# APPLYING STATISTICAL PROCESS CONTROL TO AUTOMATE THE ANALYSIS OF PERFORMANCE LOAD TESTS

by

THANH H. D. NGUYEN

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

February 2017

# Abstract

Avoiding performance regressions is very important in the evolution of ultra-large software systems. Even the addition of an extra field or control statement can degrade the performance of the software system considerably as the impact of such simple changes is multiplied by the millions of user requests that are processed simultaneously. Performance testers conduct performance load tests and analyze the results of such tests, before every new deployment, to ensure that there are no performance regressions. However, such an analysis is challenging because each test run produces a large amount of performance counter (e.g., CPU and memory readings) data.

In this thesis, we propose approaches based on a statistical process control technique, called control charts, to automatically identify and determine the causes of performance regressions. We conduct case studies on three different software systems to evaluate our proposed approaches. The results show that our approaches can determine if a new test run has performance regressions. Compared to existing approaches, our approaches can detect performance regressions accurately in most

of our case studies.

# Acknowledgments

I want to express my gratitude for the mentorship, the inspiration, and the support of my supervisor Dr. Ahmed E. Hassan. Without Ahmed's encouragement, this thesis would not be possible.

I would like to thank Dr. Bram Adams, Dr. Zhen Ming Jiang, and Dr. Mei Nagappan, and members of the Software Analysis and Intelligence Lab (SAIL) for their feedback, advices, and contributions in the studies.

I would also want to thank Dr. Dorothea Blostein, Dr. Juergen Dingel, and Dr. Patrick Martin for the advice that they gave me.

I thank BlackBerry for providing support and data access for the studies in this thesis. In particular, I appreciate the feedback of Parminder Flora, Mohamed Nasser, and other members of the two performance engineering teams that assisted me with the studies. The findings and opinions expressed in this thesis are those of the author and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliates. Moreover, my findings do not in any way reflect the quality of BlackBerry's products.

I also like to thank my parents for their love and support through my entire life. I also thank my wife, Trang T. P. Huynh, for her love, patient, and support.

# Publications

Earlier versions of the work in the thesis were published as listed below:

1. *Automated Verification of Load Tests Using Control Charts*
   Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, Parminder Flora, In Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC 2011). Ho Chi Minh, Vietnam, 5-8 Dec. 2011.

2. *Automated Detection of Performance Regressions Using Statistical Process Control Techniques*
   Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, Parminder Flora, In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012). Boston, MA, US, April 22-25, 2012.

3. *Using Control Charts for Detecting and Understanding Performance Regressions in Large Software*

Thanh H.D. Nguyen, In Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST 2012). Montreal, Canada. April 17-21, 2012.

4. *An Industrial Case Study of Automatically Identifying Performance Regression-Causes*
   Thanh H.D. Nguyen, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, Parminder Flora, In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), Hyderabad, India, May 31-June 1, 2014.

# Contents

# List of Tables

# List of Figures

# Part I

# Introduction and Background

# CHAPTER 1

# Introduction

Performance is an essential quality of software systems. In large software systems with millions of users, the ability to efficiently serve a large number of concurrent users is very important. Researchers estimate the potential revenue loss of electronic commerce due to performance related issues at about $25 billion dollars yearly (PLC, 2001). For example, users usually abandon an online transaction if they have to wait for more than 10 seconds (Ceaparu et al., 2004; Weyuker and Vokolos, 2000). Losing users due to poor performance can be disastrous.

A software system is said to have a *performance regression* when a new version of the software system performs relatively worse than the previous version. For example, a change in the database schema might increase the response time of a query by 10%. This 10% increase can push the overall response time of the software system over the ten-second threshold. In such case, the new version of the software might require additional hardware to maintain the same level of service, otherwise

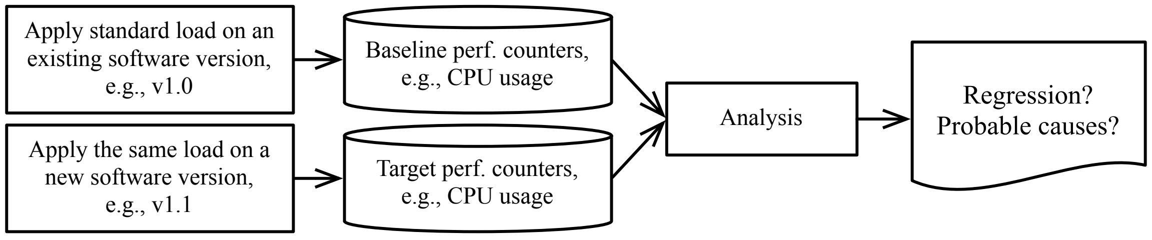

Apply standard load on an existing software version, e.g., v1.0
Apply the same load on a new software version, e.g., v1.1
Baseline perf. counters, e.g., CPU usage
Target perf. counters, e.g., CPU usage
Analysis
Regression? Probable causes?

Figure 1.1: Analysis of performance regression load tests

there is a risk of losing customers. We note that the term *performance degradation* is also used instead of the term *performance regression* in some research communities.

To avoid performance regressions, performance testers conduct performance load tests prior to the deployment of every new software version in order to identify performance regressions. Figure 1.1 shows the typical steps of performance load testing. The first step is to apply the same field-like load to both the old version and new version of the software on the same field-like hardware. This step is usually done by load generation tools such as HP's Load Runner (Hewlett Packard, 2010) or Apache's JMeter (The Apache Software Foundation, 2012). During the test runs, performance counters, such as CPU utilizations or memory utilization, are collected. Then, in the second step, the two sets of counters, which are called baseline counters for the old version and target counters for the new version, are analyzed to answer the following questions for every performance load test:

- Is there a performance regression?
- What is the root-cause of the performance regression?

One of the biggest challenges in performance analysis is the massive amount of performance counter data that must be analyzed (Nickolayev et al., 1997). In a large software system, there are thousands of counters, e.g., CPU utilization, memory utilization, or I/O operation rate. Each counter would have millions of samples. It is time consuming to analyze the counters. Since performance load testing is usually the last step of the software development cycle, performance testers usually face high pressure to rapidly complete the test analysis so the software can be released without further delays. Performance testers often rely on their experience to identify performance regressions, and to determine the probable root-causes of such regressions. This process can take a few hours or even a few days depending on the experience of the testers and the complexity of the regression.

To cope with the large amount of performance counter data, several data reduction approaches are proposed in literature. Examples of such approaches include principal component analysis (Eeckhout et al., 2003a,b,c; Malik, 2010; Malik et al., 2010; Oliner and Aiken, 2011), projection pursuit (Vetter and Reed, 1999), or statistical clustering (Nickolayev et al., 1997). These approaches have their advantages and disadvantages. One of the potential disadvantages is that they often change the form of the counter data (e.g., creating principal components) - making it difficult to explain the analysis results. In this thesis, we suggest the use of control charts for data reduction. Control charts are used in statistical process control to determine if an industrial or business process is within or out of control (Shewhart, 1931). For each performance counter, we achieve data reduction by replacing the counter and its baseline with the control charts measurement of violation ratio. Then we use the violation ratios for data analysis. Since the violation ratio is easy to understand, it is easier to explain and communicate the analysis results in comparison with other data reduction approaches.

In this thesis, we first derive an approach to help performance testers automatically identify test runs with performance regressions. Once a performance regression is identified, the cause of the performance regression needs to be identified. This task can take days or even weeks. Hence, we also derive an approach that leverages control charts as the data reduction approach to automatically determine the causes of performance regression. Finally, we apply our control charts based approaches to the performance testing process of a commercial software system and report our hands-on experience.

**Thesis statement:** Control charts can identify performance regressions in performance load tests. Moreover, control charts can assist in identifying the causes of performance regressions. Control chart based approaches can be applied in practice.

## 1.1 Thesis Contribution:

The key contributions of this thesis are as follows:

- We apply control charts into the analysis of performance load test results. We show, through empirical case studies, that control charts can be used to automatically identify test runs with performance regressions (Chapter 6 and 7).
- We apply machine learning to automatically suggest the possible causes of the performance regression using historical data (Chapter 8).

## 1.2 Summary:

In this thesis, we show that testers can adopt our approaches to automate the analysis of the test results. However, we note that there are other challenges in achieving full automation of the performance regression tests. For example, what is the suitable the length of a test? What are the suitable configurations for the system under-test? These challenges still require the knowledge and experience of the testers.

# CHAPTER 2

# Related Work

We conduct a literature survey to identify existing approaches to analyze the results of performance load tests. Performance load testing is a relatively new area in software engineering research (Jiang and Hassan, 2015). However, the performance of software and hardware systems has been studied extensively in the past. Approaches used in such studies can be applied to the area of performance load testing.

## 2.1 Process Followed for the Selection of Surveyed Studies

We start by finding relevant papers in the main publication venues for software performance studies such as: the International Conference on Dependable Systems and Networks (DSN) or the International Conference on Performance Engineering

(ICPE). Then we follow the papers' references to find other relevant publications in other venues.

To focus our inquiry, we focus on papers that aim to find performance problems in a software system. We also include papers that study the performance of operating systems, using hardware resource counters such as the number of interrupts and data or instruction cache hits/misses. We specifically focus on papers that analyze counter data. We exclude studies that applied data mining approaches on the execution traces (logs) (Jiang et al., 2008) or communication traces (Aguilera et al., 2003) to detect performance issues. We exclude such studies since the input data set (unstructured trace files) is different from what we are concentrating on in this thesis (performance counters). However, there are studies (e.g., (Gainaru et al., 2012a, 2013; Xu et al., 2009)), that used trace files as input but they parsed the trace files to produce counters, e.g. counting frequency of each event per time interval during the test runs. We include such studies since the input data is now counters. Also, because we focus on research that studies performance of software-intensive systems, we do not explicitly search for studies of hardware intensive systems such as those from the International Symposium on Performance Analysis of Systems and Software or Modelling, Analysis, and Simulation of Computer and Telecommunication Systems Conference.

## 2.2 Classification of Studies

Figure 2.1 shows the five meta steps in analyzing performance counters that we can abstract from the studies that we found:

1. Select the studied software and data: We outline the types of software system that the studies used. We also classify the studies based on the used data: real versus synthetic data (Section 2.2.5).

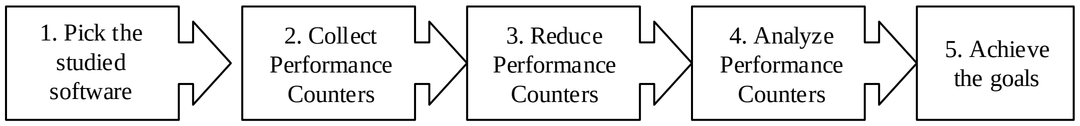

1. Pick the studied software
2. Collect Performance Counters
3. Reduce Performance Counters
4. Analyze Performance Counters
5. Achieve the goals

Figure 2.1: The five meta steps in analyzing performance counters

2. Collect the performance counter data: We classify the studies using the type of performance counter that they used: hardware (HW) layer performance counters, operating system (OS) layer performance counters, and software (SW) layer performance counters (Section 2.2.1).

3. Reduce the performance counter data: We classify the papers along the three used types of data reduction approaches: filtering counters, creating smaller counter set, and reducing counter data (Section 2.2.2).

4. Analyze the performance counter data: We classify the papers along the three types of approach that are used to analyze the performance counters: without any empirical data, with untagged data, with tagged data (Section 2.2.3).

5. Achieve the goals: We classify the papers along three types of goals: monitoring live production system, analyzing performance tests, and benchmarking hardware (Section 2.2.4).

In the next five subsections, we outline the studies along the aforementioned dimensions. We conclude and summarize all the papers and their attributes (Table 2.2) in Section 2.3.

### 2.2.1 Types of Performance Counter

Performance counters come from different layers of the hardware and software stack:

- At the hardware layer, we have hardware counters (HW). Examples of such counters include instruction types, data/instruction cache hits/misses, branch prediction hits/misses, or the number of executed parallel instructions. Hardware counters are used mainly in studies of hardware performance such as the work of Hammerly et al. (Hamerly and Elkan, 2001) or benchmarking (Section 2.2.4) studies such as the work of Eeckhout et al. (Eeckhout et al., 2003a,b,c).

- In the platform layer, we have operating system counters (OS). Examples of such counters include CPU utilization, memory utilization, disk input/output (IO), or network IO. Most performance studies use counters from this layer as they are the most accessible type of performance counters. The OS layer counters are sometime used by themselves (Vetter and Reed, 1999). However, when the performance of software is studied, OS layer counters are usually used along with the software layer performance counters.

- At the software layer, we have counters from the software itself, i.e., the software layer counters (SW). These software layer counters include throughput, queue size, response time, or processing time. These counters are specific to each software system. For example, there are many job queues in a software system. For each queue, there would be counters that report the current number of jobs in the queue, the number of jobs left the queue, and the amount of time that the jobs spent in the queue. Most studies (Jiang et al., 2009a; Malik, 2010; Malik et al., 2010; Zhao et al., 2009) use counters from both the software layer and the OS layer.

  We note that one can separate the layers further. For example, middleware can be a layer between the hardware and the platform layer. Our choices of the layers represent a simplified view of the software/hardware stack.

Going from the OS layer to the software layer represents a trade-off. Lower layer counters are usually the most accessible counters. However, these counters hold less system/domain specific information. For example, if the CPU utilization increases by a few percentage points, it would be difficult to determine the causes of the regression using just the OS layer counters. Higher layer counters would help in that case. They hold more domain information such as whether the size of a queue had increased. The increased queues would indicate the locations of the bottleneck (based on the location of the queues in the call flow of the software). However, higher layer counters are more difficult and expensive to produce and collect. For example, to add a queue size counter into a software, the software developers often have to add monitoring code into their own source code. The monitoring code has to be configurable, e.g., the frequency of reporting or storage location of the counters. The system engineers, who install and monitor the software, then have to configure and monitor the new counter. On the other hand, CPU or memory utilization counters at the OS layer are usually available on most operating systems. There exist many off-the-shelf standard tools (e.g. (Godard, 2014; Microsoft Corp., 2011)) to collect and display these counters.

Lower layer counters are more accessible but have less value for troubleshooting software specific performance regressions. Higher layer counters are expensive to collect but they are more valuable in troubleshooting.

### 2.2.2 Data Reduction Approaches

Reduction of performance counter data is necessary for studying the performance of large software systems. Not all studies use data reduction before the analysis step. Some studies, e.g. (Cherkasova et al., 2008), can detect performance problems using the full set of original counters. However, the large amount of counter data is often considered as a key challenge in analyzing performance counters. Zhao et al. (Zhao et al., 2009) and Oliner et al. (Oliner and Aiken, 2011) specifically study approaches to reduce the amount of performance counter data. Cohen et al. (Cohen et al., 2005) show that using reduced counter data can outperform using the raw counter data in terms of accuracy.

The main goal of data reduction is to reduce the amount of performance counter data. A simple reduction operation can be the removal of performance counters that did not change during the test runs. These counters will not affect the analysis results. Such simple reduction would reduce the amount of counters for analysis.

**Filter out Counters**

A more sophisticated data reduction approach would be filtering out the counters that would not influence the results of the subsequent analysis approach. The following studies employ such an approach:

- Vetter and Reed (Vetter and Reed, 1999) apply a statistical technique called projection pursuit to reduce the large number of similar counters into a smaller set of representative counters. First the counters are divided into intervals of time, e.g., three minutes. Then, for each interval, their approach picks the top most changed $m$ counters, e.g., three, for that particular time interval. Their approach reduces the number of counters from thousands to just $m$ counters.

- Another approach to reduce data is statistical clustering, which is used by Nickolayev et al. (Nickolayev et al., 1997). While analyzing the performance of a parallel system with a massive amount of computer processors, Nickolayev et al. employ K-means clustering to determine the top most representative processors for each time interval. Since most programs cannot make effective use of all processors equally, the researchers can reduce the amount of counters from all the processors to a few representative processors.

- Zhao et al. (Zhao et al., 2009) improves the clustering based reduction using self-correlating information. Instead of just feeding all the counters into a clustering algorithm and determining the top representative counters, they first apply on-the-fly correlation algorithms on the counters. These algorithms reduce the counters information by identifying temporal changes or spatial changes. For example, each server in a region usually gets the same amount of requests. So all the servers' performance counters are usually highly correlated. Thus sending the counters of one node to the monitoring server might be sufficient. The reduced counters are then fed into a clustering algorithm on the monitoring server to determine the most changed counters. Using this approach, the researchers were able to reduce the counter data further than just clustering alone.

**Create New Composite Metrics**

Instead of filtering out the counters, one can also achieve data reduction by creating another smaller set of new "composite" counters which represent the original set of counters as a whole. The following studies employ such an approach:

- Malik et al. (Malik, 2010; Malik et al., 2010) use principal component analysis (PCA) to find the valuable performance counters before analyzing the counter data to identify performance regression. The main idea behind PCA is to transform possibly correlated variables, i.e., performance counters, into non-correlated vectors, which are called components. The researchers select a subset of the components to determine which ones are the most promising for further analysis.

- Eeckhout et al. (Eeckhout et al., 2003a,b,c) determine the input workloads that are most useful for benchmarking Java applications on virtual machines. However, they first use PCA to reduce the performance counters into a smaller number of vectors. Then they use the principal components to compare the characteristics of the workload. PCA is a common technique for this kind of reduction. Using the few top components for data analysis would capture most of the characteristics of variables.

- Oliner and Aiken (Oliner and Aiken, 2011) studied logs instead of performance counters. However, they extract events from the logs and use the event frequencies to analyze the interactions between components in large software systems. Their approach also uses PCA to reduce the event frequency counters.
- Instead of using PCA, Jiang et al. (Jiang et al., 2009a) develop an approach that transforms the counters into groups of similar counters using Normalized Mutual Information (NMI). NMI is a measure of similarity between counters. Using NMI, the researchers group the counters into groups of highly correlated metrics. So, for monitoring the performance of the software system, the researchers only need to monitor the changes of groups as opposed to monitoring each individual metric.

**Reduction of Individual Counters**

Both of the aforementioned types of data reduction approach alter the set of counters to a smaller set. Another approach to reduce performance counter data is to keep the same set of counters but reduce the data points of each of the counters themselves. The following studies employ such an approach:

- Foo et al. (Foo, 2011; Foo et al., 2010) discretize the counters' values into low, medium, or high. Then they use the discretized counters to build association rules to detect performance anomalies in performance load tests.

- Bezemer et al. (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Wiertz et al., 2014) use a similar approach to Foo et al. (Foo, 2011; Foo et al., 2010). They also discretize the counters values into low, medium, and high. Then they build association rules to identify performance issues in live systems.

All the above three types of data reduction approaches have their own advantages and disadvantages. However, our experience in working with performance testers shows that preserving the actual counters improves the adoptability of an approach in practice. The first reason is that it is easier to verify the result of the approach using the reduced data set if the counters are not merged into composite counters. For example, let us assume that we have an analysis approach which uses machine learners to identify the location of performance regressions, i.e., which component performs worse than before. To justify the learner's selection, it would be easier if we can observe the relative increases in the queues of that component compared to other components' queues. If the counters are reduced using PCA, for example, it would be harder to explain and rationalize the learner's selection since all the counters are merged into components. The second reason is that it is easier to explain and communicate the analysis results if all the counters are presented. Practitioners have better understanding and familiarity with the performance counters. They can associate the counters with areas of the code or domain level use cases of their software. Hence, it is easier for practitioners to explain and communicate the analysis results if they can observe how the counters are used in the analysis. Hence, we favour such type of data reduction approach.

There are three major approaches for reducing performance counter data: filter out counters, create new composite metrics, and reduction of data points. Preserving the actual counters eases the adoptability of an approach.

### 2.2.3 Data Analysis Approaches

Most performance studies involve comparing different sets of performance counter data. For example, Malik et al. (Malik, 2010; Malik et al., 2010) compared the baseline performance load test with the new performance load test. Stehle et al. (Stehle et al., 2010) compare the performance counters when a system is under attack to when it is not.

The surveyed data analysis approaches can be classified into three types according to the availability of the data:

- Without any performance counter data: In this case, the software system usually does not exist. So performance counters cannot be collected using real test runs or from production. So the researchers usually employ analytical models.
- With untagged performance counter data: In this case, the performance counters can be collected. However, there are no tags (e.g., error or normal behaviour) associated with the counters. The researchers usually employ unsupervised learning algorithms that measure the degree to which two sets of performance counters are different from each other.
- With tagged performance counter data: When the counter data is tagged, researchers usually use supervised learning algorithms that can learn then predict the properties of performance counters using the tagged performance counters.

In the next three sub-sections, we delve into these three types of approaches. We explain the trade-offs between these types of data analysis approaches in Section 2.2.5.

### Without Any Performance Counter Data

There are cases where, a software system does not exist yet. Or it may be hard to conduct performance load tests or collect production data for a particular software system. However, there is still a need to determine bottlenecks, evaluate performance, or optimize designs for such a software system (Casale and Serazzi, 2012). For example, in the early stages of the software lifecycle, analysts might want to compare the performance of different architectures. Researchers employ approaches that essentially create performance models of the software system (Petriu et al., 2012; Woodside et al., 2014). The goal, then, is to solve the models using analytical models such as queueing network based approaches (Balsamo et al., 2004).

**Simulation Models**

To evaluate the performance of alternative software architectures, Williams and Smith (Smith and Williams, 1997, 2002; Williams and Smith, 1998) construct software executions models. These are some of the earliest literature that describe a systematic approach to analyze software performance. The models are call graphs between elements of the software. Each call is, then, associated with computing resources. Then, using simulation, one can determine the possible contentions of each proposed architecture.

Arief and Speirs (Arief and Speirs, 1999, 2000) derive an approach, which was implemented into a tool, that transforms UML diagrams into a Simulation Model Language. Then, a Java-based simulation program called JavaSIM (Little, 2015) is used to simulate the software behaviour. The researchers demonstrated how the performance of the software can be predicted using their approach.

Miguel et al. (de Miguel et al., 2001) also introduce a similar approach that transforms UML diagrams of software into simulation models. One can, then, use simulation software such as OPNET (Technology, 2015) to examine the performance characteristics of the proposed software architects.

There are a variety of commercially available software than can be used to model software and run simulations to determine the performance characteristic of the proposed architect. One example is IBM Rational Rhapsody (IBM Corporation, 2012). Rhapsody can be used to evaluate software design. Another example is OPNET (Technology, 2015) that is very popular in evaluating network performance through simulation.

**Queueing Networks**

A queueing network is a type of mathematical model (Baccelli and Bremaud, 2003). It is commonly used to model systems where incoming jobs are queued and processed by servers. For example, in a supermarket, the jobs are customers with goods that they picked from the market's shelves. The servers are check-out counters. Queueing networks provides a mathematical approach to estimate how fast a system of servers can process a given number of incoming jobs.

Instead of running a simulation on the models of software, as the approaches in the previous subsection, one can transform software models into queueing networks, then, use the mathematics of queueing models to determine the performance of the software.

Cortellessa and Morandola (Cortellessa and Mirandola, 2000; Cortellessa et al., 2000, 2001) describe approaches that transform different types of UML diagrams to queueing networks. Then they use the produced queueing networks to analyze the performance of the propose software. The researchers call their approach "PRIMA-UML". Later on, Grassi and Mirandola (Grassi and Mirandola, 2002) extend the approach to mobile applications.

Another notable example of the application of queueing network into simulation of software performance is the JMT tool by Bertoli et al. (Bertoli et al., 2006, 2009; Casale and Serazzi, 2012). The JMT tool is a Java application that allows performance testers to model and simulate the performance of hardware, network, and software systems.

Ghaith et al. (Ghaith et al., 2013) apply a queueing network simulator on existing software to learn about the effect of workload changes on a software system. The researchers first conduct real test runs to create a model of the software performance based on a single input workload. Then the researchers use that model to infer the software behaviour with other input workloads. Moreover, the researchers apply the same approach to identify software contentions (Ghaith et al., 2014).

Osman et al. (Osman et al., 2009) suggest the use of queueing networks to model the design of databases. As a proof of concept, the researchers demonstrate how one can use queueing network to determine the performance bottleneck of a simple database design for the TPC-C benchmark workload. The researchers also publish a literature overview (Osman and Knottenbelt, 2012) on the use of analytical models to evaluate the performance of database systems.

**Layered Queueing Networks**

While the above studies use queueing networks, Franks et al. (Franks et al., 2009) gives a good account for approaches that use a more advanced form of queueing networks called layered queueing networks. Layered queueing networks are created specifically for analyzing software performance (Woodside et al., 1986). The main benefit of layered queueing networks, compared to plain queueing networks, is that the simulation of such networks permits service queues to wait for upper layer service queues to finish their processing. This feature enables the simulation of blocking threads in software systems which is the predominant use of concurrent threads.

Franks et al. (Franks and Woodside, 1996) was among the first to apply layered queueing networks to understand and predict the performance improvement of concurrency in client-server architecture. They later applied the same approach to study the performance improvements of an early server reply to client requests (Franks and Woodside, 1999).

Petriu et al. (Petriu et al., 2000) apply layered queueing networks to predict performance of Common Object Request Broker Architecture based middleware systems. Experiments show that the models can predict the effects of service demands, inter-node delays, and message size with relatively low error compared to the actual measurements.

Omari et al. (Omari et al., 2007) extended layered queueing networks to build models of large software system where the topology of the subsystems can be replicated. The researchers showed that it is faster to execute their networks in comparison to other models.

**Stochastic Rendezvous Network**

Woodside et al. (Woodside et al., 1989; Woodside, 1989) introduced a new model called Stochastic Rendezvous Networks (SRVN) to model software performance. The advantage of SRVN over queuing networks is that they enable the modelling of synchronization in more modern programming languages, i.e., the synchronization between threads of the same process using programming primitives such as `synchronized` in Java. SRVN can be translated into Petri nets (Peterson, 1977). Solving Petri net allows modellers to identify performance bottlenecks.

Petriu (Petriu, 1994) proposed the use of a special Markov chain model called Task-Direct Aggregation (TDA) to solve SRVN. TDA gave better results compared to the heuristics that were used in a previous study (Woodside et al., 1989).

**Process Calculi**

Aside from queueing networks, one can also analyze software performance without performance counter data using Process Calculi (Milner, 1982).

Hermanns et al. (Hermanns et al., 1995, 2000; Herzog, 1990) extend the Process Calculi with stochastic elements to create an approach that can model software performance. They create a tool called TIPPtool that can incrementally model the software architecture and identify performance bottlenecks.

Bernado et al. (Bernardo and Gorrieri, 1998; Bernardo et al., 1995, 2000) introduces a Process Calculi based approach called ÆMPA. The approach consists of a new architecture description language that can be used to formally describe a proposed architecture and a tool to analyze the described architecture. The tool, which is called TwoTowers, can be used to analyze both functional and performance characteristics of the architecture using a Markov chain solver (Stewart, 2015).

**Summary**

In summary, simulation, Queueing Networks, Layered Queueing Networks, Stochastic Rendezvous Network, Process Calculi, and other models are commonly used when real performance load tests cannot be done because the software systems are in an early stage of the development cycle or due to the large scale of the studied deployment.

### With Untagged Performance Counter Data

If the software or the prototype of the software are available, one can exercise the actual code under various workloads and produce empirical data to analyze the performance of the software. We are more interested in these types of approaches.

We separate the studies that use empirical data into two categories: with untagged data and with tagged data. This separation is done because the type of analysis which can be done with tagged and untagged data are different.

For untagged data, one does not have the state of the test run or the production status with which to associate the performance counters. Researchers usually employ unsupervised data-mining learners for this type of data. Because of the lack of tagged data, these approaches do not make inferences on the outcome of the analysis (Hastie et al., 2008). They merely output the difference between the compared datasets. For example, if the learners output a difference of $x$ between two datasets. The researchers may decide that there is a performance problem if $x$ is greater than a certain threshold.

The performance studies that we found for this type apply unsupervised learning approaches in two different fashions: counter-to-counter and set-to-set comparisons.

**Counter-to-counter Comparisons** There are approaches that operate on a pair of counters or on the same counter across time periods. These approaches check if there are changes among the counters. We call such approaches "counter-to-counter". When a software system performs normally, the performance counters are assumed to be at a stable state. For example, at the load of ten requests per second, the CPU utilization is 40% and the heap size is 80 MB. When there is a performance problem, the counters would deviate from the stable state. For example, the same load might end up consuming 43% of the CPU and 120 MB of the memory for a new version of the software. One can observe that there is a noticeable difference in the memory consumption by comparing the pair of memory counters across the tests of the two versions.

The simplest approach is probably just comparing the average values of counter pairs (Gabel et al., 2012). For example, one can compare the average value of CPU utilization in the previous baseline test with the new test. If the CPU utilization is the same, then there is no regression. Comparing averages is not advisable. For example, the CPU utilization might fluctuate wildly, yet give the same average. Also comparing average values implies that the distribution of the counters is normal, which is usually not the case (Nguyen et al., 2012). Unfortunately, comparing averages is the most commonly used method in practice as it is the most intuitive.

The following approaches improve over comparing averages:

- A simple approach to improve over comparing averages is using quantiles (Bodik et al., 2010). Instead of comparing the average, the 25thpercentile, the 50th and the 75th percentiles are compared.

- A Wilcoxon Rank-Sum test on sliding windows of raw performance counters is more appropriate (Jiang et al., 2009a). Wilcoxon Rank-Sum is non-parametric test so it does not assume a normal distribution of the performance counters. Wilcoxon Rank-Sum of two samples can determine if the two samples are from two different populations. The performance counter can be separated into windows over time, i.e., period $1$, $2$, $3$, to $n$. Wilcoxon Rank-Sum will then be able to tell whether one period is different from the prior periods. If there is a difference, then there is likely a regression.

- Both Local Outlier Factor and Tukey tests can also be used to alleviate the issues of comparing average (Gabel et al., 2012). Local Outlier Factor test not only considers the actual value of the counter at a given time but also considers the density of the prior and following values of the counter. Turkey Test not only compares the mean values but also the distribution. Experimental results show that the Local Outliers Factor is more accurate in identifying performance problems than using both the Tukey test and performing ad hoc comparison of averages (Gabel et al., 2012).

There exists another type of approaches for comparing performance counters: treating the counters as time series. Performance of software systems often varies in a cyclical fashion. A counter, such as the number of site accesses of a business server would follow the cycles of the work day. The counter would peak during the work hour every day and drop during the off-work hour every day. Resource counters would follow the cycles of the number of site accesses. Time series analysis approaches are often used to study periodic data, e.g., forecasting temperature fluctuation throughout the year. Hence, researchers have used time series analysis approaches to study performance counters as well. The following studies employ such approaches:

- Burgess et al. (Burgess, 2006; Burgess et al., 2002) suggest the use of time series analysis to study the performance of production software systems. They demonstrate that the performance of various software systems on a university's network can be modelled using time series. The researchers show that the performance counters, such as incoming traffic or CPU utilization, exhibit strong daily/weekly periodicity. After modelling the counters using time series, one can determine if there is a performance issue on a production server by checking if the actual performance counters deviate from the expected values based on the models.

- Liang et al. (Liang et al., 2006) use simple correlation of fail-event counters to identify the occurrence of hardware failures in a large parallel processing computer system. The researchers made an observation that if a counter for the fail event increases in one period, an overall system failure follows. They use this observation to identify system failures.

- Yigitbasi et al. (Yigitbasi et al., 2010) also use time series to analyze the failures of large parallel processing computer systems. Similar to Burgess et al. (Burgess, 2006; Burgess et al., 2002), the researchers found that many of the analyzed counters exhibited strong periodicity. They also notice that failures of the studied software systems also exhibit a strong periodicity. They build time series models that can correctly identify the failures for 50-95% of the failures.

- Gainaru et al. (Gainaru et al., 2012a,b, 2013) also conduct a similar study which uses time series analysis of failure events. They first extract the events in the software execution log into counters data. Each counter represents the number of times an event occurred within a particular time window. Then the researchers run auto correlation, which is an approach to find the periodicity of time series, on the counters. If there is a time window where the auto correlation is low, they predict that such a window of time is most likely to contain a failure.

**Set-to-set Comparisons** The counter-to-counter approaches operate on pairs of counters, e.g., the CPU utilization of two test runs or the memory consumption during two periods of execution. However, there are cases where we need to compare two sets of counters together. For example, in performance regression testing, we need to compare the set of counters for the baseline test, i.e., the old version, to the set of counters for the target test, i.e., the new version. If the two sets show differing values or patterns, the system might have a performance regression. We can use counter-to-counter approaches. However, it is not clear whether we have a regression or not if only a small amount of counters are different and the rest of the hundreds of counters are the same. Moreover, in large software systems, there are thousands of performance counters. Some of the counters cannot be paired for comparisons, e.g., CPU utilization on multiple-core hardware or the multiple threads of a subsystem. So researchers have to employ data analysis approaches that can compare between two sets of performance counter data. We call these approaches: "set-to-set" approaches.

One direction to improve over the previous sub-section's counter-to-counter comparisons is the use of intercorrelations. Malik et al. (Malik, 2010; Malik et al., 2010) use the intercorrelations between the performance counters as a similarity measure. For all performance counters, they calculate the intercorrelations among the counters. They use the correlations to create a signature of a performance load test run. If a new test run has a different signature, they can determine that the new test run deviated from the original test run.

PCA, which is commonly used for reducing performance counters (Section 3.2), can also be used for set-to-set comparisons. The membership of the performance counters in the principal components can be used to infer similarity among the counters. If two counters both have relatively high membership level in a component, they are more similar to each other compared to the other counters:

- Eeckhout et al. (Eeckhout et al., 2003a,b) use PCA as a data analysis approach. The researchers use the percentage of variance that is accounted in the first six principal components as a measure of how benchmark workloads differ.

- Zheng et al. (Zheng et al., 2007) employ PCA and Euclidean distance to identify the location (node) of a performance failure in parallel computing systems. They first use PCA to identify the highly-related performance counters of the computing nodes. Then they use the Euclidean distance to detect, among the highly related performance counters, the ones that are outliers during a test run. These outliers are marked as failure nodes for further manual investigations.

Another approach that improves over pairwise comparison is hierarchical clustering. Hierarchical clustering approaches build a root-tree called a dendrogram, where the leaves of a common ancestor are more similar than those of the different ancestors. Hierarchical clustering is used extensively in bioinformatics to build genetic relationships between species or mutations within a species. Hierarchical clustering approaches can be used in the same manner to determine the relationships among the sets of performance counters:

- Eeckhout et al. (Eeckhout et al., 2003a,b) use an Unweighted Pair Group Method with Arithmetic Mean (UPGMA) to analyze the different relationships among benchmarking workloads. Applying UPGMA to the counter data that is obtained from different workloads, the researchers can group the workloads according to the similarity of the software performance.

- Ahn and Vetter (Ahn and Vetter, 2002) similarly use Agglomerative Hierarchical Method to create dendrograms of performance counters of different workloads. Depending on the location of the clusters of counters in the dendrogram, the researchers identify the similarity among the workloads. The researchers further augment the visual dendrograms with the F-ratio of the K-means clustering. Then they also apply PCA to confirm the results of the dendrograms and the F-ratio. The researchers also give a good comparison of different set-to-set comparison approaches.

Researchers also adapt other unsupervised learning approaches that are commonly used in other fields to analyze performance counters:

- Guo et al. (Guo et al., 2006) calculate the Mahalanobis distance of the performance counters at time $n-1$. If the distance at time $n$ is longer than at time $n-1$, and the subsequent distances at time $n+1$ and time $n+2$ are longer than the previous time, then an alarm is raised to indicate a performance problem.

- Stehle et al. (Stehle et al., 2010) employ an algorithm that detects a convex hull, i.e., the smallest set of data points that encapsulate all the data points, to define the stable state of the set of analyzed counters. For example, if there were three counters, the researchers determine six points in the three-dimensional space, which are the minimum and maximum of each counter in the stable state. Then the researchers build a convex hull, which encompasses the six points. Then, whenever a data point falls outside the convex hull, that data point will be marked as problematic.

- Jiang and Munawar et al. (Guofei et al., 2006; Jiang et al., 2009a; Munawar and Ward, 2007) derive a similarity measure called RatioScore from the Jaccard similarity coefficient to determine which component has performance anomalies in runtime. If the RatioScore of a cluster of performance counters from a particular component changes significantly, the metrics in that cluster must have changed relatively to each other. The researchers use the RatioScore index as an indicator of performance regressions.

- Jiang and Munawar et al. further improve the accuracy of their aforementioned approach by inventing another measurement called SigScore (Jiang et al., 2009a). This measure is adapted from PageRank, which takes into account the popularity of the components. RatioScore tends to give popular, i.e., highly connected, components higher score thus producing more false positives on these components. Their evaluation shows that SigScore is more accurate that RatioScore.

### With Tagged Performance Counter Data

There exists tagged data which associates the performance counters with certain states of the test run or production. Thus, researchers can apply supervised learning approaches, which are sometimes called classifiers, to infer the outcome of the analysis (Hastie et al., 2008). For example, one can train a classifier using past performance counter data, such as previous test runs or data from previous days of a system in production, by inputting the data and the associated outcomes, e.g., whether there is a performance regression or not. Then, the trained classifier can be used to predict the outcome of the new test run. Many studies employ this kind of approach.

- Hamerly et al. (Hamerly and Elkan, 2001) proposed approaches to predict disk failures ahead of time using SMART performance counters of the disk. The researchers use an industrial data set that consisted of hard disk performance counters for normal and abnormal cases. The researchers derived two supervised machine learning approaches based on the naive Bayes learning algorithm. Both approaches performed well in identifying normal/abnormal cases with high accuracy.

- Cohen et al. (Cohen et al., 2004, 2005) want to automatically detect problems of production software systems. They use Tree-Augmented Bayesian Networks to build classifier models using performance counters and the known state of the system, e.g., normal or problematic, at certain periods of time. They train the models using past production data. They were able to achieve high accuracy (87%-94%) in identifying performance problems with their approach.

- Zhang et al. (Zhang et al., 2005) later improve on Cohen et al. (Cohen et al., 2004, 2005) by combining the classifier models into ensembles. The ensembles improve the accuracy and significantly decrease the number of false positives compared to the use of a single classifier.

- Bronevetsky et al. (Bronevetsky et al., 2012) seek to improve the accuracy of supervised classifiers by combining them with unsupervised classifiers. A classifier is used to determines whether a test run is good or bad based on training data. The researchers then apply an unsupervised classifier, within each test, on periods of each test runs to identify the probability that each period has a performance regression. Their observation is that the distribution of the problematic periods are different between normal and problematic test runs. Combining the results of the supervised and unsupervised classifiers, the researchers improve their accuracy for identifying problematic test runs.

- Foo et al. (Foo, 2011; Foo et al., 2010) use association rules that are created using historical test runs to determine if a new test run has a performance problem. They first discretize the performance counters, e.g., high, medium, or low. Then, the researchers generate association rules which combine the discretized counters with the test results. For example, if the CPU is medium, the heap size is medium, and the network IO is low, the test is not problematic. If the CPU is medium, the heap size is high, and the network IO is high, the test has problems and is considered as a failed test. Foo et al. build rules using tagged test data and use the rules to determine whether a new test has failures.

Studies with tagged performance counter data have an advantage over studies with untagged data: the ability to automatically determine the outcome of the test. However, tagged data is difficult to obtain and the quality of such data varies. The challenges of using tagged data have been explored by researchers in the mining software repositories field (Godfrey et al., 2009). For example, performance testers often complain that the hardware and workload keep on changing. When testers run a new version's regression test, the corresponding baseline test (with the previous software version) might no longer be valid. They often need to repeat baseline tests on new hardware and with new workloads (Foo et al., 2015). Therefore, leveraging prior tests (with tagged performance counter data) is often challenging even though the data is tagged.

### 2.2.4 Goals

Table 2.1 classifies the studies that are mentioned in this chapter by the goal of the study.

In term of goals, studies can be classified into three types:

- Monitoring: In large software systems, that service user requests year round, operators need to monitor the conditions of the server nodes constantly so operators can react to problems right away. For example, if there is a memory leak, one of the nodes' memory might fill up slowly overtime. When the memory runs out, the software on that node would crash. If a few nodes crash, the clients would reconnect to the remaining nodes. This will, in turn, cause the memory of the remaining nodes to run out faster. The entire system might crash in a matter of minutes from the time since the first node crashes. Thus identifying the problem right away is very important in such a situation. Most of the relevant literature that we found falls into this category.

- Testing: While monitoring allows engineers to detect performance problems in production, performance testing prevents the problems from entering production. To conduct performance load testing, testers create load drivers to create simulated traffic. During the execution of the test, testers can monitor the environment to look for potential performance problems (Jiang et al., 2009c; Malik, 2010; Malik et al., 2010). We can only find a few research papers for this particular goal.

- Benchmarking: The goal of benchmarking is to standardize performance measurements such that the performance can be compared, communicated, and planned effectively (Eeckhout et al., 2003a,b,c). There are many benchmarking suites available such as those from the Standard Performance Evaluation

Table 2.1: Goals of the surveyed studies

| Goals | Description | Studies |
|---|---|---|
| Monitoring | Detect faults as they occur on live software systems | (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Bodik et al., 2010; Bronevetsky et al., 2012; Burgess, 2006; Burgess et al., 2002; Cohen et al., 2004, 2005; Cohen, 1995; Gabel et al., 2012; Gainaru et al., 2012a,b, 2013; Guo et al., 2006; Guofei et al., 2006; Hamerly and Elkan, 2001; Jiang et al., 2009a; Munawar and Ward, 2007; Oliner and Aiken, 2011; Stehle et al., 2010; Vetter and Reed, 1999; Wiertz et al., 2014; Zhang et al., 2005; Zhao et al., 2009) |
| Testing | Analyze performance load test results | (Ahn and Vetter, 2002; Cohen et al., 2004, 2005; Cohen, 1995; Foo, 2011; Foo et al., 2010; Malik, 2010; Malik et al., 2010) |
| Benchmarking | Standardize performance measurements among hardware | (Eeckhout et al., 2003a,b,c) |

Corporation (SPEC) (Giladi and Ahituv, 1995) and Embedded Microprocessor Benchmark Consortium (EEMBC) (Poovey et al., 2009). We found a few studies that deal with finding a suitable benchmark suite, i.e., workload, for comparing different microarchitecture designs (Eeckhout et al., 2003a,b,c).

The difference among the three goals in term of data reduction and data analysis approaches is the input dataset. For monitoring, the input dataset is the performance counters of different periods of time. For example, the counters of the previous hour and the current hour can be compared to determine if there is an arising performance problem. For performance load testing, the input dataset is the performance counters of different test runs with the same workload. For example, the counters of a new software version can be compared with previous versions' test data to uncover whether a performance regression has occurred. For benchmarking, the input dataset is also test runs but with different workloads on different hardware.

There exists little research that focuses on verifying the results of performance load tests of software systems.

### 2.2.5 Evaluation

We classify the evaluation methods of the studies based on two dimensions: the type of systems that are used for evaluation, and real versus synthetic data. The last column of Table 2.2 shows our classifications for the evaluation of the studies.

Type of systems: Researchers use a wide range of software and hardware systems to evaluate their approaches.

- Benchmarking suites (BS) - workload designed to compare hardware systems: SPECint95, SPECint2000, SPECjvm98, SPECint2000, SPECjbb2000, Java Grande Forum, Raja, TPC-D, LI (Eeckhout et al., 2003a,b,c), Dell DVD

Store (Malik, 2010; Malik et al., 2010), IBM Trade Performance Benchmark (Jiang et al., 2009a,b), Pet Store (Cohen et al., 2004; Guo et al., 2006; Zhang et al., 2005), and RUBiS (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014).

- Local scientific computing software (LSC) - scientific software running on a single node (e.g., Cactus and CG-Heat (Vetter and Reed, 1999)).

- Distributed scientific computing clusters (DSC) - scientific software running on multiple nodes: SMG98 (Vetter and Reed, 1999), Blue Gene/L (Bronevetsky et al., 2012; Liang et al., 2006; Oliner and Aiken, 2011), Thunderbird, Spirit, Liberty, Junior, Stanley (Oliner and Aiken, 2011), Planetlab, VCL (Zhao et al., 2009), Sun Wulf (Zheng et al., 2007), Failure Trace Archive (Yigitbasi et al., 2010), sPPM, Sweep3D, and UMT (Ahn and Vetter, 2002)

- Server software (SS) - single node server software such as web servers or mail servers: Unnamed (Malik, 2010; Malik et al., 2010), NanoHTTPD (Stehle et al., 2010), collection of university servers (Burgess, 2006; Burgess et al., 2002), Unnamed, Dell DVD Store (Foo, 2011; Foo et al., 2010), and sample web application (Ghaith et al., 2013, 2014),

- Distributed server software (DSS) - multiple node server software: Microsoft Bing (Gabel et al., 2012), and Exact Online (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014).

- Data centre/Cloud systems (DC) - distributed hardware systems: Virtual Computing Lab, PlanetLab (Zhao et al., 2009), and an unnamed software system (Bodik et al., 2010)

- Hardware (HW) - single hardware systems: Hard drives (Hamerly and Elkan, 2001)

Real versus injected/synthetic data:

- Real: Performance counters data that is collected from production systems with real users and workloads.
- Synthetic: Performance counters data that is collected from systems while they are under synthetic load which is automatically generated by a load driver.

## 2.3 Conclusion

This chapter presents a literature survey of prior research that analyzes performance counter data. We summarize all the studies and their attributes in Table 2.2.

As explained in the problem statement, this thesis aims to introduce a new data reduction approach by using control charts. Our approach is a reduction of data points approach. We also aim to derive approaches to identify, determine the causes, and locate performance regressions in performance test data. In the different studies of this thesis, we will use both OS level and SW level counters data. However, our approach is independent of the counters' layer. As along as the counter can be collected, we can apply our approach. We will also use both tagged and untagged data. We will also use both synthetic data and real data in the studies.

Table 2.2: Summary of studies and their attributes

**Goal:** Mon. - Monitoring, Ben. - Benchmarking, Perf. - Performance load testing **Evaluation:** BS - Benchmarking suites, LSC - Local scientific computing software, DCS - Distributed scientific computing clusters, SS - Server software, DSS - Distributed server software, DC - Data centre/Cloud systems, HW - Hardware

| **Study** | **Data type** | **Data reduction approach** | **Data analysis approach** | **Goal** | **Evaluation** |
|---|---|---|---|---|---|
| Vetter and Reed (1999) | OS | Filter out Counters | | Mon. | LSC & DSC real |
| Eeckhout et al. (2003a,b,c) | HW | Create New Composite Metrics | Set-to-set Comparison | Ben. | BS synthetic |
| Malik (2010); Malik et al. (2010) | SW & OS | Filter out Counters | Set-to-set | Perf. | SS synthetic |
| Oliner and Aiken (2011) | OS | Create New Composite Metrics | | Mon. | DSC real |
| Zhao et al. (2009) | SW & OS | Filter out Counters | | Mon. | DC real |
| Guofei et al. (2006); Jiang et al. (2009a); Munawar and Ward (2007) | SW & OS | | Set-to-set | Mon. | BS synthetic |

| **Study** | **Data type** | **Data reduction approach** | **Data analysis approach** | **Goal** | **Evaluation** |
|---|---|---|---|---|---|
| Guo et al. (2006) | SW | | Set-to-set | Mon. | BS synthetic |
| Stehle et al. (2010) | SW | | Set-to-set | Mon. | SS synthetic & injected |
| Zheng et al. (2007) | OS | | Set-to-set | Mon. | DSC injected |
| Hamerly and Elkan (2001) | HW. | | With Tagged Data | Mon. | HW real |
| Cohen et al. (2004, 2005) | SW & OS | | With Tagged Data | Mon. | BS synthetic |
| Zhang et al. (2005) | SW & OS | | With Tagged Data | Mon. | BS synthetic & injection |
| Bronevetsky et al. (2012) | OS | | With Tagged Data | Mon. | DSC synthetic |
| Bodik et al. (2010) | SW & OS | | Counter-to-counter | Mon. | DC real |
| Gabel et al. (2012) | SW & OS | Filtering based on heuristics | Counter-to-counter | Mon. | DSS real |

| **Study** | **Data type** | **Data reduction approach** | **Data analysis approach** | **Goal** | **Evaluation** |
|---|---|---|---|---|---|
| Burgess (2006); Burgess et al. (2002) | OS | | Counter-to-counter | Mon. | SS real |
| Liang et al. (2006) | SW | | Counter-to-counter | Mon. | DSC real |
| Yigitbasi et al. (2010) | SW | | Counter-to-counter | Mon. | DSC real |
| Ahn and Vetter (2002) | HW | | Set-to-set | Perf. | DSC synthetic |
| Foo (2011); Foo et al. (2010) | SW & OS | Reduction of Data Points | With Tagged Data | Perf. | SS synthetic |
| Bezemer (2014); Bezemer and Zaidman (2010, 2014); Wiertz et al. (2014) | SW & OS | Reduction of Data Points | With Tagged Data | Mon. | BS, DSS real & synthetic |
| Ghaith et al. (2013) | SW & OS | | Without Any Empirical Data | | SS synthetic |

| Study | Data type | Data reduction approach | Data analysis approach | Goal | Evaluation |
|---|---|---|---|---|---|
| Franks and Woodside (1996, 1999) | SW & OS | | Without Any Empirical Data | | |
| Petriu et al. (2000) | SW & OS | | Without Any Empirical Data | | |
| Omari et al. (2007) | SW & OS | | Without Any Empirical Data | | |

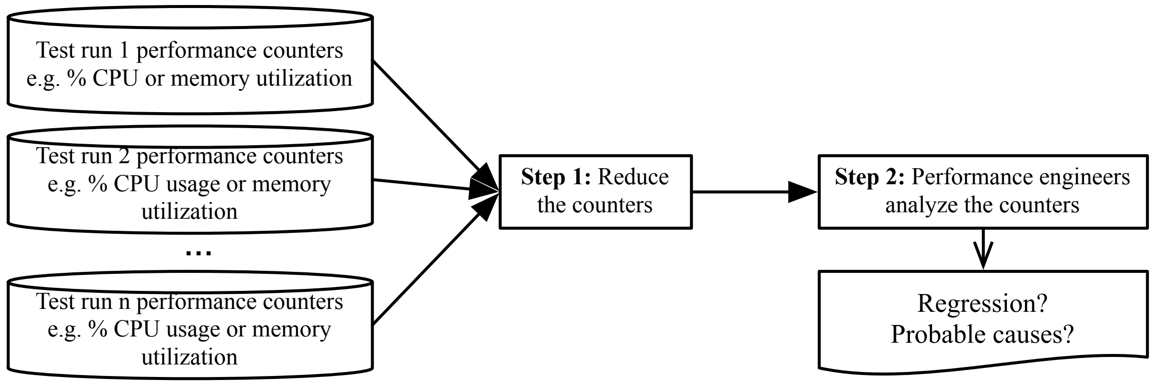
# CHAPTER 3

# Our Control Charts Based Approach

As explained in the prior chapter, studies apply data reduction approaches to deal with large amount of counters data (Section 2.2.2). In this thesis, we propose a new data reduction approach that is based on control charts (Shewhart, 1931). Control charts allow us to reduce the performance counters into a smaller yet understandable data set. Using the reduced data, we propose new approaches to identify and determine causes of performance regressions.

Control charts are a technique in quality control to identify issues of control process (Shewhart, 1931). Control charts are used widely in many fields where quality control is required (Wheeler and Chambers, 2010). For example, control charts are used to monitor quality of care in intensive care units (Abdollahian et al., 2011; Duclos and Voirin, 2010). Control charts are used to enforce the progress of software processes (Komuro, 2006). Control charts are also applied (Amin et al., 2012) to detect quality of service violations in production software system.

Figure 3.1: Steps of performance load test analysis of Figure 1.1


Test run 1 performance counters e.g. % CPU or memory utilization
Test run 2 performance counters e.g. % CPU usage or memory utilization
...
Test run n performance counters e.g. % CPU usage or memory utilization
Step 1: Reduce the counters
Step 2: Performance engineers analyze the counters
Regression? Probable causes?

## 3.1 Steps of Performance Load Test Analysis

As we see in the related work chapter (Chapter 2), there are two main steps in analyzing a performance load test (the Analysis box in Figure 1.1): data reduction and data analysis. Figure 3.1 shows these two steps. The main step is Step 2 where the performance testers analyze the performance counters. However, the amount of counter data is very large. It is important to reduce the amount of counter data in Step 1.

## 3.2 Data Reduction Using Control Charts

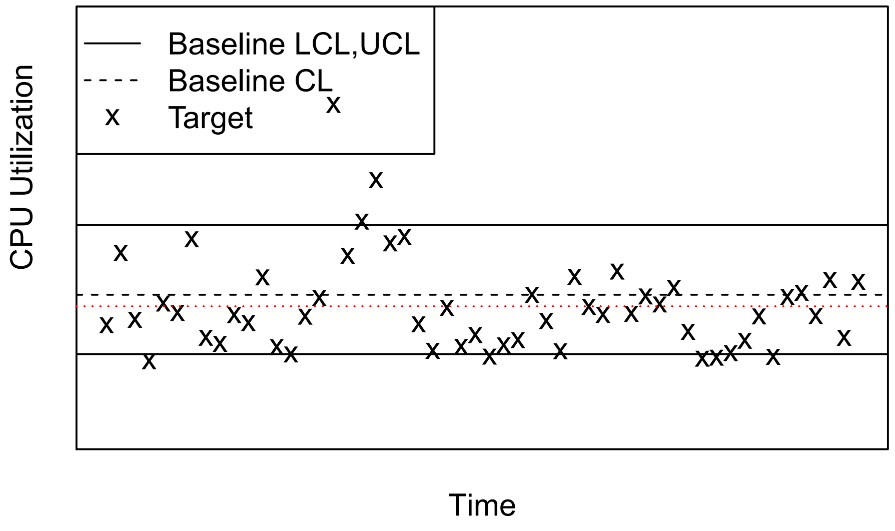
Data reduction is essential when analyzing a very large number of counters as explained in Section 2.2.2. Studies that deal with large counter data employ data reduction approaches such as principal component analysis (Eeckhout et al., 2003a; Malik, 2010), projection pursuit (Vetter and Reed, 1999), or normalized mutual information (Jiang et al., 2009a) for data reduction. However, these approaches either filter or combine the counters. The counters are filtered and combined differently from one pair of tests to another. Hence, it is difficult to understand and verify the results of the analysis. During our meetings with our industrial partners, they indicate that they are more comfortable to adopt approaches that are easy to understand and interpret. Thus, an ideal data reduction approach should preserve the performance counter set.

In this thesis, we propose the use of control charts (Shewhart, 1931) for data reduction. Similar to Foo et al. (Foo, 2011; Foo et al., 2010) and Bezemer et al. (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Wiertz et al., 2014), control charts compress the counters themselves and preserve the counter set. Thus the results of the analyses that are based on control charts would be more intuitive for the testers to adopt in practice. We will compare our control charts based approaches to other approaches in Chapter 9 (p.120).

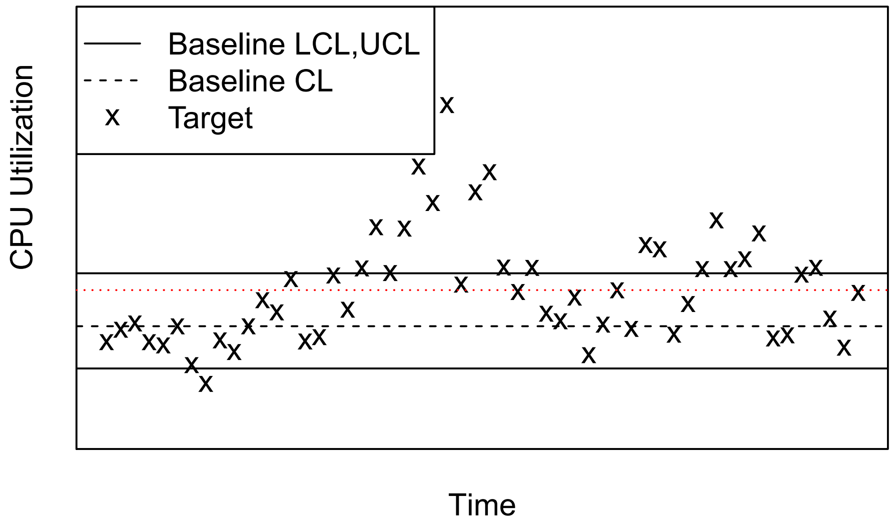
### 3.2.1 Control Charts

Figure 3.2 shows two example control charts. The x-axis is time, e.g., minutes. The y-axis is the measurement data from the metric that we are monitoring about the process. In this example, we are monitoring the CPU utilization of a web server. The two solid lines are the Upper Control Limits (UCL) and Lower Control Limit (LCL). The dashed line in the middle is the Centre Line (CL). Figure 3.2a is an example where the CPU utilization is within its control limits, which should be the normal operation of the web server. Figure 3.2b, on the other hand, is an example where the CPU utilization (process output data) is out-of-control. In this case, further investigation is needed.

A control chart is typically built using two datasets: a baseline dataset and a target dataset. The baseline dataset is used to create the control limits, i.e., LCL, CL, and UCL. In the example of Figure 3.2a and 3.2b, the baseline set would be the CPU utilization of baseline test runs. The CL line is median of all samples in the



(a) Normal



(b) Out-of-control

Figure 3.2: Examples of control charts

baseline dataset at a particular time. The LCL line is the lower limit of the normal behaviour range. The UCL line is the upper limit. There are several alternatives for choosing the LCL and the UCL. A common choice is three standard deviations from the CL (Shewhart, 1931). The target dataset is scored against the control limits of the baseline dataset. In Figure 3.2a and 3.2b, the target data are the crosses, which would be the CPU utilization time of the new target test.

The result of an analysis using control chart is the violation ratio. The violation ratio is the percentage of target dataset values that are outside the control limits. The violation ratio represents the degree to which the current operation is out-of-control. In Figure 3.2a and 3.2b, the violation ratio is the percentage of the crosses that are outside the LCL and UCL lines. This violation ratio is derived from both the baseline and the target counters.

### 3.2.2 Reducing Performance Counter Data Using Control Charts

In a performance load test, we have two sets of data, which are the old version's tests and the new version's test (Figure 1.1). Table 3.1 shows an example of a baseline test of the commercial software system. The raw data of this test has 910 counters with 945 readings each. We only include five counters:

1. **CPU:** CPU utilization of the software during the sampling interval. For example, if there are four cores and each core is busy 80% of the time during the sampling interval, this counter will be 320.
2. **NetRx:** The number of bytes that are received during the sampling interval.
3. **NetTx:** The number of bytes that are transmitted during the sampling interval.

Table 3.1: Example of counter data for a baseline test of a commercial software system

| Time | CPU | NetRx | NetTx | NotHeap | Heap |
|---:|---:|---:|---:|---:|---:|
| 21:31:30 | 327 | 26,646,406 | 48,147,554 | 752,803,824 | 19,492,862,504 |
| 21:31:45 | 345 | 30,263,189 | 55,934,776 | 752,167,104 | 9,816,646,640 |
| 21:32:00 | 331 | 29,515,664 | 53,735,490 | 752,460,032 | 11,351,510,640 |
| ... | | | | | |
| 21:35:45 | 329 | 30,524,559 | 54,766,567 | 748,294,768 | 18,171,733,472 |
| 21:36:00 | 330 | 30,606,069 | 55,102,085 | 748,203,944 | 19,667,034,368 |
| 21:36:15 | 329 | 30,672,832 | 54,884,038 | 748,254,928 | 21,007,219,816 |

Table 3.2: Example of counter data for a target test of a commercial sofware system

| Time | CPU | NetRx | NetTx | NotHeap | Heap |
|---:|---:|---:|---:|---:|---:|
| 15:49:15 | 354 | 28,450,637 | 80,352,648 | 721,905,056 | 20,973,365,904 |
| 15:49:30 | 363 | 26,918,811 | 67,882,064 | 723,436,088 | 18,827,309,040 |
| 15:49:45 | 373 | 29,085,929 | 77,025,467 | 724,739,384 | 13,710,490,664 |
| ... | | | | | |
| 15:53:30 | 365 | 30,248,695 | 78,345,863 | 724,248,104 | 21,539,781,616 |
| 15:53:45 | 359 | 30,495,700 | 81,110,628 | 724,408,272 | 21,987,294,576 |
| 15:54:00 | 361 | 30,018,038 | 80,474,501 | 724,099,520 | 21,823,276,392 |

4. **NotHeap:** The amount of memory that is currently used that is not heap space.

5. **Heap:** The amount of memory that is currently used that is heap space.

Table 3.2 shows an example of a target test. This target test was identified by performance testers of this software system to contain a performance regression. The CPU utilization in the baseline test is about 347% to 345% (CPU column on Table 3.1). The CPU utilization in the target test is about 354% to 373% (CPU column on Table 3.2).

We use control charts to reduce the counters in the two data sets into one violation ratio for each counter pair. Table 3.3 shows an example of the data reduction results for the baseline and target tests of Table 3.1 and Table 3.2. For each test

Table 3.3: Example of violation ratios (Vio.) computed from the baseline run of Table 3.1 and the target run of Table 3.2

| Counter | Vio. Lower | Vio. Upper | Vio. Average | Vio. Sum |
|---:|---:|---:|---:|---:|
| CPU | 0.00 | 95.74 | 47.87 | 95.74 |
| NetRx | 14.89 | 0.00 | 7.45 | 14.89 |
| NetTx | 0.00 | 100.00 | 50.00 | 100.00 |
| NotHeap | 100.00 | 0.00 | 50.00 | 100.00 |
| Heap | 0.00 | 0.00 | 0.00 | 0.00 |

pair, instead of having two sets of $n$ counters for the baseline and the new test, each with thousands of samples, we now have only $n$ violation ratios representing the target test run and its baseline.

Whether a counter is higher or lower in the target test compared to the baseline test is important in software performance. So, instead of using just the violation ratio as defined in Section 3.2.1, we define the following four derivations of the violation ratios to better suit our purpose:

1. **Lower violation ratio:** The percentage of data points in the target test's counter that is lower than the LCL.

2. **Upper violation ratio:** The percentage of data points in the target test's counter that is higher than the UCL.

3. **Average violation ratio:** The average of the lower and upper violation ratio.

4. **Sum violation ratio:** The sum of the lower and upper violation ratio.

We can see from Table 3.3 evidence of the performance regression in this pair of tests. The violation ratio for the CPU is high. As mentioned, the CPU utilization in the baseline test is about 347% to 345% (CPU column on Table 3.1). The CPU utilization in the target test is about 354% to 373% (CPU column on Table 3.2).

We investigate the benefits of using control charts instead of other data reduction approaches in Chapter 9.

# CHAPTER 4

# Evaluation and Studies

As explained in the problem statement, we want to demonstrate that control charts can be used to identify and determine causes of performance regressions in performance load test. We conduct four studies. In each study, we use control charts to reduce the performance counter data of the tests. Then we derive approaches to identify (Chapter 6, p.73 and Chapter 7, p.82), and determine causes (Chapter 8, p.96) of performance regressions. Then we apply our control charts based approaches in an industrial setting and report the experience (Chapter 9, p.120).

The main goal of our approaches is to help performance testers save time and effort in comparison to manually analyzing performance counters. Since our approaches can be automated, performance testers can potentially add them into the automated test execution processes, which are common in practice (Hewlett Packard, 2010; Mozilla Project, 2010).

## 4.1 Studied Systems

We evaluate the accuracy of our approaches using real-world large-scale commercial and open source software systems.

- **Commercial 1:** Our first system is a large scale software system developed by our industrial partner. The software is developed in a fast iterative process. Performance engineers have to perform load tests at the end of each development iteration to determine if the software can be released. The software has a multi-tier server client architecture. A typical load test exercises load on multiple components which reside on different servers. The behaviour of the components and the hardware servers is recorded during each test run.

- **Commercial 2:** The second system is another large scale software system which is also developed and operated by our industrial partner. The software system is deployed on thousands of servers with millions of concurrent users. Because of the high number of users and the high level of load, the performance of the system is very important to the business. A single digit percentage regression would mean hundreds of thousand dollar increase in terms of yearly operating budget. Thus, there is a rigorous performance load testing mandate for all the releases. After each development iteration, testers always conduct performance regression tests to determine if the new version has any performance regressions.

- **DS:** Our third system is the Dell DVD Store version 2. It is a simple e-commerce web application which is used by Dell to benchmark their server hardware systems. The software has web pages that allow users to search, view, and order movies. DS is an open-source software that has been previously used in several prior software performance studies (Foo, 2011; Foo et al., 2010; Jiang et al., 2008).

For each study, we either use our own lab to conduct tests or use existing data from the projects' performance testers. We will explain in more details the type of tests and performance regressions that we use in the next section (Section 4.2).

## 4.2 Evaluation Methodology

We conduct test runs or collect test runs from all three software systems described in the previous section. There are normal test runs where the system performed as required. There are test runs with performance regressions. After the test runs, we collect the raw metrics from the software monitoring systems.

It is challenging to use raw performance counters as-is for automated analysis. For the three software systems that we study, we identify three main challenges: varying input load, multimodal distribution of the counters, and always changing performance counters. We derive approaches to deal with the three challenges. We discuss in detail about those approaches in Appendix 5 (p.56). Before we use the counter data from the three software systems in our studies, we apply these approaches to correct the counter data when applicable.

We, then, perform the data reduction steps as described in Section 3.2 to compute the violation ratios. Then we apply our approaches to identify and determine the probable cause of the performance regressions. We measure the effectiveness of the approaches by measuring the accuracy of the approaches' predictions.

### 4.2.1 Introducing Performance Regressions

Depending on the system under-test, we employ the following three methods to introduce performance regressions: force resource contentions, inject performance regressions, or use real performance regressions.

### Force Resource Contentions

Since we cannot alter the code of the Commercial 1 system, we simulate the problem by starting a separate process with real-time priority that uses about 50% and 75% of the CPU time. This process will compete with the actual service processes for CPU cycles. This process effectively forces the software system under-test to perform worst than the normal runs (due to resource contention).

### Inject Performance Regressions

For other systems under-test, where we can alter the code, we introduce performance regressions into the software by injecting code or changing configurations which will lead to performance regressions. This method is more realistic as it is how performance regressions are usually introduced in the software development cycle.

The following are examples of changes that frequently lead to performance regressions:

- **R1 - Adding fields in long living objects**: Causes an increase in memory utilization. If a field is added to a class, and that class's objects are created many times by the software, even though the additional memory footprint of a field might be small, the multiple instantiations of the class can lead to a large increase in memory utilization (Gunther, 2000).
- **R2 - Adding code in a hot code path**: Causes an increase in CPU utilization. Even a small set of additional calculations added to a hot code path can lead to a large increase in CPU utilization (Malik et al., 2013).

- **R3 - Adding a DB query in a hot code path**: Causes an increase in DB requests. In large software systems, each database can connect to hundreds of nodes. Each node processes thousands of requests per minute. If a request performs one additional database query than before, the number of querying requests to the database would increase sharply (Chen et al., 2014).

- **R4 - Missing a DB column index**: Causes longer DB requests. Necessary indices are required for frequently-executed queries. However, the necessity depends on the actual load. Hence this kind of regression only emerge under a performance load test (Chen et al., 2014).

- **R5 - Missing a DB text indexes**: Causes longer DB requests. Similar to (4), text indices are required if text searches are performed on a column of the DB. If such an index is missing, the query would take a longer time and use more CPU cycles (Chen et al., 2014).

- **R6 - Missing query limit**: Causes longer DB requests. The good practice is to limit the number of rows to be returned by the DB to what would be consumed by the front-end, e.g., to be displayed. The bad practice is to fetch all the rows then filter on the front-end. This results in unnecessary data rows to be fetched from the DB (Chen et al., 2014).

- **R7 - Not reusing DB connections**: Causes excessive DB reconnections. The good practice is to reuse database connections whenever possible (Gunther, 2000). Queries to the same database should reuse the same connection as the first one. However, developers sometime create new connections for every query. This results in unnecessary opening and closing of DB connections which is often expensive in term of CPU and memory utilization.

- **R8 - Adding log statements**: Causes heavier I/O. Adding unnecessary log lines is a common mistake (Gunther, 2000). Log lines are usually required when implementing a new feature. There is a tendency to leave the logging statements behind in the source code when the software is finished. Unfortunately, if the log lines are part of a hot code path, there will be much more log printing in the field, which can lead to a drastic increase in I/O (Kabinna, 2016).

**Use Real Performance Regressions**

In many of the studies, we also use performance regressions that were found by the performance engineering teams of the Commercial 1 and Commercial 2 software systems. We determine the appropriate test runs and download the counters data from the team's performance test repository. In some cases, we also rerun the tests with the same software versions that contain the performance regressions in our lab. We rerun if we need to recollect the counter data or if we need to introduce new performane regressions.

### 4.2.2 Evaluation Measurements

For studies with tagged performance counter data, we use two measurements of accuracy: overall accuracy and gain (of Dayton Research Institute, 1963).

The overall *accuracy* is defined in Equation (4.1):

$$Accuracy = \frac{|Success|}{N} \tag{4.1}$$

In 4.1, $Success$ is the set of test runs when the prediction of the approach is the same as the actual type. $N$ is the total number of runs.

While the overall accuracy can indicate the success of the approaches, it does not take into account the number of different possible outcomes. If there are two possible outcomes, an accuracy of 50% would be the same as a random choice. Such approach is not useful at all. If there are four possible outcomes, 50% accuracy is twice better than a random choice. Such approach would be somewhat useful since they reduce the cost of inspecting all the possible outcomes manually by half. If there are eight possible outcomes, the 50% accuracy would be more useful since it is four times better than a random choice. The *gain* measurement captures this notion of usefulness. It is the ratio of gained accuracy (of Dayton Research Institute, 1963) over the accuracy of a random predictor as defined in Equation (4.2):

$$Gain = \frac{Accuracy}{r} \tag{4.2}$$

In the above equation, $r$ is the accuracy of the random predictor, which is defined in the following Equation (4.3):

$$r = 100 * \sum_{i=1}^{|Types|} \left(\frac{|Runs\,per\,type_i|}{N}\right)^2 \tag{4.3}$$

We will also use the measurement of Precision and Recall in some studies using the following formulas by comparing against the correct classification of a test:

$$Precision = \frac{|classified\ bad\ runs \cap actual\ bad\ runs|}{|classified\ bad\ runs|} \tag{4.4}$$

$$Recall = \frac{|classified\ bad\ runs \cap actual\ bad\ runs|}{|actual\ bad\ runs|} \tag{4.5}$$

Table 4.1: Summary of the studies in this thesis

| **Study** | **Studied system** | **Evaluation** | **Data type** |
|---|---|---|---|
| **Part I:** Identifying Performance Regressions Using Control Charts | | | |
| Study 1: Identifying Injected Performance Regressions | Commercial 1, DS | Injected code changes and resource contention | Untagged |
| Study 2: Identifying Regressions in Industrial Performance Tests | Commercial 1, Commercial 2 | Real performance regressions | Tagged & untagged |
| **Part II:** Determining Root Causes Using Control Charts | | | |
| Study 3: Using Control Charts to Determine Causes of Performance Regressions | Commercial 2, DS | Injected performance regressions | Tagged |
| **Part III:** Applying Control Charts in Practice | | | |
| Study 4: Applying our control charts based approaches into commercial software projects | Commercial 2 | Real performance regressions | Tagged & untagged |

## 4.3 Studies

Table 4.1 lists all the studies that we conduct in the thesis. As stated in the problem statement (Chapter 1, p.2), we want to derive approaches to answer three questions:

- **Identify - Is there a performance regression?** Study 1 and 2.
- **Determine cause - What is the root-cause of a detected performance regression?** Study 3.
- **Application - How well does our approach perform in practice?** Study 4.

In Study 1 (Chapter 6 p.73) and Study 2 (Chapter 7, p.82), we want to determine if "control charts" is a good approach to identify performance regressions. For our first and preliminarily study, we first take the software from Commercial 1 and conduct test runs which force resource contentions (Section 4.2.1, p.50). Then we take the DS software and inject regressions into the code (Section 4.2.1, p.50). We conduct the test runs with the original software and the modified software. In our first study, we do not have any real performance regression data. Instead, we force resource contentions for the Commercial 1 software system and inject regressions in the DS software system. In Study 2 and 3, we leverage an industrial test repository and apply control charts on real performance regressions. The main contribution of Study 1 and Study 2 is to demonstrate that control charts can be used to identify performance regressions in performance load tests.

In Study 3 (Chapter 8, p.96), after we establish that control charts can be used to identify performance regressions, we leverage control charts to address other tasks with which help performance teams need assistance. Identifying if there is a performance regression is only the first step. Once a performance regression is found, the performance testers need to find out what causes the regression. In Study 3 (Chapter 8, p.96), we propose an approach that applies control charts and machine learning algorithms to automatically identify the cause of a performance regression using historical test data.

In Study 4 (Chapter 9, p.120), we apply our control charts based approach to the testing of a commercial software system. We report the results and the feedback.

# CHAPTER 5

# Dealing with Data

## 5.1 Introduction

Throughout our studies, we found that dealing with performance regression test data is challenging. In this chapter, we outline the solutions that we used to deal with the three challenges that we found. In the next three sections, we describe in detail each of the proposed solutions and evaluate their effectiveness:

- **Varying Input (Section 5.2):** During a performance test, performance testers use load generators to simulate inputs from thousands of users. It maybe beneficial to apply input load that fluctuates as it is usually the case in production. However, varying the input load hinder identifying performance regressions. This is the case for Commercial 1 software systems. We apply the solution described here on all case studies that use the Commercial 1 software system in this thesis.

- **Multimodal Counter Distribution (Section 5.3):** There may be multiple types of input load during a performance load test. For example, the users can search a catalog and order items from an e-commence website. In such case, the resulting performance counters might have a multimodal distribution: one for each type. We suggest a solution that removes the undesired part of the distribution. We also apply this solution on all case studies that use the Commercial 1 software system.

- **Load independent counters (Section 5.4):** During a test runs, most performance counters would depend to the input load. For example, when there are many users searching catalogs, the CPU utilization increases. However, there may be performance counters that do not depend on the input load. For example, the amount of cached results would stay the same as there is a fixed limit. We suggest a solution that identifies and removes those counters. We apply this solution on all case studies that use the Commercial 2 software system in this thesis.

## 5.2 Varying Input

In performance regression testing, the same load is applied to both the baseline and target version. In the case of the Commercial 1 software system, the load driver would use a randomizer to create fluctuating load during each test. For example if the load profile specifies a rate of 5,000 requests per hour, the load generator will aim for 5,000 requests per hour in total using a randomizer. The randomization of the load is essential to trigger possible race conditions and to ensure a realistic test run. However, the randomization leads to varying inputs throughout the different time periods of a test run. The impact of randomization on

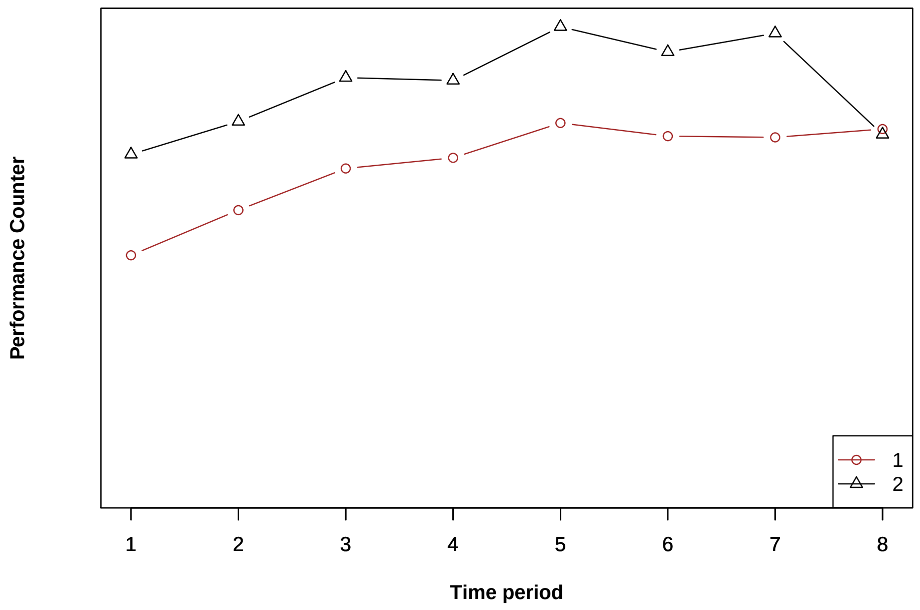


Figure 5.1: The performance counters of two test runs. The difference in performance is not a performance regression since both runs are of the same version. The difference (20% average) is due to differences in the actual load.

the input load and the output counters increases as a system becomes more complex with many subcomponents having their own processing threads and timers. Even in the DS system, which is a relatively small and simple system, the load driver employs a randomizer. This randomizer makes it impossible to get two similar test runs with the same effective load input.

If the input load are different between runs, it is difficult to identify performance regressions since the difference in the counters can be due to the variations of the input load instead of performance related changes in the code. Figure 5.1 shows a performance counter of two different runs coming from two successive versions of Commercial 1 (*see* Section 4.1 for more details). We divide the runs into eight equal periods of time (x-axis). The y-axis shows the median of the performance counter during that period. The red line with round points is the baseline, which is from an earlier build. The black line with triangular points is the target, which is from the new build. According to documentation, these two runs are of the same version. Yet, it turns out that they are different because of the variation in the effective input load due to randomization of the load. The smallest difference is 2% in the eighth period. The highest difference is 30% in the first period. On average, the difference is about 20%. The actual load inputs are about 14% different between the two runs.

### 5.2.1 Proposed Solution

Our proposal is to scale the performance counter according to the load. Under normal load and for well designed systems, we can expect that performance counters are proportionally linear to the load intensity. The higher the load, the higher the performance counters are. Thus, the relationship between counter values and the input load can be described with the following linear equation:

$$c = \alpha * l + \beta \tag{5.1}$$

In this equation, $c$ is the average value of the performance counter samples in a particular period of a run. $l$ is the actual input load in that period. $\alpha$ and $\beta$ are the coefficients of the linear model.

To minimize the effect of load differences on the counters, we derive the following solution to scale each counter of the target run:

- We collect the counter samples ($c_b$) and corresponding loads ($l_b$) in the baseline runs. For example, at the third minute the CPU utilization is 30% when the load is 4,000 requests per minute. In the next minute, the load increases to 4,100 so the CPU utilization increases to 32%.

- We determine $\alpha$ and $\beta$ by fitting counter samples, e.g. the CPU utilization, and the corresponding load, e.g. the number of requests per second, into the linear model: $c_b = \alpha * l_b + \beta$ as in (5.1). The baseline runs usually have thousands of samples, which is enough to fit the linear model.
- Using the fitted $\alpha$ and $\beta$, we can then scale the corresponding counter of the target run ($c_t$) using the corresponding load value ($l_t$) as in (5.2).

$$c_t = c_b * \frac{\alpha * l_t + \beta}{\alpha * l_b + \beta} \tag{5.2}$$

### 5.2.2 Evaluation

**Evaluation Approach.** The accuracy of the scaling solution depends on the accuracy of the linear model in Equation (5.1). To evaluate the accuracy of the linear model, we use the test runs of Commercial 1.

We run a set of controlled test runs to build the linear model as in Equation (5.1). The control runs use the same stable version. We pick a different target load for each test. For example, if the first three runs have actual loads of 1,000, 1,200, and 1,000, we will aim for a load of 1,200 for the fourth run. This choice ensures that we have enough data samples for each load level, i.e., two runs with 1,000 and two runs with 1,200 in this example.

For each test run, we extract the total amount of load $l$ and the mean performance counter $c$ for each period of the runs. Then we train a linear model to determine $\alpha$ and $\beta$ using part of the data. Then we can test the accuracy of the model using the rest of the data. This evaluation approach is used commonly to evaluate linear models. A common ratio for train and testing data is 2:1. We randomly sample two-thirds of the periods to train the linear model. Then we use the

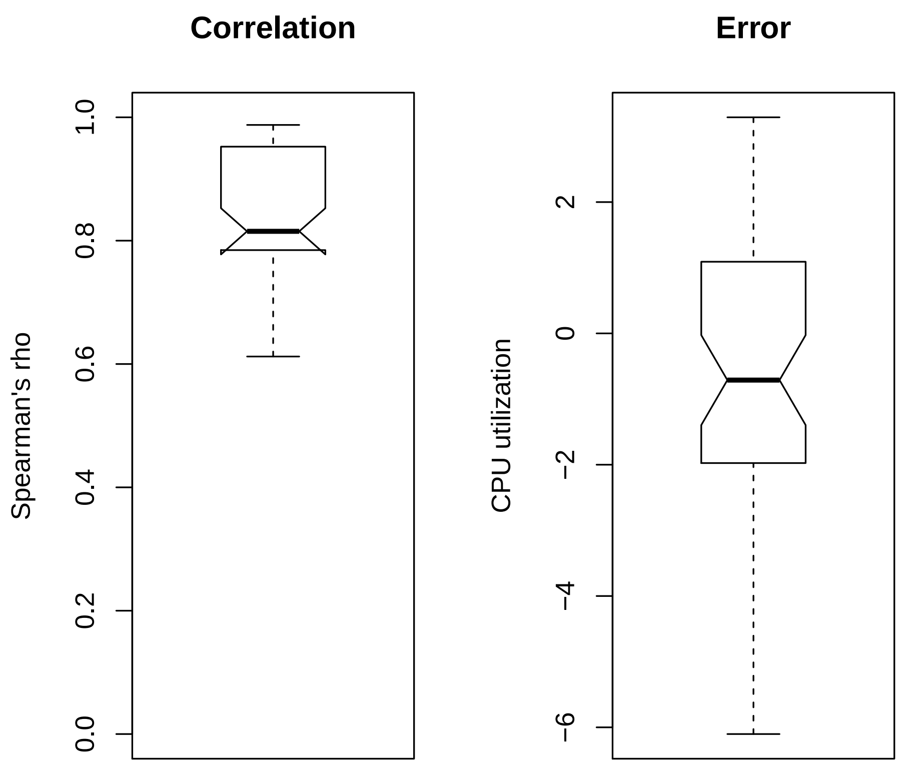


Figure 5.2: The accuracy of the linear model in Equation (5.1). Left: The Spearman correlations between the predicted and actual performance counter values of 50 random splits. Right: The differences between the predict and actual value in relative percentage. 0% means no difference.

remaining one-third to test the model. We repeat this process 50 times to eliminate any possible sampling bias.

**Results.** Figure 5.2 shows the result of our evaluation. The graph on the left is the box plot of the Spearman correlation between the predicted and the actual counter values. If the Spearman correlation nears zero, the model is a bad one. If the Spearman correlation nears one, then the model fits well. As we can see, all 50

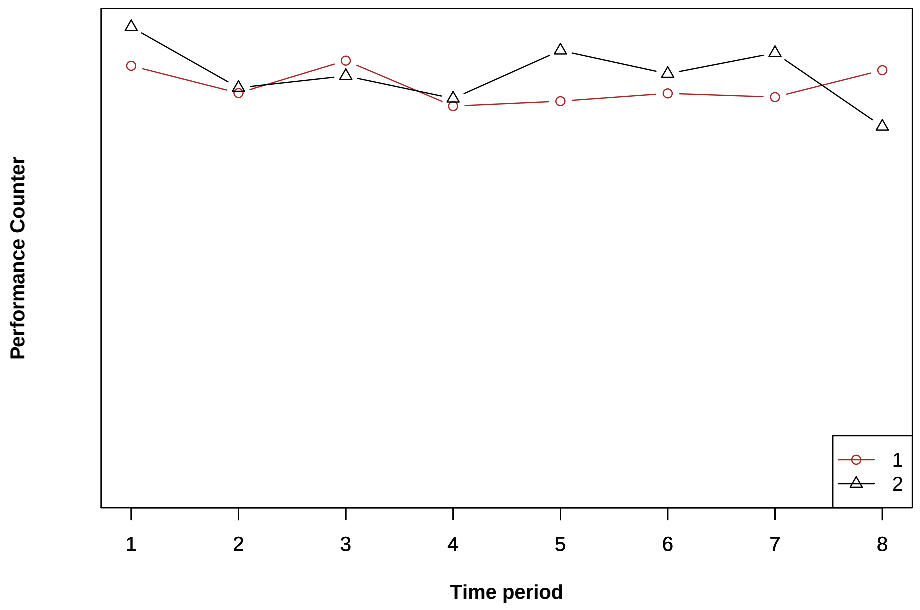


Figure 5.3: The scaled performance counters of two test runs in Figure 5.1. We scale each performance counter according to the load to minimize the effect of load differences between test runs. The average difference between the two runs is only 9% as compared to 20% before scaling.

random splits yield very high correlations. The graph on the right is the box plot of the mean error between the predicted and the actual counter values. As we can see, the errors, which are less than 2% in most cases, are very small. These results show that the linear model used to scale the performance counters based on the actual load is very accurate.

**Example.** Figure 5.3 shows the performance counters of the two runs in Figure 5.1 after our scaling. The counters of the two tests are now very similar. As we can see, after scaling, the target runs fluctuate closer to the baseline test. The difference between the two runs is about 9% (between 2% to 15%) after scaling compared to 20% (between 2% to 30%) without scaling (Figure 5.1).

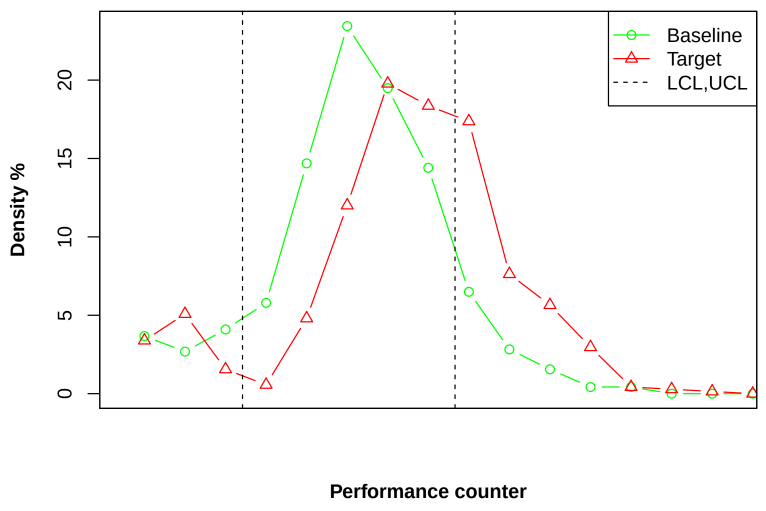
## 5.3 Multimodal Counter Distribution

Process output is usually proportional to the input. So the distribution of the counter samples should be a uni-modal normal distribution (unless the load is pushed to the maximum, as in stress testing, the counter distribution will skew to the right).

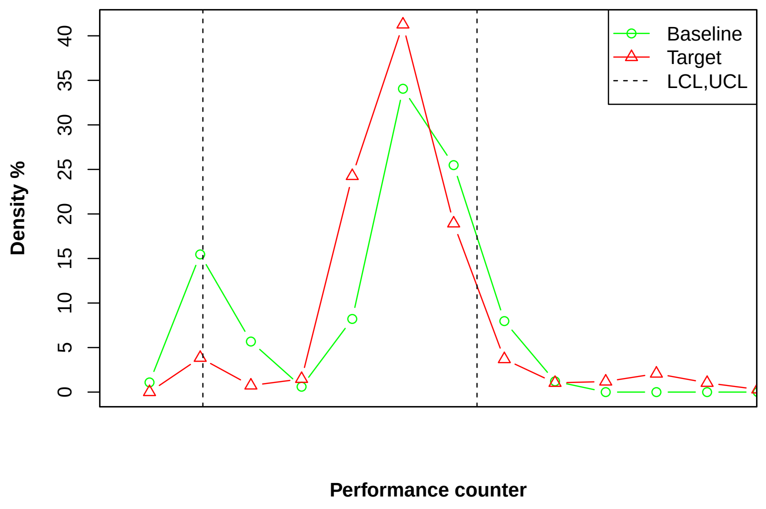
However, the assumption here is that there is only one kind of effective load input. If there are many kinds of input, the distribution of counters would be a multi-modal distribution. The Commercial 1 software system is very susceptible to this problem. Because the input load varies during the test, sometimes only one kind of input is being exercised. Sometimes, the other kind of input is being exercised. The Commercial 2 software system is not susceptible to this problem because its input load stays constant during the test runs.

Figure 5.5a shows the normal QQ plot of a performance counter of Commercial 1. If the counter is uni-modal, its samples should form a diagonal straight line. As we can see, the upper part of the data fluctuates around a diagonal straight line. However, the lower end does not. Unfortunately, the data points at the lower end are not outliers; they appear in all test runs of the Commercial 1 software system. As such, the overall distribution is not uni-modal. A Shapiro-Wilk normality test on the data confirms that with $p < 0.05$.

Figure 5.4a shows a density plot of a performance counter of two test runs of the same software under test. These two runs come from two successive versions. The green line with round points is the baseline run, which is based on an older version. The red line with triangles is the target, which is from a new version. As we can see in the graph, the distribution of the performance counters resembles a normal distribution. However, the left tail of the distribution always stays up instead of decreasing to zero.



(a) Density plot of two test runs.



(b) Density plot of another two different test runs. These two runs are on a better hardware system so the left peak is much higher than the two runs in Figure 5.4a.

Figure 5.4: Example of multi-modal performance counters

When the system is under a load input, the performance counters respond to the load input in a normal distribution. However, in between periods of high load, the performance counter is zero since there is no load input. The first point on the density plot for both runs is about 4%. Hence, the performance counter is at zero for about 4% of the time during both test runs. The target run spent 5% of its time at semi-idle state (the second point from the left on the red curve). When there is no load, the system would usually perform bookkeeping tasks. These tasks require only small amounts of resources. We can consider these tasks as a different kind of load input. These secondary tasks create a second peak in the distribution curve, which explains the long tail in the QQ plot of Figure 5.5a.

Unfortunately, this is a common behaviour. For example, on a web server, a web page would contain images. When a web page is requested by the client, the web server has to process and serve the page itself and its associated images. The CPU cycles required to process an image are almost minimal. Web pages, on the other hand, require much more processing since there might be server side scripts on them. Thus the distribution of the CPU utilization would be a bi-modal distribution. The main peak would correspond to processing the web pages. The secondary peak would correspond to processing the images.

In the Commercial 1's software system, only 16% of the studied test runs are uni-modal. We confirm that these runs are, in fact, normal as confirmed by Shapiro-Wilk tests ($p > 0.05$). The majority of the runs, which is about 84%, have a bi-modal distribution similar to that of the target run in Figure 5.4a. In the bi-modal runs, the left peak corresponds to the idle-time spent on book-keeping tasks. The right peak corresponds to the actual task of handling the load. The relative scale of the two peaks depends on the capacity of the hardware. The more capable the hardware, the more time the system idles, i.e., the left peak will be higher. The less capable the hardware, the less time the system has to idle because it has to use more resource to process the load. The right peak will be higher and the left peak might not be there. The two runs shown in Figure 5.4a are on relatively standard equipment. Figure 5.4b shows another two test runs that are performed on more capable hardware configurations. As we can see, the left peaks in the density plots are more prominent in the runs with more capable hardware.

### 5.3.1 Proposed Solution

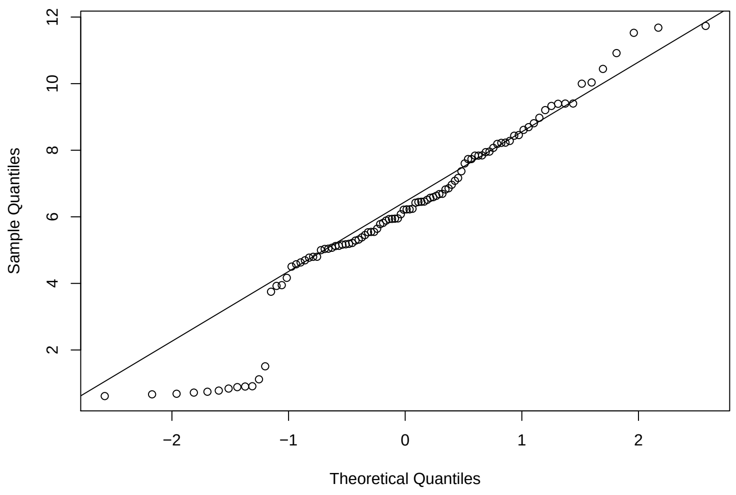
Our proposed solution is to filter out the counters' samples that correspond to the secondary task. To implement the filtering solution, we derive a simple algorithm to detect the local minima, i.e., the lowest point between the two peaks of a bi-modal distribution. Then, we simply remove the data points on the left of the local minima. The algorithm is similar to a high pass filter in an audio system. For example, after the filtering is applied, the first three points on the target run's density plot in Figure 5.4a (the red line with triangles) would become zeros.

An alternative solution is to increase the load as the server hardware becomes more capable. The increased load will make sure that the system spends less time idling and more time processing the load, thus, removing the left peak. However, artificially increasing the load for the sake of normality defeats the purpose of performance regression testing and compromises our ability to compare new tests to old tests.

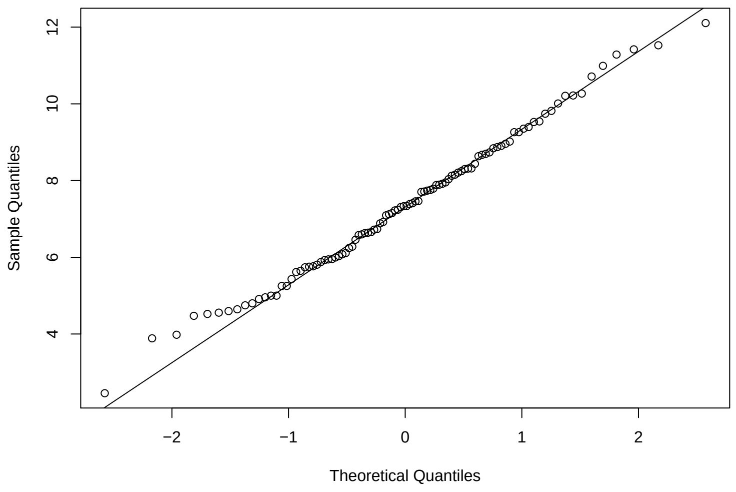
### 5.3.2 Evaluation

**Evaluation Approach.** To evaluate the effectiveness of our filtering solution, we pick three major versions of Commercial 1 with which we are most familiar. For each run of the three versions, we generate a QQ plot and run the Shapiro-Wilk normality test to determine if the runs' performance counters are normal. Then, we apply our filtering solution and check for normality again.

**Results.** We first manually inspect the density plots of each run in the three versions. About 88% of the runs have a bi-modal distribution. About 66% do not



(a) Normal QQ plot of CPU utilization counters. We show only 100 random samples of CPU utilization from the run to improve readability.



(b) Normal QQ plot of the performance counters after our filtering process. The data is the same as in the QQ plot of Figure 5.5a. After we remove the data that corresponds to the idle-time tasks, the distribution becomes normal.

Figure 5.5: Example of our solution to normalize performance counters

have a normal distribution, i.e., the Shapiro-Wilk tests on these runs return $p < 0.05$. If the left peak is small enough, it will pass the normality test. After filtering, 91% of the non-normal runs become normal. We believe the result demonstrates the effectiveness of our filtering.

**Example.** Figure 5.5b shows the QQ plot of the same data as in Figure 5.5a after our filtering solution. As we can see, the data points are clustered more around the diagonal line. This means that the distribution is more normal. We perform the Shapiro-Wilk normality test on the data to confirm. The test confirms that the data is normal ($p > 0.05$). We can now use the counter data to build a control chart.

## 5.4 Load independent counters

The goal of performance regression analysis is to detect the changes in the performance counters, which are collected using various tools during test runs, to identify performance regressions. However, one will find that some performance counters are not related to the load, i.e., they will change regardless of the inputs to the system. For example, the amount of cache used for disk IO would change periodically. Counters that are related to other software that are not related to the one being under load would also change. Other examples are pruning processes. Pruning processes are invoked once in a while to re-index the system internal data storage. The performance of such processes depends on a timer instead of the system's input load.

These load independent counters are likely to introduce false positives in our analyses. This is true even if we do not use control charts to compress the counter data. We need to find a solution to remove such counters from the pool of to-be-analyzed counters.

### 5.4.1 Proposed Solution

Our solution is to identify those counters using two normal tests. We first run the performance test with one version of the software that does not have known performance regression. We run the same test again. Afterward, we build control charts for each counter. If there are counters with high violation ratios (even when there is no change in the software), these counters would be the ones that are likely to yield false positives.

### 5.4.2 Evaluation

Both the Commercial 1 and Commercial 2 software systems are susceptible to the problem of load independent counters. To evaluate our approach, we apply this data treatment on an analysis of the Commercial 1 software system.

The goal of our analysis is to identify the software process that exhibits regression behaviour. Similar to Study 1 (*see* Section 6.2), we conduct test runs of the Commercial 1 software system. During the test runs, we apply forced resource contentions to the front end and a back end process. We create the following pair of tests:

- **Normal:** The baseline is the Normal run. The target is another normal run. We expect to see no regression in this pair.
- **P1a:** The baseline is the Normal run. The target is the run with 50% forced CPU contention in the front end (the P1a run). We expect to see a performance regression in the front end's counters in this pair.
- **P1b:** The baseline is also the Normal run. The target is the run with 75% forced CPU contention in the front end (the P1b run). We expect to see a performance regression in the front end's counters in this pair.

Table 5.1: The ten most out-of-control counters in Commercial 1's test runs

| **Norm.** | **Vio%** | **P1a** | **Vio%** | **P2a** | **Vio%** | **P1b** | **Vio%** | **P2b** | **Vio%** |
|---|---|---|---|---|---|---|---|---|---|
| FE* | 98% | FE | 100% | FE* | 100% | FE | 100% | FE* | 95% |
| FE* | 83% | FE | 98% | BE | 98% | FE | 100% | BE | 93% |
| oBE* | 58% | FE* | 93% | BE | 98% | FE | 100% | BE | 93% |
| FE* | 23% | FE | 78% | BE | 98% | FE | 100% | BE | 93% |
| BE | 8% | FE | 75% | BE | 98% | FE | 100% | BE | 91% |
| BE | 8% | FE | 75% | FE* | 81% | FE | 100% | FE* | 83% |
| BE | 6% | FE | 73% | oBE* | 70% | FE | 100% | oBE* | 53% |
| BE | 6% | oBE* | 61% | BE | 25% | FE* | 98% | FE* | 41% |
| BE | 5% | oBE | 46% | FE* | 25% | FE* | 71% | FE | 20% |
| BE | 5% | FE* | 43% | oBE | 15% | FE* | 66% | oBE | 10% |

FE - front end, BE - back end, oBE - the other back end
* - counters that are identified as load independent counters

- **P2a:** The baseline is also the Normal run. The target is the run with 50% forced CPU contention in the back end (the P2a run). We expect to see a performance regression in the back end's counters in this pair.

- **P2b:** The baseline is also the Normal run. The target is the run with 75% forced CPU contention in the back end (the P2b run). We expect to see a performance regression in the back end's counters in this pair.

Table 5.1 shows the top 10 counters by violation ratio. If the counter belongs to the front end process, we label it as FE. If the counter belongs to the back end process, we label it as BE. If the counter belong to other processes, we label it as oBE. For P1a and P1b, we should see the top counters with label FE as we force resource contentions on the front end process. For P2a and P2b, we should see the top counters with label BE as we force resource contentions on the back end process.

We apply our solution to identify load independent counters. We use the two

Normal runs and mark all the counters that have more than 10% violation ratio. In Table 5.1, we mark these counters with a ”*”. In the Normal vs Normal run, we are able to identify four counters that always have high violation ratios (in the first column group from the left). We mark these four counters with ”*” on the rest of the test pairs.

As we can see, these counters show up in the top 10 counters for the other regression cases: 3 for P1a and P1b (second and third column group) and all 4 for P2a and P2b (third and fifth column group). Once we remove these counters, the results are much clearer. For example, once we remove the three load independent counters for the P1a case, all the remaining counters in the top 10 are those of the front end. The front end is the regression's location, so the result is as expected.

# Part II

# Identifying Performance Regressions Using Control Charts

# CHAPTER 6

# Study 1: Identifying Injected Performance Regressions

## 6.1 Introduction

In this first and preliminary study, we showcase how we can use control charts to reduce performance counter data. We derive a with-untagged-data approach which analyzes the reduced data to classify a load test run as good or problematic. For our first study, we use the force resource contention method and injected performance regressions method (*see* Section 4.2 p.49) to evaluate our control charts based approach. We use the Commercial 1 and DS (*see* Section 4.1 p.48) software systems for this study.

The chapter is organized as follows. Section 6.2 explains the setting and experimental design of our case studies. Section 6.3 shows the results. We discuss our findings and conclude in Section 6.4.

Table 6.1: Configuration of the Dell DVD store load generator

| **Property** | **Value** |
|---|---|
| Database size | Medium (1GB) |
| Threads | 50 |
| Ramp rate | 25 |
| Warm up time | 1 minutes |
| Run duration | 60 minutes |
| Customer think time | 0 seconds |
| Percentage of new customers | 20% |
| Average number of searches per order | 5 |
| Average number of items returned in each search | 5 |
| Average number of items purchased per order | 5 |

## 6.2 Case Studies

### 6.2.1 Study setting

We use the Commercial 1 and DS software systems (*see* Section 4.1 for an introduction about these systems) in this study:

- For Commercial 1, the details are explained in Section 4.1. We install the software in a lab with the same components as required in its production environment.

- For DS, Table 6.1 shows our configuration for the load generator so others can replicate our analyses if needed. The front end web server is an Apache Tomcat application server (The Apache Software Foundation, 2010). The backend database is a MySQL database server (MySQL AB, 2011). We host each component on a separate server with a single-core Pentium 4 processor at 3.2 Ghz with 4 GB of RAM.

### 6.2.2 Experiment Design

We run two types of tests on both the Commercial 1 and the DS systems:

- The first type is the good runs. We pick a very stable version of the Commercial 1 software system and push it through a stable load profile. For DS, we perform the test runs using the profile that is specified in Table 6.1. Both systems perform comfortably under these load profiles. For DS, the CPU utilization of the web server is about 40% on average. The CPU utilization of the database server is about 50% on average. Neither system is sensitive to memory or disk IO.

- The second type of test runs is the problematic test runs which are test runs with simulated performance anomalies. We apply the same load on both systems as in the good runs. However, we simulate different anomalies to either the front end server or one of the back end servers.

Table 6.2 outlines the four anomalies that we simulate. Since we cannot alter the code of Commercial 1, we employ the force resource contention method (*see* Section 4.2.1) to simulate performance regressions in the front end's code (P1) in both systems by starting a separate process with realtime priority that uses about 50% (P1a) and 75% (P1b) of the CPU time. This process will compete with the actual service processes for CPU cycles. This first change forces the front end code to perform about 10% and 25% worse on DS. We apply the same method on one of the back end servers of Commercial 1 and the MySQL database server of DS (P2). This second change will slow down the backend subsystem.

Since we can alter the code of DS, we employ the inject performance regressions method (*see* Section 4.2.1) to introduce performance regressions. We introduce two changes to the code. The first change is a busy waiting loop in the credit

Table 6.2: Problems introduced to simulate problematic test runs

| **Problem ID** | **System** | **Description** |
|---|---|---|
| P1a P1b | Both | CPU hog a) 50% and b) 75% of CPU on the front end server |
| P2a P2b | Both | CPU hog a) 50% and b) 75% of CPU on one of the back end servers |
| R2 | DS | Busy waiting in the front end server code |
| R3 | DS | Extra query to the database back end |

card processing function (R2). This first change slows down the Tomcat front end process about 15 milliseconds per ordered item. The second change is to issue extra calls to the MySQL server (R3). This second change slows the MySQL server down to about 50% of its original capacity. More details on R2 and R3 can be found in Section 4.2.1.

## 6.3 Result

We refer to a pair of baseline and target runs as a *test pair*. For example, Normal $\rightarrow$ Normal means that we build control charts using a normal run and test another normal run.

In a Normal $\rightarrow$ Normal pair, we build a control chart for a good baseline run. This control chart represents the normal behaviour of the software. Then we pick a target run which is also a good run. We determine the violation ratio of the target run using the control chart of the baseline's counter. If the violation ratio is relatively low, then our control charts based approach performs well since the target run is supposed to be within the control limits. Hence, the run should be marked as good. A high violation ratio for the Normal $\rightarrow$ Normal pair means that the counter's behaviour is too random for a control chart. Hence, our control chart

Table 6.3: Study 1 results

| **Test pair** | **Violation Ratio** | |
|---|---|---|
| | **Commercial 1** | **DS** |
| Normal $\rightarrow$ Normal | 4.67% | 12.37% |
| Normal $\rightarrow$ P1a | 18.51% | 100% |
| Normal $\rightarrow$ P1b | 94.33% | 100% |
| Normal $\rightarrow$ P2a | 11.11% | 100% |
| Normal $\rightarrow$ P2b | 15.09% | 96.66% |
| Normal $\rightarrow$ R2 | N/A | 28% |
| Normal $\rightarrow$ R3 | N/A | 100% |

base approach performs poorly.

On the other hand, if we pick a target run that is a problematic run with one of the anomalies in Table 6.2, the violation ratio should be relatively high. The counter of the problematic target run should be mostly outside the control limits. So the run should fail. On contrary, if the violation ratio is low, then our control charts based approach is not useful since it cannot distinguish between a problematic and a good run.

In both case studies, we only use the response time counter to identify performance regressions. The response time counter was identified by the Commercial 1's performance testers as the main indicator of performance regressions. In later studies, we do use all of the collected counters for both Commercial 1 and DS software systems.

For the control chart limits, we use the $1^{th}$ percentile and the $99^{th}$ percentile for the lower and upper control limits respectively. These limits are equivalent to about $\pm 3$ standard deviations from the mean response time.

Table 6.3 shows the violation ratio for different test pairs for both studied systems. As we can observe, when we use a control chart to verify a good run against

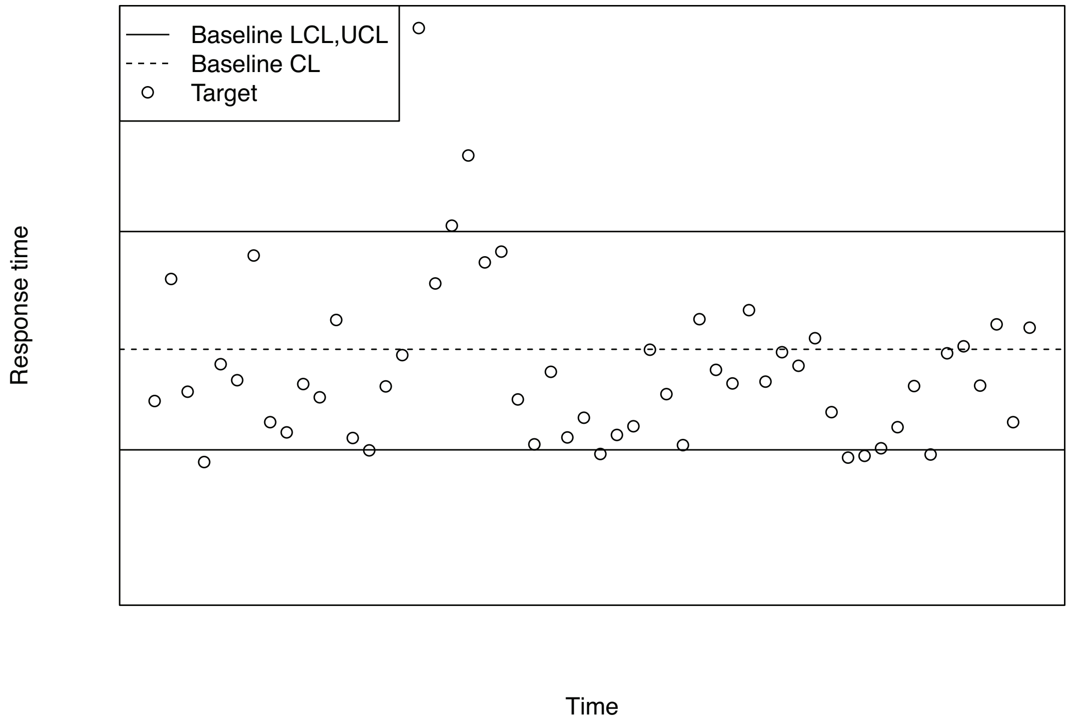


Figure 6.1: Control chart of a good baseline run with a good target run in Commercial 1 (Normal $\rightarrow$ Normal)

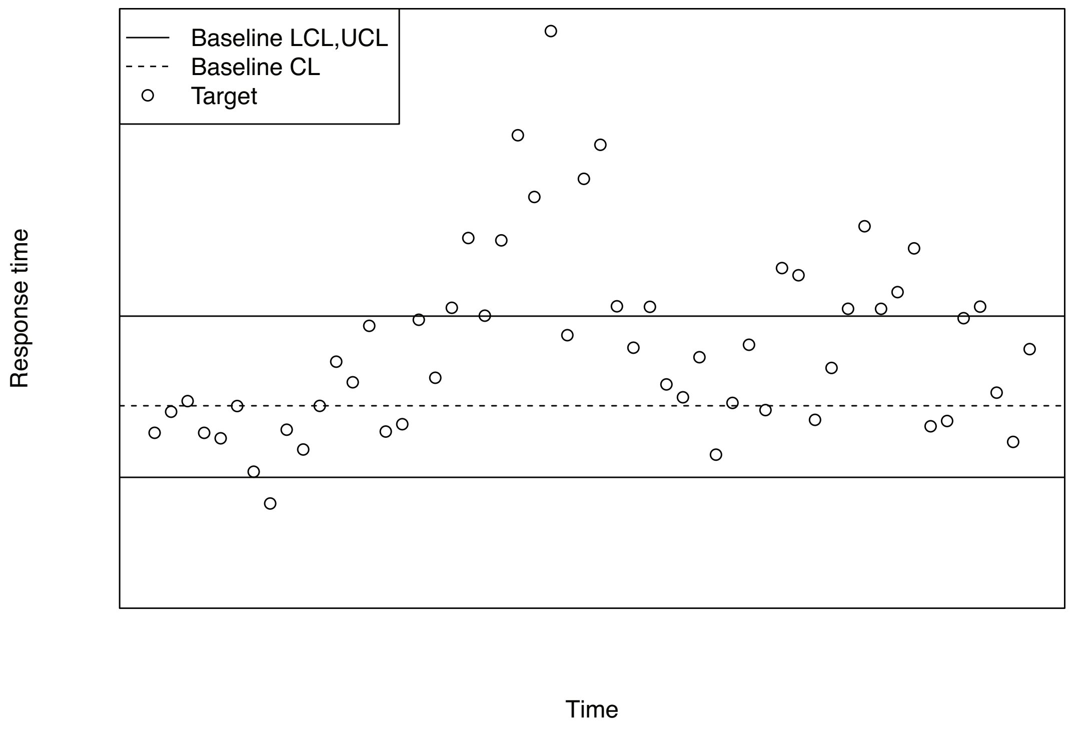


Figure 6.2: Control chart of a good baseline run with a problematic target run (P2a) in Commercial 1 (Normal → P2a)

another good run (Normal $\rightarrow$ Normal), the average violation ratios are relatively low at 4.67% and 12.37% for Commercial 1 and DS respectively. Figure 6.1 shows an example of a control chart with a good baseline run and a good target run in Commercial 1. The x-axis shows the timeline of the test run. The y-axis shows the response time. As we can see, most of the target run's response times are within the control limits set by the baseline. This result indicates that control charts based approach can actually mark a good run as a good run.

On the other hand, the violation ratio is relatively high when the target runs are problematic. For the Commercial 1 system, if we mark all runs with a violation ratio that is higher than 10% as problematic runs, we should be able to mark all problematic runs for further inspection. For the DS system, if we mark all runs with a violation ratio that is higher than 15%, we can catch all the problematic runs. Figure 6.2 shows an example of a control chart with a good baseline run and a problematic target run for Commercial 1 (Normal $\rightarrow$ P2a). This target run has the P2a anomaly. As we can observe, there are more data points (response time samples) outside of the control limits set by the normal run. For R2 and R3, both runs of DS exhibit a high violation ratios when compared to a normal DS run. These results indicate that our control charts based approach can automatically verify the result of load test runs.

Interestingly, despite being much a simpler and smaller system, the counter fluctuations in DS are much higher. When the target runs are normal, the average violation ratio is 12.37% compared to 4.67% on the Commercial 1 system. Commercial1 is a commercial product which undergoes constant performance engineering tests. DS, on the other hand, is a sample application that is not as throughly tested.

## 6.4 Conclusion

Our preliminary study shows that a control charts based approach can automatically determine if a test run is normal or problematic. Our control charts based approach has a large potential application in practice because the approach reduces the need for the human intervention in the load test verification process. For example, the development team can set up their build system to automatically perform load tests in the same fashion that unit tests are automated. When a developer checks in a new code, the build system will perform the integration build. Then it starts the load test and records the counters. Then the build system can count the violation ratio of the new run on control charts of past test runs. If the violation ratio is too high, the developer can be notified immediately. Being able to understand the performance impact of their code right away would make developers more conscious about performance (Kamei et al., 2016). Such ability can have a strong potential impact on the performance of software products like what unit tests have done for functional verification.

However, as a preliminary study, we only use the force resource contentions and injected performance regressions methods to simulate performance regressions. It would be early to conclude that our control charts based approach works based on just this preliminary study with injected faults. So, in the next study, we attempt to apply our control charts based approach to identify performance regressions based on data from real performance load test runs.

CHAPTER 7

# Study 2: Identifying Regressions in Industrial Performance Tests

## 7.1 Introduction

Chapter 6 describes our preliminary examination of the use of control charts for performance testing. The main limitation of that study is that we use injected data, i.e., there was no input from testers and we did not use any real data. So in this second study, we want to have input from the testers and use real data to evaluate if we can use our control charts based approach to distinguish between real test runs with performance regressions and those without. To evaluate our approach, we conduct three case studies on Commercial 1 and Commercial 2 (*see* Section 4.1, p.48).

The chapter is organized as follows. Section 7.2 presents our approach and case study design. Section 7.3 explains how we evaluate our case studies. Section 7.4 discusses the results of our case studies. We conclude in Section 7.6.

## 7.2 Approach and Case Studies

### 7.2.1 Case Study 1: Commercial 1

Our industrial partner maintains a repository of the results of the performance regression test runs for the Commercial 1 software system. As explained in Chapter 3, the Commercial 1 system is a typical multiple-tier server client architecture. Performance testers perform performance regression tests at the end of each development iteration. The tests that we use in this study exercise load on multiple components residing on different servers. The behaviour of the components and the hardware servers is recorded during the test run. The testers then analyze the performance counters. After the runs are analyzed, they are saved to a repository so testers can compare future test runs with these runs.

We pick a few recent versions of the Commercial 1 software system. We pick 110 runs in total. These runs include past normal runs without performance regressions and failed runs with performance regressions.

The main focus of the performance regression testing for the Commercial 1 system is to keep the CPU utilization low because the testers already know that CPU is the most performance sensitive resource. So we extract the test runs' CPU utilization counters from the repository.

We build the control charts using the method that we detailed in Chapter 3. Unfortunately, we do not know which test run is the baseline of which. So we adopt a leave-one-out approach to build the control charts. For each test, we use the CPU utilization counter of that test as the target. Then we take a random sample of the

CPU utilization counter of all the other tests as the baseline test. We use the random sampled baseline data to calculate the control limits. These control limits are, then, used to score the target test. So for each test run, we will have the violation ratios for the CPU utilization counter.

### 7.2.2 Case Study 2: Commercial 2

The purpose of this case study is to show, in details, how our control charts based approach works with real performance regressions using multiple counters.

We pick a release of the Commercial 2 software system. During the time frame of one of the major release iteration, there was one major performance regression that was uncovered by the software system's performance team. We take the build that contains the regression and conduct three performance load tests:

1. **Baseline run 1:** This run is with a build without the regression.
2. **Baseline run 2:** This run is with another build without the regression.
3. **Target run:** This run is with the build that has the regression.

### 7.2.3 Case Study 3: Commercial 2

Similar to case study 1, we collect 22 test pairs of a newer release of the Commercial 2 software system. Unlike case study 1, we know that 11 of the pairs have performance regressions that were identified by the testers. We also collect 11 test pairs without performance regressions so there is an equal number of tests with and without regressions. Also, similar to case study 2, we consider all the performance counters in case study 3 instead of just the CPU utilization as in case study 1.

## 7.3 Evaluation

### 7.3.1 Case study 1: Commercial 1

For each test run, we ask the testers to determine if the run contains performance regressions, i.e., it is a failed run, or not, i.e., whether it is a normal run. We used the testers' evaluation instead of ours because we lack domain knowledge and may be biased.

We compare our classifications with the testers' classification and report the performance. However, we need to decide on a threshold, $t$, such that if the violation ratio $V_x > t$, run $x$ is marked as a failed run. We explore a range of values for $t$, to understand the sensitivity of our approach to the choice of $t$. At each $t$, we report the Precision, Recall, and F-measure of our classification compared to the tester's classification.

### 7.3.2 Case study 2: Commercial 2

We build control charts to identify regressions between:

1. **Normal-Normal pair:** Baseline run 1 versus 2: If our approach works, the control charts should indicate that no regression exists.

2. **Normal-Failed pair:** Baseline run 1 versus Target run: Because the Target run in our case study has a known performance regression, the control charts should indicate that there is a regression.

For each pair, we show the violation ratios as defined in Section 3.2.2 for the top counters.

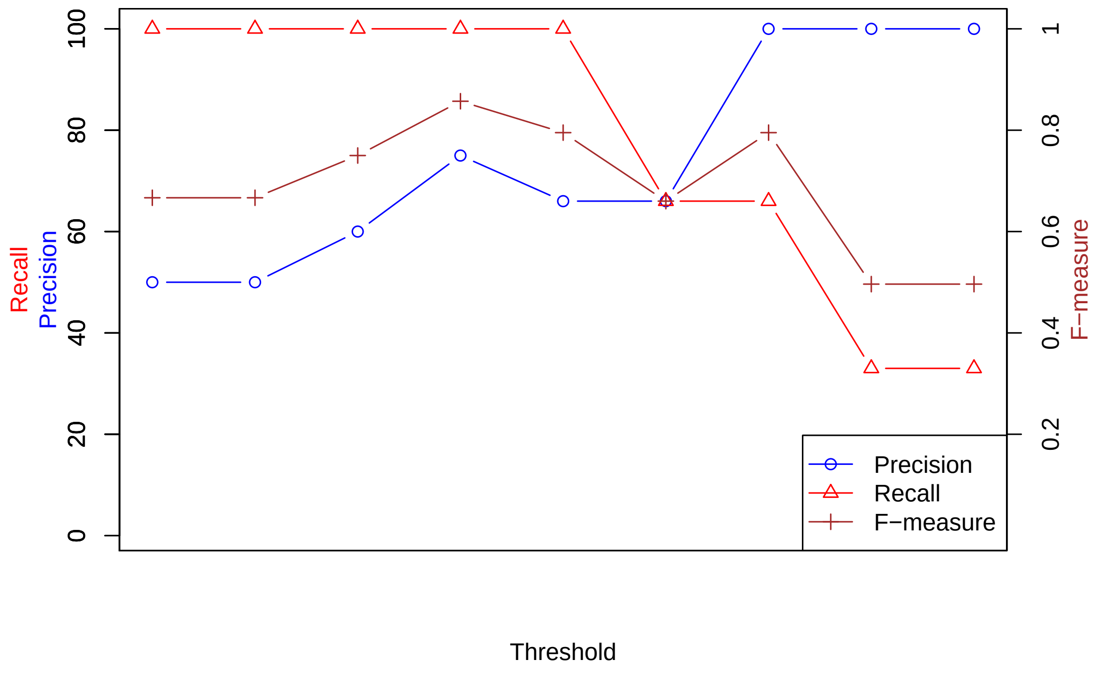


Figure 7.1: The performance of our control charts based approach compared to the testers' evaluation in case study 1.

### 7.3.3 Case Study 3: Commercial 2

Similar to case study 1, we compare the control charts based approach classifications with the real classification (normal versus failed) of the test pairs using the same analysis.

## 7.4 Results

### 7.4.1 Case Study 1: Commercial 1

Figure 7.1 shows the Precision, Recall, and F-measure (*see* Section 4.2.2) when comparing the control charts classifications with the testers at different thresholds. The threshold increases from left to right. We hide the actual threshold values for confidentiality reasons. When the threshold increases, we mark more runs as failed, in which case the Precision (blue circles) increases but the Recall (red triangles) decreases. The F-measure (brown pluses) is the highest when the Precision is at 75% and the Recall is at 100%.

Our approach can identify test runs with performance regressions at a Precision of 75% and a Recall of 100% when applied on real test data.

### 7.4.2 Case Study 2: Commercial 2

Table 7.1 shows the top 10 counters by average violation ratio (*see* Section 3.2.2 for definition) for the Normal-Normal pair. The Normal-Normal pair represents the runs with builds that have no performance regressions. As we can see here, all the average violation ratios are very low.

Table 7.2 show the top 10 counters by average violation ratio (*see* Section 3.2.2 for definition) for the Normal-Failed pair. The target test in this case contains a known performance regression. As we can see, all of the counters in Table 7.2 have a lower or upper violation ratio of 100%.

The results from Table 7.1 and Table 7.2 show that control charts are able to accurately identify the performance regressions in our case study. For the Normal pairs, all the violation ratios of the 10 most out-of-control counters are very low. For the Regression pairs, all the violation ratios of the 10 most out-of-control counters are very high.

Table 7.1: The ten most out-of-control counters in **Normal-Normal pair**

| **Counter** | **Vio Lower %** | **Vio Upper %** | **Vio Average %** |
|---|---|---|---|
| TG A Thread 1 Last Write Unsecured ms | 0 | 0.073 | 0.037 |
| TG A Thread 24 Last Write Secured ms | 0 | 0.073 | 0.037 |
| TG A Thread 1 Last Write Secured ms | 0 | 0.073 | 0.037 |
| TG A Thread 28 Last Write Secured ms | 0 | 0.049 | 0.024 |
| TG A Thread 14 Last Write Secured ms | 0 | 0.049 | 0.024 |
| Current Time-Waited DB Connections | 0 | 0.049 | 0.024 |
| Last Write Secured ms | 0 | 0.049 | 0.024 |
| Online Users | 0.073 | 0 | 0.037 |
| Current connections | 0.073 | 0 | 0.037 |
| Open File Descriptor Count | 0.073 | 0 | 0.037 |

### 7.4.3 Case Study 3: Commercial 2

Figure 7.2 shows the Precision, Recall, and F-measure (*see* Section 4.2.2 p.52) when comparing the control charts classifications with the testers's (i.e., correct) classifications at different thresholds. The threshold increases from left to right. We hide the actual threshold values for confidentiality reasons. When the threshold increases, we mark more runs as failed, in which case the Precision (blue circles) increases but the Recall (red triangles) decreases. The F-measure (green pluses) is maximized when the Precision is about 91% and the Recall is at 100%.

Table 7.2: The ten most out-of-control counters in **Normal-Failed pair**

| **Counter** | **Vio Lower %** | **Vio Upper %** | **Vio Average %** |
|---|---|---|---|
| Committed Virtual Memory Size | 0 | 1 | 0.5 |
| Memory Buffers KB | 0 | 1 | 0.5 |
| CPU Utilization Total System | 0 | 1 | 0.5 |
| CPU Utilization Total Idle | 1 | 0 | 0.5 |
| CPU Utilization Cpu0 Soft IRQ | 1 | 0 | 0.5 |
| Retrieve Total Delayed Requests With Results | 1 | 0 | 0.5 |
| Network Utilization Eth0 Sent p/s | 1 | 0 | 0.5 |
| Network Utilization Eth0 Received B/s | 1 | 0 | 0.5 |
| Network Utilization Eth0 Received p/s | 1 | 0 | 0.5 |
| Garbage Collector PS Marksweep LastGC Duration | 1 | 0 | 0.5 |

## 7.5 Discussion

### 7.5.1 Can Our Control Charts Based Approach Be Used to Identify Performance Regressions?

In case study 1, the best F-measure is achieved when the Precision is at 75% Precision and the Recall is at 100% (*see* Figure 7.1). Hence, at the best threshold, our approach can identify all the performance regressions runs with about a 25% false positive. Performance testers would need to double check the classifications of our approach to rule out such false positive runs. While the performance of our approach is not perfect, we are expected to have non-perfect results in this case study as we do not know which tests are normal or failed in this case study. The test runs in the Commercial 1's repository is not marked. The testers did not identify the baseline of the tests in the repository. So, the baseline is sampled data from all other tests including the failed ones. Hence, the baseline's control limits might also

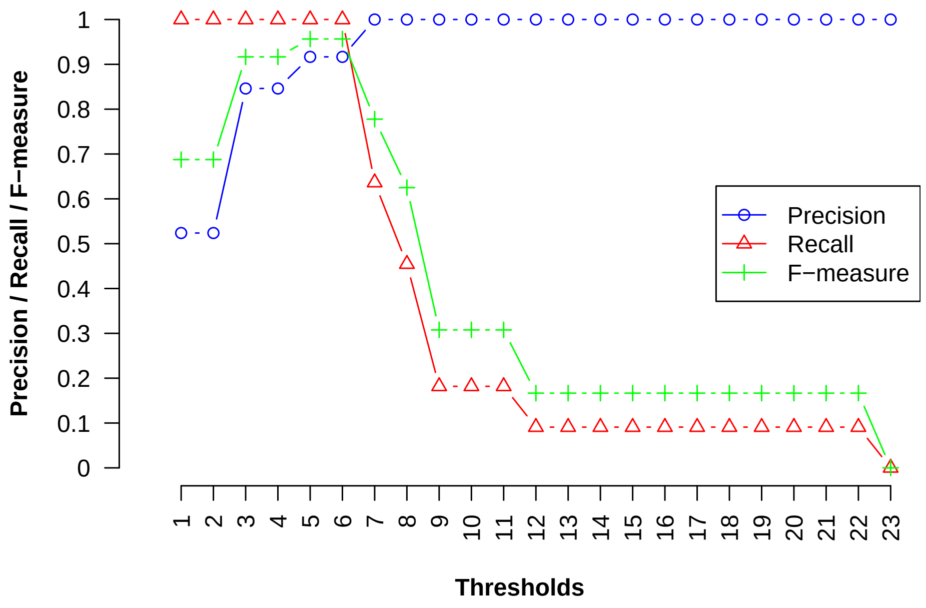


Figure 7.2: The performance of our control charts based approach compared to the correct classifications in case study 3.

include out of control data.

In case study 3, we do know the baselines of all the tests and whether each test is normal or failed beforehand. So the baseline data is from the actual baseline test of a particular test run. The reported performance of our approach is better. The F-measure is maximized when the Precision is about 91% and Recall is at 100% (*see* Figure 7.2). Hence, our false positive rate is only 9% compared to 25% of case study 1.

### 7.5.2 Why Are There False Positive Cases in Our Approach?

According to case study 2, the counters of the Normal-Normal pairs have very low violation ratios. The highest average violation ratio is 0.037% (*see* Table 7.1). For the Normal-Failed pairs, the violation ratios for a few counters are high. Table 7.2 shows that at least 10 of the counters have very high average violation ratio. So why is it that we have 9% false positive in case study 3?

The problem is the number of counters. Our approach takes the average of all the average violation ratios (of all the counters) in a test pair and compares that average with a fixed threshold. The average depends on the number of counters that are impacted by the regression. Some regressions might affect a large number of counters. However, other regressions only affect a small number of counters. When the number of affected counters is small, it is hard distinguish between the normal and the failed pairs because the change in the average would be small.

To demonstrate that the number of counters has an effect on performance, we rerun the analysis for case study 3. This time, instead of taking the average of the average violation ratios (of all the counters), we only take the average of the top 50% of the counters (based on the average violation ratios). Figure 7.3 shows the Precision, Recall, and F-measure. Compared to the original analysis (Figure 7.2), the best F-measure is achieved at a Precision of 100% and a Recall of 100% instead of just a Recall of 91% as in the original analysis.

So, with some tuning, one can achieve very high performance with our control charts based approach. Unfortunately, the number of counters that should be considered during the analysis would depend on how the counters are implemented and how the regressions would affect the counters. These two factors are different from one software system to another.

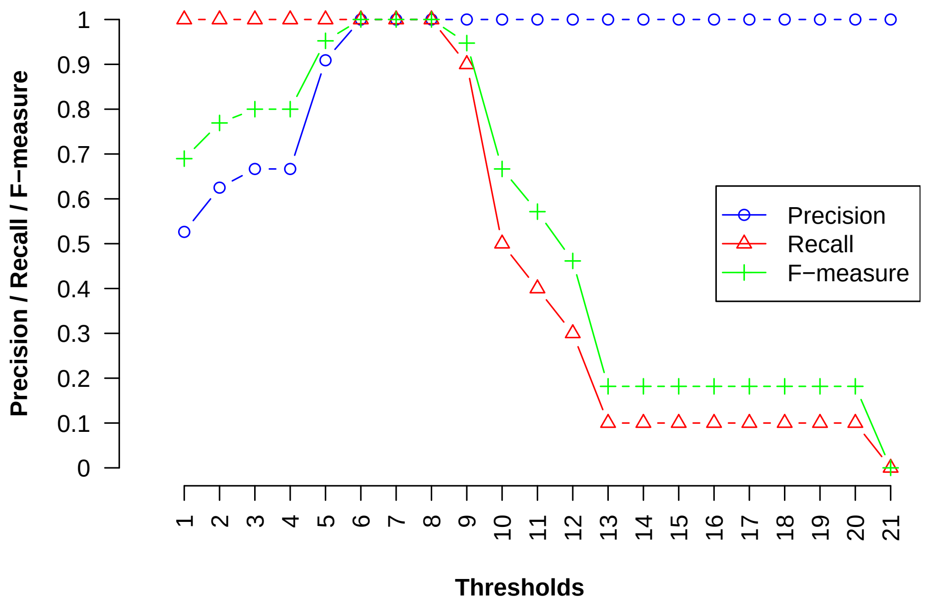


Figure 7.3: The performance of our control charts based approach compared to the correct classifications in case study 3 when we only consider the only top 50% of the counters based on the average violation ratios.

### 7.5.3 Which Threshold Should Be Used?

In case study 1 and case study 3, we show the performance of our approach over a range of thresholds (Figure 7.1 and Figure 7.2). An obvious question would be: What threshold should one use if one wants to implement our approach?

A fixed threshold for every performance regression is difficult to determine. As we showed in the previous subsection, the Precision and Recall are affected by the number of counters that were influenced by the performance regressions. Hence choosing a threshold should be a tuning exercise.

We propose this approach to start with: One can use the maximum of the average violation ratios of all the normal pairs as the threshold. For case study 3, that number would correspond to threshold 7. The best threshold is 5. So it would not be a good choice. For the analysis in Figure 7.3, that number would correspond to threshold 6. At this threshold, the Precision and Recall are both at 100%. So it is a good choice.

Unfortunately, the threshold changes from software to software and from regression to regression. More importantly, our experience working with the Commercial 2 project shows that the definition of a regression also changes over time. In earlier releases, when there are many code changes and the number of users is small, a relatively small increase in resource utilization would not be considered a regression. However, as the software matures and there are more users, a change of the same amount in resource usage would be considered a regression.

So choosing a threshold definitely requires some adjusting depending on the software system, its maturity, its criticality, and the nature of the regressions.

## 7.6 Conclusion

In Chapter 6, we show that control charts can identify performance regressions which we injected into the under-test software system. In this chapter, we show that control charts can identify performance regressions from industrial performance load tests. Our case studies include all the test runs for a few releases of the Commercial 1 software system, a performance regressions that were identified in one release of the Commercial 2 software system, and series of test runs for a major release of Commercial 2 software system. In all three case studies, we show that our control control charts based approach can identify the performance regressions that were manually identified in practice. From both Study 1 and Study 2, we can conclude that our control charts based approach can be used to automatically identify performance regressions in performance load tests.

# Part III

# Determining Root Causes Using Control Charts

CHAPTER 8

# Study 3: Using Control Charts to Determine Causes of Performance Regressions

## 8.1 Introduction

The previous part of this thesis mainly dealt with identifying performance regression in a new test run. That is the first goal as mentioned on the thesis's problem statement (Chapter 1, p.2). The second goal is to determine the possible causes of the identified performance regressions. Once a performance regression is identified, the performance team must also provide guidance to the development team as to what the possible causes (e.g., added fields in a long-living object) are. Developers use such guidance to narrow down their investigation to the offending change(s). In an ideal setting, performance tests would be run after each checkin so the causes of regressions could be easily mapped to the very specific checkins (i.e., code change).

However, given the complexity of performance tests (e.g., requiring large lab setups, complex manual configurations, and lengthy executions times), per-checkin performance tests are rarely feasible in a large scale industrial setting. Instead performance tests are conducted across versions which often contain a large number of changes. Hence determining the cause of an identified regression (which we call as a regression-cause in the rest of the chapter) is often a very time consuming effort – requiring hours or even days, depending on the experience of the testers, the complexity of the system, and the performance regression itself. The testers would need to use their experience to match the behaviour of the performance counters with the new code or configuration changes. Then, the testers would need to remove the change that is in doubt from the code. The testers would also need to rerun the performance test to confirm if the removed change is indeed the cause of the detected regression.

In this chapter, we investigate a control charts based approach to automate the process of determining regression-causes. We use data mining techniques to learn the cause of the performance regression from historical performance test runs that already have verified regression-causes. Then we match the behaviour of a new test run (through the performance counters) to determine the regression cause in the new test run.

We evaluate our approach using the Commercial 2 and the DS systems (*see* Section 4.1). Using the two case studies, we answer the following research questions:

- **RQ1: How accurate is our approach?** We find that in 74%-80% of the cases, the regression-cause that was determined by our approach was indeed the actual cause.

- **RQ2: How much training data is needed?** Since the number of historical performance regressions might be limited, we examine the amount of training data that is needed by our approach. We find that, even though having more training data improves the performance of our approach, we can get high accuracy with as little as four runs per regression-cause.

The chapter is organized as follows. Section 8.2 introduces regression-causes and the challenge determining regression-causes. Section 8.3 explains the details of our approach. Section 8.4 introduces the two case studies that we used to evaluate our approach. Section 8.5 presents the results for our two research questions. We conclude in Section 8.7.

## 8.2 Background

### 8.2.1 What Are Performance Regression Causes?

To answer this question, we study the performance regressions causes that were reported by the Commercial 2 team over a period of one year. With the help of the testers, we analyze the content of bug reports using an approach similar to Jin et al. (Jin et al., 2012), where bug reports were manually analyzed and grouped into different types of causes.

Table 8.1: Typical performance regression causes for Commercial 2 software project

| **Type** | **Regression-cause** | **%** |
|---|---|---|
| R1 | Added frequently accessed fields and objects | 30.18% |
| R2 | Added frequently executed logic | 16.67% |
| R3 & R5 | Added frequently executed DB query or miss matched DB indices | 30.54% |
| R8 | Added blocking I/O access: | 5.55% |
| N/A | Symptom of regression is identified (e.g., response time increased) but no regression-cause can be determined. | 16.67% |

Table 8.1 shows a breakdown of the various causes of regressions over a one-year period (only percentages are shown due to confidentiality reasons). We name the regression-causes according to the wording in the bug reports. However, these regression-causes are widely exhibited in practice with similar names. Altman et al. (Altman et al., 2010) and Malik et al. (Malik et al., 2013) also report similar regression-causes, e.g., Memory Leak, Database Bottleneck, or Disk I/O. We explained, in more details, the different regression types in Section 4.2.1.

As we can see, four of the regression-causes can explain 82.94% of the encountered regression problems throughout a year. Only for 17% of the identified performance regressions, no regression-cause was identified. Such cases are common since not all performance metrics are exposed. These cases usually require extensive debugging.

### 8.2.2 What are the Challenges of Analyzing the Results of Performance Tests?

Unfortunately, determining the regression-cause is not a simple task because:

- As we mentioned in Chapter 1, there exists a large amount of counters. Apart from the standard OS level counters, system-specific counters are also added to monitor the activities of different components. It is possible to have thousands of counters for each test run.

- Most performance regressions change a combination of counters. For example, if a database index is missing, both the CPU utilization and memory are likely to increase. If the database slows down, the front end will have to wait more for queries to return. Hence, the front end will use less CPU. Thus, looking at just one counter would not be sufficient.

Yet, the analysis of the test results is usually done manually - a very time consuming and error-prone process (Jiang et al., 2008). To analyze this massive amount of counter data, performance testers usually have to rely on their experience and gut feelings when selecting a subset of counters to manually compare. It can take several hours or even days to analyze each performance test.

## 8.3 Approach

Software companies sometime maintain a repository of regression test results. An example of that is the Commercial 2 software system. We leverage the regression test repository of Commercial 2 in our Second Study (*see* Chapter 7). Based on our prior experience with mining software repositories, we believe that we can also leverage information from prior tests in such repository to aid with the regression cause analysis as well.

As explained in Chapter 2 (p.6) and Chapter 3 (p.39), there are two conceptual steps for our approach:

- **Step A - Data reduction:** As in the previous studies, we reduce the counter data into a smaller set of data using control charts as explained in Section 3.2 (p.40). Thus a simpler and easier to understand performance counter dataset is created for use in the next step of our approach.

- **Step B - Identify regression and determine regression-cause:** Using machine learning techniques, we match the regression-cause of the current test run with the regression-cause of historical runs. One of the regression-cause is “Normal”, which means there is no regression. So we can say a) whether a regression occurred and b) what is the probable cause of that regression.

### 8.3.1 Step A: Data Reduction Using Control Charts

We first reduce the data by applying our control charts based approach for data reduction as outlined in Section 3.2 (40). The resulting data for analysis is similar to the example in Table 8.2. Each row represents a test run (with relation to its baseline test run). Each column is the violation ratio of the corresponding counter when comparing the target run against its baseline run. We have one column for each counter of every component of the software. For example, if we have 15 components and each component has 36 counters (the standard counters that Microsoft Windows collects for a process), we will have $15 * 36 = 540$ columns.

The top part of Table 8.2 shows the historical tests which are stored in a repository. The last column shows the determined cause of the performance regression in that run. If there is no regression, the regression-cause would be R0 ('Normal'). If there is a performance regression, the verified regression-cause is recorded. For example, in the first row, there is no performance regression, so the regression-cause is R0 ('Normal'). In the second row, there is a performance regression due to adding code to a frequently accessed logic, so the regression-cause is recorded as R2 ('Added hot code') (*see* Section 4.2.1 for a detailed explanation of the various

Table 8.2: Example of the data that is used in the machine learning of Step B.

Co.X = Counter X, and VR = Violation Ratio

| **Data** | **Co.1 CPU VR** | **Co.2 Mem VR** | **Co.3 Net VR** | ... | **Co.n VR** | **Regression-Cause (*see* Sec. 4.1)** |
|---|---|---|---|---|---|---|
| Historical test runs | 3% | 5% | 6% | ... | ... | R0 |
| | 14% | 0% | 3% | ... | ... | R2 |
| | 13% | 2% | 5% | ... | ... | R2 |
| | 3% | 23% | 4% | ... | ... | R1 |
| | 3% | 4% | 2% | ... | ... | R0 |
| | ... | ... | ... | ... | ... | ... |
| New test run | $X_1$ | $X_2$ | $X_3$ | ... | $X_n$ | **?** |

regression-causes).

### 8.3.2 Step B: Determining the Regression Types Using Machine Learners

The last row of Table 8.2 is the new test run, which corresponds to the new version, and hence has an "unidentified" regression-cause (denoted as a '?'). To determine the regression-cause (if a regression occurred) of this new test, an tester would have to match the characteristics of the performance counters of this new test with the performance counters of all prior tests. In simple cases, only a few counters of a component will have high violation ratios, i.e., the regression is isolated to that component. It would be easy to determine the regression-cause. As we mentioned in the previous section, because the components are dependent on each other, a regression in one component will also affect other connected components. In such a case, many of the counters would have high violation ratios therefore the tester must spend considerable time to isolate the regression-cause using his/her experience. Our motivation is to introduce machine learners to capture the experience of the testers.

The matching of the current test to previous tests can be done using machine leaners such as Naive Bayes Classifier (John and Langley, 1995), J48 Decision Trees (Quinlan, 1993), or Multi-Layer Perceptron (Minsky and Seymour, 1969). We train the learners using the data (violation ratios of performance counters and the verified regression-causes) from prior performance test runs. Then, we use the trained learners to determine the regression-cause of the new test run (for which we have the violation ratios of the performance counters).

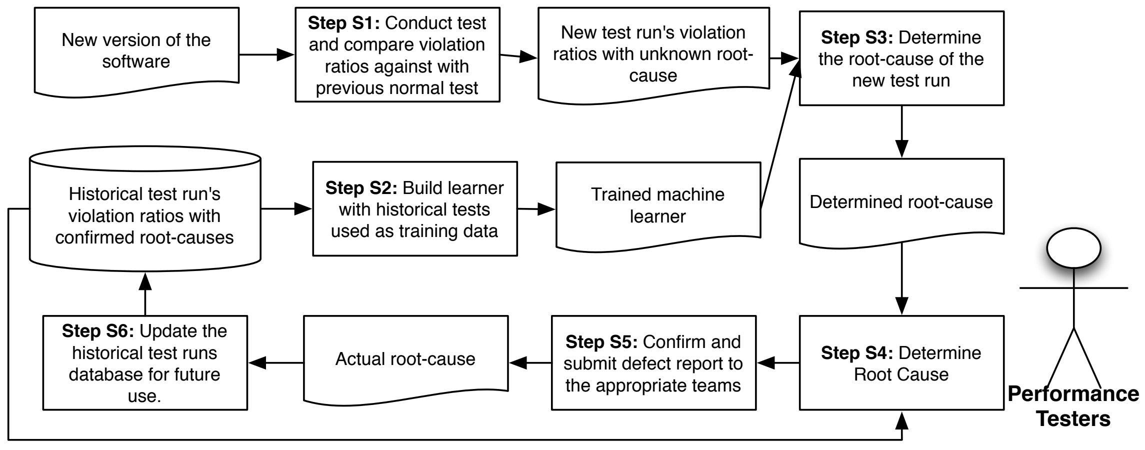

New version of the software
Step S1: Conduct test and compare violation ratios against with previous normal test
New test run's violation ratios with unknown root-cause
Step S3: Determine the root-cause of the new test run
Historical test run's violation ratios with confirmed root-causes
Step S2: Build learner with historical tests used as training data
Trained machine learner
Determined root-cause
Step S6: Update the historical test runs database for future use.
Actual root-cause
Step S5: Confirm and submit defect report to the appropriate teams
Step S4: Determine Root Cause
Performance Testers

Figure 8.1: Our proposed approach for determining the regression-cause.

### 8.3.3 Our Control Charts Based Approach to Automatically Determine Root-causes

Figure 8.1 shows the steps our proposed approach:

- **Step S1:** We compare the violation ratios using the new test data and the previous test data (the baseline) as explained in Step A.
- **Step S2:** Then we use the prior tests in the repository with known regression-causes to train a machine learner.
- **Step S3:** We use the machine learner to suggest the regression-cause of the new test run.
- **Step S4:** At this point, tester can confirm the determined regression-cause. If determined regression-cause is correct, the tester can then file the defect report and communicate with the right development team for further investigation.
- **Step S5:** Once the correct regression-cause is determined, we add the test data as well as the identified regression-cause into the repository so that we

can use it for future tests.

The regression-causes can be as simple as “Added hot code”. Or they can be more complex such as “Added hot code into X component” or even “Added hot code into X component in thread pool A”. The only requirement is that there must be enough prior tests with the same regression-causes. The number of tests needed for each regression-cause is explored in Section 8.5.2. One can start with simpler/more generic regression-causes. Then as the number of tests increases, one can create a more detailed coding system for more specific regression-causes.

## 8.4 Case Study Setup

To evaluate our approach (Figure 8.1), we apply the approach on the Commercial 2 software system. We also apply the approach on the DS software system. Details about the two case studies can be found in Section 4.1 (p.48).

In both case studies, we employ the Inject Performance Regressions method (*see* Section 4.2.1 p.50) to introduce six types of performance regression (i.e., regression-causes): R1, R2, R3, R4, R5, R8, R0. These regression-causes include those that are determined in Section 8.2.1 (in the future additional regression-causes can be added and explored – we limit our regression-causes to ones that the testers noted to be frequently occurring based on their experience).

For the Commercial 2 case study, we inject issues corresponding to the first four regression-causes (R1, R2, R3 and R4). Regression-cause R5 is not applicable since there is no text search in all transactions. We could not implement regression-cause R6 without drastically altering the functioning of the software, which we want to avoid. For each cause, we inject the actual problem (change in source code) in six different parts of the source code. Every one of these six different parts of the code lies in the same execution path for an end user action (e.g., the checkout process in an e-commerce application). We conduct a test run for each code location of each regression-cause. For each of these test runs that will cause a performance regression, we calculate the violation ratios using the counters of that run and the counters of a normal run. We also calculate the violation ratio of the normal runs (R0) by comparing the counters of the examined run against a historical normal run. Thus, we will have the performance counter data (violation ratios) of all 30 test runs: six each, for the normal case and the four regression-causes.

For the DS case study, we use the runs with performance regression of the five regression-causes (R1, R2, R3, R5, and R8), by injecting the issues into six different areas of the JSP code. For regression-causes R4 and R5 we change the database configuration, to cause a performance regression. For regression-cause R8, the logging library was configured to write directly to disk, so that a performance regression with blocking I/O would occur. Normally, most production logging systems use asynchronous I/O instead, which does not cause a blocking I/O regression. For regression-causes R1, R2, and R5, we introduce the problems in two different execution paths of the software. For example, R1-1 is adding fields in long living objects on the execution path for the searching transaction. R1-2 is adding fields in long living objects on the execution path for the ordering transaction. We tag R1-1 and R1-2 as two different regression-causes, because they are on two different execution paths from two different transactions, each of which has a different performance profile. Then, for each code location of each regression-cause, we conduct a test run. We calculate the violation ratios of each run by comparing that run against the 'Normal' runs (R0), which we also run six times. We also calculate the violation ratios for the normal runs by scoring that run over the other normal runs. At the end, we have a table with 54 rows of violation ratios for the nine different causes (six test runs each, for the normal case, and eight problems - R1-1, R1-2, R2-1, R2-2, D, R5-1, R5-2, and R8).

In both case studies, we collect only the OS level performance counters (*see* Section 2.2.1), which produces 32 counters per process, for each test run. Then, we calculate the violation ratios using that run as the target and one of the 'Normal' regression-cause runs (R0) as the baseline. As a result, we have a table for each case study that is similar to the example in Table 8.2. We use these two tables for the analysis.

## 8.5 Results

In this section, we discuss the results of each research question.

### 8.5.1 RQ1: How Accurate is our Automated Approach for Determining Root-causes?

#### Evaluation Approach

We use Weka's machine learners in our experiment. Weka (Group, 2012) is a popular open-source data mining program. It has a comprehensive collection of machine learning techniques. All the test run data (*see* Section 8.4 for more information on the test runs) is imported into R. From R, we can access Weka through an open-source interface called RWeka. We run all our analysis from R.

We use learners from Weka. We use learners that accept numeric independent variables (since violation ratios of counters are numeric) and a categorical dependent variable (determined regression-cause) as inputs to determine the learner with the best accuracy. If the learner can take our inputs, we use it for our approach. We found 17 such learners in Weka version 3.6.10. Table 8.3 lists all the used learners.

For evaluation, we perform a leave-one-out evaluation for both case studies (Section 8.4). We first pick one random test run out of all the runs of all the regression-causes. The rest of the runs are used for training the learner. Then, a learner is used to suggest the regression-cause of this test run, by finding the closest matching regression-cause of all the other test runs that were used as training data. Since regression-causes are known for each test run, we compare the suggested and the actual to evaluate the accuracy. If they match, then it is a success. For example, if the test run's actual regression-cause is "Normal", meaning no regression, and if the suggested regression-cause is also "Normal", then it is a success. Otherwise, it is a failure. For example, if the suggested cause is "Adding hot code", then it is a failure. We repeat this procedure for all test runs.

For each learner, we report the accuracy and gain in overall accuracy compared to the random predictor as explained in Section 4.2.2.

For the Commercial 2 software system, we have five regression-causes in the test runs (four of them with performance regressions, and one is normal), with six test runs for each regression-cause. For the DS case study, we have nine regression-causes in the test runs (eight of them with performance regressions, and one is normal), with six test runs for each. Hence $N$ is $30$ and $54$ and the value of $r$ (the random predictor's accuracy) is 20% and 11% for Commercial 2 and DS respectively.

## Results and Discussion

Table 8.3 shows the accuracy for each of the studied learners in both case studies. The first row shows the random accuracy $r$ (Equation (4.3)). If a machine learner performs worse than $r$, then that learner is not useful.

The higher the accuracy of a learner compared to the random predictor, the better the learner is. The seventh column (L.G.) shows the average accuracy gained

(Equation (4.2)) compared to the random predictor over the two case studies. This average gain indicates the usefulness of a learner. The last column (C.G.) shows the average accuracy gained of all the learners of a particular machine learning algorithm class (column two). The average gain shows which class of learners is most suitable for our approach.

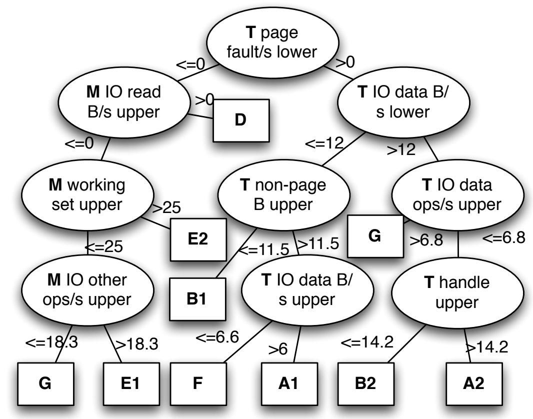
The first observation that we can make from the results is that: **most learners outperform the random predictor**. The worst learner (LWL) has a gain of 2.73 times over the random predictor. The best learner (RandomForest) has a gain of 4.99 times. So, at Step 2e of our automated approach, when the testers try to confirm the regression-cause, they already have two to almost five times advantage compared to random guessing. This advantage would translate into saved time and effort.

The second observation we can make from the results of Table 8.3 is that: **the most suitable learning technique is system dependent**. For the Commercial 2 system, J48, which is a Java implementation of the C4.5 (Quinlan, 1993) learner, is the best learner with 80% accuracy. For DS, the best learner is MultilayerPerceptron (Minsky and Seymour, 1969), a lazy type learner, with 74% accuracy. This means that there is no universal best learner to suggest probable regression-causes for all software systems. The learner can be accurate in suggesting the probable regression-causes of one systems but not the other. MultilayerPerceptron and J48 (Quinlan, 1993) had the opposite accuracy for the two case studies. MultilayerPerceptron can predict with 74% accuracy for DS (best out of 17) but can only achieve 57% for the Commercial 2 system (11th out of 17). On the other hand, J48 works the best for Commercial 2 (80%) but is the 14th best out of 17 learners for DS (56%).

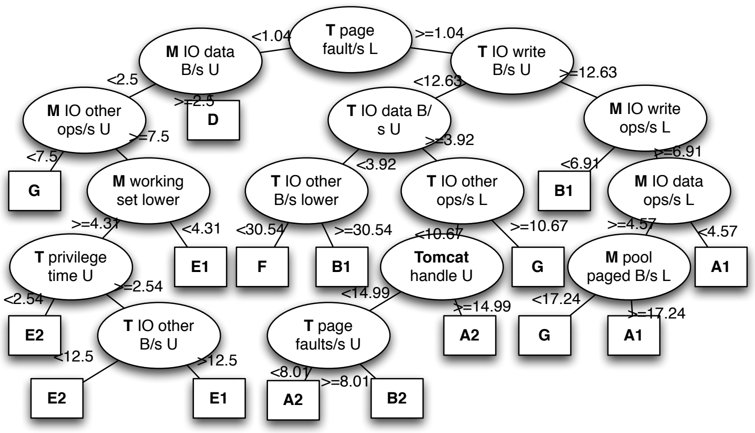
The third observation is that: although there is no universal best learner, **there are few good learners (and classes of learners) for both case studies**. Those learners are identified by the high average gain column (column 7 - L.G.). RandomForest (Breiman, 2001) achieves an average gain of 4.99 times compared to the random predictor (which is 70% and 72% for the Commercial 2 and DS respectively). Similarly SMO achieves average gain of 4.98. Hence using SMO, we gain about three and a half times compared to the random predictor for the Commercial 2 system, and more than six times for DS. In general, the neural network class, which SMO is part of, has a good class-average gain which is 4.87 (last column of Table 8.3)). Note that we take the average to quantify the class level gain, but the median produced very similar results too. The decision tree class, which RandomForest is part of, also has a good class-average gain of 4.60. So, if the best learner for a software system is not known yet, learners from these two classes can be used.

The fourth observation is that: **there are unsuitable learners (and classes of learners) for both case studies**. In general, the rule and lazy based learners are the worst (class-average gain column of Table 8.3 - C.G.). While most of the classes have class-average gain greater than four, these two classes has less than four average gain. Rule based and lazy learners are the simplest learners. They are more suitable for a low number of features (i.e., fewer counters), with a large amount of data, and low amount of noisy data (Mitchell, 1997). However, these learners can benefit from preprocessing approaches (Nguyen et al., 2011), to remove the noisy counters. The fact that both classes of learners perform badly suggests that the relationships among the performance counters are not simple.

To further examine as to why certain class of learners performed very well (observation 2), we look at the ”decision tree” class of learners in Table 8.3 for the DS case study. In particular we compare J48 and RandomForest. In the commercial case study, both learners have a similar performance. While in the DS case study there is a much larger difference in the performance of these two learners (almost 16%). Therefore, we compare the trees that are created by these two learners in the DS case study to understand why there exists such a difference in performance.



(a) J48 decision tree of the DS case study



(b) RandomForest decision tree of the DS case study

Figure 8.2: Comparing the decision tree structure of the DS case study

Figure 8.2 are the decision trees for the J48 and RandomForest learners of the DS case study. As we can see, J48's decision tree is simple. On the left subtree of the root, the decisions are based on the MySQL related counters. On the right subtree, the decisions are based on the Tomcat related counters. RandomForest's decision tree, on the other hand, uses both set of counters data throughout its structure.

The J48 learner is not able to take advantage of all the relationships among the counters. So the achieved gain and accuracy is low. The RandomForest learner was introduced for this kind of situation (Mitchell, 1997). It works by generating different decision trees on a subset of counters. The decisions are the most popular among all generated trees. The resulting tree was able to take advantage of more relationships among the counters. Thus, it has better gain and accuracy.

RandomForest learner can also be used to determine the importance of different counters for a specific classification problem. We run this analysis on the DS case study (Figure 8.2b) for the decision tree that is built by the RandomForest learner. Table 8.4 shows the top 20 most important counters. If we look at the regression-causes of the DS case study in Section 8.4, most of the causes should change the CPU and memory utilization of the MySQL and Tomcat process with the exception of cause R8. However, Table 8.4 shows that the top most important counters are mostly IO related. This is an example of the complex relationship among counters which the machine learners, such as RandomForest, are able to capture.

### 8.5.2 RQ2: How Much Training Data is Needed?

The results of RQ1 shows good accuracy for both case studies. So we have evidence that our approach is useful. Assuming that a performance team wants to adopt our approach, they would first need to build a repository of test runs with verified regression-causes. Building such a repository can be very time consuming. In RQ2, we want to understand how much training data is needed before one can start using our approach.

#### Evaluation Approach

To determine the number of performance test runs that are needed in the repository (i.e., size of training set), we modify the leave-one-out procedure in RQ1. The goal is to observe the change in accuracy if we use one, two, three, four, or five runs of each regression-cause for training instead of using all six runs as we did in RQ1. So, after picking one random run for testing, we will only use $n$ run(s), selected randomly (where $n$ is one to six), of each regression-cause from the rest of the runs to train the learner. This will limit the size of training data to $n$ for each regression-cause. We perform the procedure six times for each $n$ and report the accuracy. We note that, similar to RQ1, under no circumstance is the testing run used for training.

**Results and Discussion**

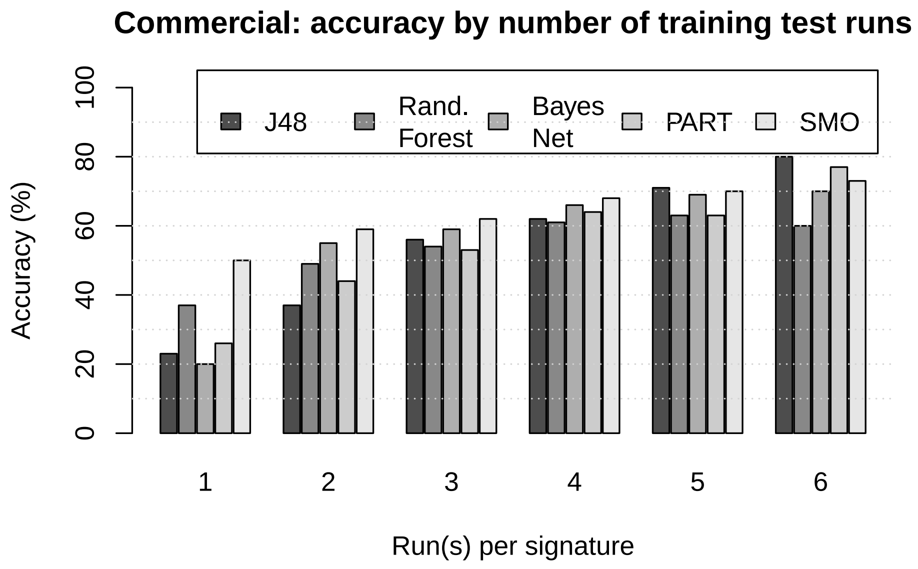
Figure 8.3a and 8.3b shows the results for both case studies. We perform the procedure mentioned above using the top five learners according to the gain columns (G.) in Table 8.3. The first set of bars in Figure 8.3a and 8.3b show the accuracy when only one run of each regression-cause is used for training. The second set of bars is the accuracy when we use two test runs per regression-cause and so on.

We can make two observations from the results shown in Figure 8.3a and Figure 8.3b.

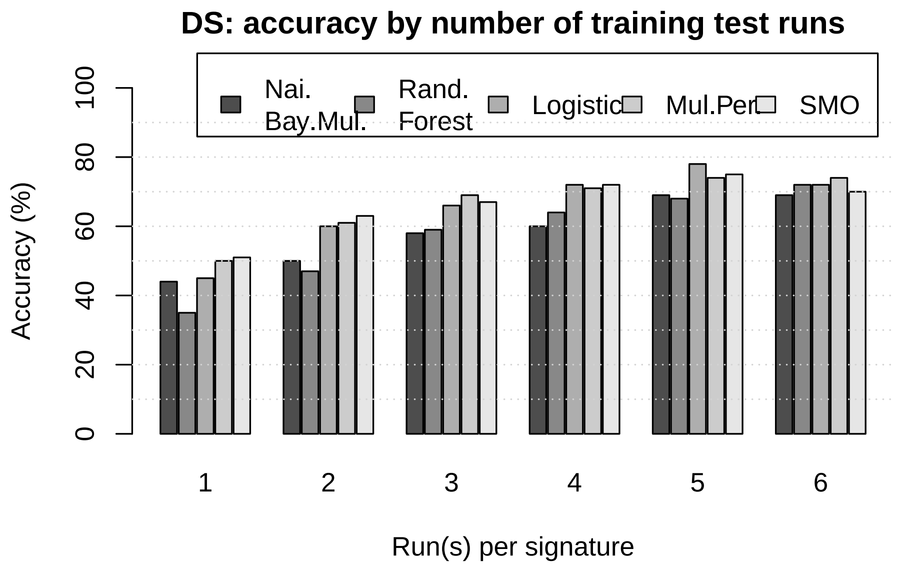
Firstly, in most cases, the **accuracy of determining regression-causes increases with the increase in training data size**. Such result is an indication of good learners for any data mining application (of Dayton Research Institute, 1963). Since our approach (Figure 8.1) is iterative, the more iterations there is, the more historical data is available, and thus the better accuracy.

Secondly, for the Commercial 2 case study, we did not reach the top accuracy with six runs in our case study. For J48, which is the best learner for this system, the accuracy increases steadily from 30% when one run per regression-cause is used, to 80% when six runs per regression-cause are used. So the results have not reached a plateau yet. Hence, there is still room for improvement (when additional runs can be added to the repository) in the Commercial 2 case study.

In contrast, we reach a plateau for the DS case study with about four runs. This



(a) Commercial 2



(b) DS

Figure 8.3: Accuracy of the top five learners using different number of test runs per regression-cause for training (RQ2).

probably means that we have enough runs for this case study. For Logistic, MultilayerPerceptron, and SMO, we reach 70% accuracy with four runs per regression-cause. At five runs per regression-cause, we do not have any additional gain in accuracy, thus indicating that we have reached the maximum accuracy for the three learners. At six runs per regression-cause, the accuracy of MultilayerPerceptron does not improve. Logistic and SMO's accuracy even decreases. Such results are evidence of over-fitting (of Dayton Research Institute, 1963).

## 8.6 Application on Real Life Performance Regression

To collect feedback from the performance testers, we applied our proposed approach on a few recently available test runs of the commercial system as a proof of concept.

**Setup:** Since we cannot build a historical repository because of the unavailability of certain data, we decided to use the same injected runs as we used in RQ1 and RQ2 as our historical test repository. These test runs are hence considered as synthetic test runs.

We asked two performance testers of the commercial system for three test runs with actual performance regressions. All three runs have performance regressions as confirmed by the performance testers of the commercial system in a recent testing cycle. The regression-cause has been confirmed to be similar to one of the regression-causes that exist in our repository. We also obtain three runs of the previous version to use as baseline.

For each of the three runs, we apply our approach to determining the regression-cause using the top five learners in Table 8.3. Similar to RQ2, we train the learners with one randomly chosen run per regression-cause. Then we train with data from two, tree, four, five, and all 6 runs for each regression-cause. For each of the three new test runs we measure the accuracy of each of the learners (i.e., whether the predicted regression-cause is the actual regression-cause). We repeat this procedure 6 times, so that in the case where just one run is chosen per regression-cause, there is chance for each of the 6 run for that regression-cause to be chosen.

**Results:** The accuracy is recorded in Figure 8.4. The first set of bars show the accuracy when we use only one run per regression-cause. The next set of bars show the accuracy when we use two runs per regression-cause and so on.

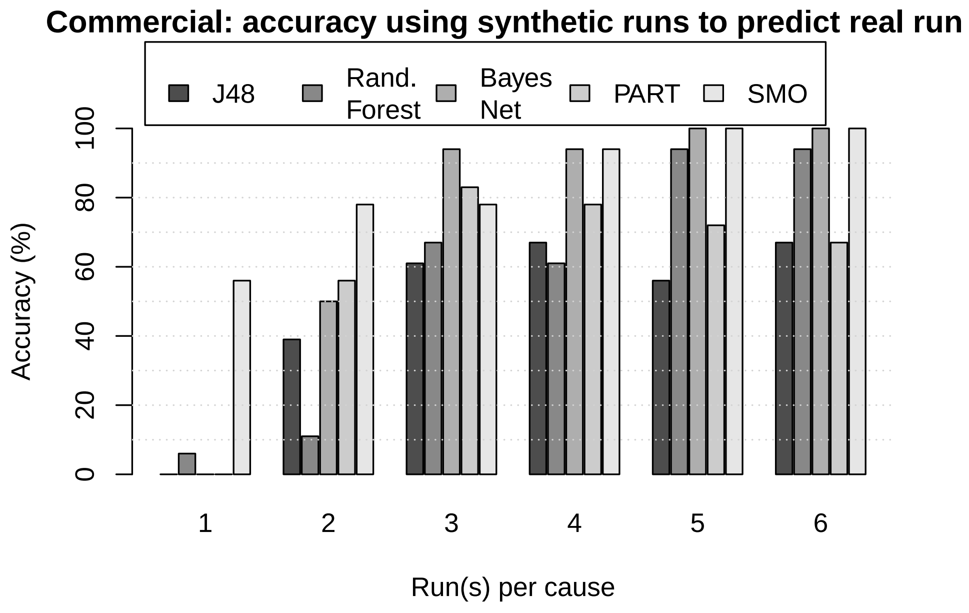


Figure 8.4: Accuracy of the top five learners when using the synthetic runs to determining the regression-causes of three actual test runs of the commercial system.

For these particular runs with actual performance regression, our approach can accurately determine the actual regression-cause using the repository of synthetic test runs. Both BayesNet and SMO reach 100% accuracy when only five test runs per regression-cause are used as training data. At four runs per cause, both learners reach 94% accuracy already.

The results support the possibility of adopting our approach in an industrial setting. Since not many software system have a performance test repository with verified regression, one can boot-strap our approach by creating a synthetic repository of test runs with injected problems (due to various causes). Using this repository, our approach can be adopted without having to wait for test runs with actual performance regressions.

## 8.7 Conclusion

This chapter proposes a new type of software repository and demonstrated its value through an industrial setting. This repository of performance test runs allow us to automatically determine the cause of performance regressions in new test runs.

We conducted two case studies to develop and evaluate our approach. The results show that our approach can accurately suggest (up to 80% accuracy) the regression-cause with a very small training dataset (sometimes with as few as three or four test runs per regression-cause). Moreover, this approach can be bootstrapped using synthetic test runs with injected problems.

The results thus far are encouraging. One possible avenue for future research is to adopt a code mutation approach which randomly injects regressions into various parts of the code. Then, we can use the same approach we have here to determine the different regression-causes in different areas of the code, thus, eliminating the need for prior tests in our approach. The testers can use such mutation injections to rapidly create their repository.

Table 8.3: Success rates of different learners for determining performance regression-causes using violation ratios (RQ1).

| Machine learners | Class | Com. 2 | | DS | | L.G. | C.G. |
|---|---|---|---|---|---|---|---|
| | | A. | G. | A. | G. | | |
| Random $r$ (Eq.(4.3)) | | 20% | 0 | 11% | 0 | | |
| J48 (Quinlan, 1993) | D.Tree | **80%** | **4.00** | 56% | 4.95 | 4.48 | 4.60 |
| LMT (Landwehr et al., 2005) | D.Tree | 57% | 2.83 | 67% | 5.94 | 4.39 | |
| R.Forest (Breiman, 2001) | D.Tree | 70% | 3.50 | 72% | 6.48 | **4.99** | |
| R.Tree (Frank and Kirkby, 2012) | D.Tree | 67% | 3.33 | 64% | 5.76 | 4.55 | |
| N.Bayes (John and Langley, 1995) | Bayes | 53% | 2.67 | 62% | 5.58 | 4.12 | 4.39 |
| N.Bayes Mul. | Bayes | 63% | 3.17 | 68% | 6.12 | 4.64 | |
| BayesNet | Bayes | 70% | 3.50 | 59% | 5.31 | 4.41 | |
| Dec.Tab. (Kohavi, 1995) | Rule | 60% | 3.00 | 39% | 3.42 | 3.21 | 3.70 |
| JRip (Cohen, 1995) | Rule | 47% | 2.33 | 57% | 5.13 | 3.73 | |
| PART (Frank and Witten, 1998) | Rule | 77% | 3.83 | 50% | 4.50 | 4.17 | |
| IBk (Aha et al., 1991) | Lazy | 63% | 3.17 | 63% | 5.58 | 4.37 | 3.82 |
| KStar (Cleary and Trigg, 1995) | Lazy | 63% | 3.17 | 63% | 5.58 | 4.37 | |
| LWL (Frank et al., 2003) | Lazy | 53% | 2.67 | 31% | 2.79 | 2.73 | |
| Logistic (Cessie and Houwelingen, 1992) | Reg. | 57% | 2.83 | 72% | 6.48 | 4.66 | 4.44 |
| Sim.Logis. (Landwehr et al., 2005) | Reg. | 50% | 2.50 | 67% | 5.94 | 4.22 | |
| Mul.Perc. (Minsky and Seymour, 1969) | Neu.Net. | 57% | 2.83 | **74%** | **6.66** | 4.75 | **4.87** |
| SMO (Platt, 1999) | Neu.Net. | 73% | 3.67 | 70% | 6.30 | 4.98 | |

(A.) Accuracy (Equation (4.1)) (G.) Gain (Equation (4.2))
(L.G) Learner Avg. Gain (C.G.) Class Avg. Gain

Table 8.4: Variable importances (VI) of the counters in the RandomForest decision tree (in $10^{-2}$)

| **VI** | **Counter** | **VI** | **Counter** |
|---|---|---|---|
| .93 | SQL IO R B/s U | .34 | Tom. IO W B/s U |
| .78 | SQL IO W B/s L | .33 | Tom. pool paged U |
| .78 | Tom. IO data B/s L | .31 | Tom. pri. time L |
| .76 | Tom. IO W B/s L | .31 | Tom. working set L |
| .69 | SQL work. set U | .28 | SQL IO data O/s L |
| .66 | SQL IO W O/s L | .28 | SQL IO R B/s L |
| .64 | Tom. IO data B/s U | .27 | Tom. IO other B/s U |
| .53 | SQL IO R O/s U | .24 | SQL IO R O/s U |
| .47 | SQL IO data O/s U | .23 | Tom. page faults/s U |
| .46 | Tom. page faults/s L | .22 | Tom. pool nonpaged B U |
| .35 | SQL IO data B/s L | .19 | SQL User Time U |

R=read W=write B=bytes O=operations U=Upper vio. ratio L=Lower vio. ratio

# Part IV

# Applying Control Charts in Practice

# CHAPTER 9

# Study 4: An Experience Report of Applying Our Control Charts Based Approaches

## 9.1 Introduction

The results in Study 1 and Study 2 (Chapter 6, p.73 and Chapter 7, p.82) show that control charts can be used to identify performance regressions in performance load tests. The results in Study 3 (in Chapter 8, p.96) show that control charts can be used to determine the causes of identified performance regressions. While the results are positive, we want to understand if our control charts based approaches can be applied into practice.

In this chapter, we first present the feedback from our industrial partners on our control charts based approaches. Then we describe our experience in applying our control charts based approach into the Commercial 2 software system. We work closely with the performance team of the Commercial 2 software system. During this time, we observe their activities. We first try to apply our approach to identify test runs with performance regression (Study 1 and Study 2's approach) on the performance regressions that the teams discovered. Then we learn that they also need to identify the counters that exhibit the regressions. So we derive automated approaches that are based on control charts to help the team in these tasks.

Our experience is loosely grouped into three parts:

- Part 1: During the course of our research, we give regular presentations of our results to our industrial partners at the Commercial 1 and Commercial 2 project. We present here some of the feedback we got for the approaches.
- Part 2: We shadow the Commercial 2 performance testing team for one release. During this release, the testers discover and correct three performance regressions. Similar to Study 1 and Study 2, we first apply our control charts based approach to identify the runs with performance regressions. Then, we also derive an approach to identify the performance counters that exhibit regression behaviour. During this time, we also replicated approaches from previous research and compared the results with our control charts based approaches.
- Part 3: Finally, we work with the performance testers of the Commercial 2 project to build a tool that creates a regression analysis report automatically for all the test runs of the Commercial 2 software system.

Table 9.1: Practitioners' feedback on the three approaches

| **Approach** | **Advantage** | **Disadvantage** |
|---|---|---|
| Foo et al. (Foo et al., 2010) | Provide support for root cause analysis of bad runs | Complicated to explain |
| Malik et al. (Malik, 2010) | Compresses counters into a small number of important indices | Complicated to communicate findings due to the use of compressed counters |
| Control charts | Simple and easy to communicate | |

## 9.2 Part 1: Feedback from Industrial Partners

### 9.2.1 Feedback on Identifying Performance Regressions in Study 1 and 2

We seek feedback from the performance testers of the Commercial 1 and 2 software systems on our control charts based approach to identify performance regressions. Prior to helping us with this research, the testers had worked with Malik et al.'s PCA based approach (Malik, 2010) and Foo et al.'s discretization based approach (Foo et al., 2010). Thus their feedback on our approach is relative to these two approaches.

The feedback is summarized in Table 9.1. In general, the advantage of our approach compared to the other two approaches is the simplicity and intuitiveness. Control charts quantify the performance quality of a piece of software into a measurable and easy to explain quantity, i.e., the violation ratio. Thus performance testers can easily communicate the test results with others. It is much harder to convince the developers that some statistical model determined the failure of a test than to say that some counters have many more violations than before. Because of that, practitioners felt that our control charts based approach has a high chance of adoption in practice.

The practitioners noted that a disadvantage of our approach is that it does not provide support for root cause analysis of the performance regressions. Foo et al. (Foo et al., 2010), through their association rules, can give a probable cause to the performance regressions. That feedback inspired us to conduct Study 3 (*see* Chapter 8, p.96) in which we derive a control charts based approach to determine the probable causes of the regressions.

The practitioners also noted that our approach to scale the load using a linear model might not work for systems with complex queuing. Instead, it might be worthwhile exploring the use of Queuing Network models to do the scaling for such systems. We have not found a software system that would exhibit such a queuing profile to better understand the negative impact of such a profile on our assumptions about the linear relation between load inputs and load outputs.

The practitioners want to see a comparison of our control charts based approach and the other two approaches on one of the performance regressions. So, in Part 2 of this study, where we apply our control charts based approach in real performance regressions, we also apply the PCA based and the discretization based approaches along with a couple of other approaches to compare.

### 9.2.2 Feedback on Regression-Cause Analysis of Study 3

Apart from the high accuracy of our approach, the feedback from the performance testers was positive. The testers were impressed with the potential time savings. They compared our approach with a push information mechanism such as Amazon's book suggestions. Having the suggestions would help them completing the analysis earlier since they do not have to go through all the data.

More importantly, the testers said that the violation ratios that we produced in Step A (Section 8.3) are simple and easy to explain. It is easier to adopt an approach which can be explained easily to other teams.

The testers were also impressed that we only use the standard resource counters which are available on any operating system. In reality, the software also implements their own counters such as queue sizes or response-times of different processing threads. Such counters could potentially improve the accuracy of finding regression causes by our approach. Future work should examine this.

However, the testers also noted some potential limitations of our approach:

- First, we demonstrated our approach on only one of their regressions. A larger industrial study is required to justify the investment cost of applying our approach.

- Second, we did not verify the accuracy of our approach when there are multiple regression-causes in the same test run. However, identifying standalone regression-causes is also helpful to developers. Future works should examine the case of multiple regression-causes.

- Third, although the results show that any machine learner performs better than the random predictor, the training set must have sufficient number of test runs for each regression-cause. If the regression-cause of the new test is new to the software system, there would be no similar runs in the historical test repository (Figure 8.1). Thus, the learner will output the closest regression-cause that it can identify. This can be misleading. The testers can potentially mitigate this risk by introducing synthetic runs for possible regression-causes into the training set, but there is no guarantee.

- Fourth, while our approach can save time, from manually checking all the possible regression-causes, it might create unwanted distractions from the mismatched regression-causes when the approach fails to produce a correct match.

- Fifth, as the software evolves, the regression data in the repository might become invalid over time, which might cause the learners to produce misleading suggestions. A longitudinal study is required to understand the effect of evolution on the accuracy of older runs.
- Finally, using our approach we can identify the type of regression, but not the location of it. Although finding the location would be be even more helpful to the performance testers, knowing that regression type is the first step in the right direction.

Even with these limitations, the possibility of saving time makes our approach very attractive to the testers.

## 9.3 Part 2: Identify Test Runs with Performance Regressions and The Counters That Exhibit Regression Behaviour

We work with the performance testing team of the Commercial 2 software system for one release. During the release, the team discovered and fixed three performance regressions. So we take the test run data from the three performance regressions and apply our control charts based approaches with two purposes:

- First, we want to apply our control charts based approach that is described in Study 1 and Study 2 to see if our approach can identify the test runs with performance regressions. As mentioned Part 1, we also want to compare our approach with other approaches including the PCA based (Malik, 2010) and discretization based (Foo et al., 2010) approaches. As mentioned in Chapter 2 (p.6), there are three types of approach to reduce performance counters data: reducing individual counters data, filtering-out-by-mean, and combining counters data. Control charts are a type of reducing individual counters data. So we first apply our control charts based approach on the three performance regressions. Then we derive approaches that are inspired by prior research. We apply the approaches on the three performance regressions. We compare the results with our control charts based approach.

- Second, the team also wishes to identify the performance counters that exhibit regression behaviour. So we derive a control charts based approach that can automatically identify the counters that exhibit the behaviour of the uncovered performance regression. As mentioned in Study 3 (Chapter 8), after the test run has been identified as a failed test, testers would need to investigate the root causes of the performance regressions in the failed run. Being able to automatically flag the performance counters that exhibit regression behaviour will help the testers to speed up the process. We also try to replicate approaches from previous research to identify the counters that exhibit regression behaviour and compare the results with our approach.

### 9.3.1 Data Collection

#### Test Pairs

As explained in the background section, a performance regression test includes one or more baseline test runs of the previous version and one or more target test runs of the new version of the software. For each regression, we collect the performance counters of the following test runs:

- **Baseline run 1:** This is a run of the previous version of the software that was performance tested and released into production.

- **Baseline run 2:** This is another run of the same version as Baseline run 1.
- **Target run:** This is the run of the new version of the software with a regression.

Similar to the evaluation in Section 7.3, from these three runs, we obtain two pairs of runs:

- **Normal pair:** The first pair, Baseline run 1 vs Baseline run 2, is the normal pair. We expect no regression in this pair.
- **Regression pair:** We expect a regression in the second pair, Target run vs Baseline run 1 or Baseline run 2. There is a regression in this pair.

We not only know which test pair is a regression pair (as in Study 2), we also know which counters exhibit the detected performance regression for each regression pair. This is a crucial piece of information for the root cause investigation that were conducted by the testers. We call this the *matched counter set* for the detected performance regression.

#### Studied Performance Regressions

We use three performance regressions that were identified during the testing of the Commercial 2 software system. As mentioned in Section 4.1, Commercial 2 is an ultra-large scale system with thousands of nodes. The operating of Commercial 2 costs multi-million dollars per year. Each release of the software is performance tested to prevent performance regressions.

These three regressions were caught by the performance testing team. After the regressions were identified, the performance testers conduct a root cause analysis process to find the cause of the performance regression. They identify all the performance counters that exhibit regression behaviour. Then, using their knowledge of the system, they work with other teams to isolate the changes that might have caused the performance regression. The regressions were corrected/justified before the release was deployed in production.

**Performance regression 1:** For performance regression 1, the performance testers, through manual inspection of the counters, found that this particular new version has multiple performance regressions. Figure 9.1 shows the test report from the tester's analysis:

1. CPU utilization increased (15 counters)
2. Java scavenger garbage collection increased (1 counters)
3. Ethernet sent increased (1 counters)
4. Java process CPU time increased (2 counters)
5. Loopback sent and received increased (1 counters)

Figure 9.1: The counters that exhibit performance regression in performance regression 1.

99 performance counters are collected for each test run. The performance testers of the Commercial 2 system analyze the counters manually. They found 20 counters that were exhibiting regression behaviour as outlined in Figure 9.1. Each of the items that are listed in Figure 9.1 associates with one or more related performance counters. For example, there are many counters that are related to CPU utilizations such as CPU idle, CPU utilization, CPU system utilization, or CPU user utilization. Together, the 20 counters constitute the matched counter set for performance regression 1. We use this matched counter set for evaluation 2.

**Performance regression 2:** According to the report for Performance regression 2, there are 22 counters that show symptoms of regressions. There are 295 performance counters that are collected for each test run.

We note that the number of performance counters that are collected during for the test runs varies even though it is the same system under-test. Different labs have different setups to record and archive the metrics. The metrics are also being added to the system as new features are implemented.

**Performance regression 3:** This regression's report indicates 18 counters that exhibit regressions. There are in total 311 counters that were collected.

### 9.3.2 Evaluation Methodology

#### Approaches that We Evaluated

We evaluate the following approaches:

- **Our control charts based approach:** We apply our control charts based approach on the three performance regression.
- **Discretizing counter approach:** We apply an approach that discretize the performance counters as in previous research (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Foo, 2011; Foo et al., 2010; Wiertz et al., 2014).
- **Hierarchical clustering approach:** Inspired by previous research (Ahn and Vetter, 2002; Eeckhout et al., 2003a,b; Nickolayev et al., 1997; Zhao et al., 2009), we derive an approach that uses a clustering algorithm to reduce and analyze performance counters data.
- **Filtering-out-by-mean approach:** Inspired by previous research (Malik, 2010; Malik et al., 2010; Nickolayev et al., 1997; Vetter and Reed, 1999; Zhao et al., 2009), we derive an approach that filters out counters by the mean to reduce the number of counters that need to be analyzed.
- **Using PCA to combine counter approach:** Inspired by previous research (Malik, 2010; Malik et al., 2010), we derive an approach that uses PCA to combine

the counters data.

For the discretizing, we replicate the original research's methodology using our performance regressions' counter data. This is possible because the original research's goal was also detecting performance regressions in load tests. For hierarchical clustering, filtering-out-by-mean, and PCA based approaches, we derive approaches that are inspired by the original research. The original research' goal was not detecting performance regressions, so we cannot just replicate these approaches as is.

We note that there are other approaches that can be adapted to identify performance regressions. However, due to the time constraint of the thesis, we evaluate only the four that we can adapt and implement.

### Evaluation Criteria

For each of the approaches, we use two criteria for our evaluation:

- **Evaluation Criterion 1 - Distinguish the regression and the normal pair:** We apply the approaches on the normal and regression pairs. We check if the approaches can distinguish between the regression pair and the normal pair. This is the same criteria that we use for Study 1 and 2.

- **Evaluation Criterion 2 - Identify the matched counters:** As discussed, from our interaction with the testers of the Commercial 2 system, we found that the ability to automatically identify the counters that exhibit regression behaviour, i.e., changed compared to the baseline, is crucial for further analysis. Thus, for each performance regression, we evaluate the approaches' ability to identify the matched counters in the *matched counter set*. This is a new criterion compared to Study 1 and 2. We use each approach to rank the performance counters. The counter with the highest chance of showing the regression is at the top. Then we create a receiver operating characteristic (ROC) curve for each approach (Hastie et al., 2008). ROC curves are used to compare performance of binary classifiers. Our binary operation is whether a counter in the ranked list is in the *matched counter set* or not. We compare the ROC curve of each approach with the ROC curve of the control charts based approach. If the approach is better in identifying the matched counters, then its ROC curve should be higher than the control charts based approach's ROC curve and vice versa.

### 9.3.3 Results

#### Our Control Charts Based Approach

**Evaluation Criterion 1:** Table 9.2 shows the violation ratios of the highest changed counters for the normal and regression pairs of Performance regression 1. On the left is the counters' average violation ratio (*see* Section 3.2.2) for the Normal pair. On the right is the counters's average violation ratio of the Regression pair. We only show the top ten counters by the average violation ratios.

For the Normal pair, the average violation ratio is relatively low and there are no counters of the *matched counter set* in the top ten counters. Some of those counters are actually not suitable for comparison. For example, memory used, memory used kb, memory free, and memory free kb are counters that would always increase or decrease even when there is no regression. This is because the Linux operating system uses memory for caching IO pages. The longer the system is up, the higher the cache size, the higher the used memory, and the lower the free memory. We discuss such always changing counters in detail in Section 5.4. So if we discount these counters from Table 9.2, the average violation ratios for the Normal pair would be

Table 9.2: Using control charts to identify regression in Performance regression 1

| Normal pair | | Regression pair | |
|---|---|---|---|
| **Counter** | **Avg. Vio.** | **Counter** | **Avg. Vio.** |
| memory used | 0.38 | **NET lo sent bytes per second** | 0.5 |
| memory free | 0.35 | memory free kb | 0.5 |
| unicast num xmits | 0.35 | **OS process cpu time** | 0.5 |
| memory free kb | 0.34 | unicast xmit table num moves | 0.5 |
| memory used kb | 0.34 | **NET eth0 sent bytes per second** | 0.5 |
| OS free physical memory size | 0.34 | **GC ps scavenge collectiontime** | 0.5 |
| GC ps scavenge collectiontime | 0.34 | unicast xmit table num purges | 0.5 |
| GC ps scavenge lastgcinfo id | 0.32 | OS free physical memory size | 0.5 |
| unicast num acks received | 0.31 | memory used | 0.5 |
| memory cached kb | 0.31 | fd number of heartbeats sent | 0.5 |

**Bold** - Counter is in the *matched counter set*

even lower. On the other hand, the average violation ratios of the Regression pair is high. All the top 10 counters have average violation ratios of 0.5. Hence, those counters were either 100% higher or lower than the control limits. So one can already separate the Normal and Regression pair by just looking at the top counters' average violation ratios.

Table 9.3 shows the average of all the average violation ratios of all the counters for all the performance regressions. As we can see, similar to the Performance regression 1's result, the violation ratios for the normal pairs are all low in comparison to the regression pairs in both Performance regression 2 and 3.

**Evaluation Criterion 2:** We create ROC curves to evaluate the control charts performance in identifying the matched counters set. ROC curves are commonly used to evaluate classifiers' performance (Hastie et al., 2008). For example, let us assume that there are five counters in total (e.g., a to e) and that the matched counter set has two counters (e.g., a and b). If a classifier C1 outputs: b, d, a, c, e, then the

Table 9.3: Evaluation criterion 1 - Using control charts based approach to identify the target (with regression) runs

| **Test pair** | **Average of violation ratio** |
|---|---|
| Performance regression 1 - Normal | 3.82% |
| Performance regression 2 - Normal | 4.35% |
| Performance regression 3 - Normal | 5.60% |
| Performance regression 1 - Regression | 15.29% |
| Performance regression 2 - Regression | 9.84% |
| Performance regression 3 - Regression | 9.69% |

ROC curve for C1 will be: 1, 1, 2, 2, 2. If a classifier C2 outputs: b, a, c, d, e, then the ROC curve for C2 will be: 1, 2, 2, 2, 2. Hence, C2 performs better than C1. C2's ROC curve will be higher than C1's ROC curve.

Figure 9.2 shows the ROC curves for finding the matched counters in each performance regression for both the normal and regression pairs. Our approach ranks the counters by the highest average violation ratio first. The green line is the ROC curve for the regression pair. We also show the ROC curve in red for the normal pair. We show the ROC curve for the normal pair as well because it is our control group in this evaluation. If the ROC curve of the normal pair has the same pattern as the ROC curve for the regression pair, our approach has failed.

As we can see from Figure 9.2, for all the regression pairs, our approach can match most of the counters of the matching set. The green lines reach 100% faster. The red lines, on the other hand, are trailing along the diagonal line as expected.

### Discretizing Counter Approach

Similar to our control charts based approach, other studies (Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Foo, 2011; Foo et al., 2010; Wiertz et al., 2014) also reduce individual counter data. So we replicated their approach on the three performance regressions.

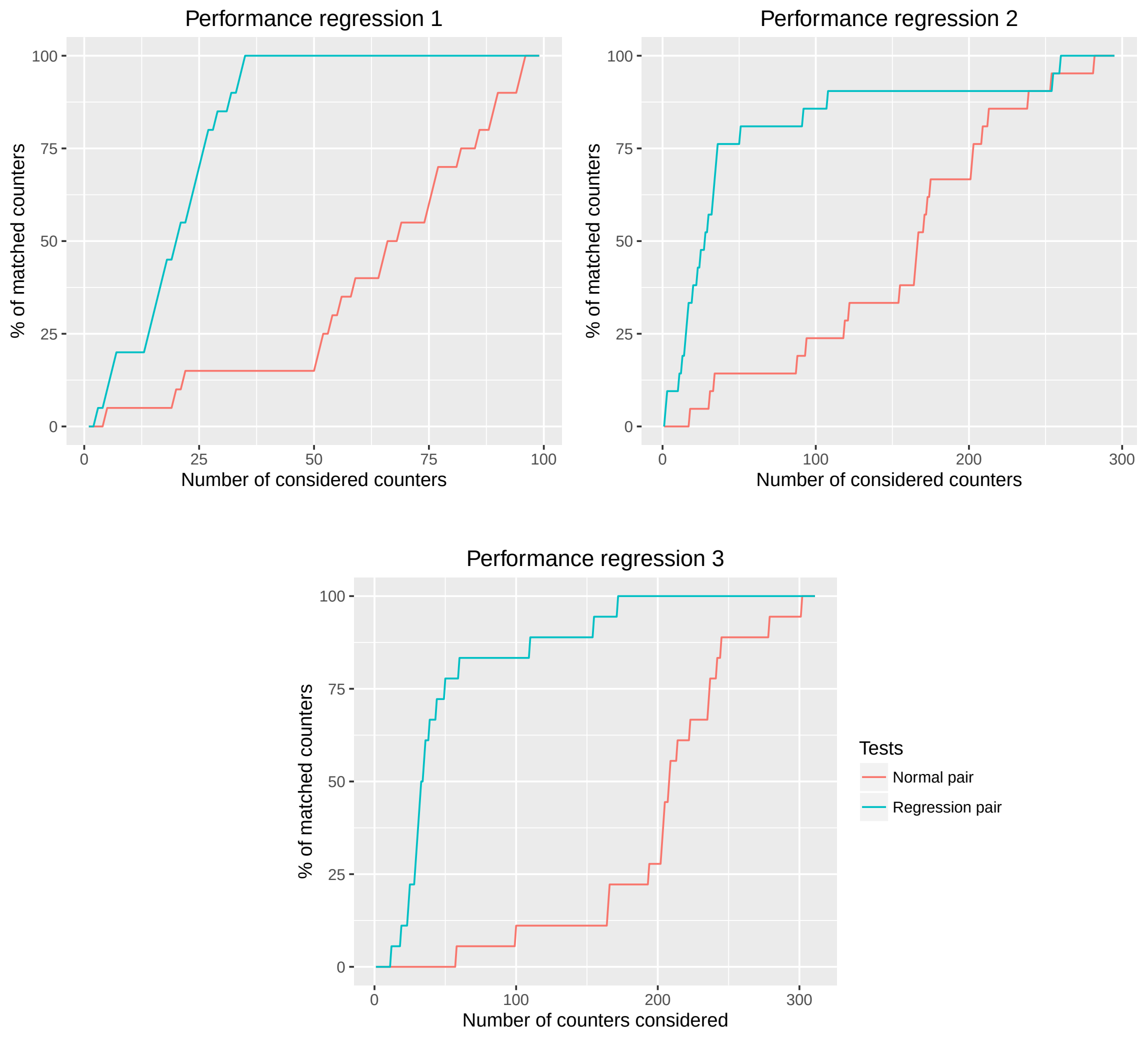


Figure 9.2: Evaluation criterion 2 - Identify the matched counter set using our control charts based approach

For each counter, we calculate the range of all possible values using the counter data in the baseline and target runs. Then we divide the range into three equal ranges that we call *low*, *normal*, and *high*. Then, for each run, we classify the counters value into low, normal, and high based on the average value of the counter during the run. For example, the range of CPU utilization is 0 to 400% (4 cores). If the average CPU utilization of the baseline run is 230%, then we mark the counter as normal. If the average CPU utilization of the target run is 340%, we mark the counter as high.

For each counter pair, we calculate the level of changes from the baseline run to the target run of that counter. For example, if the baseline CPU utilization is normal and the target CPU utilization is high, we mark that counter as changed 1 level (normal to high). If the counter changes from low to high, then we mark that counter as changed 2 levels. The invert is marked the same. For example, if a counter changes from high to low, we mark the counter as changed 2 levels as well.
**Evaluation Criterion 1:** Using the change level, we detect if the test run is a normal pair or a regression pair. In a normal pair, there would be relatively fewer counters that change and, if change happened, the level of change would be relatively lower. In a regression pair, there would be relatively more counters that change and the level of change would be relatively higher.

Table 9.4 shows the results when we apply the discretizing approach on the three regressions. The second column shows the number of counters that do not change between the baseline and the target test of the pair. The third column shows the number of counters that change 1 level, i.e., low to normal and normal to high or vice versa. The fourth column is the number of counters that change 2 levels, i.e., low to high or vice versa.

As we can see from Table 9.4, for Performance regression 1 and 3, the normal pairs have more counters that do not change compared to the regression pair (84

Table 9.4: Evaluation criterion 1 - Using discretizing counters approach to identify the target (with regression) runs

| **Test pair** | **# counters that don't change** | **# counters that change 1 level**[1] | **# counters that change 2 levels**[2] |
|---|---|---|---|
| **Regression 1** | | | |
| Normal | 84 | 12 | 3 |
| Regression | 66 | 24 | 9 |
| **Regression 2** | | | |
| Normal | 219 | 59 | 17 |
| Regression | 250 | 40 | 5 |
| **Regression 3** | | | |
| Normal | 242 | 39 | 5 |
| Regression | 215 | 63 | 8 |

[1] E.g., normal to high or low to normal or vice versa
[2] E.g., low to high or vice versa

versus 66 and 242 versus 215 ). The regression pairs have more counters that change either 1 level (24 versus 12 and 63 versus 39) or 2 levels ( 9 versus 3 and 8 versus 5). However, for Performance regression 2, the normal pair has less unchanged counters (219 vs 250) but have more 1 level (59 versus 40) and 2 levels (17 versus 5) changed counters compared to the regression pair. So the discretizing counters approach did not perform well for Performance regression 2.

**Evaluation Criterion 2:** We apply evaluation criterion 2 using the discretizing counters approach. For each of the performance regressions, we rank the counters according to the number of changed levels. If a counter has a higher level of changes, that counter should be in the matched counter set.

Figure 9.3 shows the ROC curves for the discretizing counters based approach compared to our control charts based approach on the three performance regressions. The green curve in each graph is for the discretizing counters approach. The

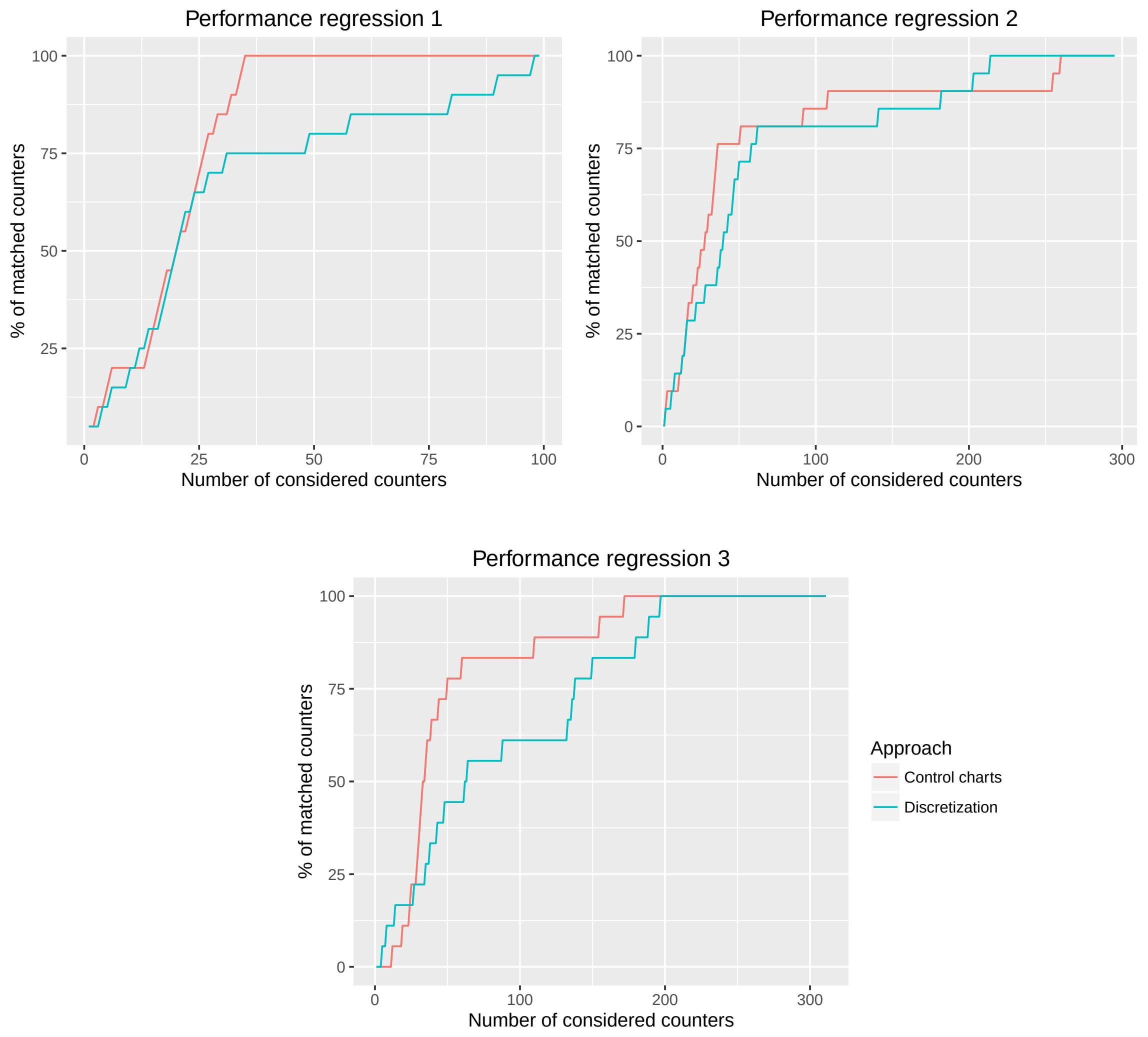


Figure 9.3: Evaluation criterion 2 - Identify the matched counter set using a discretizing counters approach versus our control charts based approach

red curve is for our control charts based approach which ranks the counters by average violation ratio as in Figure 9.2. For clarity, we do not show the ROC curves for the normal pair in the same graph as in Figure 9.2. Both curves are for the regression pair.

As we can see in Figure 9.3, our control charts based approach outperforms the discretizing counters approach for Performance regression 1 and 3. As explained previously, the higher the ROC curve, the better the approach is at finding counters in the matched counter set. In Performance regression 2, both approaches perform similarly.

### Hierarchical Clustering Approach

As mentioned in the related work (Section 2), researchers employed different clustering algorithms to reduce performance counters data (Nickolayev et al., 1997; Zhao et al., 2009) or to identify problems in performance counters (Ahn and Vetter, 2002; Eeckhout et al., 2003a,b). So we derive and apply a hierarchical clustering approach on the three performance regressions and compare the results with our control charts based approach.

Hierarchical clustering is commonly used in genome analysis. For example, suppose we have three samples of a gene's genome for a species that are collected from different locations. Because the gene sample are from different locations, they probably evolved in different ways. Hierarchical clustering algorithms are used to determine how far apart the gene samples are in terms of evolution.

The input to a hierarchical clustering algorithm are different lists of items. In case of the gene samples, each gene sample is a list of denine (A), guanine (G), cytosine (C) and thymine (T). Then we have three lists of the same length. A hierarchical clustering algorithm, then, can be used to compute the distance between all the pair of lists. The output is a matrix of distances. In the case of three gene samples, the output would be a 2x2 matrix that indicates the distances between the all gene pair. However, the output is usually visualized using a hierarchical tree. If a pair is further apart, it would be in subtrees that are further apart. If a pair is closer to each other, they would show up in the same or relatively closer subtrees.

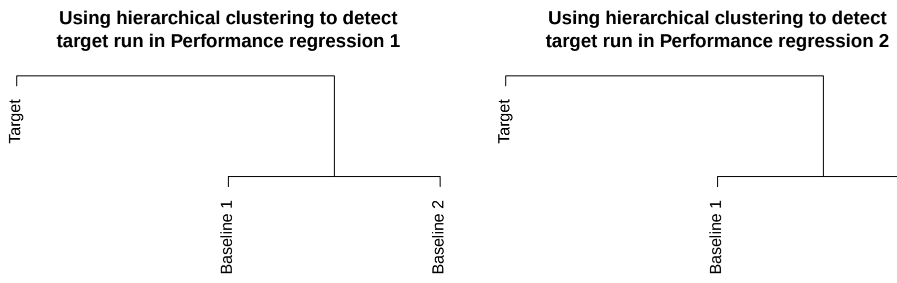
In this section, we pick a hierarchical clustering algorithm called *complete-linkage clustering* (Everitt et al., 2001) and derive an approach to identify the regression run and identify the matched counters. Complete-linkage clustering is one of the most established clustering algorithms.

**Evaluation Criterion 1:** We first try to determine if a hierarchical clustering based approach can identify performance regressions in our three performance regressions.

We have three runs (two normal and one with regression). For each counter in each run, we calculate the mean of the particular counter. So, for each run, we have a list of $n$ values. Then, we apply the complete-linkage clustering algorithm to compute the distances between the runs using the values. The input to the algorithm is the list of means for the three runs. The algorithm will output the distances between the runs. These distances can be, then, visualized as a clustering tree.

If the approach can identify the regression run, we expect to see the two baseline (normal) runs clustered together, i.e., they are closer to each other. The target (with regression) run should stand out as it should be further away from the baseline runs.

Figure 9.4 shows the results when we applied hierarchical clustering to the test runs of each performance regressions. As we can see, the hierarchical clustering approach was able to identify all performance regressions. In each performance regression, the two baseline runs cluster together while the target run with performance regression stands out.


Using hierarchical clustering to detect target run in Performance regression 1
Target
Baseline 1
Baseline 2
Using hierarchical clustering to detect target run in Performance regression 2
Target
Baseline 1
Baseline 2

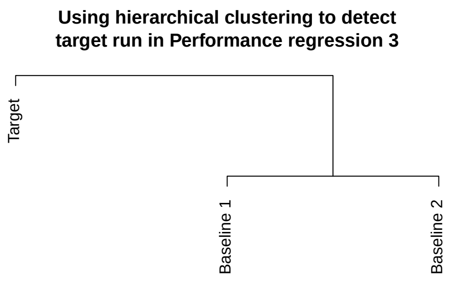

Using hierarchical clustering to detect target run in Performance regression 3
Target
Baseline 1
Baseline 2

Figure 9.4: Evaluation criterion 1 - Identify performance regression runs using a hierarchical clustering based approach

**Evaluation Criterion 2:** To evaluate the ability of the hierarchical clustering approach to identify the matched counters, we apply the same hierarchical clustering algorithm as in Evaluation 1. This time, we use the runs to calculate the distance between the counters instead of using the counters to calculate the distances between the runs. For each counter, we calculate the mean value of the counter for the three runs (two baseline runs and one target run). So for each counter, we have a list of three values. Then we apply the *complete-linkage clustering* algorithm.

Let us consider any pair of counter:

- If a pair of counters is not part of the matched counter set, both should not change for all three tests. Thus, their lists should be relatively constant. Hence, the clustering algorithm should output a relatively closer distance between these two counters.

- If a pair of counters is part of the matched counter set, both counters should change for all three tests. Thus, their lists should be changed together, i.e., the value for the target test (with regression) should be changed on both list. Hence, the clustering algorithm should output a relatively closer distance between these two counters.

- If one counter is part of the matched counter set but the other one is not, then one counter should change but the other one should not, i.e., the value for the target test (with regression) should be changed on one list but not the other. Hence the clustering algorithm should output a relatively larger distance between these two counters.

Thus, if the hierarchical clustering approach can identify the matched counters, we expect to see the matched counters clustered together and the rest of the counters clustered together as well.

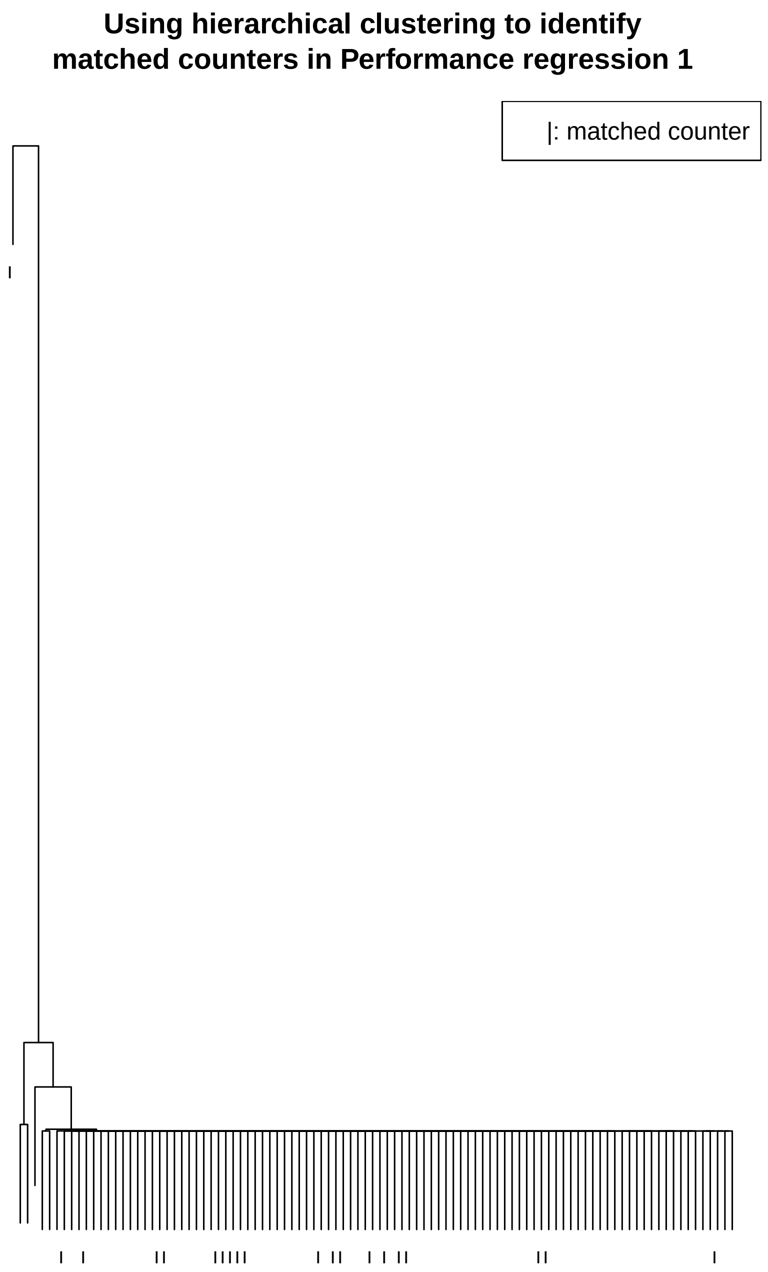


Figure 9.5: Evaluation criterion 2 - Identify the matched counter set using hierarchical clustering for Performance regression 1

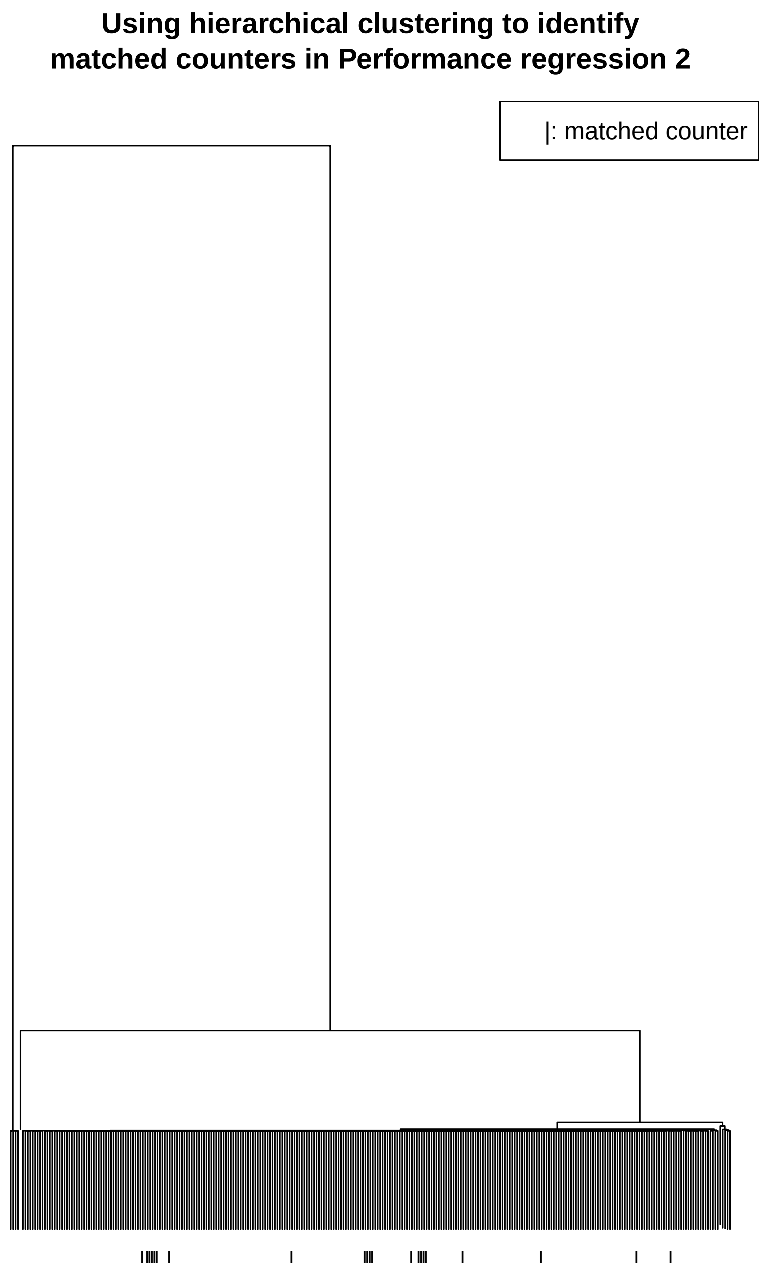


Figure 9.6: Evaluation criterion 2 - Identify the matched counter set using hierarchical clustering for Performance regression 2

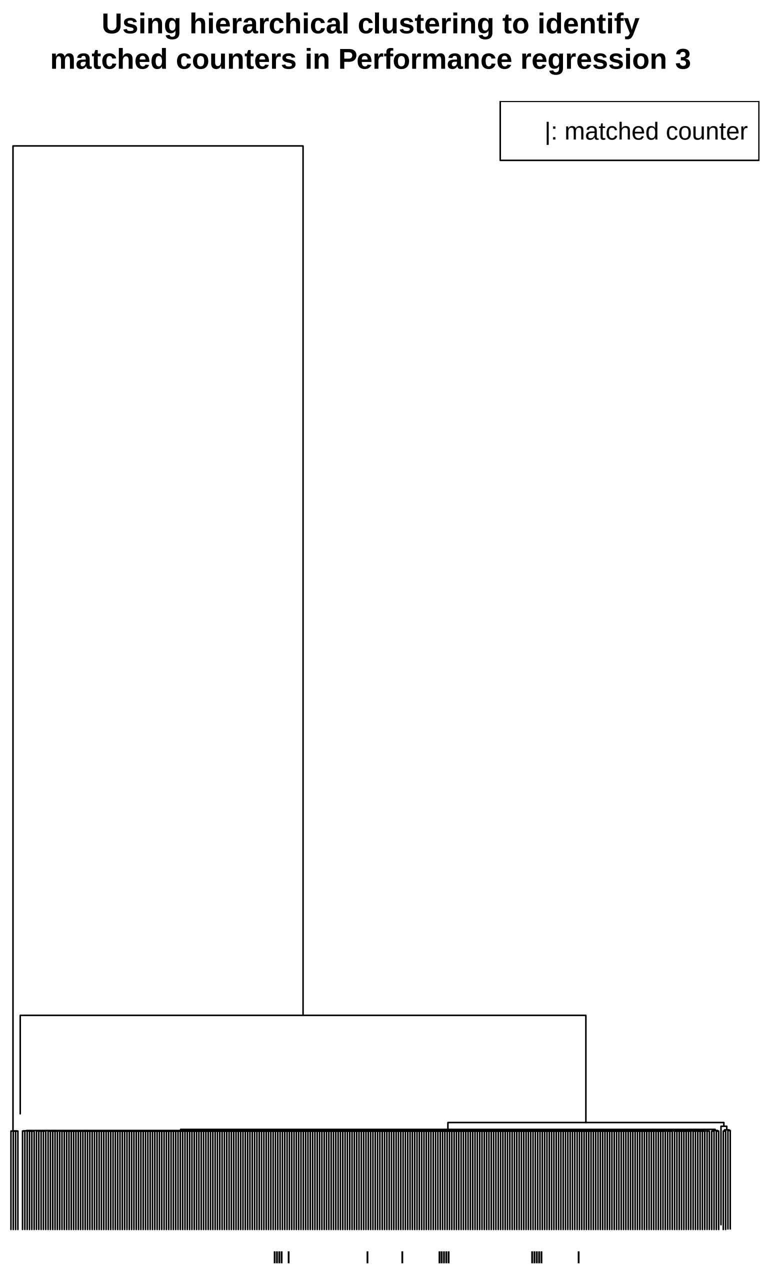


Figure 9.7: Evaluation criterion 2 - Identify the matched counter set using hierarchical clustering for Performance regression 3

Figures 9.5, 9.6, and 9.7 show the clusters that are created using the hierarchical clustering approach on our three performance regressions. We indicate the matched counters with — so we can visually inspect if the matched counters are clustered together.

For Performance regression 1 (Figure 9.5), one of the counters stands out as we expected. However, the rest of the matched counters do not. For the other two performance regressions, the matched counters do not seem to cluster together or separate from other counters. The results show that hierarchical clustering approach cannot be used to identify matched counters.

## Filtering-out-by-mean Approach

As mentioned in the Chapter 3, there are approaches (Malik, 2010; Malik et al., 2010; Nickolayev et al., 1997; Vetter and Reed, 1999; Zhao et al., 2009) that filter out counters to reduce the amount of counters that need to be analyzed. We pick a simple filter out counters approach by using the mean. We call it filtering-out-by-mean approach. We apply the filtering-out-by-mean approach on the three performance regressions.

**Evaluation Criterion 1:** The purpose of filtering counter approaches (Malik, 2010; Malik et al., 2010; Nickolayev et al., 1997; Vetter and Reed, 1999; Zhao et al., 2009) is not identifying performance regressions. Hence we do not perform evaluation 1.

**Evaluation Criterion 2:** We conduct evaluation 2 on the filtering-out-by-mean approach. Filtering-out-by-mean is a simple approach that uses the relative differences to rank the counters. We calculate the mean of the relative differences of each counter pair of the normal and regression pair. The relative difference is defined in Equation 9.1):

$$Relative\ difference = \frac{|T - B|}{B} \tag{9.1}$$

In Equation 9.1, T is the value of the counter in the target test. B is the value of the counter in the baseline test. For example, if the CPU utilization in the baseline run is 80% and the CPU utilization in the target run is 85%. Then the relative difference is (85-80)/80=6.25%.

We rank the counters in descending order by the relative differences. We have a ranked list of counters for each test pair. If a counter appears higher in the list, that counter should be kept after filtering out because that counter changes more significantly compared to the counters lower in the list. Thus, if the approach can identify the matched counters, more of the top ranked counters should match *matched counter set* of the performance regression. Hence, the ROC curve should be the higher toward the left.

We compare the ROC curve of the filtering-out-by-mean approach with the ROC curve of our control charts based approach. The better the approach, the higher the ROC curve.

Figure 9.8 shows the ROC curves for the regression pairs in the three performance regressions for both approaches, i.e., filtering out by control charts' violation ratios and filtering-out-by-means. Because there is a regression in each test pairs on Figure 9.8, both ROC curves should be closer to top left corner, i.e., the top ranked counters should match the ones in the *matched counter set* that are identified by the testers manually.

As we can see in Figure 9.8, both the ROC curves in each graph are indeed closer to the top left corner. Thus, both approaches work reasonably well. However, the ROC curves of the control charts based approach are higher than those of the filtering-out-by-means approach. As we mention earlier, the better the approach, the higher the ROC curve. So, the control charts based approach works better for all three case studies.

Figure 9.9 shows the ROC curves for the normal pairs in the three performance

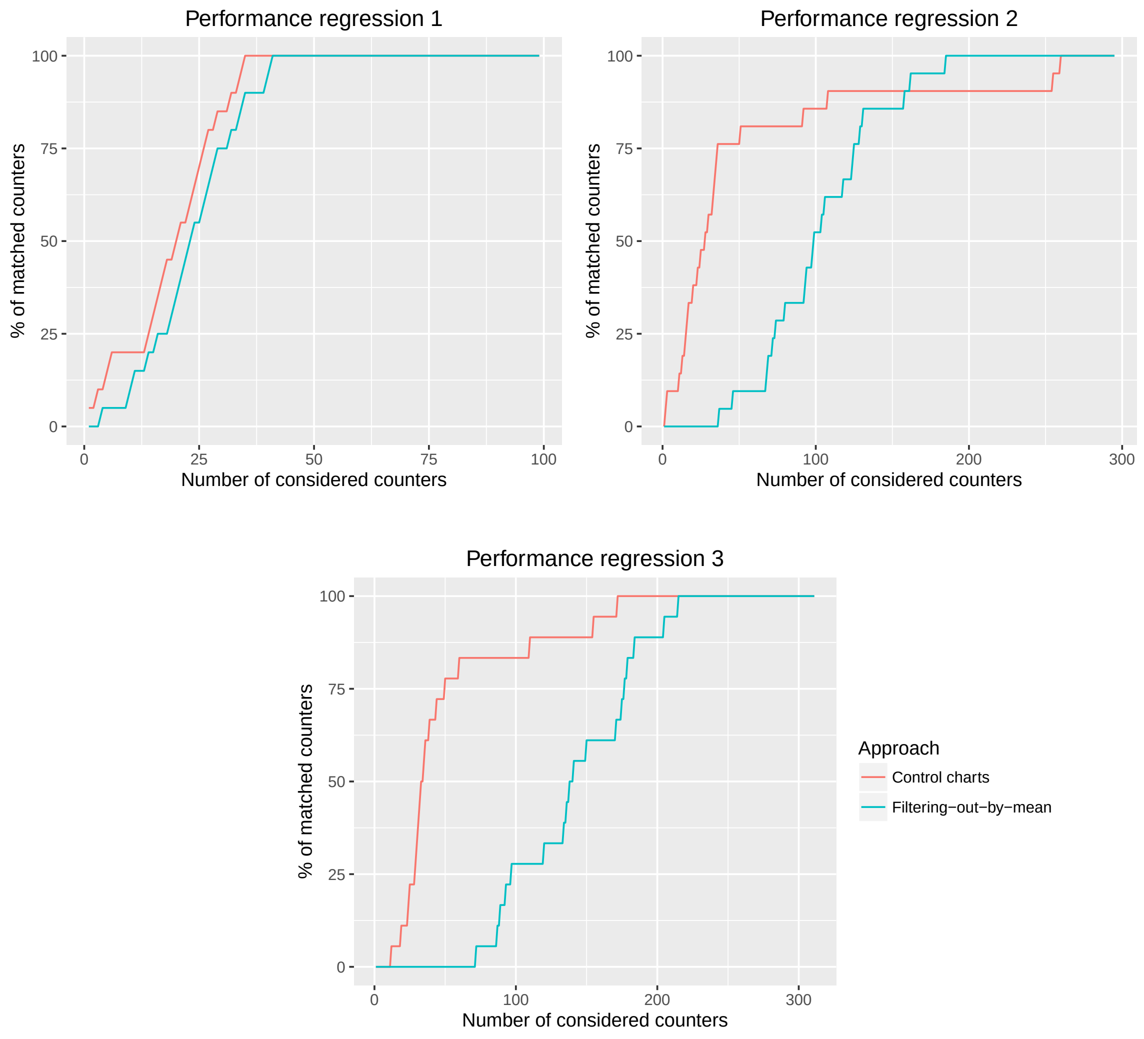


Figure 9.8: Evaluation criterion 2 - Identify the matched counter set using filtering-out-by-mean approach - the regression pairs.

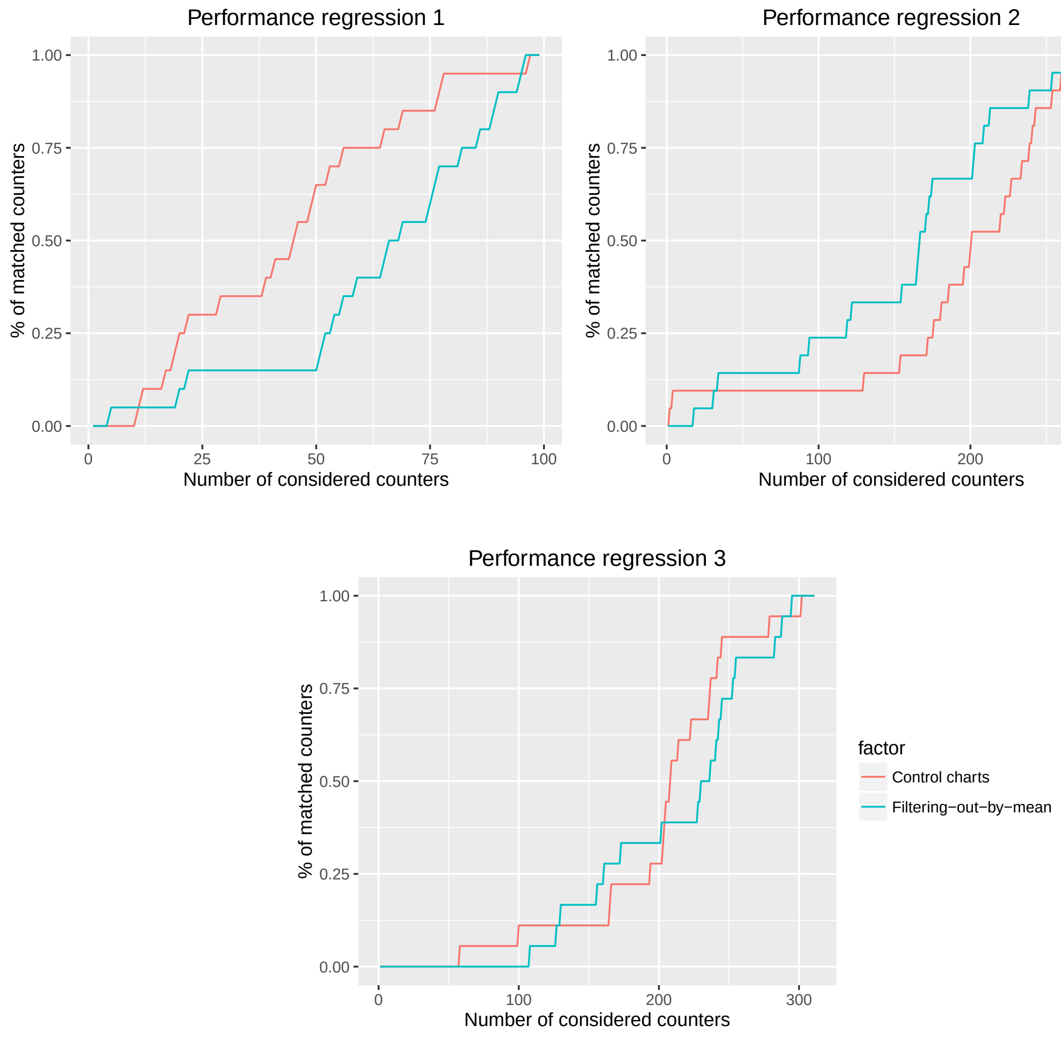


Figure 9.9: Evaluation criterion 2 - Identify the matched counter set using a filtering-out-by-mean approach - the normal pairs

regressions. Because there is no regression in these test pairs, both ROC curves on Figure 9.9 should be close to the diagonal line running from the bottom left corner to the top right corner. As we can see in Figure 9.9, this is indeed the case.

The results in Figure 9.8 and Figure 9.9 illustrate that our control charts based approach works better that the simple filtering-out-by-mean approach. For the cases where there are no regressions, the control charts based approach's ROC curve is relatively closer to the diagonal line. For the cases where there are regressions, the control charts based approach's ROC curve is relatively higher.

### Using PCA Based Approach

**Evaluation Criterion 1:** We derive an approach that uses PCA to identify the runs with performance regression. Our approach is inspired by Malik et al. (Malik, 2010; Malik et al., 2010)'s PCA based approach. The approach has two steps:

- **Step 1 - Compute the principle components:** Use PCA to reduce the performance counters into principal components for both the performance counters of the baseline and target test runs.
- **Step 2 - Analyzing top components' membership:** Analyze the counter membership in the top components on the baseline test and the target test. If the membership is similar, i.e., the same counters present in similar order in the top components of both the baseline and the target test, then there are no regressions. Otherwise, there is a regression.

*Step 1 - Compute the principle components:*

We apply the PCA based approach on all three performance regressions. The output of the PCA algorithm are the components. If there are $n$ counters, the algorithm would outputs $n$ components. Each component can be thought of as a composition of the counters at different weights. If a group of counters are highly related, i.e., they change together, the counters would most likely be included in the same component. If all the counters that changed have high weight in just one component, that component would be the most important component. Usually, most of the changes can be contained in just a few components. So instead of analyzing $n$ counters, one can just analyze those few components. That is the intuition behind PCA.

The degree to which a component contains the valuable information is called *variance explained*. All the $n$ components would explain 100% of the variance. Researchers usually use only the top $k$ components if these $k$ components's variance explained passes a threshold. It is very popular to use the 95% threshold. For example, the first component might explain 76% of the variance. The second component might explain 20% of the variance. The rest of the components would explain the last 4% of the variance. Researchers would only examine the first two components since they already explain 96% of the variance. This is primarily how PCA is used to reduce counter data.

So in our three performance regressions, for all the test runs, the variance explained of the first three components for all the test runs are greater than 95%. So we only examine the first three components.

*Step 2 - Analyzing the membership of top components:*

Once we have the top three components, we examine the membership of these components to detect performance regressions. The intuition is:

- If the test pair is a normal pair, the counters should behave the same way in both test runs. So the membership of the counters in the top three components should be relatively the same in both runs.

- If the test pair is a regression pair, the counters should behave differently

Table 9.5: Evaluation criterion 1 - Identify the target (with regression) runs with PCA approach - Comparing the membership of the top 10 counters of the first three components

| **Performance regression** | $\|$**Baseline 1** $\cap$ **Baseline 2**$\|$ | $\|$**Baseline 1** $\cap$ **Target**$\|$ | $\|$**Baseline 2** $\cap$ **Target**$\|$ |
|---|---|---|---|
| Performance regression 1 | | | |
| 1st component | 100% | 90% | 90% |
| 2nd component | 90% | 100% | 90% |
| 3rd component | 90% | 90% | 100% |
| Performance regression 2 | | | |
| 1st component | 60% | 80% | 50% |
| 2nd component | 70% | 80% | 70% |
| 3rd component | 70% | 70% | 80% |
| Performance regression 3 | | | |
| 1st component | 50% | 80% | 50% |
| 2nd component | 70% | 80% | 90% |
| 3rd component | 50% | 60% | 50% |

in the target run compared to the baseline run. So the membership of the counters in the top three components should be relatively different.

To examine the counter membership of the top three components of the runs, we compute the intersections of the first 10 counters that have the highest weight. If the PCA approach can identify the target (with a regression run), we expect to see the intersection between the Baseline runs to be relatively greater than the intersection between the Target run and the baseline runs. Hence, we should see:

- $|$Baseline 1 $\cap$ Baseline 2$| > |$Baseline 1 $\cap$ Target$|$
- $|$Baseline 1 $\cap$ Baseline 2$| > |$Baseline 2 $\cap$ Target$|$

Table 9.5 show all the intersections for all three components between all the test pairs for all three performance regressions.

For example, let us look into the three runs of Performance regression 1:

- Table 9.6 shows the top 2 components of Baseline run 1.
- Table 9.7 shows the top 2 components for Baseline run 2.
- Table 9.8 shows the top 2 components for the Target run.

The first component of Baseline run 1 (Table 9.6) and Baseline run 2 (Table 9.7) has the same top 10 counters. Thus, $|\text{Baseline } 1 \cap \text{Baseline } 2|$ for 1st component of Performance regression 1 is 100% (first cell on the first row of Table 9.5)).

The first component of Baseline run 1 (Table 9.6) and Target run (Table 9.8) has 1 different counter in the top 10 counters. Thus, $|\text{Baseline } 1 \cap \text{Target}|$ for 1st component of Performance regression 1 is 90% (second cell on the first row of Table 9.5).

The first component of Baseline run 2 (Table 9.6) and Target run (Table 9.8) also has 1 different counter in the top 10 counters. Thus, $|\text{Baseline } 2 \cap \text{Target}|$ for the 1st component of Performance regression 1 is 90% (third cell on the first row of Table 9.5).

Table 9.6: Performance regression 1 - Baseline run 1 - Top 10 counters of the first two components.

| **Component 1** | **Component 2** |
|---|---|
| **OS process cpu time** | OS free physical memory size |
| OS free physical memory size | heap memory usage used |
| heap memory usage used | GC heap bytes in use |
| GC heap bytes in use | **OS process cpu time** |
| nonheap memory usage used | heap memory usage committed |
| heap memory usage committed | nonheap memory usage used |
| memory free kb | memory free kb |
| unicast xmit table num purges | unicast xmit table num purges |
| unicast num messages sent | unicast num messages sent |
| NET lo sent bytes per second | NET eth0 sent bytes per second |

**Bold** - Counter is in the *matched counter set*

Table 9.7: Performance regression 1 - Baseline run 2 - Top 10 counters of the first two components.

| **Component 1** | **Component 2** |
|---|---|
| **OS process cpu time** | heap memory usage used |
| heap memory usage used | OS free physical memory size |
| OS free physical memory size | GC heap bytes in use |
| GC heap bytes in use | **OS process cpu time** |
| heap memory usage committed | nonheap memory usage used |
| nonheap memory usage used | heap memory usage committed |
| memory free kb | memory free kb |
| unicast xmit table num purges | unicast xmit table num purges |
| unicast num messages sent | unicast num messages sent |
| NET lo sent bytes per second | GC ps scavenge collectiontime |

**Bold** - Counter is in the *matched counter set*

Table 9.8: Performance regression 1 - Target run - Top 10 counters of the first two components.

| **Component 1** | **Component 2** |
|---|---|
| **OS process cpu time** | heap memory usage used |
| heap memory usage used | OS free physical memory size |
| OS free physical memory size | **OS process cpu time** |
| heap memory usage committed | GC heap bytes in use |
| GC heap bytes in use | heap memory usage committed |
| nonheap memory usage used | nonheap memory usage used |
| unicast xmit table num purges | unicast xmit table num purges |
| unicast num messages sent | unicast num messages sent |
| NET eth0 sent bytes per second | memory free kb |
| memory free kb | GC ps scavenge collectiontime |

**Bold** - Counter is in the *matched counter set*

The result from Table 9.5 indicates that the PCA approach can not identify the target (with regression) run in our three performance regressions. As mentioned earlier, we should see $|\text{Baseline } 1 \cap \text{Baseline } 2| > |\text{Baseline } 1 \cap \text{Target}|$ and $|\text{Baseline } 1 \cap \text{Baseline } 2| > |\text{Baseline } 2 \cap \text{Target}|$. From Table 9.5, we can see that these two conditions are not met. For example, in Performance regression 1 component 1, the $|\text{Baseline } 1 \cap \text{Baseline } 2| = 100\%$. The $|\text{Baseline } 1 \cap \text{Target}|$ and $|\text{Baseline } 2 \cap \text{Target}|$ is 90%. There is only one different counter. It is not a big difference. For component 2 and component 3, the Target test run seems be more common with the baselines (100% common counters) than the baseline runs themselves (90% common counters).

**Evaluation Criterion 2:** In other studies (Malik, 2010; Malik et al., 2010), PCA is not used to determine the matched counters. So we do not perform evaluation 2

Table 9.9: Summary

| Approach | Evaluation 1 | Evaluation 2 |
|---|---|---|
| Control charts | Works for all performance regressions | Works for all performance regressions |
| Discretizing counter | Works for 2 of the 3 performance regressions | Works but control charts show better results |
| Hierarchical clustering | Works for all performance regressions | Does not work for the performance regressions |
| Filtering-out-by-mean | N/A | Works but control charts show better results |
| Combining counters | Does not work for the performance regressions | N/A |

for PCA.

### 9.3.4 Conclusion

Table 9.9 summarizes the result of Part 2. The results show that our control charts based approach can be applied on the Commercial 2 software system. The architecture and technologies behind Commercial 2 software system is common for ultra-large scale software systems. So our control charts based approach has a large potential of application in identifying performance regressions in other ultra-large scale software systems.

Table 9.9 also shows that our control charts based approach can automatically detect the counters that exhibit regression behaviour (evaluation 2). Using our approach, the testers can proceed with cause analysis faster instead of having to analyzing the counters manually.

Table 9.9 also shows that our control charts based approach performs as well if not better than other approaches for the three performance regressions. For both evaluation criteria, i.e., identifying target (with regressions) runs and identifying the counters that are related to the regressions, our control charts based approach yields actionable results for all three performance regressions. For evaluation 2, i.e., identifying the counters that are related to the regression, our control charts based approach works better than the discretizing counter and filtering-out-by-mean approaches.

We wish to stress that the results of our experiments should not be interpreted as evidence against the other approaches that we experimented with. The goal of the performance regressions is to identify performance regressions in performance load tests. The approaches might not work well for this purpose because of the unique characteristics of the counter data, the type of executed performance load tests, or the identified performance regressions:

- We believe that the PCA-based approach does not work well because there was not enough variance in the counters during the test runs. In a production system, the input (the requests from the users) varies across the time or events such as failures. Hence, the PCA algorithm needs more variance to build more accurate components. In the test runs, the input stays relatively constant.
- We believe the reason that hierarchical clustering and combining counters using PCA approaches do not work well is because of the amount of counter data. For monitoring production software, as in the previous studies (Ahn and Vetter, 2002; Eeckhout et al., 2003a,b)), there are many incidents that can be used as data for the clustering algorithm. For performance load tests, as in our performance regressions, there are usually just the baseline and the target tests.

While we cannot comprehensively demonstrate that our control charts based approach is better than other approaches, we show the potential advantage of our approach compared to representative approaches the previous research (*see* Section 2.2.2).

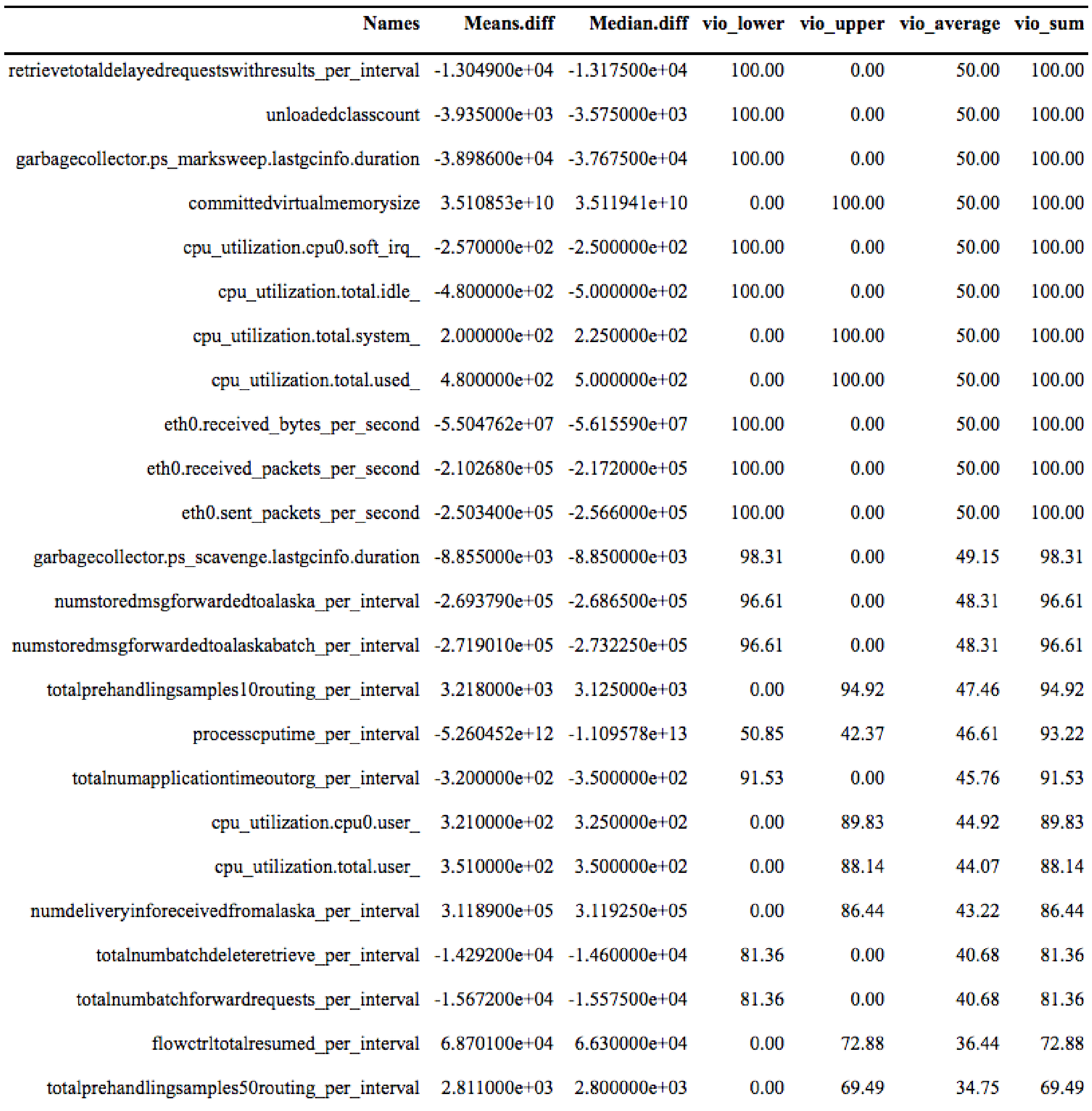
## 9.4 Part 3: Automate Regression Analysis in the Commercial 2 Software System

Prior to our involvement, the Commercial 2 software system's performance testers only conduct regression analysis manually. Manually analyzing all the counters is a time consuming and erroneous process. We implement an automatic analysis system using the majority of R script written for this thesis:

- The baseline and target test runs' counters are automatically collected from the tests' repository.
- We apply the solution that filters out load independent counters as described in Section 5.4 (p.68) using just the counters of the baseline test.
- We calculate the control charts violation ratios and create a web page with the results.
- The web page is sent to the performance testers as test run's report.

Figure 9.10 shows an example of the report. This test pair is the same as the one that we examine in Chapter 7 (p.82). There was a new database driver that caused more CPU utilization.

For each counter, we calculate that lower, upper, average, and sum violation ratios as described in Section 3.2.2 (p.43). For example, the first counter with name *retrieve total delayed requests with results* has 100% lower violation ratio. It means that, in the target test, 100% of the data points for this counter was lower than the control limits set by the baseline test. The upper violation ratio is 0%. Thus, the avarage violation ratio is 50% (out of maximum 100%) and the sum violation is 100% (out of maximum 200%).



| Names | Means.diff | Median.diff | vio_lower | vio_upper | vio_average | vio_sum |
|---|---|---|---|---|---|---|
| retrievetotaldelayedrequestswithresults_per_interval | -1.304900e+04 | -1.317500e+04 | 100.00 | 0.00 | 50.00 | 100.00 |
| unloadedclasscount | -3.935000e+03 | -3.575000e+03 | 100.00 | 0.00 | 50.00 | 100.00 |
| garbagecollector.ps_marksweep.lastgcinfo.duration | -3.898600e+04 | -3.767500e+04 | 100.00 | 0.00 | 50.00 | 100.00 |
| committedvirtualmemorysize | 3.510853e+10 | 3.511941e+10 | 0.00 | 100.00 | 50.00 | 100.00 |
| cpu_utilization.cpu0.soft_irq_ | -2.570000e+02 | -2.500000e+02 | 100.00 | 0.00 | 50.00 | 100.00 |
| cpu_utilization.total.idle_ | -4.800000e+02 | -5.000000e+02 | 100.00 | 0.00 | 50.00 | 100.00 |
| cpu_utilization.total.system_ | 2.000000e+02 | 2.250000e+02 | 0.00 | 100.00 | 50.00 | 100.00 |
| cpu_utilization.total.used_ | 4.800000e+02 | 5.000000e+02 | 0.00 | 100.00 | 50.00 | 100.00 |
| eth0.received_bytes_per_second | -5.504762e+07 | -5.615590e+07 | 100.00 | 0.00 | 50.00 | 100.00 |
| eth0.received_packets_per_second | -2.102680e+05 | -2.172000e+05 | 100.00 | 0.00 | 50.00 | 100.00 |
| eth0.sent_packets_per_second | -2.503400e+05 | -2.566000e+05 | 100.00 | 0.00 | 50.00 | 100.00 |
| garbagecollector.ps_scavenge.lastgcinfo.duration | -8.855000e+03 | -8.850000e+03 | 98.31 | 0.00 | 49.15 | 98.31 |
| numstoredmsgforwardedtoalaska_per_interval | -2.693790e+05 | -2.686500e+05 | 96.61 | 0.00 | 48.31 | 96.61 |
| numstoredmsgforwardedtoalaskabatch_per_interval | -2.719010e+05 | -2.732250e+05 | 96.61 | 0.00 | 48.31 | 96.61 |
| totalprehandlingsamples10routing_per_interval | 3.218000e+03 | 3.125000e+03 | 0.00 | 94.92 | 47.46 | 94.92 |
| processcputime_per_interval | -5.260452e+12 | -1.109578e+13 | 50.85 | 42.37 | 46.61 | 93.22 |
| totalnumapplicationtimeoutorg_per_interval | -3.200000e+02 | -3.500000e+02 | 91.53 | 0.00 | 45.76 | 91.53 |
| cpu_utilization.cpu0.user_ | 3.210000e+02 | 3.250000e+02 | 0.00 | 89.83 | 44.92 | 89.83 |
| cpu_utilization.total.user_ | 3.510000e+02 | 3.500000e+02 | 0.00 | 88.14 | 44.07 | 88.14 |
| numdeliveryinforeceivedfromalaska_per_interval | 3.118900e+05 | 3.119250e+05 | 0.00 | 86.44 | 43.22 | 86.44 |
| totalnumbatchdeleteretrieve_per_interval | -1.429200e+04 | -1.460000e+04 | 81.36 | 0.00 | 40.68 | 81.36 |
| totalnumbatchforwardrequests_per_interval | -1.567200e+04 | -1.557500e+04 | 81.36 | 0.00 | 40.68 | 81.36 |
| flowctrltotalresumed_per_interval | 6.870100e+04 | 6.630000e+04 | 0.00 | 72.88 | 36.44 | 72.88 |
| totalprehandlingsamples50routing_per_interval | 2.811000e+03 | 2.800000e+03 | 0.00 | 69.49 | 34.75 | 69.49 |

Figure 9.10: Example of an automatic regression analysis result of the Commercial 2 software system

In the automated test report (e.g. Figure 9.10), we display counters decreasingly based on their average violation ratio. We only show the top ranked counters due to space constraint in Figure 9.10. There are 255 counters in total for this test pair. We filtered out 39 always changing counters using the approach discussed in Appendix 5.4. There are still 216 counters left. Yet, looking at the report the performance testers can infer relatively quickly that:

- There is a CPU utilization regression. Many of the top counters with high violation ratio are CPU related.
- Something creates more objects to be collected in the new version of the software. *lastgcinfo.duration* counters indicate the time that is required for the garbage collectors to run.
- The *retrieve total delayed requests with results* is related to the database access code.
- Judging by the *eth0* related counters, the software also sent and received more network messages.

Compared to manually analyzing 255 counters, this automated approach significantly reduces the amount of time required for each regression analysis. Using the report, the performance testers should be able to quickly realize that there is a regression. They would also be able to diagnose that the regression is related to accessing the database. The actual cause of this particular regression is a new database driver in the new version. This means that the performance testers would be able to arrive at the cause faster than before.

# Part V

# Conclusions

# CHAPTER 10

# Conclusion

## 10.1 Summary

Performance testing is an important software activity in the engineering of large scale software systems. Performance testers need to make sure that a new version of the software performs as well as the previous version. Otherwise, the operating cost may have to be increased or user experience might be compromised.

Analyzing performance counters data is a time consuming process. Previous studies (*see* Chapter 2) suggested approaches to reduce and analyze performance counters of performance load tests. In this thesis, we propose the use of control charts to reduce performance counter data. We use control charts to identify performance regressions in test runs (Study 1 - Chapter 6, p.73 and Study 2 - Chapter 7, p.82). Then, we develop approaches that are based on control charts to automatically suggest the cause of the performance regression (Study 3 - Chapter 8, p.96). We use performance test runs from two large scale software systems from our industrial partner and one open-source software system to evaluate our control charts based approaches. We also evaluate the applicability of our control charts based approaches by applying our approaches to different releases of two commercial software systems (Study 4 - Chapter 9, p.120).

The results warrant future application of control charts into performance testing. We show that control charts can identify performance regression in test runs as well as or better than existing approaches (Ahn and Vetter, 2002; Bezemer, 2014; Bezemer and Zaidman, 2010, 2014; Eeckhout et al., 2003a,b; Foo, 2011; Foo et al., 2010; Malik, 2010; Malik et al., 2010; Nickolayev et al., 1997; Vetter and Reed, 1999; Wiertz et al., 2014; Zhao et al., 2009) (Study 4 - Chapter 9, p.120). The feedback from practitioners is also positive (Section 9.2). We believe our approaches can effectively help automate performance load test analysis. We apply our approach to automatically identify performance regression into our industrial partner's software system (Section 9.4, p.158).

The biggest threat to the validity of this thesis is the external validity. We were able to evaluate our approaches on three different software systems, of which two are in-production large scale industrial software systems. However, they are both from the same company (yet they are developed by two different teams). Further studies with more industrial partners are required to validate the effectiveness of using control charts in analyzing performance load tests' result.

## 10.2 Future work

We believe that there are other applications of control charts to performance load testing. For example, one can apply control charts to detect the location of performance regressions by analyzing the violation ratios of different thread pools' counters. One can also extract keywords from the logs that are collected during the performance test runs. The frequency of the keywords can be treated as performance counters. Control charts, then, can be used to analyze the log counters in order to identify problems during the test run.

Researchers (Trubiani, 2015; Trubiani et al., 2014) have proposed several ways to identify performance anti-patterns. Control charts can be adapted to detect these anti-patterns in performance tests or in production environments. The regression causes that we studied in Study 3 (Chapter 8) are very similar to performance anti-patterns.

We note that the performance regressions that we are considering here are very similar to performance degradations in research on the performance of hardware and middleware (Najibi and Pedram, 2008). Future work can extend the use of control charts on hardware performance counters to detect performance degradations.

In Chapter 9, we apply four other approaches to automatically identify performance regressions. We show that our control charts based approach works better. However, we believe that future work can improve the accuracy of those approaches. We also believe that further comparison studies between the control charts based approach and other approaches are warranted.

# Bibliography

M. Abdollahian, S. Ahmad, and S. Huda. Multivariate control charts for surgical procedures. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, ISABEL '11, pages 18:1–18:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0913-4. doi: 10.1145/2093698.2093716. URL http://doi.acm.org/10.1145/2093698.2093716.

Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Symposium on Operating Systems Principles*, pages 74–89. ACM, 2003.

David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, Jan 1991. ISSN 0885-6125.

Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Conference on Supercomputing*, pages 1–16. IEEE Computer Society Press, 2002.

Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Conference on Object-oriented Prog., Sys., Lang., and Apps*, pages 739–753. ACM, 2010.

Ayman Amin, Alan Colman, and Lars Grunske. Statistical detection of qos violations based on cusum control charts. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 97–108, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1202-8. doi: 10.1145/2188286.2188302. URL http://doi.acm.org/10.1145/2188286.2188302.

L. B. Arief and N. A. Speirs. Automatic generation of distributed system simulations. In *European Simulation Multiconference*, pages 85–91, 1999.

L. B. Arief and N. A. Speirs. A uml tool for an automatic generation of simulation programs. In *International Workshop on Software and Performance*, pages 71–76. ACM, 2000. doi: 10.1145/350391.350408.

Francois Baccelli and Pierre Bremaud. *Elements of Queueing Theory: Palm Martingale Calculus and Stochastic Recurrences*. Springer, 2003. ISBN 9783540660880. URL http://books.google.ca/books?id=REcVXcy2sb0C.

Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: a survey. *Transactions on Software Engineering*, 30(5):295–310, 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.9. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1291833.

Marco Bernardo and Roberto Gorrieri. A tutorial on empa: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

Marco Bernardo, Lorenzo Donatiello, and Roberto Gorrieri. Integrating performance and functional analysis of concurrent systems with empa. Report, University of Bologna, 1995.

Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Aempa: a process algebraic description language for the performance analysis of software architectures. In *International Workshop on Software and Performance*. ACM, 2000. doi: 10.1145/350391.350394.

Marco Bertoli, Giuliano Casale, and Giuseppe Serazzri. Java modelling tools: an open source suite for queueing network modelling andworkload analysis. In *International Conference on Quantitative Evaluation of Systems*, pages 119–120, 2006. doi: 10.1109/QEST.2006.22.

Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: performance engineering tools for system modelling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. ISSN 0163-5999. doi: 10.1145/1530873.1530877.

Cor-Paul Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. PhD thesis, TU Delft, Netherlands, 2014.

Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *Joint Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 88–92, 1862393, 2010. ACM.

Cor-Paul Bezemer and Andy Zaidman. Performance optimization of deployed software-as-a-service applications. *Journal of Systems and Software*, 87(0):87–103, 2014. ISSN 0164-1212.

Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *European Conference on Computer systems*, pages 111–124. ACM, 2010.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 0885-6125.

Greg Bronevetsky, Ignacio Laguna, Bronis R. de Supinski, and Saurabh Bagchi. Automatic fault characterization via abnormality-enhanced classification. In *International Conference on Dependable Systems and Networks*, pages 1–12, 2012.

Mark Burgess. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming*, 60(1):1–26, 2006.

Mark Burgess, Harek Haugerud, Sigmund Straumsnes, and Trond Reitan. Measuring system normality. *Transactions Computer System*, 20(2):125–160, 2002.

Giuliano Casale and Giuseppe Serazzi. Quantitative system evaluation with java modelling tools. In *International Conference on Performance Engineering*, Karlsruhe, Germany, 2012.

Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human-Computer Interaction*, 77(3):333–356, 2004.

S. Le Cessie and J. C. Van Houwelingen. Ridge estimators in logistic regression. *Journal of the Royal Statistical Society*, 41(1):191–201, 1992.

Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *International Conference* *on Software Engineering*, ICSE 2014, pages 1001–1012, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568259. URL http://doi.acm.org/10.1145/2568225.2568259.

Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *International Conference on Dependable Systems and Networks*, pages 452–461, 2008.

John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In *International Conference on Machine Learning*, pages 108–114. Morgan Kaufmann, 1995.

Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Symposium on Opearting Systems Design and Implementation*, pages 231–244, San Francisco, CA, 2004. USENIX Association.

Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Symposium on Operating Systems Principles*, pages 105–118. ACM, 2005.

William W. Cohen. Fast effective rule induction. In *International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

Vittorio Cortellessa and Raffaela Mirandola. Deriving a queueing network based performance model from uml diagrams. In *International Workshop on Software and performance*. ACM, 2000. doi: 10.1145/350391.350406. URL http://dl.acm.org/citation.cfm?doid=350391.350406.

Vittorio Cortellessa, Giuseppe Lazeolla, and Raffaela Mirandola. Early generation of performance models for object-oriented systems. *IEEE Software*, 147(3):61–72, 2000. ISSN 1462-5970. doi: 10.1049/iip-sen:20000755.

Vittorio Cortellessa, Andrea D'Ambrogio, and Giuseppe Lazeolla. Automatic derivation of software performance models from case documents. *Performance Evaluation*, 45(2-3):81–105, 2001. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/S0166-5316(01)00036-0. URL `http://www.sciencedirect.com/science/article/pii/S0166531601000360`.

Miguel de Miguel, Thomas Lambolais, Sophie Piekarec, Stephane Betge-Brezetz, and Jerome Pequery. *Automatic Generation of Simulation Models for the Evaluation of Performance and Reliability of Architectures Specified in UML*, volume 1999 of *Lecture Notes in Computer Science*, book section 9, pages 83–101. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-41792-7. doi: 10.1007/3-540-45254-0_9. URL `http://dx.doi.org/10.1007/3-540-45254-0_9`.

Antoine Duclos and Nicolas Voirin. The p-control chart: a tool for care improvement. *International Journal for Quality in Health Care*, 22(5):402, 2010. doi: 10.1093/intqhc/mzq037. URL `+http://dx.doi.org/10.1093/intqhc/mzq037`.

Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 169–186, Anaheim, California, USA, 2003a. ACM.

Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 15:1–33, 2003b.

Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Designing computer architecture research workloads. *Computer*, 36(2):65–71, 2003c.

Brian S. Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. *Cluster Analysis*. Wiley, 2001. ISBN 9780340761199.

King C. Foo. Automated discovery of performance regressions in enterprise applications. Master's thesis, Queen's University, Kingston, Ontario, Canada, 2011.

King C. Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Zou Ying, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *International Conference on Quality Software*, pages 32–41, 2010.

King C. Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *International Conference on Software Engineering*, volume 2, pages 159–168, 2015. ISBN 0270-5257. doi: 10.1109/ICSE.2015.144.

Eibe Frank and Richard Kirkby. Randomtree, 2012.

Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In *International Conference on Machine Learning*, San Francisco, CA, 1998. Morgan Kaufmann Publishers.

Eibe Frank, Mark Hall, and Bernhard Pfahringer. Locally weighted naive bayes, 2003.

Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. In *International Workshop on Software and Performance*, pages 120–130. ACM, 1996. doi: 10.1145/287318.287346.

Greg Franks and Murray Woodside. Effectiveness of early replies in client server systems. *Performance Evaluation*, 36-37(0):165–183, 1999. ISSN 0166-5316.

Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modelling and solution of layered queueing networks. *Transactions on Software Engineering*, 35(2):148–161, 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.74.

Moshe Gabel, Assaf Schuster, Ran-Gilad Bachrach, and Nikolaj Bjorner. Latent fault detection in large scale services. In *International Conference on Dependable Systems and Networks*, pages 1–12, 2012.

Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale hpc systems. In *International Parallel and Distributed Processing Symposium*, pages 1168–1179, 2012a.

Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: a closer look into hpc systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, Series Fault prediction under the microscope: a closer look into HPC systems. IEEE Computer Society Press, 2012b.

Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for hpc systems and applications: Current situation and open issues. *International Journal of High Performance Computing Applications*, 27(3):273–282, 2013.

Shadi Ghaith, Miao Wang, Philip Perry, and John Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–383, 2013.

Shadi Ghaith, Miao Wang, Philip Perry, and Liam Murphy. Software contention aware queueing network model of three-tier web systems. In *International Conference on Performance Engineering*, pages 273–276. ACM, 2014. doi: 10.1145/2568088.2576760.

Ran Giladi and Niv Ahituv. Spec as a performance evaluation measure. *Computer*, 28(8):33–42, 1995. ISSN 0018-9162. doi: 10.1109/2.402073.

Sebastien Godard. Sysstat, 2014.

Michael W. Godfrey, Ahmed E. Hassan, James Herbsleb, Gail C. Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: A roundtable. *Software, IEEE*, 26(1):67–70, 2009. ISSN 0740-7459.

Vincenzo Grassi and Raffaela Mirandola. Primamob-uml: a methodology for performance analysis of mobile software architectures. In *International Workshop on Software and performance*. ACM, 2002. doi: 10.1145/584369.584411.

Machine Learning Group. Weka 3: Data mining software in java, 2012.

Harvey W. Gunther. Websphere application server development best practices for performance and scalability. *IBM WebSphere Application Server Standard and Advanced Editions - White paper*, 2000.

Zhen Guo, Guofei Jiang, Haifeng Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *International Conference on Dependable Systems and Networks*, pages 259–268, 2006.

Jiang Guofei, Haifeng Chen, and Kenji Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *Transactions on Dependable and Secure Computing*, 3(4):312–326, 2006.

Greg Hamerly and Charles Elkan. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning*, pages 202–209. Morgan Kaufmann Publishers Inc., 2001.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer-Verlag, 2008.

Holger Hermanns, Ulrich Herzog, and Vassilis Mertsiotakis. Stochastic process algebras as a tool for performance and dependability modelling. In *International Computer Performance and Dependability Symposium*, pages 102–111, 1995. doi: 10.1109/IPDS.1995.395813.

Holger Hermanns, Ulrich Herzog, Ulrich Klehmet, Vassilis Mertsiotakis, and Markus Siegle. Compositional performance modelling with the tipptool. *Perform. Eval.*, 39(1-4):5–35, 2000. ISSN 0166-5316. doi: 10.1016/s0166-5316(99)00056-5. URL http://www.sciencedirect.com/science/article/pii/S0166531699000565.

Ulrich Herzog. *Formal Description, Time and Performance Analysis a Framework*, volume 264 of *Informatik-Fachberichte*, pages 172–190. Springer Berlin Heidelberg, 1990. ISBN 978-3-540-53490-7. doi: 10.1007/978-3-642-76309-0_10. URL http://dx.doi.org/10.1007/978-3-642-76309-0_10.

Hewlett Packard. Loadrunner, 2010.

IBM Corporation. The rational rhapsody family from ibm, 2012. URL http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=BR&infotype=PM&appname=SWGE_RA_YA_USEN&htmlfid=RAB14010USEN&attachment=RAB14010USEN.PDF#loaded.

Miao Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *International Conference on Dependable Systems and Networks*, pages 285–294, 2009a.

Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A.S. Ward. System monitoring with metric-correlation models: problems and solutions. In *International Conference on Autonomic computing*, pages 13–22. ACM, 2009b.

Zhen Ming Jiang and Ahmed E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, Nov 2015. ISSN 0098-5589.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *International Conference on Software Maintenance*, pages 307–316, 2008.

Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic performance analysis of load tests. In *International Conference in Software Maintenance*, pages 125–134, Edmonton, 2009c.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Conference on Prog. Lang. Design and Impl.*, pages 77–88. ACM, 2012.

George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

Suhas Kabinna. An exploration of challenges associated with software logging in large systems. Master's thesis, Queen's University, Kingston, Ontario, Canada, 2016.

Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

Ron Kohavi. The power of decision tables. In *European Conference on Machine Learning*, Heraklion, Crete, Greece, 1995.

Mutsumi Komuro. Experiences of applying spc techniques to software development processes. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 577–584, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134367. URL http://doi.acm.org/10.1145/1134285.1134367.

Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. *Machine Learning*, 59(1-2):161–205, May 2005.

Yinglung Liang, Yanyong Zhang, Morris Jette, Sivasubramaniam Anand, and Ramendra Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *International Conference on Dependable Systems and Networks*, pages 425–434, 2006.

Mark C. Little. JavaSim, 2015.

Haroon Malik. A methodology to support load test analysis. In *International Conference on Software Engineering*, pages 421–424, Cape Town, South Africa, 2010. ACM.

Haroon Malik, Bram Adams, and Ahmed E. Hassan. Pinpointing the subsystems responsible for the performance deviations in a load test. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 201–210, 2010.

Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *International Conference on Software Engineering*, ICSE '13, pages 1012–1021, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486927.

Microsoft Corp. Windows reliability and performance monitor, 2011.

Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982. ISBN 0387102353.

Marvin Minsky and Papert Seymour. *Perceptrons*. M.I.T. Press, Oxford, England, 1969.

Tom Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

Mozilla Project. Talos, 2010.

MohammadAhmad Munawar and PaulA S. Ward. *Leveraging Many Simple Statistical Models to Adaptively Monitor Software Systems*, volume 4742 of *Lecture Notes in Computer Science*, chapter 42, pages 457–470. Springer Berlin Heidelberg, 2007.

MySQL AB. Mysql community server, 2011. Ver. 5.5.

M. Najibi and H. Pedram. Compensating algorithmic-loop performance degradation in asynchronous circuits using hardware multi-threading. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pages 507–510, April 2008. doi: 10.1109/ISVLSI.2008.66.

Thanh H. D. Nguyen, B. Adams, Jiang Zhen Ming, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated verification of load tests using control charts. In *Asia Pacific Software Engineering Conference*, pages 282–289, 2011.

Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *International Conference on Performance Engineering*, pages 299–310, Boston, Massachusetts, USA, 2012. ACM.

Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-time statistical clustering for event trace reduction. *International Journal of High Performance Computing Applications*, 11(2):144–159, 1997.

Information Retrieval Project. University of Dayton Research Institute. *Information Retrieval: Ground Rules for Indexing*. University of Dayton Res. Institute, 1963.

Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *International Conference on Dependable Systems and Networks*, pages 49–60, 2011.

Tariq Omari, Greg Franks, Murray Woodside, and Amy Pan. Efficient performance models for layered server systems with replicated servers and parallel behaviour. *Journal of Systems and Software*, 80(4):510–527, 2007. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2006.07.022. URL http://www.sciencedirect.com/science/article/pii/S0164121206002032.

Rasha Osman and William J. Knottenbelt. Database system performance evaluation models: A survey. *Performance Evaluation*, 69(10):471–493, 2012. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/j.peva.2012.05.006. URL http://www.sciencedirect.com/science/article/pii/S0166531612000442.

Rasha Osman, Irfan Awan, and Michael E. Woodward. Application of queueing network models in the performance evaluation of database designs. *Electron. Notes Theor. Comput. Sci.*, 232:101–124, 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.02.053.

James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977. ISSN 0360-0300. doi: 10.1145/356698.356702. 356702.

Dorina Petriu, Hoda Amer, Shikharesh Majumdar, and Istabrak Abdull-Fatah. Using analytic models predicting middleware performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 189–194, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: 10.1145/350391.350431. URL http://doi.acm.org.proxy.queensu.ca/10.1145/350391.350431.

Dorina C. Petriu. Approximate mean value analysis of client-server systems with multi-class requests. *SIGMETRICS Perform. Eval. Rev.*, 22(1):77–86, 1994. ISSN 0163-5999. doi: 10.1145/183019.183027. 183027.

Dorina C. Petriu, Mohammad Alhaj, and Rasha Tawhid. *Software Performance Modeling*, pages 219–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30982-3. doi: 10.1007/978-3-642-30982-3_7. URL http://dx.doi.org/10.1007/978-3-642-30982-3_7.

John C. Platt. Advances in kernel methods. chapter Fast training of support vector machines using sequential minimal optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

WPP Group PLC. The need for speed ii. Technical report, Zona Research, 2001.

Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, PP(99):1–1, 2009. ISSN 0272-1732. doi: 10.1109/MM.2009.50.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

Walter A. Shewhart. *Economic control of quality of manufactured product*. American Society for Quality Control, 1931.

Connie U. Smith and Lloyd G. Williams. *Performance engineering evaluation of object-oriented systems with SPEED TM*, volume 1245 of *Lecture Notes in Computer Science*, book section 12, pages 135–154. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63101-9. doi: 10.1007/BFb0022203. URL http://dx.doi.org/10.1007/BFb0022203http://link.springer.com/chapter/10.1007%2FBFb0022203.

Connie U. Smith and Lloyd G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-72229-1.

Edward Stehle, Kevin Lynch, Maxim Shevertalov, Chris Rorres, and Spiros Mancoridis. On the use of computational geometry to detect software faults at runtime. In *International Conference on Autonomic Computing*, pages 109–118, 1809069, 2010. ACM.

William J. Stewart. Marca: Markov chain analyzer, 2015.

Riverbed Technology. Riverbed modeler, 2015.

The Apache Software Foundation. Tomcat, 2010. Ver. 5.5.

The Apache Software Foundation. Jmeter, 2012.

Catia Trubiani. Introducing software performance antipatterns in cloud computing environments: Does it help or hurt? In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 207–210, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3248-4. doi: 10.1145/2668930.2695528. URL http://doi.acm.org/10.1145/2668930.2695528.

Catia Trubiani, Antinisca Di Marco, Vittorio Cortellessa, Nariman Mani, and Dorina Petriu. Exploring synergies between bottleneck analysis and performance antipatterns. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 75–86, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2733-6. doi: 10.1145/2568088.2568092. URL http://doi.acm.org/10.1145/2568088.2568092.

Jeffrey S. Vetter and Daniel A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Conference on Supercomputing*, page 44, Portland, Oregon, United States, 1999. ACM.

Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, 26(12):1147–1156, 2000.

D.J. Wheeler and D.S. Chambers. *Understanding Statistical Process Control*. SPC Press, 2010. ISBN 9780945320692. URL https://books.google.ca/books?id=8JlHSAAACAAJ.

Maarten Wiertz, Andy Zaidman, Ad van der Hoeven, Remko Weijers, Andre van de Graaf, and Cor-Paul Bezemer. Locating performance improvement opportunities in an industrial software-as-a-service application. In *International Conference on Software Maintenance*, pages 547–556. IEEE Computer Society, 2014.

Lloyd G. Williams and Connie U. Smith. Performance evaluation of software architectures. In *International Workshop on Software and Performance*. ACM, 1998. URL http://dl.acm.org/citation.cfm?doid=287318.287353.

C. M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar. The rendezvous network model for performance synchronization multitasking distributed software. Technical report, Carleton University, 1989.

C. Murray Woodside. Throughput calculation for basic stochastic rendezvous networks. *Perform. Eval.*, 9(2):143–160, 1989. ISSN 0166-5316. doi: 10.1016/0166-5316(89)90039-4. 82986.

Murray Woodside, E. Neron, E. D. S. Ho, and B. Mondoux. An ”active server” model for the performance of parallel programs written using rendezvous. *Journal of System and Software*, 6(1-2):125–131, 1986. ISSN 0164-1212. doi: 10.1016/0164-1212(86)90031-2.

Murray Woodside, Dorina C. Petriu, José Merseguer, Dorin B. Petriu, and Mohammad Alhaj. Transformation challenges: from software models to performance models. *Software & Systems Modeling*, 13(4):1529–1552, 2014. ISSN 1619-1374. doi: 10.1007/s10270-013-0385-x. URL http://dx.doi.org/10.1007/s10270-013-0385-x.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Symposium on Operating Systems Principles*, pages 117–132. ACM, 2009.

Nezih Yigitbasi, Matthieu Gallet, Derrick Kondo, Alexandru Iosup, and Dick Epema. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *International Conference on Grid Computing*, pages 65–72, 2010.

Steve Zhang, Ira Cohen, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. In *International Conference on Dependable Systems and Networks*, pages 644–653. IEEE Computer Society, 2005.

Ying Zhao, Yongmin Tan, Zhenhuan Gong, Xiaohui Gu, and Mike Wamboldt. Self-correlating predictive information tracking for large-scale production systems. In *International Conference on Autonomic Computing*, pages 33–42. ACM, 2009.

Ziming Zheng, Yawei Li, and Zhiling Lan. Anomaly localization in large-scale clusters. In *International Conference on Cluster Computing*, pages 322–330, 2007.