Automated Approaches for Reducing the Execution Time of Performance Tests

by

Hammam M. AlGhamdi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2017

Abstract

Performance issues are one of the primary causes of failures in today's large-scale software systems. Hence, performance testing has become an essential software quality assurance tool. However, performance testing faces many challenges. One challenge is determining how long a performance test must run. Performance tests often run for hours or days to uncover performance issues (e.g., memory leaks). However, much of the data that is generated during a performance test is repetitive. Performance testers can stop their performance tests (to reduce the time to market and the costs of performance testing) when the test no longer offers new information about the system's performance. Therefore, in this thesis, we propose two automated approaches that reduce the execution time of performance tests by measuring the repetitiveness in the performance metrics that are collected during a test.

The first approach measures repetitiveness in the values of the performance metric, e.g., CPU utilization, during a performance test. Then, the approach provides a recommendation to stop the test when the generated performance metric values become highly repetitive and the repetitiveness is stabilized (i.e., relatively little new information about the systems' performance is generated). The second approach also recommends the stopping of a performance test, by measuring the repetitiveness in

the inter-metrics relations. Our second approach combines the values of the performance metrics that are generated at a given time into so-called a *system state*. For instance, a system state can be the combination of a low CPU utilization value and a high response time value. Our second approach recommends a stopping time when the generated system states during a performance test becomes highly repetitive.

We performed experiments on three open source systems (i.e., CloudStore, PetClinic and Dell DVD Store). Our experiments show that our first approach reduces the duration of 24-hour performance tests by up to 75% while capturing more than 91.9% of the collected performance metric values. In addition, our approach recommends a stopping time that is close to the most cost-effective stopping time (i.e., the stopping time that minimizes the duration of a test while maximizing the amount of information about the system's performance as provided by the performance test). Our second approach was also evaluated using the same set of systems. We find that the second approach saves approximately 70% of the execution time while preserving at least 95% of the collected system states during a 24-hour time period. Our comparison of the two approaches reveals that the second approach recommends the stopping of tests at more cost-effective times, in comparison to our first approach.

Co-authorship

An earlier version of Chapter 3 was published as below:

- An Automated Approach for Recommending When to Stop Performance Tests. Hammam M. Alghamdi, Mark D. Syer, Weiyi Shang and Ahmed E. Hassan, 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME). Raleigh, USA 2016. Oct 4-8 (acceptance rate: 37/127 (29%))

  My contribution: Drafting the research plan, conducting experiments, collecting the data, analyzing the data, writing and polishing the paper drafts, and presenting the paper.

Acknowledgments

I would like to express my sincerest gratitude to my supervisor, Professor Ahmed E. Hassan, for his patience during my MSc study, as well as his unparalleled motivation and immense knowledge, without which this thesis would not have been completed. I would never forget my first meeting with him, during which he enlightened me right away with just a glance at my research. His support and guidance were unwavering during the entire course of my study. I could not have imagined having a better supervisor and mentor for my MSc study than Professor Hassan.

My sincere thanks also go to Dr. Weiyi Shang, Dr. Cor-Paul Bezemer, and Dr. Mark D. Syer, who afforded me the opportunity to be a part of the team. Their dedication, commitment, and support contributed significantly to my research achievements. What I learned from them is invaluable, and I will forever be grateful to them.

I likewise thank my fellow lab mates for the friendship and companionship. In particular, I express my gratitude to Ravjot who has been more than a friendhe has been a true brother to me. He has helped me considerably, and I would forever be indebted to him. I am also grateful to Safwat, Sami, Tariq, Suhas, Hend, Daniel, and Sumit for the friendship and the great times together.

I also thank my parents for always being there for me. The greatest motivation

that makes me continue my path toward success is seeing joy and pride in their faces. May Allah reward you for everything you have done for us. I likewise thank my brothers and sister for their unwavering moral support and encouragement.

Finally, I thank my dear wife, Khulud, my greatest gift from God. You are my inspiration throughout this academic journey and my stalwart partner in life's ups and downs.

Contents

Abstract **i**

Co-authorship **iii**

Acknowledgments **iv**

Contents **vi**

List of Tables **ix**

List of Figures **xi**

Chapter 1: Introduction **1**
1.1 Research Statement 3
1.2 Thesis Overview 4
1.3 Major Thesis Contributions 5
1.4 Organization of Thesis 6

Chapter 2: Background and Related Work **7**
2.1 Workload Scenarios 8
2.1.1 Covering the Source Code: 8
2.1.2 Covering the Scenarios that are Seen in the Field: 8
2.1.3 Covering Scenarios that are Likely to Expose Performance Issues. 9
2.2 Workload Intensity 11
2.2.1 Steady Workload: 11
2.2.2 Step-wise Workload: 11
2.3 Test Execution Time 12

Chapter 3: Cost-effective Stopping of a Performance Test Using the Repetitiveness in the Collected Performance Metric Values **13**
3.1 Chapter Introduction 14

3.2 A Motivating Example . . . 15
3.3 Our Approach for Determining a Cost-Effective Length of a Performance Test . . . 18
3.3.1 Collecting Performance Test Data . . . 19
3.3.2 Determining the Use of Raw or Delta Values of a Metric . . . 20
3.3.3 Measuring the Repetitiveness . . . 21
3.3.4 Smoothing Likelihood of Repetitiveness . . . 27
3.3.5 Calculating the First Derivative of Repetitiveness . . . 27
3.3.6 Determining Whether to Stop the Test . . . 28
3.4 Experiment Setup . . . 28
3.4.1 Subject Systems . . . 29
3.4.2 Deployment of the Subject Systems . . . 29
3.4.3 Performance Tests . . . 29
3.4.4 Data collection . . . 30
3.4.5 Parameters of Our Approach . . . 31
3.4.6 Leveraging Existing (Jain's) Approach to Stop Performance Tests 32
3.4.7 Preliminary Analysis . . . 33
3.5 Case Study Results . . . 33
3.6 Threats to validity . . . 38
3.6.1 Threats to Internal Validity . . . 38
3.6.2 Threats to Construct Validity . . . 39
3.7 Chapter Conclusion . . . 41

Chapter 4: Cost-effective Stopping of a Performance Test Using the Repetitiveness in the collected Inter-Metrics Relations 42

4.1 Chapter Introduction . . . 43
4.2 A Motivating Example . . . 44
4.3 Our Approach for Determining a Cost-Effective Length of a Performance Test . . . 46
4.3.1 Collecting Data . . . 47
4.3.2 Transforming the Collected Metrics into States . . . 48
4.3.3 Determining Whether to Stop the Test . . . 51
4.4 Experiment Setup . . . 51
4.4.1 Experiment Environment . . . 52
4.4.2 Parameters of Our Approach . . . 53
4.5 Case Study Results . . . 54
4.6 Threats to validity . . . 61
4.6.1 Threats to Internal Validity . . . 62
4.6.2 Threats to External Validity . . . 62

4.6.3 Threats to Construct Validity 63
4.7 Chapter Conclusion 63

Chapter 5: Conclusion **65**
5.1 Summary 66
5.1.1 Thesis Summary 66
5.2 Future Work 67

Bibliography **70**

List of Tables

3.1 A motivational example of our approach for stopping a test using the repetitiveness of performance metric values 17

3.2 Wilcoxon test results for our working example 25

3.3 The stopping times that are recommended by our approach for performance tests. The values of the stopping times are hours after the start of the tests. 34

3.4 The percentages of the post-stopping generated data is repetitive of the pre-stopping generated data. 37

3.5 The delay between our recommended stopping times and the most cost-effective stopping times. 38

4.1 A motivational example of a performance issue that is missed by a test of which the execution time is reduced using our previous approach. As our previous approach detects repetition on metric values, period p_3 is considered repetitive (i.e., with p_1 for the response time (RT) metric and with p_2 for the CPU metric). The approach presented in this chapter does not consider any of the periods repetitive. 45

4.2 An example of performance metrics and metric states. Orange cells show the low value metric state (i.e., S_1), blue cells show the medium value metric state (i.e., S_2), and yellow cells show the high value metric state (i.e., S_3) 50
4.3 The $state_{unique}$ for every 5 minutes in the example in Table 4.2 53
4.4 The recommended stopping times and the state coverage when applying our approach to the tests 55
4.5 Comparison between the state coverage and the stopping times of our new approach and our previous approach that is presented in Chapter 3 57
4.6 The states that are missed by our new approach and captured by the 24-hour performance tests 60
4.7 The states that are missed by our previous approach that is presented in Chapter 3 and captured by the 24-hour performance tests 61
4.8 The *coverage-opportunity* scores of our approaches. 61

List of Figures

3.1 An Overview of Our Approach 19

3.2 Our Approach that Evaluates Whether the Recommended Stopping Times is too Early 35

4.1 An overview of our approach 47

4.2 The MEM, △ MEM, and RT metrics from the example in Table 4.2 . 49

4.3 Transforming the values into states for the example from Table 4.2. The red vertical lines are the state boundaries determining the states 52

4.4 State coverage of our experiments over time. The red line is the stopping time that is recommended by our proposed approach. The blue line is the stopping time recommended by our previous approach that is presented in Chapter 3. 56

4.5 An imaginary example showing the difference between a *borderline-state* and a *clearly-identified-state*. A metric value that is inside the light grey background area is considered as close to a state boundary 58

Chapter 1

Introduction

The performance of large-scale software systems (e.g., Gmail and Amazon) is a critical concern as they must concurrently support millions of users. Failures in these systems are often due to performance issues rather than functional issues [1, 2]. Such failures may have significant financial and reputational consequences. For example, a failure in Yahoo mail on December 12, 2013, resulted in a service outage for a large number of users [3]. A 25-minute service outage on August 14, 2013 (caused by a performance issue) cost Amazon around $1.7 million [3]. Microsoft's cloud-based platform, Azure, experienced an outage due to a performance issue on November 20, 2014 [4, 5], which affected users worldwide for 11 hours. These performance failures affect the competitive position and reputation of these large-scale software systems because customers expect high performance and reliability.

To ensure the performance of large-scale software systems, performance tests are conducted to determine whether a given system satisfies its performance requirements (e.g., minimum throughput or maximum response time) [6]. Performance tests are conducted by examining the system's performance under a workload in order to gain understanding of the system's expected performance in the field [7]. Therefore, performance tests are often carefully designed to mimic real system behaviours in the field and uncover performance issues.

One of the challenges that is associated with designing performance tests is determining how long a performance test should last. Some performance issues (e.g., memory leaks) might only appear after a test is executed for an extended period of time. Therefore, performance tests may potentially last for days to uncover such issues [8]. Although in practice, performance tests are often pre-defined (typically by prior experience or team convention) to last a certain execution time, there is no guarantee that

all the potential performance issues would appear before the end of the tests. Moreover, performance tests are often the last step in already-delayed and over-budget release cycles [9] and consume significant resources. Therefore, it is necessary to reduce the execution time of a performance test when possible, while preserving the amount of information about the system the test reveals.

There exists a point in time at which the cost-effectiveness of the test is the highest, i.e., if the test continues to execute after this point, the amount of new information that is generated during the test is too minor to be beneficial. Determining the cost-effective length of a performance test may speed up the releases cycles, i.e., reduce the time to market, while saving testing resources, i.e., reducing the computing resources that are required to run the test and to analyze the results.

A performance test often repeats the execution of test cases multiple times [10]. This repetition generates repetitiveness in the results of the test, i.e., performance metrics. Therefore, in this thesis, we aim to identify the cost-effective stopping time of a performance test by measuring the repetitiveness in the collected performance metrics.

1.1 Research Statement

With the increase of the complexity of large-scale systems, performance testing has become a very important phase before the deployment of any new version of these systems. Prior research studied many aspects of performance testing, including detecting performance regression [11, 12, 13], and test workload selection [14, 15, 16]. However, to the best of our knowledge, there exists very limited research about reducing the execution time of performance testing [17, 18].

Running the tests longer or shorter than the cost-effective execution time results in undesirable consequences (e.g., missing the identification of performance issues or delaying release cycles). Therefore, a study of when to stop a performance test is needed in order to avoid such undesirable consequences.

1.2 Thesis Overview

This thesis presents two approaches for recommending the stopping of a performance test:

1. *Cost-effective Stopping of a Performance Test Using the Repetitiveness in the Collected Performance Metric Values (Chapter 3):* Intuitively, performance testers may consider stopping a performance test if they believe that 1) the newly-generated metric values from continuing the test would likely be similar to the already-collected metric values, or 2) the trends in the performance metric values from continuing the test would likely be similar to the already-collected trends. Therefore, our first approach measures the repetitiveness of performance metric values and recommends the stopping of a test when all metric values become highly repetitive.

   To evaluate our approach, we conduct 24-hour performance tests on three online open source systems (i.e., CloudStore, PetClinic and Dell DVD Store). We find that our approach saves up to 75% of the execution time and preserves more than 91% of the performance metric values that are produced during the 24 hours.

2. *Cost-effective Stopping of a Performance Test Using the Repetitiveness in the*

collected Inter-Metrics Relations (Chapter 4): Our second approach addresses a limitation of our first approach. For example, a test generates at hour 1: low CPU utilization values and high memory usage values, at hour 2: low CPU utilization values and low memory usage values, and at hour 3: high CPU utilization values and low memory usage values. Our first approach recommends to stop this test at hour 3 because the CPU and memory metrics generate repetitive values. However, if we look at the inter-metrics relations, i.e., combination between the CPU and memory metrics, the test does not produce any repetitive combination.

The inter-metrics relations captures so-called a *system state*. Such state describes how the metrics correlate with each other, which is important for capturing the complex performance behaviour of a system [19]. Our second approach recommends to stop a performance test when it no longer produces new system states.

We evaluate the approach using the same set of software systems. We find that our second approach saves approximately 70% of the execution time and preserves more than 95% of the inter-metrics relations, i.e., system states.

1.3 Major Thesis Contributions

The main contributions of this thesis are as follows:

1. We are the first work to ever study in depth the important research problem of cost- effective stopping of a performance test.

2. We propose two approaches that automatically recommends the stopping of a

performance test by examining the repetitiveness of performance metric values or the inter-metrics relations.

1.4 Organization of Thesis

We proceed by discussing the background and related work of the thesis in Chapter 2. Chapter 3 describes our first approach for recommending the cost-effective execution time of performance tests. Chapter 4 discusses the limitation of our first approach and describes our second approach that addresses the discussed limitation. Chapter 5 concludes the thesis and outlines avenues for future work.

Chapter 2

Background and Related Work

Performance testing is the process of evaluating a system's behaviour under a workload [20]. The goals of performance testing are varied and include identifying performance issues [21], verifying whether the system meets its requirements [22], and comparing the performance of two different versions of a system [12, 13].

Performance tests need to be properly designed in order to achieve such goals [23]. There are three aspects that need to be considered when designing a performance test: 1) workload scenarios, 2) workload intensity and 3) test length. In this chapter, we discuss prior research along these three aspects.

2.1 Workload Scenarios

The first aspect of designing a performance test is to design the workload scenarios that will execute against the system. There are a number of strategies for designing these scenarios. Therefore, researchers have proposed approaches to assist in the design of workload scenarios.

2.1.1 Covering the Source Code:

A relatively naïve way of designing workload scenarios is to ensure that the scenarios cover a certain amount of the source code [24].

2.1.2 Covering the Scenarios that are Seen in the Field:

A more advanced way of designing workload scenarios is to ensure that the scenarios cover a certain amount of the field workload [7].

2.1.3 Covering Scenarios that are Likely to Expose Performance Issues.

A typical test case prioritization approach is based on the number of potential faults that the test case is likely to expose [25] or the similarity between test cases [15].

Noor et al. conducted two studies that prioritize or rank test cases. In their first study [25], they define a new risk measure (i.e., *risk-driven* metric) that assigns a risk factor to a test case. The classical prioritizing metric for test cases is based on previously-failing test cases. A failing test case in a prior release has a high potential to fail in the new release. Their new metric does not only rely on that factor. It also measures the similarity between the two test cases, i.e., from the previous and new releases. The reason behind measuring the similarity is that a new release may introduce new modifications, which did not exist in the prior version of the system. In their second study [15], they propose an approach (i.e., a similarity-based approach) that prioritizes the test cases using their new metric. They rank the test cases based on two factors: 1) whether a test case detected a failure in the previous release and 2) the similarity between that test case and the same test case in the previous release. Then based on the rank of the test cases, they are prioritized.

These approaches often generate workloads that are too large or complex to be used for performance testing. Therefore, performance testers must determine the most important aspects of the workload using a reduction approach. Several approaches exist to reduce these workload.

Reduction based on Code Coverage as a Baseline: Avritzer et al. propose a technique that limits the number of test cases by selecting a subset of the test cases [26]. First, a given system is modelled as a Markov chain, which consists of a

finite number of states (i.e., each state consists of a sequence of rules that were fired as a result of a change in object memory). A subset of the test suite is selected to maximize test coverage (i.e., percentage of unique states that are covered by the test case). Jiang et al. propose a technique that is related to Avritzer's technique [26] in that both techniques define a set of different system states. However, Jiang et al. use a different definition for a state, which is the active scenarios that are extracted from the execution logs of the system under the test. Their technique reduces the time of User Acceptance Testing by comparing the scenarios (i.e., extracted from workload scenarios logs) of a current test and a baseline test [27]. The intuition of their technique is to measure whether all possible combinations of workload scenarios have already been covered in the test. The technique first identifies all the combinations of workload scenarios from prior tests. In a new test, if most of the combinations of workload scenarios have appeared, the test can be stopped. However, such a technique may not be effective for a performance test. Some workload scenarios with performance issues, such as memory leaks, would not have a large impact on system performance if they are only executed once. Such workload scenarios need to be repeated for a large number of times in order to unveil performance issues.

Hybrid Reduction Approaches: Several researchers have proposed reducing the number of workload scenarios based on analyzing multiple dimensions. Mondal et al. propose a metric to prioritize the selection of test cases that maximizes both *code coverage* and *diversity* among the selected test cases using a multi-objective optimization approach [16]. Shihab et al. propose an approach that prioritizes lists of functions that are recommended for unit testing by extracting the development history of a given system [28]. Cangussu et al. propose an approach based on an adaptive

or random selection of test cases by using polynomial curve fitting techniques [14]. Hemmati et al. evaluate the effectiveness of three metrics (i.e., *coverage*, *diversity*, and *risk*) for test case prioritization [29]. Their work concludes that historical riskiness is effective in prioritizing test case in a rapid release setting. Elbaum et al. [30] propose a test case selection and prioritization techniques in a continuous integration development environment. Test cases that would execute modules that are related to newly-changed code are prioritized over other tests.

2.2 Workload Intensity

The second aspect of designing a performance test is to specify the intensity of the workload (e.g., the rate of incoming requests or the number of concurrent requests). There are two strategies for designing a performance test:

2.2.1 Steady Workload:

Using this strategy, the intensity remains steady throughout the test. The objective of this strategy, for example, can be to verify the resource requirements such as CPU and response time for a system under a test [31], or to identify performance issues (e.g., memory leaks) [32].

2.2.2 Step-wise Workload:

Using this strategy, the intensity of a workload varies throughout the test. For example, a system may receive light usage late at night in comparison to other peak hours. Therefore, another strategy of designing a test is by changing usage rates (i.e., workload intensity). The *step-wise* strategy refers to increasing the workload

intensity periodically. This strategy may be used to evaluate the scalability of a system [2]. Increasing the workload intensity may uncover the ability of the system to handle heavy workloads. Furthermore, Hayes et al. [33] describe the approach of adjusting the number of concurrent users as *not realistic* as it may give misleading results (i.e., any serious workload variation in the field may lead to performance-related failures).

2.3 Test Execution Time

The third aspect of designing a performance test is to specify the duration of the test. The chosen test cases or workload scenarios can have a finite length. During a performance test, the execution of these test cases are repeated multiple times [34]. Jain [17] designs an approach to stop a performance test by measuring the variances between response time observations. His approach recommends stopping the performance test when the variance is lower than 5% of the overall mean of the response time values. Busany et al. [18] propose an approach that can be used to reduce test length by measuring the repetitiveness of log traces.

Intuitively, a performance tester may stop a test if the test does no longer offer new information. Therefore, in this thesis, we propose two automated approaches that reduce the execution time for performance tests by measuring the repetitiveness in the performance metrics that are collected during a test.

Chapter 3

Cost-effective Stopping of a Performance Test Using the Repetitiveness in the Collected Performance Metric Values

3.1 Chapter Introduction

In this chapter we present an approach that automatically determines when to stop a performance test. Our approach measures the repetitiveness of the performance metrics during a performance test. We use the raw values of these metrics to measure the repetitiveness of the metric values and we use the delta of the raw values, i.e., the differences in the raw metric values between two consecutive observations of these performance metrics, to measure the repetitiveness of the observed trends in the performance metrics. To automatically determine whether it is cost-effective to stop a test, our approach examines whether the repetitiveness has stabilized. Our intuition is that as the test progresses, the system's performance is increasingly repetitive. However, this repetitiveness eventually stabilizes at less than 100% because of transient or unpredictable events (e.g., Java garbage collection). Our approach would recommend stopping the test if the repetitiveness (for either raw values or delta values) stabilizes. In addition, the repetitiveness measured by our approach can be leveraged as a reference for performance testers to subjectively determine whether to stop a performance test or not.

To evaluate our approach, we conduct a case study on three open-source systems (i.e., CloudStore, PetClinic and Dell DVD Store). We conducted performance tests on these systems. We use a random workload for Dell DVD Store, and use a steady workload for CloudStore, PetClinic. We then used our approach to recommend when each test should stop. We measure the repetitiveness in performance metrics by running each test for 24 hours.

We find that the data that would be generated after the recommended stopping time is 91.9% to 100% repetitive of the data that is collected before our recommended

stopping time. Such results show that continuing the test after the recommended stopping time generates mostly repetitive data, i.e., little new information about the performance. In addition, we calculate such repetitiveness for every hour during the test and we find that the delay between the most cost-effective stopping time and our recommended stopping time is short, i.e., within a three-hour delay.

This chapter is organized as follows. Section 3.2 provides a motivation example in order to give a clearer idea of where and when to use our approach. Section 3.3 explains the phases of our approach. We discuss our case study in Section 3.4, followed by the results in Section 3.5. Then threats to the validity of our work are presented in Section 3.6. We conclude the chapter in Section 3.7.

3.2 A Motivating Example

Eric is a performance tester for a large-scale software system. His job is to conduct performance tests before every release of the system. The performance tests need to finish within a short period of time, such that the system can be released on time. In order to finish the performance tests before the release deadline, Eric needs to know the most cost-effective length of a test. Eric usually performs the appropriate length of a performance test based on his experience, gut feelings, and (unfortunately) release timelines.

A performance test consists of the repeated execution of workload scenarios. Hence, Eric develops a naïve approach that verifies whether a performance test has executed all the scenarios. Once the test has executed all workload scenarios at least once, the test is stopped. However, some performance issues (e.g., memory leaks) may only appear after a large number of executions. Stopping the performance test after

the single execution of each workload scenario would not detect such performance issues.

Eric uses performance metrics to analyze the system's performance. If the performance metrics become repetitive, continuing the test would not provide much additional information. In addition, Eric found that if he observed trends of a performance metric in the beginning of a test, he can use the trend to calculate the metric values in the rest of the test. Table 3.1 shows an imaginary example of four performance metrics that are generated during a performance test. During the performance test, the memory usage has an increasing trend. Therefore, the delta between every two consecutive memory usage values are also shown in the table.

In this example, i.e., Table 3.1, the values of the performance metrics from time stamp 304 to 308, i.e., the time period rep_1 in Table 3.1, and 310 to 314 (i.e., rep_2), are repetitive to (i.e., exactly the same values as) time stamp 326 to 330 (i.e., rep'_1), and 321 to 325 (i.e., rep'_2). If the test is stopped at time stamp 321, Eric would not miss any performance metric value from the test, while the total duration of the test case would be reduced to 321 minutes.

Therefore, Eric re-designed his automated approach to recommend the stopping of a performance test based on the repetitiveness of values or trends of the performance metrics that are generated during the test. Once the data that is generated during a performance test becomes highly repetitive, the approach recommends the stopping of the test.

In practice, tests last for hours or days and hundreds or thousands of performance metrics are generated. Therefore, performance tests need a scalable and automated approach to determine when to stop. In the next section of this chapter, we explain this automated approach in detail.

Table 3.1: A motivational example of our approach for stopping a test using the repetitiveness of performance metric values

Time	RT	CPU	MEM	△ MEM	IO	
301	82	9	86	86	90	
302	26	7	113	27	3	
303	80	7	135	22	70	
304	**81**	**52**	**182**	**47**	**77**	
305	**83**	**99**	**224**	**42**	**74**	
306	**12**	**53**	**229**	**5**	**57**	rep_1
307	**5**	**99**	**229**	**0**	**67**	
308	**17**	**93**	**240**	**11**	**37**	
309	37	40	319	79	25	
310	**62**	**29**	**396**	**77**	**47**	
311	**83**	**61**	**411**	**15**	**10**	
312	**98**	**69**	**428**	**17**	**93**	rep_2
313	**71**	**22**	**494**	**66**	**2**	
314	**31**	**79**	**556**	**62**	**13**	
315	68	15	568	12	86	
316	25	26	616	48	77	
317	18	65	706	90	27	
318	36	53	770	64	87	
319	51	1	843	73	53	
320	100	93	942	99	62	
321	**62**	**29**	**1,019**	**77**	**47**	
322	**83**	**61**	**1,034**	**15**	**10**	
323	**98**	**69**	**1,051**	**17**	**93**	rep'_2
324	**71**	**22**	**1,117**	**66**	**2**	
325	**31**	**79**	**1,179**	**62**	**13**	
326	**81**	**52**	**1,226**	**47**	**77**	
327	**83**	**99**	**1,268**	**42**	**74**	
328	**12**	**53**	**1,273**	**5**	**57**	rep'_1
329	**5**	**99**	**1,273**	**0**	**67**	
330	**17**	**93**	**1,284**	**11**	**37**	

3.3 Our Approach for Determining a Cost-Effective Length of a Performance Test

In this section, we present our approach for determining a cost-effective length for a performance test. Figure 3.1 presents an overview of our approach.

Table 3.1 shows a motivational example[1] of performance metrics that would be collected during a performance test. The performance metrics in Table 3.1 are collected every minute. The values of the performance metrics at each time stamp are called observations. To ease the illustration of our approach, we show a small example with only four performance metrics (i.e., response time, CPU usage, memory usage, and I/O traffic) and 30 observations. However, the number of performance metrics and observations is much larger in practice.

To determine when to stop the performance test, we periodically (e.g., every minute) collect performance metrics during the test. After collecting the metrics, we determine whether to use the raw values or the delta values of each metric. We then measure the likelihood of repetitiveness. Finally, we determine when repetitiveness stabilizes (and the test can be stopped) using the *first derivative* of each metric.

We fit a smoothing splines to the likelihood and determine whether the likelihood of repetitiveness has stabilized using the first derivative. Our intuition is that as the test progresses, the system's performance is increasingly repetitive. However, this repetitiveness eventually stabilizes at less than 100% because of transient or unpredictable events. Therefore, we aim to identify this stabilization point. In the rest of this section, we present our approach in detail.

[1]We do not use real performance data, as such data tends to contain large values, which would hinder the readability of this chapter.

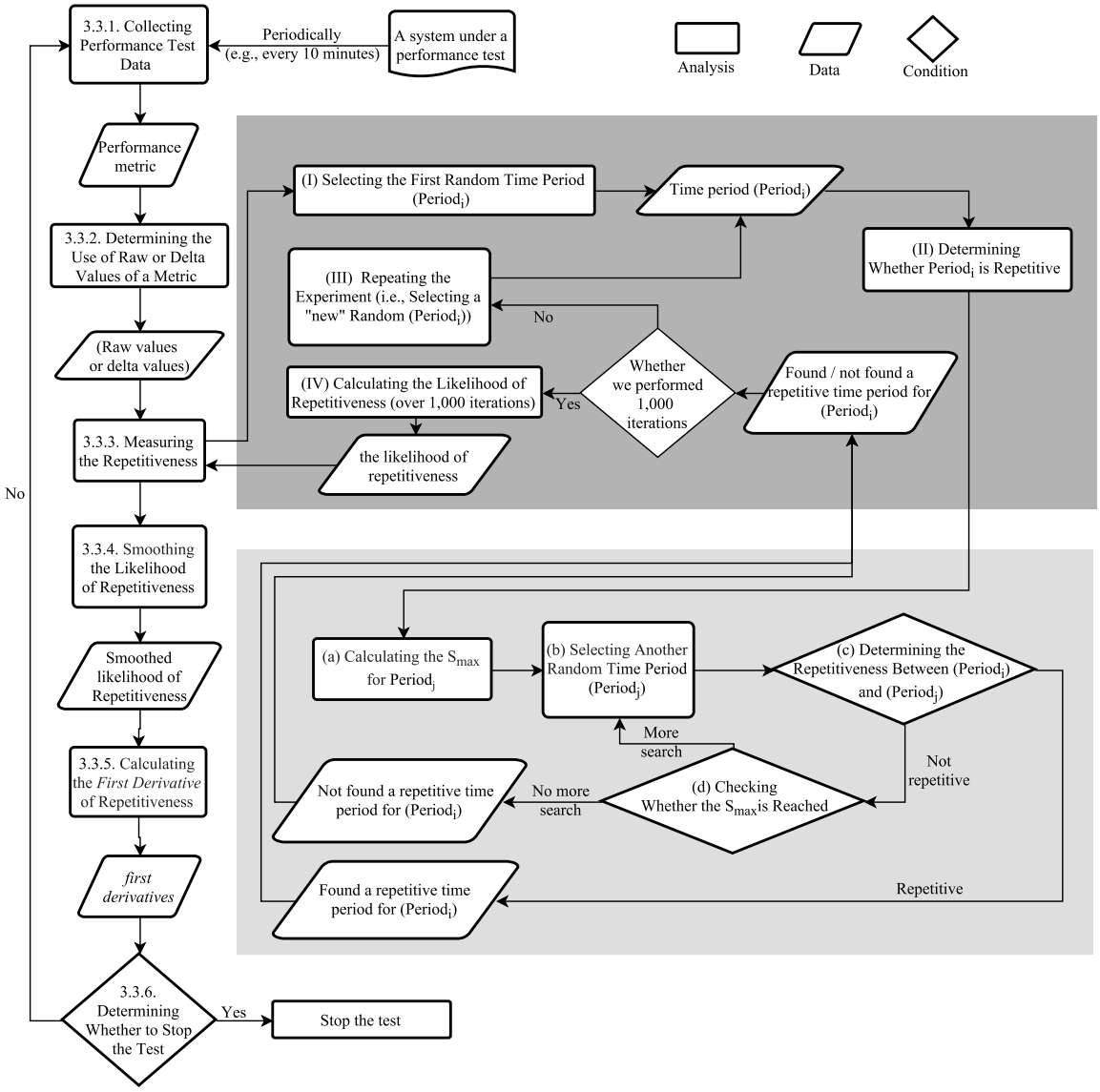

Figure 3.1: An Overview of Our Approach

3.3.1 Collecting Performance Test Data

The collected performance metrics are the input of our approach. Typical performance monitoring techniques, such as PerfMon [35], allow users to examine and analyze the performance metric values during the monitoring.

We collect both raw values of the metrics and the delta of metrics between two consecutive observations of each metric. We design our approach such that we either see highly repetitive values of the metrics, or repetitive trends of the metrics. Therefore, we collect raw values of the metrics for measuring the repetitiveness of the metric values and we collect the delta of the metrics to measure the repetitiveness of the metric trends.

Since our approach runs periodically, in our working example, the first time when we run our approach, we collect metrics from the beginning of the test to the time stamp 320. The second time when we run our approach, we collect metrics from the period of time from the beginning of the test to the 321 minute since we choose to run our approach periodically every minute.

3.3.2 Determining the Use of Raw or Delta Values of a Metric

Performance metrics may illustrate trends during performance tests. For example, memory usage may keep increasing when there is a memory leak. On the other hand, some metrics do not show any trends during a performance test. In this step, i.e., every time before we measure the repetitiveness of the generated performance metric values, we determine for each metric, whether we should use the raw or delta values. We leverage a statistical test (Mann-Kendall Test [36]) to examine whether the generated values of a metric has a monotonic trend or not. The null hypothesis of the test is that there is no monotonic trend in the values of a metric. A p-value smaller than 0.05 means that we can reject the null hypothesis and accept the alternative hypothesis, i.e., there exists a monotonic trend in the values of a metric. If there exists a statistically significant trend, we would use the delta values of the metric,

otherwise, we would use the raw values of the metric.

For our working example, for the first time period, i.e., the 301^{st} to the 320^{th}, the memory metric is the only metric that has a statistically significant trend (i.e., a p-value that is <0.05 in the Mann-Kendall Test). Therefore, for our working example, we use delta memory instead of raw memory values.

3.3.3 Measuring the Repetitiveness

The goal of this step is to measure the repetitiveness of the performance metrics that have already been collected since the beginning of a performance test. The more repetitive, the more likely that the test should be stopped, since the data to be collected by continuing the performance test is more likely to be repetitive.

It is challenging to measure repetitiveness of performance metrics. Performance metrics are measurements of resource utilizations. Such measurements typically do not have exact matches. For example, two CPU usage values may be 50.1% and 50.2%. It is hard to determine whether such differences between two performance metric values correspond to an actual performance difference or if such differences are due to noise [37]. Statistical tests have been used in prior research and in practice to detect whether performance metric values from two tests reveal performance regressions [12, 38]. Therefore, we leverage statistical tests (e.g., Wilcoxon test [39]) to determine whether the performance metric values are repetitive between two time periods. We choose the Wilcoxon test as it does not have any assumptions on the distribution of the data.

(I) **Selecting the First Random Time Period ($Period_i$)**

In order to measure the repetitiveness of performance metrics, we randomly

select a time period ($Period_i$) from the performance test and check whether there is another period during the performance test that has generated similar data. The length of the time period len_τ is a configurable parameter of our approach. In our working example, the performance test data shown in Table 3.1 has 30 observations. With a time period of 5 observations ($len_\tau = 5$), we may select time period $Period_i$ from time stamp 317 to 321.

(II) **Determining Whether $Period_i$ is Repetitive**

In this step, we determine whether there is another time period during the performance test that has similar data to $Period_i$. First, we randomly select another time period ($Period_j$). We do not consider overlapping time periods $Period_i$. For example, in Table 3.1, any time period that contains observations from time stamp 317 to 321 is not considered. Second, we compare the data from $Period_i$ to the data from $Period_j$ to determine whether they are repetitive. If not, another time period is randomly selected (a "new" $Period_j$) and compared to $Period_i$. If we cannot find a time period $Period_j$ that is similar to $Period_i$, we consider $Period_i$ as a non-repetitive time period.

(a) **Calculating the S_{max} for $Period_j$:** One may perform an exhaustive search on all the possible time periods to find a time period $Period_j$ that generates similar data as $Period_i$. For our working example, since a time period consists of five observations (configured as a parameter), there are 16 possible time periods (i.e., those do not overlap with $Period_i$). Usually a long running performance test, leads to a substantial number of possible time periods. For example, a typical performance test that runs for 48 hours and collects performance metrics every 30 seconds, would contain

over 5,700 possible time periods of 30 minutes. Thus, our approach would likely take a long time to execute. Since our approach aims to stop the performance test to reduce the performance test duration, we determine the number of searches (S_{max}) based on a statistically representative sample size [40].

We consider all possible time periods that do not overlap with $Period_i$ as a population and we select a statistical representative sample with 95% confidence level and ± 5 confidence interval as recommended by prior research [40]. The confidence interval is the error margin that is included in reporting the results of the representative samples.

For example, if we choose a confidence interval of 5, and based on a random sample, we find that 40% of the time periods are repetitive. Such results mean that the actual likelihood of having repetitive time periods is between 35% (40-5) and 45% (40+5). The confidence level indicates how often the true percentage of having repetitive time periods lies within the confidence interval. For the same previous example, there is a 95% likelihood that the actual likelihood of having repetitive time periods is between 35% and 45%.

By selecting a statistical sample of time periods to search for $Period_j$, we can reduce the number of searches as in comparison to exhaustive search. For example, if there are $5,000$ time periods, we only need to randomly sample 357 time periods to compare to $Period_i$. In our working example, the number of random time periods to search for $Period_j$ is 15. Because we use a small size of data (i.e., 16) in our working motivational example,

the size of statistical sample does not have a large difference to the size of the entire data. With larger data, we would have a considerably larger reduction in the size of the statistical sample relative to the entire data.

First of all, *Sample Size (SS)* is determined as a constant value (i.e., 384) in our approach because we use fixed configurations. Our configurations are 95% confident level, and a confidence interval of ±5 as Then we calculate the representative sample, i.e., the number of random time periods to search for $Period_j$, out of the total number of possible time periods *Pop*. To compute that, we use the formula in [40].

$$S_{max} = \frac{384}{1 + (\frac{(384)}{Pop})} \tag{3.1}$$

To compute *Pop*, we filter out the time periods and we eliminate the time periods that are less than the selected time period length. To compute *Pop*, we use following formula:

$$Pop = N - (len(\tau) * 3) + 1 \tag{3.2}$$

Where N is the length of the performance metrics file and len_τ is the length of selected time period. For our working example, let say that $Period_i$ is in the range *(317,322)*, and our parameters stay the same (i.e., len_τ= 5 and N=30). We filter out the time periods that are in the range *(313,322)* as per (1) and time periods *(327,330)* as per (2). Therefore, *Pop* as per the Equation 3.2 will be 16.

(b) **Selecting Another Random Time Period ($Period_j$):** We select a

Table 3.2: Wilcoxon test results for our working example

	Performance Metrics			
	RT	CPU	△ Memory	IO
p-values	0.0258	0.313	0.687	0.645

second random time period $Period_j$ with the same length as $Period_i$ and determine whether $Period_i$ and $Period_j$ are repetitive. In our working example, the time period *(304,308)* is randomly sampled as $Period_j$.

(c) **Determining the Repetitiveness Between ($Period_i$) and ($Period_j$):** To determine whether $Period_i$ (e.g., *(317,321)*) and $Period_j$ (e.g., *(304,308)*) are repetitive, we determine whether there is a statistically significant difference between the performance metric values during these two time periods using a two-tailed unpaired Wilcoxon test [39]. Wilcoxon tests do not assume a particular distribution of the population. A $p - value > 0.05$ means that the difference between the metric values from both time periods is not statistically significant and we cannot reject the hypothesis (i.e., there is no statistically significant difference of the metric values between $Period_i$ and $Period_j$). Failure to reject the null hypothesis means that the difference between the metric values from two time periods is not statistically significant. In such cases, we consider $Period_i$ and $Period_j$ to be repetitive. The *Wilcoxon test* is applied to all metrics from both time periods. The p-values of *Wilcoxon tests* for the metrics of our working example are shown in Table 3.2.

The two time periods ($Period_i$ and $Period_j$) are considered **repetitive** if

all the differences of all the metrics are not statistically significantly different between two time periods. For our working example, the difference between the response time (RT) in two time periods is statistically significant. Therefore, we do not consider the two time periods as repetitive. In this case, another random time period $Period_j$ would be selected if the number of search iterations is not reached.

(d) **Checking Whether the S_{max} is Reached:** If the total number of searches for a repetitive time period for $Period_i$ is not yet reached, our approach will continue the search by selecting another random time period as $Period_j$. Otherwise, the search will stop and $Period_i$ will be reported as **not repetitive**.

For our working example, when a random time period that consists of observation 301 to 305 is selected as $Period_j$, the two time periods are repetitive (i.e., p-values of all metrics are greater than 0.05). Thus, $Period_i$ is reported as **repetitive**.

(III) **Repeating the Experiment**

The entire process (i.e., randomly select $Period_i$ and search for a repetitive time period $Period_j$) is repeated for a large number of times (i.e., $1,000$ times) to calculate the repetitiveness of the performance metrics. Bootstrap is a statistical technique that makes an inference of a data set. Efron et al., state that when using bootstrap, 1,000 replications from a data set can make an inference [41]. Therefore, our approach repeats this process for 1,000 iterations. In every iteration, our approach will report whether the $Period_i$ is **repetitive** or **not repetitive**.

(IV) **Calculating the Likelihood of Repetitiveness**

We determine the repetitiveness by calculating the likelihood that a randomly selected $Period_i$ is determined to be **repetitive**. In particular, we divide the number of times that $Period_i$ is reported as **repetitive** by $1,000$. After repeating the process in our working example $1,000$ times, the calculated likelihood of repetitiveness is 81.1%. This means that after $1,000$ iterations, a repetitive time period is found 811 times.

3.3.4 Smoothing Likelihood of Repetitiveness

We fit a smoothing spline to the likelihood of repetitiveness that is measured so far while running the test to identify the overall trend in the likelihood of repetitiveness (i.e., increasing or decreasing). The smoothing spline helps reduce the influence of short term variations in repetitiveness and increases the influence of the long term trends. We use the `loess()` function in R [42] to fit the smoothing spline.

3.3.5 Calculating the First Derivative of Repetitiveness

To identify the time point at which the likelihood of repetitiveness stabilizes, we calculate the *first derivative*. These *derivatives* quantify the difference between successive likelihoods. When a *derivative* reaches ~ 0, the test can be stopped. A *derivative* is calculated as follows:

$$Derivative = \frac{likelihood_{current} - likelihood_{previous}}{Periodically_{diff}} \tag{3.3}$$

Where the $Periodically_{diff}$ shows the number of minutes between the last time stamp at which the previous likelihood is calculated, and the last time stamp at

which the current likelihood is calculated in minutes. For our working example, the difference between calculating a likelihood and another is one minute.

3.3.6 Determining Whether to Stop the Test

In the final phase, we identify two parameters as configurations of our approach: 1) the threshold of first derivatives of repetitiveness (*Threshold*) and 2) the length of time that the first derivative of repetitiveness is below the threshold (*Duration*). If the first derivatives of repetitiveness is below the *Threshold* for equal or more than the *Duration*, our approach would recommend stopping the test. Moreover, *Duration* not only checks for the *Threshold*, but it also makes sure that the type of data of every metric does not change, i.e., every metric should stick with the same type of data (either raw data or delta data).

For our working example, a *Threshold* and a *Duration* are determined (i.e., as 0.1 and 3-minute, respectively). Therefore, the test can be stopped at the 326 minute because its first derivative is 0.031, the two following (i.e., 0.037 and 0.093) are less than our *Threshold* (i.e., 0.01), and the type of data does not change.

3.4 Experiment Setup

We evaluate our approach for determining when to stop a performance test, using three different large systems, i.e., CloudStore [43], PetClinic [44] and Dell DVD Store (DS2) [45]. In this section, we present the subject systems, the workloads that are applied to them, and our experimental environment.

3.4.1 Subject Systems

CloudStore [43] is an open-source e-commerce system, which follows the TPC-W performance benchmark standard [46]. CloudStore is built for performance benchmarking.

PetClinic [44] is an open-source web system providing a simple yet realistic design of a web system. PetClinic was used in prior performance engineering studies [1, 11, 47].

Dell DVD store (DS2) is an open-source web system [45] that benchmarks new hardware system installations by simulating an electronic commerce system. DS2 was also used in prior performance engineering research [12, 13].

3.4.2 Deployment of the Subject Systems

We deploy the three subject systems in the same experimental environment, which consists of 2 Intel CORE i7 servers running Windows 8 with 16G of RAM. One server is used to deploy the systems and the other server is used to run the load driver. For all subject systems, we use Tomcat [48] 7.0.57 as our web system server and MySQL [49] 5.6.21 as our database.

3.4.3 Performance Tests

To test our subject system, we use the workloads that mimic the real-life usage of the system under the test, and cover all of the common features [50]. The workloads of CloudStore and PetClinic are generated using Apache JMeter [51]. We used a workload that was used in performance engineering research studies: for PetClinic [47], and for CloudStore [52]. The DS2 system has its own load generator, which generates

a workload. The performance tests of all three subject systems are run for 24 hours. We used two different types of load intensities: random and steady. During DS2 test, the load intensity is randomly changed every 10 minutes, while we used steady load for CloudStore and PetClinic. This way, we evaluate our approach using two different types of load intensities.

3.4.4 Data collection

We collect two types of performances metrics during the execution of the performance tests.

Physical metrics. The physical performance metrics capture the performance of the execution environment during the performance tests. In particular, we use a performance monitoring system, PerfMon [35], to collect physical performance metrics. In this experiment, we collect the CPU utilization, memory usage, and disk usage (I/O) of the processes of our subject systems.

Domain metric. The domain performance metrics capture the performance of the system from the users' perspective. We collected the average response time (RT) as our domain performance metric. RT measures the average time that a request takes to execute, from the moment that a user sends a request to the time the user receives the response. RT is used in prior performance engineering studies to measure the performance of a system [53, 54]. In our experiment, the DS2 load driver automatically calculates RT during the tests. We calculate RT in the CloudStore and PetClinic experiments by analyzing log files that are generated by JMeter during the tests.

It is challenging to analyze performance metrics that are generated from different sources. Because of the clock skew [38], the data that is generated from PerfMon,

JMeter log files, and the DS2's load generator may not be generated at exactly the same time. Those resources generate the performance metric measurements at every 10 seconds. To address this challenge, we calculate the average value of every three consecutive measurements. Hence, every 30 seconds we take the average of the metric values at 10, 20, and 30 seconds. For example, PerfMon may record the CPU utilization at 12:01 (i.e., 12 minutes and 1 second), 12:11 and 12:21, while the response time measurements are generated at 12:05, 12:15 and 12:25. When joining the two metrics, we use the average values of the metric as the value at 12:30.

3.4.5 Parameters of Our Approach

To determine when to stop a performance test, we apply our approach on the performance metrics of the subject systems. Our approach requires three parameters to be configured: 1) the length of a time period (len_τ), 2) the threshold of the first derivative of repetitiveness (*Threshold*) and 3) the length of time that the first derivative of repetitiveness is below the threshold (*Duration*).

We choose three values of each parameter in order to evaluate the sensitivity of our approach with different parameters. For the length of a time period, we choose two values including 15 minutes and 30 minutes. For the threshold of derivatives of repetitiveness, we choose 0.1, 0.05 and 0.01. Finally, for the length of time that the first derivative of repetitiveness is below the threshold, we choose 30 minutes, 40 minutes and 50 minutes. Therefore, we have 18 different combinations of configurations for each subject system to evaluate our approach. In addition, our approach needs to run periodically with the performance test. In our case study, we choose to run our approach every 10 minutes in order to get frequent feedback on the repetitiveness of

the performance metrics.

3.4.6 Leveraging Existing (Jain's) Approach to Stop Performance Tests

Jain proposes an approach to stop a performance test by measuring the variances between response time observations [17]. We benchmark our approach by comparing the recommended stopping time of our approach to the recommended stopping time of Jain's approach. First, we group every consecutive number of response time observations into a number of batches. To determine the optimal size of batches, we keep increasing the size of batches and measure the mean of variance. The optimal size of a batch is the size before the mean variance drops [17]. In our experiments, we find the optimal sizes of batches are 1.5 minutes for CloudStore and 2 minutes for both DS2 and PetClinic. We then calculate the means of response time observations for every batch along with the overall means of response time values. Using the variances among the means of every batch, we calculate the confidence interval of the batch means of response time observations. During the test, the moment the confidence interval drops less than 5% of the overall means, the approach stops the test.

Jain's approach recommends extremely early stopping time. We find that Jain's approach recommends stopping our tests shortly after starting the test. The recommended stopping times are 7.5 minutes, 6 minutes and 20 minutes for CloudStore, DS2 and PetClinic, respectively. Intuitively, such extremely early stopping times cannot be used in practice since many performance issues, e.g., memory leak, can only appear after running the system for a long period of time.

3.4.7 Preliminary Analysis

The assumption and the most important intuition of our approach is that the performance tests results are highly repetitive. Therefore, before evaluating our approach, we perform a preliminary analysis to examine whether the performance tests results are repetitive. We measure the repetitiveness of the performance metrics that are collected during a 24-hour performance test.

All performance tests from the three subject systems are highly repetitive. The repetitiveness of PetClinic and DS2 is close to 100% and repetitiveness of CloudStore is between 92% to 98%. Such results mean that if we randomly pick performance metrics from a 15 or 30 minutes len_τ period from the performance tests, there is an over 92% likelihood that we can find another time period during the performance tests that either generates similar performance metric values or shows a similar trend. Such a high repetitiveness confirms that our approach may be able to stop the test within a much smaller amount of time.

3.5 Case Study Results

In this section, we present the results of evaluating our approach. Table 3.3 shows when our approach recommends that the performance test be stopped with different parameters. An undesired stopping time may be ether too early or too late. If a performance test stops too early, important behaviour may be missed. On the other hand, one may design an approach that stops the tests very late to ensure that there is no new information (not repeated data) is generated after the stopping time. However, stopping the tests late is against our purpose of reducing the length of a performance test.

Table 3.3: The stopping times that are recommended by our approach for performance tests. The values of the stopping times are hours after the start of the tests.

	Duration	**30 minutes**			**40 minutes**			**50 minutes**		
	Threshold	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**
CloudStore	15 minutes	5:10	6:20	7:10	5:10	6:20	7:10	5:10	6:20	7:10
	30 minutes	5:30	6:10	11:20	5:30	6:10	14:10	5:30	6:10	18:20
DS2	15 minutes	7:50	7:50	9:10	7:50	7:50	9:10	7:50	7:50	9:10
	30 minutes	8:10	8:20	9:40	8:10	8:20	9:40	8:10	8:20	9:40
PetClinic	15 minutes	4:20	4:30	9:30	4:20	4:30	9:30	4:20	4:30	9:30
	30 minutes	4:20	6:20	7:30	4:20	6:20	7:30	4:20	6:20	7:30

To evaluate whether our recommended stopping time is too early, we measure how much of the generated data after our recommended stopping time is repetitive of the generated data before our recommended stopping time. The repetitiveness captures how much behaviour is missed when we stop the test early. To evaluate whether our recommended stopping time is too late, we evaluate how long is the delay between the recommended stopping time and and the cost-effective stopping time.

Evaluating whether the recommended stopping time is too early. We run a performance test for 24 hours. We note the time when our approach recommends that the test be stopped. We then divide the data that is generated from the test into data generated before the stopping time (i.e., *pre-stopping data* and (a) in Figure 3.2) and data generated after the stopping time (i.e., *post-stopping data* and (b) in Figure 3.2).

We follow a similar approach as described in Section 3.3 to measure the repetitiveness between the pre-stopping data and the post-stopping data. First, we first select a random time period ($Period_i$) from the post-stopping data. Second, we determine whether $Period_i$ is repetitive by searching for a $Period_j$ with performance metric data that is not statistically significantly different from $Period_i$. Third, we repeat

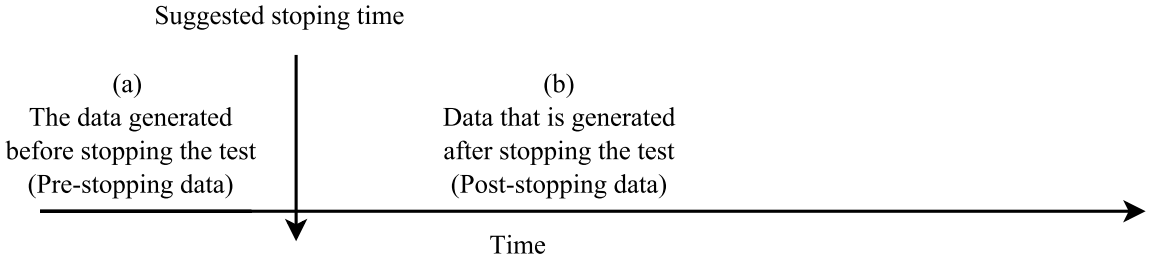

Figure 3.2: Our Approach that Evaluates Whether the Recommended Stopping Times is too Early

this process, i.e., selecting random $Period_i$ from (b) in Figure 3.2), and $Period_j$ from (a), $1,000$ times. Finally, we calculate the likelihood that we can find a repetitive time period between the pre-stopping data and the post-stopping data.

To compute the likelihood that we can find a repetitive time period (i.e., the repetitiveness likelihood), we assume that B is the set of time periods before the stopping time, and A is the set of time periods after the stopping time. $\bar{A} \subset A$ where $\forall\ \bar{a} \in \bar{A}$ there exists a $b \in B$ that is repetitive of $\bar{a} \in \bar{A}$. Therefore, the repetitiveness likelihood is the size of $\bar{A}$ divided by the size of A.

A very high repetitiveness likelihood indicates that continuing the test is not likely to reveal much new information about the system's behaviour. When the repetitiveness likelihood is high, then our approach recommended a cost-effective time to stop the test. Conversely, when the repetitiveness likelihood is low, then our approach recommended stopping the test too early (i.e., we stopped the test before all of the system's behaviour could be observed). Therefore, the higher repetitiveness likelihood, the better (i.e., more cost-effective) our decision to stop the test.

Evaluating whether the recommended stopping time is too late. First, we identify the most cost-effective stopping time. In particular, we calculate the repetitiveness likelihood if we naïvely stopped the test at the end of every hour during

the test. Then at every hour, we calculate *EffectivenessScore* using the following formula:

$$EffectivenessScore = (R_h - R_{h-1}) - (R_{h+1} - R_h) \tag{3.4}$$

where R is a repetitiveness likelihood at the hour h. We use the *EffectivenessScore* to find the hour during the test that has maximum increase of repetitiveness before the hour and minimum increase of repetitiveness after the hour. The hour with highest score is considered the most cost-effective stopping time. Finally, we measure the delay between the recommended stopping time and the most cost-effective stopping time.

Note that, one cannot know such most cost-effective time before finishing the test to gather the complete data set. We use the most cost-effective stopping time to evaluate whether our approach is able to recommend the stopping of the test with minimal delay.

For example, if we find that a sequence of likelihood of repetitiveness from the 1_{st} hour to the 6_{th} is (12%, 16%, 89%, 92%, 97%, 97.5%), and the recommended stopping time is at the 6_{th} hour. We first calculate the *EffectivenessScore*, and they are from the 2_{nd} to the 5_{th} hour as follows: (-69, 70, -2, -4.5). Therefore, the most cost-effective stopping time is the 3_{th} hour, and the delay is 3 hours.

Results. **There is a low likelihood of encountering new data after our recommended stopping times.** Table 3.4 shows the stopping time and the likelihood of having repetitive data after the stopping time. We find that the likelihood of seeing repetitive data after the stopping time is between 91.9% to 100%. Therefore, the results of our approach are not overly impacted by choosing different values for the

Table 3.4: The percentages of the post-stopping generated data is repetitive of the pre-stopping generated data.

	Duration	**30 minutes**			**40 minutes**			**50 minutes**		
	Threshold	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**
CloudStore	15 minutes	100	100	100	100	100	100	100	100	100
	30 minutes	100	100	100	100	100	100	100	100	100
DS2	15 minutes	97	97	100	97	98	100	98	97	99.9
	30 minutes	91	98	99	94	97	99	92	98	100
PetClinic	15 minutes	100	100	100	100	100	100	100	100	100
	30 minutes	100	100	100	100	100	100	100	100	100

parameters.

There is a short delay between the most cost-effective stopping times and the stopping times that are recommended by our approach. Table 3.5 shows the delay of our approach to find cost-effective stopping times for the tests. Out of 54 stopping times in Table 3.5, 27 are under three hours away from the most cost-effective stopping times. Whereas, only six of the stopping times that are recommended by our approach are more than six hours away from the cost-effective times.

The measurement of repetitiveness can be used by stakeholders to subjectively decide when to stop a performance test. Our approach recommends when to stop a performance test and measures the repetitiveness of the performance metrics, during the performance test. Such measurements quantify the risk that is associated with stopping a test early. Performance testers and other stakeholders (e.g., project managers and release engineers) can leverage such information to subjectively decide whether they are willing to take the risk of stopping a the test early (i.e., comfort level).

Table 3.5: The delay between our recommended stopping times and the most cost-effective stopping times.

	Duration	**30 minutes**			**40 minutes**			**50 minutes**		
	Threshold	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**	**0.1**	**0.05**	**0.01**
CloudStore	15 minutes	1:10	2:20	3:10	1:10	2:20	3:10	1:10	2:20	3:10
	30 minutes	0:30	1:10	6:20	0:30	1:10	9:10	0:30	1:10	13:20
DS2	15 minutes	3:50	3:50	5:10	3:50	3:50	5:10	3:50	3:50	5:10
	30 minutes	5:10	5:20	6:40	5:10	5:20	6:40	5:10	5:20	6:40
PetClinic	15 minutes	0:20	0:30	5:30	0:20	0:30	5:30	0:20	0:30	5:30
	30 minutes	0:00	1:20	2:30	0:00	1:20	2:30	0:00	1:20	2:30

The post-stopping data is highly repetitive (91.9% to 100%) relative to the pre-stopping data. There is only a short delay between the recommended stopping times by our approach and the most cost-effective stopping times. In 27 out of 54 cases, the delay is under 3 hours away from the optimal stopping times.

3.6 Threats to validity

In this section, we discuss the threats to the validity of our first study.

3.6.1 Threats to Internal Validity

Statistical Tests. Our approach leverages two statistical tests. *Wilcoxon tests* are used to determine whether two time periods are repetitive, and *Mann-Kendall tests* are used to determine whether to use either deltas or raw data. Our use of statistical tests poses a threat to validity in that the choices of the tests (whether the time periods are repetitive, or whether to use delta or raw data) is determined by a threshold of p-value (i.e., $p < 0.05$). However, such a threshold may be too

lenient or too stringent. For example, *Mann-Kendall tests* may cause Type I error (i.e., accepting the null hypothesis, when it is true) or Type II error (i.e., rejecting the null hypothesis, when it is false) [55].

Choices of Thresholds. Our approach requires three parameters as thresholds to determine whether to stop a performance test. To evaluate the sensitivity of our approach against different thresholds, we chose different settings for each threshold. Then, we re-evaluate our approach with the combinations of the different settings. We find that the choices of two parameters (i.e., len_{τ} and *Threshold*) impact our recommended stopping time, while the choice of *Duration* does not impact our recommended stopping time.

Randomness in the Approach. Our time periods, i.e., $Period_i$ and $Period_j$, are selected randomly. The choice of making random selection was made to speed up our approach. Also, to avoid the negative effect of this random selection, we repeat this selection process for 1,000 iterations. Future studies may consider more evaluation of this randomness by running our approach multiple times.

3.6.2 Threats to Construct Validity

The Length of the Performance Test. To conduct an evaluation of our approach we did not stop our tests at the time when our approach recommends to stop. As it is impossible to run the test forever to know all the possible system states that a test may generate, we chose to run our experiments for 24-hour. However, conducting longer test length may lead to different conclusions.

Determining Repetitiveness. In our approach, we consider two time periods to be repetitive only if none of the performance metrics are statistically significantly

different between two time periods. In practice, domain experts may consider two time periods repetitive even if some of the performance metrics are not statistically significantly different. However, our approach uses a stricter rule to make sure that the two time periods are repetitive to ensure that performance testers will have confidence in our recommendations. Performance testers may choose to use a less strict rule for repetitiveness based on the subject systems and their level of comfort.

Measuring Trends in Performance Metrics. We use the Mann-Kendall Test to determine whether a performance metric has a monotonic trend since the beginning of the test and we use delta values of every two consecutive observations of a performance metrics to measure the trend. However, there might exist other ways to measure trends. For example, one may check the effect size of a trend by calculating the correlation between the raw values and time. In addition, a performance metric may exhibit more complex (e.g., sinusoid trends) trends or the trends may not start at the beginning of the examined test.

The Duration of our Approach. Our approach needs to run periodically during a performance test in order to determine the cost-effective time to stop the test. In our evaluation, we run our approach every 10 minutes. Running our approach more often may have a more accurate data about the repetitiveness of the performance metrics. However, our approach may not be able to finish analysis in time if we run it too often (e.g., every half a minute). However, our approach is scalable since all the iterations for calculating the repetitiveness (see Section 3.3) can be calculated in parallel. In addition, we chose to iterate 1,000 times to calculate the repetitiveness. In practice, one may choose more or fewer number of iterations based on the frequency of running our approach during the test.

3.7 Chapter Conclusion

Performance testing is critical for ensuring the performance of large-scale software systems. Determining the cost-effective length of a performance test is challenging, yet important task for performance testers. Therefore, we propose an approach to automatically recommend when to stop a performance test. Our approach measures the repetitiveness of the performance metric values that are generated since the start of a performance test. Repetitiveness of the performance metrics is due to repeating values or repeating trends in the metrics. If the repetitiveness of performance metrics stabilizes, our approach recommends the stopping time of the test.

The highlights of this chapter are:

- We propose an approach to measure the repetitiveness of performance metrics.
- We propose an approach to automatically recommend when to stop a performance test based on the repetitiveness of performance metrics.
- Our approach recommends a stopping time that is close to the most cost-effective stopping time (i.e., the stopping time that minimize the duration of the test and maximizes the amount of information about the system's performance provided by performance testing).

Chapter 4

Cost-effective Stopping of a Performance Test Using the Repetitiveness in the collected Inter-Metrics Relations

4.1 Chapter Introduction

In Chapter 3, we propose an approach that attempts to reduce the time that is required to run a performance test, by recommending to stop when performance metric values that are monitored during a test become repetitive. The disadvantage of this approach is that it discards the inter-metrics relations, i.e., the combinations of the metrics at a given time during the test. Such a combination shows the relation between the metrics, which is significantly helpful to understand the performance behaviour of a system under a test [19].

Therefore, in this chapter, we revisit our previous approach in Chapter 3 by focusing on the inter-metrics relations, i.e., so-called a *system state*. Throughout the execution of a performance test, the system goes through a variety of system states. A system state is a unique combination of performance metric states. For example, a system state can be the combination of low CPU utilization and high memory usage. Intuitively, as the execution of the test progresses, it exercises system states that are already exercised at an earlier stage of the test. Hence, the longer the test is executing, the less likely it becomes that new system states are being exercised. As a result, the cost-effectiveness of a performance test, i.e., the number of new states that are exercised in a time frame, goes down as the test progresses.

Our proposed approach periodically monitors how many new system states a performance test exercises. Our approach recommends to stop the test when the number of new system states no longer increases. We evaluate our approach by applying it to the same experiments of our three open source systems (CloudStore, PetClinic and Dell DVD Store), in which we use our approach to reduce the execution time of a 24-hour performance test. We find that our approach recommends the stopping of

the tests within 8 hours, which is a reduction of approximately 70% of the original 24-hour execution time. At the recommended stopping times, the tests were able to exercise more than 95% of the system states that are exercised by the 24-hour test.

In summary, the main contributions of our paper are:

1. An approach for reducing the execution time of a performance test, based on checking when a test no longer exercises new system states.

2. A comparison in which we show that our new approach recommends more cost-effective stopping times than our previous approach.

This chapter is organized as follows. Section 4.2 presents a motivational example for our approach. Section 4.3 describes our approach. Section 4.4 describes our experimental setup, followed by the results of our experiments in Section 4.5. Section 4.6 discusses threats to the validity. We conclude the chapter in Section 4.7.

4.2 A Motivating Example

Eric is a performance engineer who works for a large software company. Eric's task is to ensure that every new version of a software product satisfies the performance requirements. However, the company uses continuous delivery to deploy several new versions of the product a day. One major challenge that Eric faces is how to conduct performance tests in the small time-frame that is available between the deployment of two versions of the product.

To make more efficient use of the time that is available for testing, Eric uses our previous approach in Chapter 3 to reduce the execution time of the existing performance tests. However, Eric notices that some performance issues are missed by

the tests. Table 4.1 shows parts of the performance metrics that are collected during the execution of the performance test.

Table 4.1: A motivational example of a performance issue that is missed by a test of which the execution time is reduced using our previous approach. As our previous approach detects repetition on metric values, period p_3 is considered repetitive (i.e., with p_1 for the response time (RT) metric and with p_2 for the CPU metric). The approach presented in this chapter does not consider any of the periods repetitive.

Time	**RT**	**CPU**	
100	11 (S_1)	10 (S_1)	} period p_1, low load
100	10 (S_1)	12 (S_1)	
101	10 (S_1)	10 (S_1)	
⋮	⋮	⋮	
149	49 (S_2)	100 (S_2)	} period p_2, high load
150	50 (S_2)	100 (S_2)	
151	50 (S_2)	100 (S_2)	
⋮	⋮	⋮	
199	11 (S_1)	100 (S_2)	} period p_3, overload
200	10 (S_1)	100 (S_2)	
201	10 (S_1)	100 (S_2)	

The table highlights three periods p_1, p_2 and p_3, which represent the system state, under low load, high load and being overloaded. Using our previous approach in Chapter 3, which searches for repetitiveness in the metric values, p_3 would be considered repetitive and therefore redundant, as the observed response time is similar to that of p_1 and the observed CPU utilization is similar to that of p_2.

However, p_3 clearly represents a state of the system, i.e., an overloaded state, that is not observed earlier in the test. Table 4.1 shows that studying the repetitiveness based on the individual metric is not enough. Studying the combination between metrics can give more information about the test, such as the load type, e.g., high or

low load.

Therefore, Eric transforms the observed metrics into the metric states S_1 (Low) and S_2 (High) and studies the combination in which the states occur. Eric concludes that our previous approach stops the test too early, as the system state S_1 for RT and S_2 for CPU is not covered. Instead, Eric now stops the performance test when there are no new states that are exercised by the test anymore, i.e., most of the exercised states are repetitive.

In the remainder of this chapter, we present our approach for determining when to stop a performance test based on the number of observed new system states.

4.3 Our Approach for Determining a Cost-Effective Length of a Performance Test

Intuitively, during the execution of a long-running performance test, there exists a point in time after which the test no longer provides new information about the performance of the system under test. Hence, the performance test can be stopped at that point in time. The goal of our approach is to find this point in time after which the performance test becomes repetitive.

In this section, we present our approach for determining when to stop a performance test. Figure 4.1 gives an overview of our approach, whose steps are detailed in this section. First, we collect performance metrics from the system under test. We transform the collected metrics into metric states. We define a system state as the combination of metric states at a given time. Finally, we monitor the number of newly-exercised system states during the test execution. Once the approach finds that the test no longer exercises new system states, it recommends to stop the test.

In the rest of this section, we explain our approach in detail.

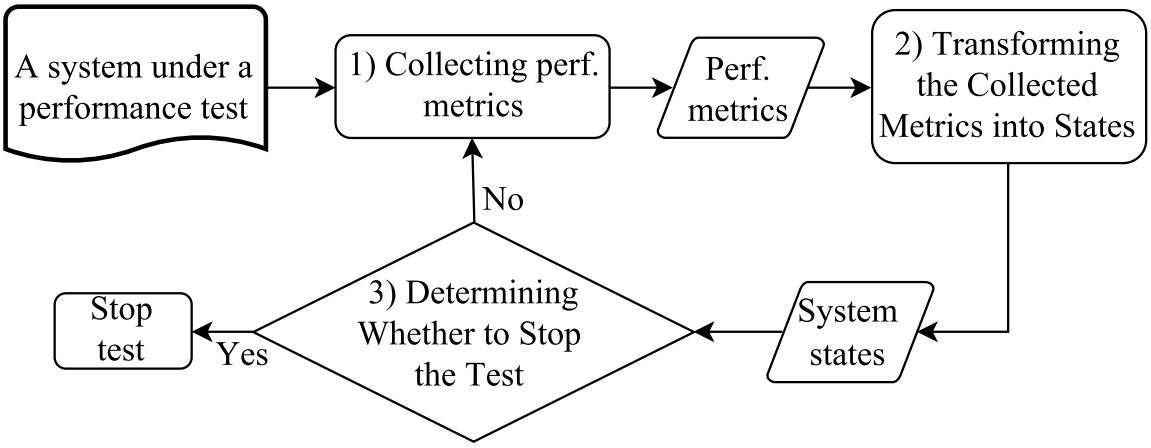

Figure 4.1: An overview of our approach

4.3.1 Collecting Data

We start the performance test and collect performance metrics every s seconds during the execution of the test. Our approach has no limitation on the number and selection of performance metrics that are collected. In this chapter, we demonstrate our approach using the following metrics:

- Response time (RT): The average response time of the requests that were handled in the last s seconds.
- CPU utilization (CPU): The average CPU utilization in the last s seconds.
- Memory usage (MEM): The memory usage in the last s seconds.
- Disk I/O (IO): The average number of bytes that are read and written in the last s seconds.

Table 4.2 contains an example of metric values that are collected during 30 minutes of an imaginary performance test. We will use the example in Table 4.2 throughout this section to demonstrate our approach.

4.3.2 Transforming the Collected Metrics into States

When testing performance at the system level, we are not particularly interested in exercising the precise performance values. Instead, we are interested in exercising a range of values. For example, the difference between 75% and 76% CPU utilization is minor, while the difference between 15% and 75% usage is relevant. Therefore, we transforme the performance metrics values into metric states, i.e., ranges of values. The combination of these metric states at a given time represents a system state.

Handling Metrics with Upward/Downward trends

In some cases, a performance metric may show a trend, in which the exercised values are monotonically increasing or decreasing during the test. Such a monotonic increase or decrease in the values of a so-called *trended* metric has the consequence that the system is exercised with new performance behaviour throughout the test. However, if the rate with which the metric increases or decreases is repetitive, the test exercises repetitive behaviour, and therefore we can stop the test.

To capture repetitiveness in the trend of a metric, we calculate the delta ($\triangle$) values. The $\triangle$ value of a metric is the difference between the current and the previous value of a metric. Figure 4.2 shows the raw and $\triangle$ values for the memory usage metric from the example in Table 4.2, together with a non-trended metric (RT) for reference.

To determine whether to use the raw or delta values of a metric, we calculate

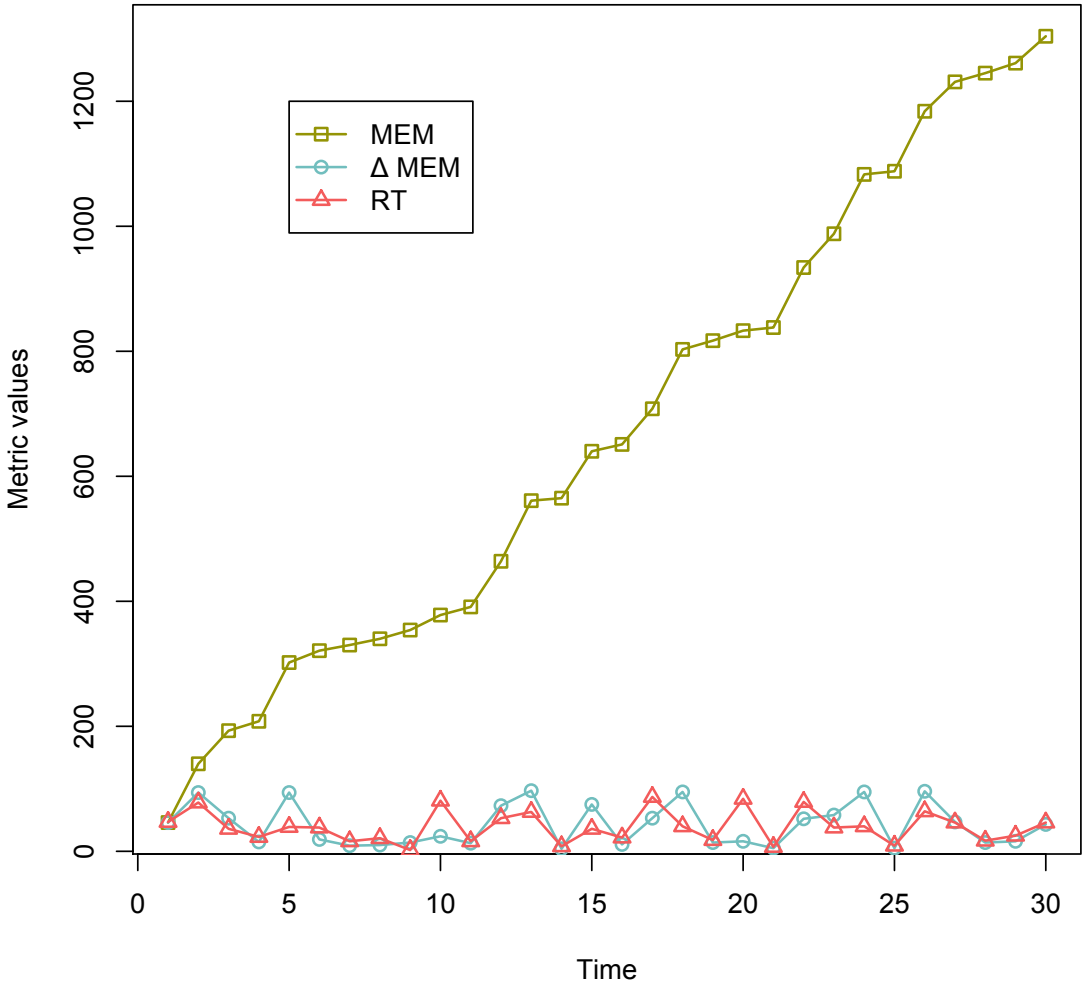

Figure 4.2: The MEM, △ MEM, and RT metrics from the example in Table 4.2

the Spearman correlation between the metric values and the time [56]. We calculate Spearman correlation by using the `cor()` function in `R`. If the metric is significantly correlated with the time (i.e., |correlation value| > 0.5 and p-value < 0.05), we use the △ metric values. Otherwise, we use the raw metric values. For our working example

in Table 4.2, the memory metric has |correlation value| = 1.00 and p-value < 0.05. Therefore, we use the $\triangle$ values for the memory metric, whereas for the other metrics we use raw values.

Table 4.2: An example of performance metrics and metric states. Orange cells show the low value metric state (i.e., S_1), blue cells show the medium value metric state (i.e., S_2), and yellow cells show the high value metric state (i.e., S_3)

t	Performance metric values					Metric state				System state
(mins)	RT	CPU	MEM	$\triangle$ MEM	IO	RT	CPU	$\triangle$ MEM	IO	exercised earlier?
1	47	43	46	46	69	S_3	S_2	S_3	S_2	
2	78	59	140	94	41	S_3	S_2	S_2	S_3	
3	36	32	193	53	12	S_2	S_2	S_1	S_2	
4	23	56	208	15	94	S_2	S_2	S_3	S_2	
5	39	10	302	94	13	S_2	S_1	S_1	S_3	
6	38	50	321	19	78	S_2	S_2	S_3	S_2	✓
7	16	100	330	9	36	S_1	S_3	S_2	S_1	
8	21	83	340	10	48	S_1	S_3	S_2	S_1	✓
9	2	13	354	14	75	S_1	S_1	S_3	S_1	
10	81	42	378	24	35	S_3	S_2	S_1	S_2	
11	16	88	391	13	66	S_1	S_3	S_3	S_1	
12	53	80	464	73	15	S_3	S_3	S_1	S_3	
13	63	23	561	97	48	S_3	S_1	S_2	S_3	
14	8	27	565	4	20	S_1	S_1	S_1	S_1	
15	36	64	640	75	92	S_2	S_3	S_3	S_3	
16	22	84	651	11	47	S_1	S_3	S_2	S_1	✓
17	87	84	704	53	61	S_3	S_3	S_2	S_2	
18	40	9	799	95	12	S_2	S_1	S_1	S_3	✓
19	18	89	813	14	65	S_1	S_3	S_2	S_1	✓
20	84	41	829	16	50	S_3	S_2	S_2	S_2	
21	7	28	834	5	21	S_1	S_1	S_1	S_1	✓
22	79	60	886	52	42	S_3	S_3	S_2	S_2	✓
23	38	31	944	58	13	S_2	S_1	S_1	S_3	✓
24	40	11	1039	95	14	S_2	S_1	S_1	S_3	✓
25	9	29	1044	5	19	S_1	S_1	S_1	S_1	✓
26	64	24	1140	96	49	S_3	S_1	S_2	S_3	✓
27	46	45	1187	47	70	S_2	S_2	S_3	S_2	✓
28	17	89	1201	14	67	S_1	S_3	S_3	S_1	✓
29	25	57	1217	16	93	S_2	S_2	S_3	S_2	✓
30	46	45	1260	43	70	S_2	S_2	S_3	S_2	✓

Transforming the Metric Values into States

We use a binning algorithm (using the `bins()` function in `R` [57]) to transforme the metric values into n states. A well-known usage of this algorithm is by the histogram (`hist()`) function, which uses the binning algorithm to divide a data set into bins to visualize its distribution. Figure 4.3 shows the division of metrics from the example in Table 4.2 into 3 states. The state boundaries of the states are the red vertical lines in the graphs.

4.3.3 Determining Whether to Stop the Test

We count the number of unique system states $state_{unique}$ that the test has exercised every τ minutes. Early in the test, $state_{unique}$ increases rapidly. However, as the test progresses, the exercised system states become repetitive, and $state_{unique}$ stabilizes. Our approach recommends to stop the test when $state_{unique}$ no longer increases.

Table 4.3 shows $state_{unique}$ of the example in Table 4.2 for $\tau = 5$. $state_{unique}$ grows rapidly at the early minutes but stabilizes after $t = 20$. Our approach waits for the $state_{unique}$ for d minutes. For our working example, $d = 5$ minutes. Therefore, our approach recommends to stop the test at 25 minutes, as $state_{unique}$ did not change for 5 minutes.

4.4 Experiment Setup

We conduct an experiment to evaluate our new approach for reducing the execution time of a performance test.

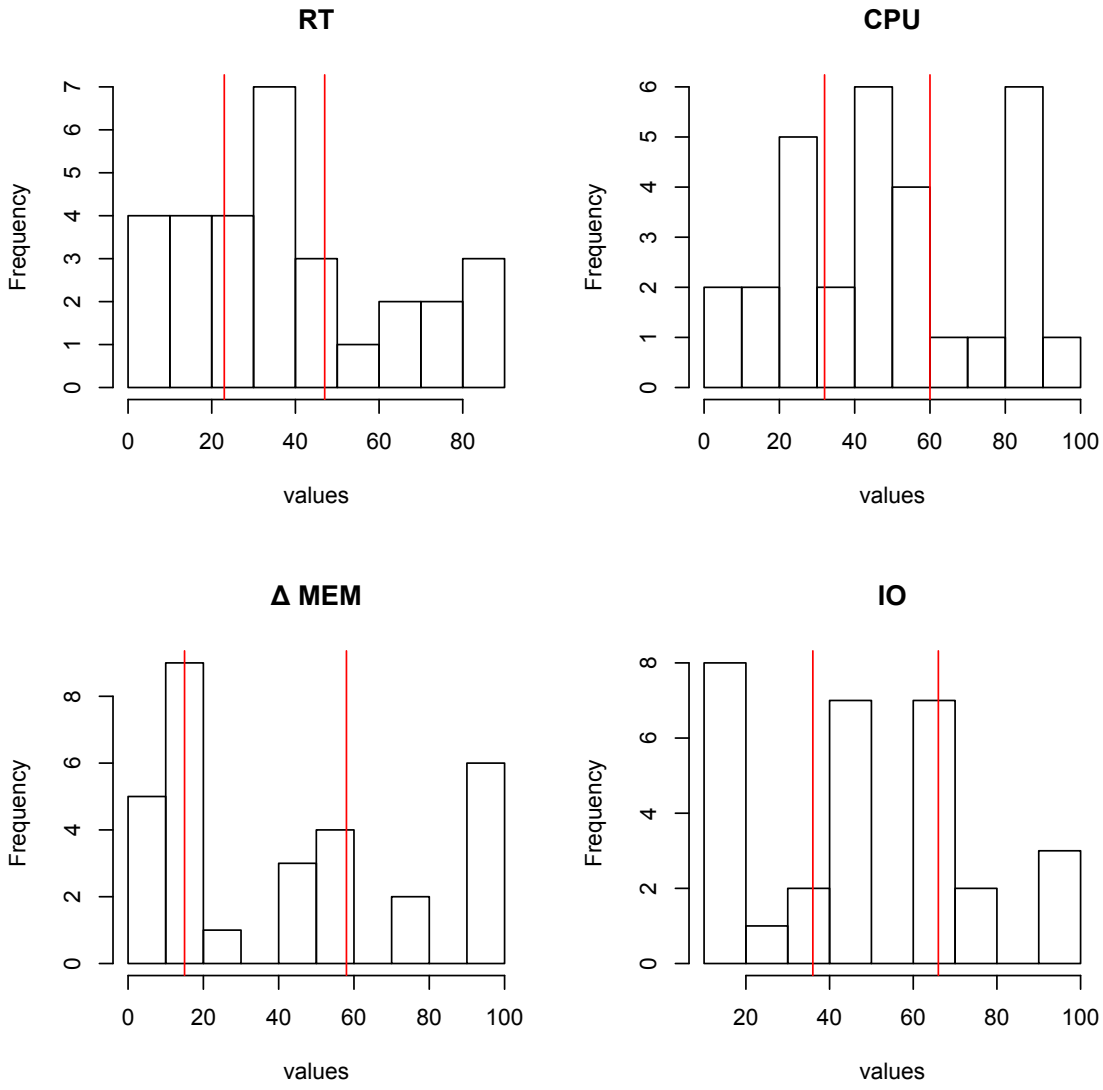

Figure 4.3: Transforming the values into states for the example from Table 4.2. The red vertical lines are the state boundaries determining the states

4.4.1 Experiment Environment

In order to conduct an evaluation between our two approaches, we used the same performance test metric data generated from the experiment in Chapter 3. The experiment consists of three online open source systems: CloudStore [43], PetClinic [44]

Table 4.3: The $state_{unique}$ for every 5 minutes in the example in Table 4.2

t	$state_{unique}$
5	5
10	8
15	13
20	15
25	15
30	15

and Dell DVD Store 2 (DS2) [45]. The workloads of CloudStore and PetClinic are based on Apache JMeter [51]. While, we run DS2 using its own load generator. We collect the same four performance metrics from Chapter 3:

Physical metrics. We collect the CPU utilization, memory usage, and (I/O) usage.

Domain metric. We collect response time.

4.4.2 Parameters of Our Approach

During the experiment, we configure our approach with the following parameters:

1. n: The number of states in which we transform the metrics. In our experiments, we use $n = 3$: S_1, S_2, and S_3.

2. τ: The number of minutes, after which we measure $state_{unique}$ during the execution of the tests. We use $\tau = 10$.

3. d: The length of time that we wait for the test not exercising new system states. We use $d = 40$ minutes.

4.5 Case Study Results

In this section, we present the results of the experiments that we conducted to evaluate our approach. In addition, we compare our approach with our previous approach that is presented in Chapter 3 that recommends to stop a performance test based on repetitiveness at the performance metric level. In this section, we evaluate our approach by answering two research questions:

- RQ1: What is the state coverage at the stopping times that are recommended by our approach?
- RQ2: What is the relevance of the states that are not covered when stopping a performance test early?

RQ1: What is the state coverage at the stopping times that are recommended by our approach?

Motivation. The goal of our approach is to reduce the execution time of a performance test without reducing the information that the test provides. Stopping a test later increases the chance of receiving more information from the test, i.e., as it covers more system states, but comes at the cost of increased execution time of the test. On the other hand, stopping a test too early results in missing useful information about the systems' performance.

Our approach recommends to stop the execution of a performance test when it stops providing new information for d minutes. In this RQ, we evaluate the state coverage at the recommended stopping times.

Approach. We use the data of the 24-hour test executions to evaluate the cost-effectiveness of the recommended stopping times. We calculate the state coverage by

Table 4.4: The recommended stopping times and the state coverage when applying our approach to the tests

Subject systems	**# of Covered states**[1]	**Stopping time**[2]	**Coverage**[3]	**Duration**[4]
CloudStore	81	7:30	96%	31%
PetClinic	81	8:10	98%	34%
DS2	66	5:30	95%	23%

[1] the number of states that are covered by the 24-hour tests.
[2] reported as hours : minutes.
[3] percentage of all states that are exercised by the test at the stopping time.
[4] percentage of the full execution time, i.e., 24 hours.

dividing $state_{unique}$ at the stopping time by $state_{unique}$ after exercising the tests for 24 hours. In addition, we compare the state coverage at the stopping times that are recommended by our new and previous approach.

Results. **Our approach reduces the execution time of the test by 70% while covering more than 95% of the system states.** Table 4.4 shows the stopping times that are recommended by our approach and the state coverage at those times. Table 4.4 shows that running the tests as long as the recommended stopping times will only yield 5% additional state coverage.

Once the number of unique states that are covered stabilizes, it stabilizes throughout the test. Figure 4.4 shows the $state_{unique}$ as the tests progress. In these graphs, the *x-axis* shows the percentage of time, and the *y-axis* shows the state coverage. Overall, while the tests running, the increase rate of $state_{unique}$ decelerates over the time, until it stabilizes. We find that once the state coverage is stabilized, it remains stabilized until the end of the test.

Our previous approach misses information about the interaction between the metrics for PetClinic and CloudStore. When stopping the test using our previous approach, some information that our newly proposed approach

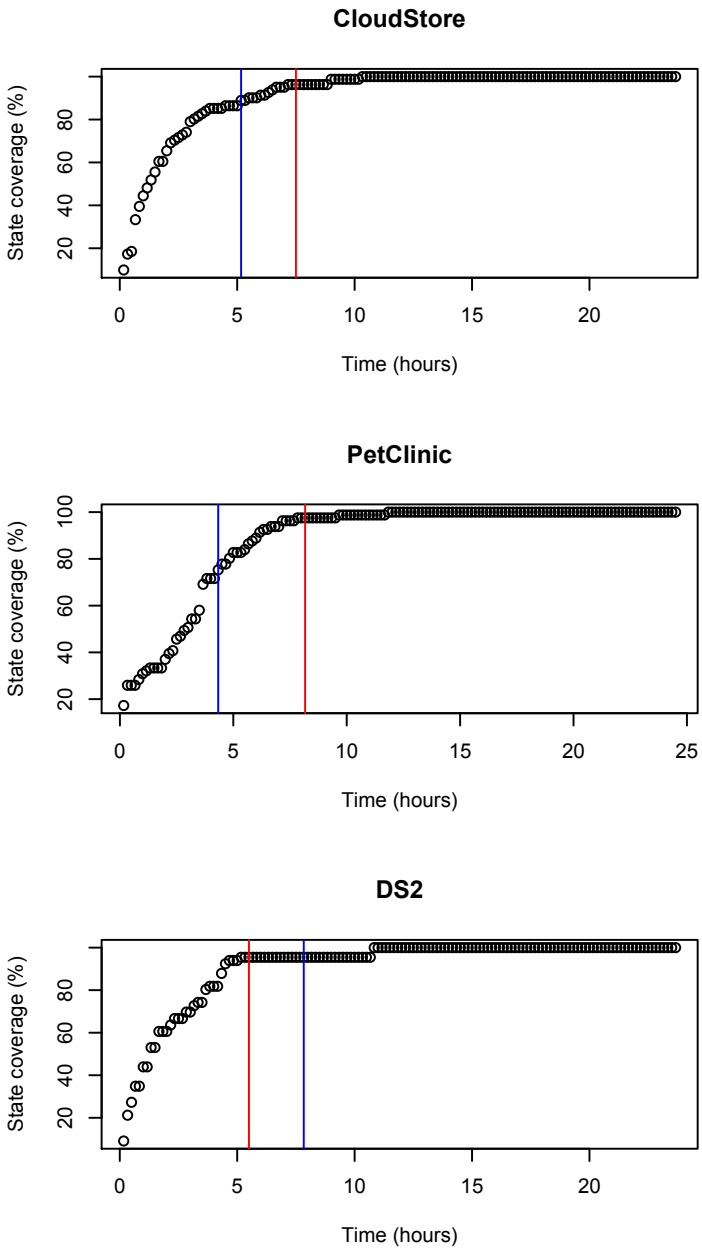

Figure 4.4: State coverage of our experiments over time. The red line is the stopping time that is recommended by our proposed approach. The blue line is the stopping time recommended by our previous approach that is presented in Chapter 3.

Table 4.5: Comparison between the state coverage and the stopping times of our new approach and our previous approach that is presented in Chapter 3

	New approach		Previous approach	
Subject systems	**Stopping times**	**State coverage**	**Stopping times**	**State coverage**
CloudStore	7:30	96%	5:10 (earlier)	89%
PetClinic	8:10	98%	4:20 (earlier)	75%
DS2	5:30	95%	7:50 (later)	95%

discovers is missed. Table 4.5 shows that stopping the tests at the stopping times that are recommended by our previous approach for CloudStore and PetClinic cover only 89% and 75% of the states, compared to 95% and 98% in our new approach. The stopping time that is recommended by our previous approach for DS2 covers exactly the same amount of information that the stopping time of our proposed approach does, i.e., 95%, but with a delay of more than two hours.

Our approach is capable of reducing the execution time of a 24-hour performance test for the three subject systems by $23-34\%$*, while preserving a state coverage of at least* 95%*.*

RQ2: What is the relevance of the states that are not covered when stopping a performance test early?

Motivation. Although our new approach recommends early stopping times at which a test achieves a high state coverage (more than 95%), there are still states that are missed by our approach but captured by the 24-hour performance tests. In this research question, we study the relevance of these missed states.

Figure 4.5 shows an example of the distribution of the performance metrics. In each distribution, two missed system states, i.e., m_1 and m_2 are indicated by the blue lines. The MEM value for m_1 is close to a state boundary, i.e., inside the light

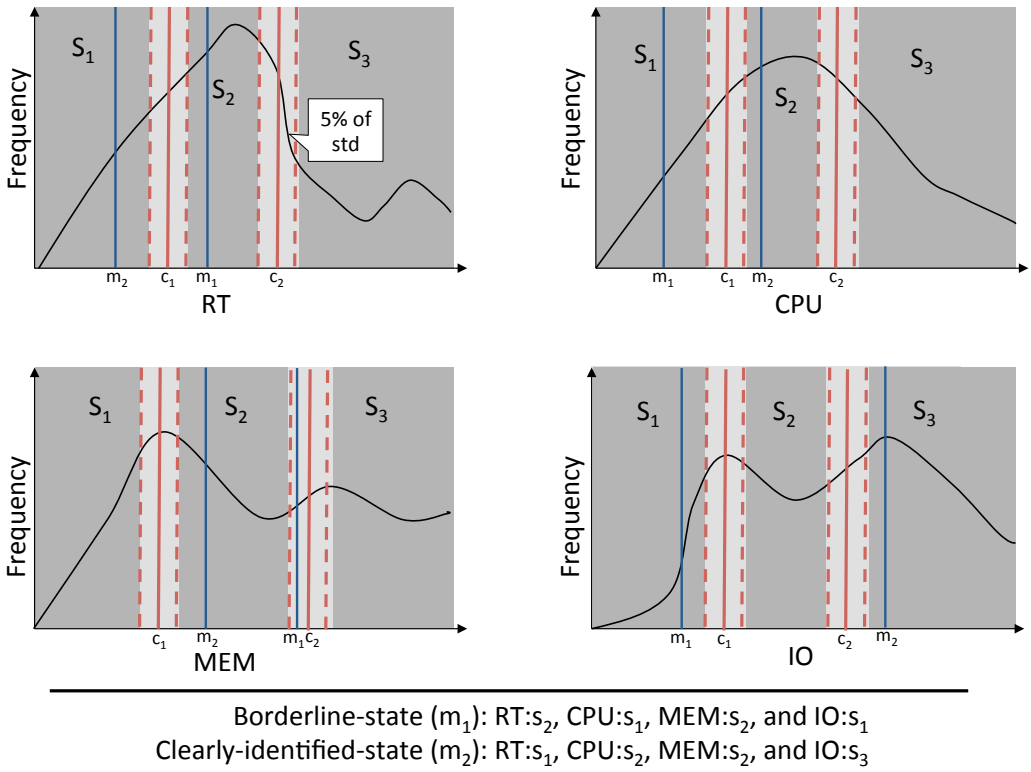

Figure 4.5: An imaginary example showing the difference between a *borderline-state* and a *clearly-identified-state*. A metric value that is inside the light grey background area is considered as close to a state boundary

grey background area. The state, into which a metric value that is close to a state boundary is indicated, might be affected by a few outliers of metric values as such outliers can slightly change the state boundaries. Therefore, such borderline values might be less relevant if one of the states that the value is close to is already covered.

Approach. To check the relevance of the missed states, we categorize them into two types of states:

- *Borderline-states*: states of which at least one of their metric values is close to a state boundary.

- *Clearly-identified-states*: states of which none of their values is close to a state boundary.

We consider a metric value close to a state boundary when the difference between the value and the state boundary is less than 5% of the standard deviation (*std*) of the entire metric distribution. The solid red lines in Figure 4.5 are the state boundaries for the metrics, which transform the metric values into states. The dashed red lines indicate 5% of the *std* around the state boundaries, which we use to categorize the missed states into either *borderline-states* or *clearly-identified-states*. Because the MEM value of m_1 is close to one of the state boundaries, we categorize m_1 as a *borderline-state*. We categorize m_2 as a *clearly-identified-state*, because none of its metric values are close to a state boundary.

Furthermore, a *borderline-state* is considered *relevant*, if the *borderline-state* contains new information about the test. The example *borderline-state* in Figure 4.5 can represent two system states, depending on the state boundaries:

- RT:S_2, CPU:S_1, MEM:S_3, and IO:S_1, or
- RT:S_2, CPU:S_1, MEM:S_2, and IO:S_1

If none of these two system states are covered by the test before the recommended stopping time, we consider the *borderline-state* as a *relevant-borderline-state*, because the state contains new information that our approach misses to capture, regardless of the state boundary of the MEM metric.

We compare our two approaches using the *coverage-opportunity* score. *Coverage-opportunity* is measured in the number of new relevant states (*clearly-identified-states* or *relevant-borderline-states*) that can be covered by running the test longer. We

calculate and compare the *coverage-opportunity* scores of the three subject systems at the stopping times that are recommended by our two approaches. The *coverage-opportunity* score is calculated as follows:

$$\textit{coverage-opportunity} = \frac{\textit{missed-clearly-identified} + \textit{missed-relevant-borderline}}{\textit{required-execution-time}} \quad (4.1)$$

Where the *required-execution-time* is the duration of time (in hours) the test requires to cover the missed states. If the *coverage-opportunity* is small, it is not cost-effective to run the test longer.

Results.

A small number of relevant states are missed by our new approach in comparison to the 24-hour performance tests. Table 4.6 shows that our new approach misses 1 relevant state out of 3 missed states for CloudStore, 2 relevant states out of 2 missed states for PetClinic, and 1 relevant state out 3 missed states for DS2.

Table 4.6: The states that are missed by our new approach and captured by the 24-hour performance tests

	Missed states	**Borderline-states**	**Relevant-borderline**	**Clearly-identified**
CloudStore	3	2	1	0
PetClinic	2	0	0	2
DS2	3	2	1	0

Most of states that are missed by our previous approach and captured by the 24-hour tests are relevant. Table 4.7 shows that our previous approach misses 3 relevant states out of 9 missed states for CloudStore, 20 relevant states out of 20 missed states for PetClinic, and 2 relevant states out 3 missed states for DS2.

Table 4.7: The states that are missed by our previous approach that is presented in Chapter 3 and captured by the 24-hour performance tests

	Missed states	**Borderline-states**	**Relevant-borderline**	**Clearly-identified**
CloudStore	9	7	2	1
PetClinic	20	0	0	20
DS2	3	2	1	0

For all subject systems, the *coverage-opportunity* scores for our new approach is smaller than our previous approach. Table 4.8 shows the *coverage-opportunity* scores for all three subject systems. Table 4.8 shows that the stopping times that are recommended by our new approach are more cost-effective than the stopping times that are recommended by our previous approach.

Table 4.8: The *coverage-opportunity* scores of our approaches.

	Coverage-opportunity score	
	Previous approach	**New approach**
CloudStore	0.6	0.4
PetClinic	3.2	0.8
DS2	0.4	0.2

Our new approach misses a small number of relevant states for all subject systems. The coverage-opportunity scores for the approach that is presented in this chapter are smaller across all system than the coverage-opportunity scores for our previous approach, which indicates that our new approach recommends more cost-effective stopping times.

4.6 Threats to validity

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

4.6.1 Threats to Internal Validity

Determining the System States. In our approach, every state is represented by a range of values for a metric. We determine a state by calculating cutting lines for each metric. These cutting lines might be too stringent. For example, if high CPU utilization is between 100% and 75%, and the test generates two consecutive values for the CPU metric of 74% and 76%, they are transformed into two different states, even though their difference is small.

4.6.2 Threats to External Validity

Our Subject Systems. We used three open-source e-commence systems (i.e., CloudStore, PetClinic, and DS2) to evaluate our approach. The programming language used for DS2 is PHP, while Java language is used for both CloudStore and PetClinic. Our approach may not have similar results when applied to other systems. However, the goal of this chapter is not to recommend a universal "stopping time", but to propose an approach that helps performance testers determine a cost-effective stopping time for their performance tests. More case studies on additional software systems (e.g., commercial systems) in additional domains and additional tests are needed to comprehensively evaluate our approach.

Our Performance Tests. Our approach assumes that the system's performance eventually becomes repetitive. However, our approach is agnostic to whether this repetition occurs when the system is performing well, experiencing performance issues or encountering a performance bottleneck. We evaluate our approach by running the two commonly-used types of load intensities: random and steady. In the performance tests of DS2, the load drivers periodically send pre-defined combinations of requests

to the systems to evaluate performance. However, large software system may leverage different types of load intensities (e.g., step-wise workload) of performance tests.

4.6.3 Threats to Construct Validity

The Quality of Performance Metrics. In our case study, we leverage PerfMon and load drivers (i.e., JMeter and DS2 driver) to provide performance metrics. In particular, the response time provided by our load drivers is an average value of response time of all the requests that are responded during a time period. However, by calculating the average response time, we may fail to identify extreme values. Similarly, the CPU usage, Memory usage and I/O traffic are also calculated by averaging their values over a small time period.

The Choice of Our Performance Metrics. In our experimental case study, four performance metrics are used. During conducting a typical performance test, hundreds of performance metrics are collected, most of which are highly correlated [8], and fit under a limited number of categories. When conducting our experimental study we use only four metrics but from different categories. However, our approach is not constrained to a specific number or types of metrics. Future studies may consider evaluating our approach with different sets of metrics.

4.7 Chapter Conclusion

Conducting a proper performance test before releasing a software system is critical, as such a test helps to ensure the performance of the system in the field. However, the time that is available to conduct a performance test is usually limited. In our previous approach that is presented in Chapter 3, we proposed an approach that

recommends when to stop a performance test based on repetitiveness in observed metric values. In this chapter, we propose an improved approach that considers the interactions between performance metrics as well.

Our approach measures the number of new system states, i.e., the combinations of metric value states, that a test exercises. Once a test no longer exercises new system states, our approach recommends to stop the test. We evaluate our approach in experiments on three open source systems (i.e., CloudStore, PetClinic and Dell DVD Store). The most important results of our experiments are:

1. Our proposed approach stops our performance tests on average within 8 hours, while covering at least 95% of the system states that are covered by the original 24-hour performance tests.

2. Our proposed approach recommends more cost-effective stopping times than our previous approach that is presented in Chapter 3.

Testers can use our approach to reduce the execution time of their performance tests, thereby taking a step towards more cost-effective performance testing.

Chapter 5

Conclusion

This chapter concludes the thesis. The findings, that are presented throughout this thesis, are summarized and possible directions for future work are presented below.

5.1 Summary

Conducting performance tests before the deployment of every new version of a system is important. Performance tests help to evaluate the expected performance and to catch potential performance issues. However, performance testing itself is associated with challenges, one of which is determining the cost-effective length of a test. Too long tests consume resources, such as machine time, and might lead to miss release deadlines. One way to avoid the cost of performance tests is by running shorter tests. However, shorter tests might miss identifying performance issues. Therefore, performance testers are in need for approaches that assist them to determine the cost-effective length of performance tests.

In this thesis, we present two approaches that recommend to stop performance tests at cost-effective execution timing.

5.1.1 Thesis Summary

- **Cost-effective Stopping of a Performance Test Using the Repetitiveness in the Collected Performance Metric Values (Chapter 3):**

  We propose our first approach that measures the repetitiveness of performance metric values. During a performance test, the approach collects the values of the metric that is so-far generated. After that, it measures the likelihood of repetitiveness. Our approach recommends to stop the test when the test

generates too little new information, i.e., performance metric values.

We conduct an experiment of three 24-hour performance tests to evaluate the approach. We find that our approach is able to stop the tests at early stopping times, i.e., saving up to 75% of the original 24-hour length, while preserving more than 91% of the performance metric values.

- **Cost-effective Stopping of a Performance Test Using the Repetitiveness in the collected Inter-Metrics Relations (Chapter 4):**

  One limitation of our first approach is that it disregards the inter-metrics relations (i.e., system states), which may miss important information about the performance of a system [19]. Therefore, we design another approach that reduces the execution time of a performance test by detecting whether a test no longer exercises new system states.

  We evaluate our second approach using the same experiment of the first approach. We find that our approach reduces the the execution time of the test to less than 8.5 hours, while preserving a coverage of at least 95% of the system states that are observed during the 24-hour tests. In addition, we find that the second approach recommends to stop at better cost-effective times, compared to the first approach.

5.2 Future Work

We believe that our thesis makes a major contribution towards determining the cost-effective execution time of performance tests. However, there remains many other

open challenges and opportunities for further improvements. In this section we highlight some of these opportunities.

- We evaluate our approach by conducting an experiment on three online open source systems. However, open source systems may not reflect industrial software systems. Therefore, evaluating our approach with large industrial systems is needed to provide a better understanding of whether the approach can be adopted in practice. Moreover, we leveraged Apache JMeter and DS2's load generator. However, industrial systems often use their own load generators. Hence, using other load generators may also provide a better understanding of the generality of our proposed approaches.

- Our approaches require a pre-configuration of their parameters. We find that different configurations can impact our approaches. Therefore, further studies should consider automatically optimizing the configurations of our approaches.

- In our experiment, due to the fact that performance metrics are highly correlated [8], we constrain the metric selection to only four metrics, each of which captures a different performance aspect (i.e., disk I/O verses CPU utilization). Different metric selection may provide a better understanding of our approaches. Thus, future studies should consider other types of performance metrics.

- To evaluate whether the next hours of the test, after the recommended stopping times, provide new information, we did not stop the tests when our approaches recommended so. The initial execution time of the tests are predefined, i.e., 24-hour. Future evaluations of our approach against different initial test lengths can lead to different conclusions.

- In the second approach, i.e., presented in Chapter 4, a few outliers can slightly change the state boundaries, which may affect the division of metric values to states. For example, if low CPU utilization is between 10% and 35%, and the test generates two consecutive values for the CPU metric of 34% and 36%, they are transformed into two different states, even though their difference is small. Therefore, future studies should consider improving our approach that is presented in Chapter 4, thereby finding other techniques that define less strict (i.e., fuzzy) state boundaries.

Bibliography

[1] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *Proceedings of the International Conference on Software Maintenance*, pages 125–134. IEEE, 2009.

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

[3] Yahoo outage and amazon loss. `http://blog.smartbear.com/performance/top-10-web-outages-of-2013/`. last visited: Dec 12 2016.

[4] Azure outage 1. `http://www.entrepreneur.com/article/240029`. last visited: Dec 12 2016.

[5] Azure outage 2. `http://www.gallop.net/blog/top-10-mega-software-failures-of-2014/`. last visited: Dec 12 2016.

[6] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, pages 94–103, New York, NY, USA, 2004. ACM.

[7] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Continuous validation of load test suites. In

Proceedings of the International Conference on Performance Engineering, pages 259–270. ACM, 2014.

[8] Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 222–231. IEEE, 2010.

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

[10] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In *Proceedings of the International Conference on Software Engineering*, pages 571–579. ACM, 2005.

[11] King Chun 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 *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 159–168, Piscataway, NJ, USA, 2015. IEEE Press.

[12] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the International Conference on Performance Engineering*, pages 15–26. ACM, 2015.

[13] Thanh HD 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 *Proceedings of the International Conference on Performance Engineering*, pages 299–310. ACM, 2012.

[14] Joao W Cangussu, Kendra Cooper, and W Eric Wong. Reducing the number of test cases for performance evaluation of components. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 145–150, 2007.

[15] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 58–68. IEEE, 2015.

[16] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[17] Raj Jain. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling, 1992.

[18] Nimrod Busany and Shahar Maoz. Behavioral log analysis with statistical guarantees. In *Proceedings of the International Conference on Software Engineering*, pages 877–887. ACM, 2016.

[19] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, volume 4, pages 16–16, 2004.

[20] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Autonomic load-testing framework. In *Proceedings of the International Conference on Autonomic Computing*, pages 91–100. ACM, 2011.

[21] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Leveraging performance counters and execution logs to diagnose memory-related ppmance issues. In *Proceedings of the International Conference on Software Maintenance*, pages 110–119. IEEE, 2013.

[22] BA Pozin and Igor V Galakhov. Models in performance testing. *Programming and Computer Software*, 37(1):15–25, 2011.

[23] 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, 2015.

[24] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. pages 442–453, 2003.

[25] Tanzeem Noor and Hadi Hemmati. Test case analytics: Mining test case traces to improve risk-driven testing. In *Proceedings of the International Workshop on Software Analytics*, pages 13–16. IEEE, 2015.

[26] Alberto Avritzer, Johannes P. Ros, and Elaine J. Weyuker. Reliability testing of rule-based systems. *IEEE Softw.*, 13(5):76–82, September 1996.

[27] Zhen Ming Jiang, Alberto Avritzer, Emad Shihab, Ahmed E. Hassan, and Parminder Flora. An industrial case study on speeding up user acceptance testing by mining execution logs. In *Proceedings of the International Conference on Secure Software Integration and Reliability Improvement*, pages 131–140. IEEE, 2010.

[28] Emad Shihab, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, and Robert Bowerman. Prioritizing the creation of unit tests in legacy software systems. *Software: Practice and Experience*, 41(10):1027–1048, 2011.

[29] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[30] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 235–245. ACM, 2014.

[31] Niclas Snellma, Adnan Ashraf, and Ivan Porres. Towards automatic performance and scalability testing of rich internet applications in the cloud. In *Proceedings of the International Conference on Software Engineering and Advanced Applications*, pages 161–169. IEEE, 2011.

[32] André B Bondi. Automating the analysis of load test results to assess the scalability and stability of a component. In *Proceedings of the Computer Management Group Conference*, pages 133–146, 2007.

[33] Roger Hayes and Alberto Savoia. How to load test e-commerce applications. In *Proceedings of the Computer Management Group Conference*, pages 275–282. Citeseer, 2000.

[34] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999.

[35] PerfMon. `http://technet.microsoft.com/en-us/library/bb490957.aspx`.

last visited: Dec 12 2016.

[36] Henry B Mann. Nonparametric tests against trend. *Econometrica: Journal of the Econometric Society*, pages 245–259, 1945.

[37] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. *ACM SIGPLAN Notices*, 48(11):63–74, 2013.

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

[39] Edmund A Gehan. A generalized two-sample Wilcoxon test for doubly censored data. *Biometrika*, pages 650–653, 1965.

[40] JWKJW Kotrlik and CCHCC Higgins. Organizational research: Determining appropriate sample size in survey research appropriate sample size in survey research. *Information technology, learning, and performance journal*, 19:43, 2001.

[41] Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994.

[42] Fitting function. `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/loess.html`. last visited: Dec 12 2016.

[43] Spring CloudStore. `http://github.com/cloudstore/cloudstore`. last visited: Dec 12 2016.

[44] Spring PetClinic. `http://docs.spring.io/docs/petclinic.html`. last visited: Dec 12 2016.

[45] Dell DVD store. `http://linux.dell.com/dvdstore`. last visited: Dec 12 2016.

[46] Transaction processing performance council. `http://www.tpc.org/tpcw/`. last

visited: Dec 12 2016.

[47] 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 *Proceedings of the International Conference on Software Engineering*, pages 1001–1012. ACM, 2014.

[48] Apache Tomcat. `http://tomcat.apache.org`. last visited: Dec 12 2016.

[49] MySQL. `http://www.mysql.com`. last visited: Dec 12 2016.

[50] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 2000.

[51] Apache JMeter. `http://jmeter.apache.org`. last visited: Dec 12 2016.

[52] Tse-Hsun Chen, S Weiyi, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2016.

[53] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Symposium on Operating systems principles*, volume 39, pages 105–118. ACM, 2005.

[54] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the International Conference on Performance Engineering*, pages 271–282. ACM, 2013.

[55] Sheng Yue, Paul Pilon, and George Cavadias. Power of the Mann-Kendall and

Spearman's rho tests for detecting monotonic trends in hydrological series. *Journal of Hydrology*, 259(1):254–271, 2002.

[56] Thomas D Gautheir. Detecting trends using spearman's rank correlation coefficient. *Environmental forensics*, 2(4):359–362, 2001.

[57] Binning algorithm. `https://cran.r-project.org/web/packages/binr/binr.pdf`.