M. Woodside, "A three-view model for performance engineering of concurrent software," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 754-767 , Sep. 1995.

[60] C.M. Woodside, "Throughput calculation for basic stochastic rendezvous networks,"

Performance Evaluation, vol. 9, no. 2, pp. 143-160, Apr. 1989.

[61] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Proceedings of the 2007 International Conference on Software Engineering (FOSE)*, Washington, DC, USA, 2007, pp. 171-187.

[62] M. Woodside, C. Hrishchuk, B. Selic, and S. Bayarov, "Automated Performance Modeling of Software Generated by a Design Environment," *Performance Evaluation*, vol. 45, no. 2-3, pp. 107-123, July 2001.

[63] C.M., Neilson, J.E. Woodside, D.C. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20-34, Jan. 1995.

[64] (2011, Jan.) WordPress. [Online]. http://wordpress.org/

[65] J. Xu and J. Kuusela, "Modeling execution architecture of software system using colored Petri nets," in *Proceedings of the 1st International Workshop on Software and Performance (WOSP)*, Santa Fe, New Mexico, USA, 1998, pp. 70-75.

[67] S. Zhang, I. Cohen, M. Goldszmidt, 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)*, Yokohama, Japan, 2005, pp. 644-653.