An Automated Approach for Recommending When to Stop Performance Tests

Hammam M. AlGhmadi†, Mark D. Syer†, Weiyi Shang‡, Ahmed E. Hassan†
Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada†
Department of Computer Science and Software Engineering, Concordia University, Canada‡
{alghamdi, mdsyer, ahmed}@cs.queensu.ca†, shang@encs.concordia.ca‡

Abstract—**Performance issues are often the cause of failures in today's large-scale software systems. These issues make performance testing essential during software maintenance. However, performance testing is faced with many challenges. One challenge is determining how long a performance test must run. Although performance tests often run for hours or days to uncover performance issues (e.g., memory leaks), much of the data that is generated during a performance test is repetitive. Performance analysts can stop their performance tests (to reduce the time to market and the costs of performance testing) if they know that continuing the test will not provide any new information about the system's performance. To assist performance analysts in deciding when to stop a performance test, we propose an automated approach that measures how much of the data that is generated during a performance test is repetitive. Our approach then provides a recommendation to stop the test when the data becomes highly repetitive and the repetitiveness has stabilized (i.e., little new information about the systems' performance is generated).**

We performed a case study on three open source systems (i.e., CloudStore, PetClinic and Dell DVD Store). Our case study shows that our approach reduces the duration of 24-hour performance tests by 75% while preserving more than 91.9% of the information about the system's performance. In addition, 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).

I. Introduction

The performance of large-scale software systems (e.g., Gmail and Amazon) is a critical concern because they must concurrently support millions of users. Failures in these systems are often due to performance issues rather than functional issues [28], [50]. 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 [1]. A 25-minute service outage on August 14, 2013 (caused by a performance issue) cost Amazon around $1.7 million [1]. Microsoft's cloud-based platform, Azure, experienced an outage due to a performance issue on November 20, 2014 [3], [4], which affected users worldwide for 11 hours. These performance failures affect the competitive positions 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) [14]. 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 [47]. Therefore, performance tests are often carefully designed to mimic real system behaviours in the field and uncover performance issues.

One of the challenges associated with designing performance tests is determining how long a performance test should last. Some performance issues (e.g., memory leaks) only appear after the test runs for an extended period of time. Therefore, performance tests may potentially last for days to uncover such issues [32]. Although in practice, performance tests are often pre-defined (typically by prior experience or team convention) to last a certain length of 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 [27] and consume significant resources. Therefore, determining the most cost-effective performance test length 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 [6]. This repetition generates repetitiveness in the results of the test, i.e., performance counters. Intuitively, a performance analyst may consider stopping a performance test by knowing 1) that the newly-generated data by continuing the test would likely be similar to the data that has already been collected, or 2) that the trend of the performance data by continuing the test would likely be similar to the data that has already been collected. Based on our intuition, in this paper we present an approach that automatically determines when to stop a performance test. Our approach measures the repetitiveness of the performance counters during a performance test. We use the raw values of these counters to measure the repetitiveness of the counter values and we use the delta of the raw values, i.e., the differences in the raw counter values between two consecutive observations of these performance counters, to measure the repetitiveness of the observed trends in the performance counters. 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 analysts to subjectively determine whether to stop a performance test.

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 with random workloads. We then used our approach to recommend when the tests should stop. We measure the repetitiveness in performance counters by running the 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 two-hour delay.

This paper makes two contributions:

1) We propose an approach to measure the repetitiveness of performance counters. Such measurements can be leveraged in future performance test research.
2) We propose a novel approach to automatically recommend the most cost-effective time to stop a performance test by examining the repetitiveness of performance counters.

This paper is organized as follows. Section II gives an overview of designing a performance test and prior research that assists in designing performance tests. Section III provides a motivation example in order to give a clearer idea of where and when to use our approach. Section IV explains the phases of our approach. We discuss our case study in Section V, followed by the results in Section VI. Then threats to the validity of our work are presented in Section VII. We conclude the paper in Section VIII.

II. Background and Related Work: Designing Performance Tests

Performance testing is the process of evaluating a system's behaviour under a workload [5]. The goals of performance testing are varied and include identifying performance issues [46], verifying whether the system meets its requirements [42], and comparing the performance of two different versions of a system [36], [43].

Performance tests need to be properly designed in order to achieve such goals. 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 section, we discuss prior research along these three aspects.

A. 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.

- **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.
- **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 [47].
- **Covering scenarios that may expose performance issues.** A typical test case prioritization approach is based on the number of potential faults that the test case may expose [37] or the similarity between test cases [38].

These approaches often generate workloads that are too large or complex to be used for performance testing. Therefore, performance analysts 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 in a baseline: Avritzer et al. propose a technique that limits the number of test cases by selecting a subset of the test cases [2]. 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 covered by the test case). Jiang et al., propose a technique that is related to Avritzer's technique [2] in that both techniques define a set of different system states. However, Jiang et al., use a different definition for the 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 [26]. 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 approach: 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 [34]. 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 [44]. Cangussu et al. propose an approach based on an adaptive or random selection of test cases by using polynomial curve fitting techniques [11]. Hemmati et al. evaluate the effectiveness of three metrics (i.e., *coverage*, *diversity*, and *risk*) for test case prioritization [25]. Their work concludes that historical riskiness is effective in prioritizing test case in a rapid release setting. Elbaum et al. [19] 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.

B. 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:

- **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 [45], and identifying performance issues (e.g., memory leaks) [8].
- **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 [50]. Increasing the workload intensity may uncover the ability of the system to handle heavy workloads. Furthermore, Hayes et al., [24] 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 a performance-related failures).

C. Test Length

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 [17]. Jain [9] designs an approach to stops a performance test by measuring the variances between response time observations. Their approach recommends stopping the performance test when the variance is lower than 5% of the overall means. Busany et al. [10] propose an approach that can be used to reduce test length by measuring the repetitiveness of log traces.

Although there is little research on determining the length of a performance test, intuitively, a performance test may stop if 1) almost every possible raw value of the performance counters is observed, or 2) a trend of the performance test data is observed. In the first case, continuing the performance test would not generate any new data, while in the second case, one may calculate (i.e., interpolate or extrapolate) the un-observed data based on the trend. Otherwise, if a test does not meet either stopping criteria, the test can be stopped based on a time length that is required or agreed upon by performance analysts. In practice, the decisions on stopping performance tests are often made based on performance analysts' experiences and intuition. Therefore, in this paper, we propose an automated approach that recommends the most cost-effective time to stop a performance test by measuring the repetitiveness of values and trends of performance test data.

III. A Motivating Example

Eric is a performance analyst 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 execution of each workload scenario once would not detect such performance issues.

Eric uses performance counters to analyze the system's performance. If the performance counters become repetitive, continuing the test would not provide much additional information. In addition, Eric found that if he observed trends of a performance counter in the beginning of the test, he can use the trend to calculate the counter values in the rest of the test. Table I shows an imaginary example of four performance counters 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 I, the values of the performance counters from time stamp 304 to 308, i.e., the time period *rep-1* in Table I, 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 counter 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 whether a performance test should continue or stop based on the repetitiveness of values or trends of the performance counters that are generated during the test. Once the data generated during a performance test becomes highly repetitive, the approach recommends that the test be stopped.

TABLE I: An imaginary example of a performance counters file

		Performance Counters			
	Time stamp	RT	CPU	Memory (delta)	IO
	301	82	9	86 (86)	90
	302	26	7	113 (27)	3
	303	80	7	135 (22)	70
rep-1	**304**	**81**	**52**	**182 (47)**	**77**
	305	**83**	**99**	**224 (42)**	**74**
	306	**12**	**53**	**229 (5)**	**57**
	307	**5**	**99**	**229 (0)**	**67**
	308	**17**	**93**	**240 (11)**	**37**
	309	37	40	319 (79)	25
rep-2	**310**	**62**	**29**	**396 (77)**	**47**
	311	**83**	**61**	**411 (15)**	**10**
	312	**98**	**69**	**428 (17)**	**93**
	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
rep-2’	**321**	**62**	**29**	**1,019 (77)**	**47**
	322	**83**	**61**	**1,034 (15)**	**10**
	323	**98**	**69**	**1,051 (17)**	**93**
	324	**71**	**22**	**1,117 (66)**	**2**
	325	**31**	**79**	**1,179 (62)**	**13**
rep-1’	**326**	**81**	**52**	**1,226 (47)**	**77**
	327	**83**	**99**	**1,268 (42)**	**74**
	328	**12**	**53**	**1,273 (5)**	**57**
	329	**5**	**99**	**1,273 (0)**	**67**
	330	**17**	**93**	**1,284 (11)**	**37**

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

IV. 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 of a performance test. Figure 1 presents an overview of our approach.

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

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

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 details.

A. Collecting Performance Test Data

The collected performance counters are the input of our approach. Typical performance monitoring techniques, such as PerfMon [39], allow users to examine and analyze the performance counter values during the monitoring.

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

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

B. Determining the Use of Counter (Raw or Delta Values)

Performance counters may illustrate trends during performance tests. For example, memory usage may keep increasing when there is a memory leak. On the other hand, some counters 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 counter values, we determine for each counter, whether we should use the raw or delta values. We leverage a statistical test (Mann-Kendall Test [33]) to examine whether the values of the counters that have already been generated during the test have a monotonic trend. The null hypothesis of the test is that there is no monotonic trend in values of a counter. 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 counter. If there exists a statistically significant trend, we would use the delta values of the counter, otherwise, we would use the raw values of the counter.

For our working example, for the first time period, i.e., the 301^{st} to the 320^{th}, the memory counter is the only counter 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.

C. Measuring the Repetitiveness

The goal of this step is to measure the repetitiveness of the performance counters 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 counters. Performance counters are measurements of resource utilizations. Such measurements typically do not have exact matches. For example, two CPU usage values may be 50.1%

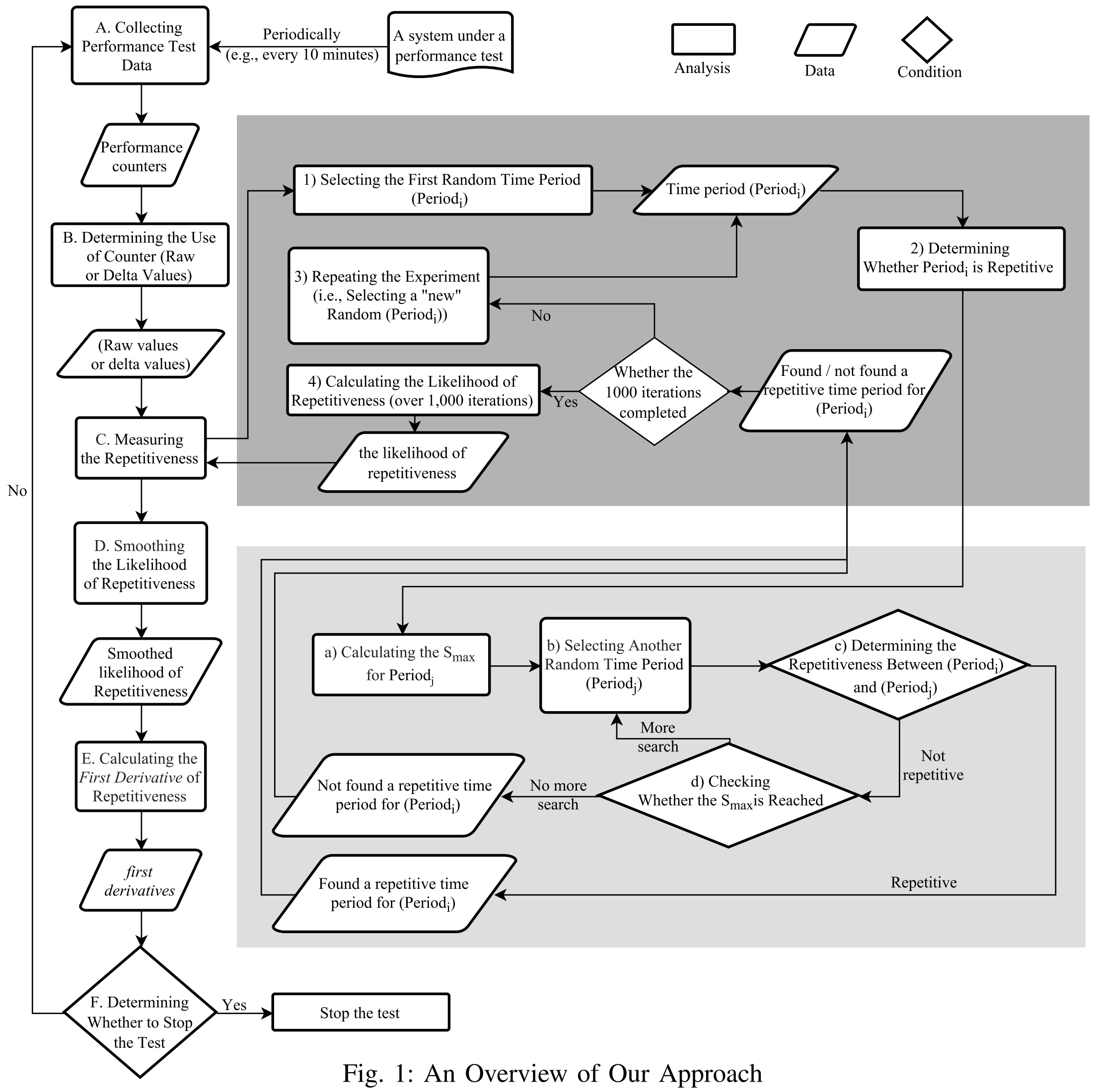
A. Collecting Performance Test Data
Periodically (e.g., every 10 minutes)
A system under a performance test
Analysis
Data
Condition
Performance counters
B. Determining the Use of Counter (Raw or Delta Values)
(Raw values or delta values)
C. Measuring the Repetitiveness
D. Smoothing the Likelihood of Repetitiveness
Smoothed likelihood of Repetitiveness
E. Calculating the First Derivative of Repetitiveness
first derivatives
F. Determining Whether to Stop the Test
Yes
No
Stop the test
1) Selecting the First Random Time Period (Period_i)
Time period (Period_i)
2) Determining Whether Period_i is Repetitive
3) Repeating the Experiment (i.e., Selecting a "new" Random (Period_i))
No
Whether the 1000 iterations completed
Yes
4) Calculating the Likelihood of Repetitiveness (over 1,000 iterations)
Found / not found a repetitive time period for (Period_i)
the likelihood of repetitiveness
a) Calculating the S_max for Period_j
b) Selecting Another Random Time Period (Period_j)
c) Determining the Repetitiveness Between (Period_i) and (Period_j)
Not repetitive
More search
d) Checking Whether the S_max is Reached
No more search
Not found a repetitive time period for (Period_i)
Found a repetitive time period for (Period_i)
Repetitive

Fig. 1: An Overview of Our Approach

and 50.2%. It is hard to determine whether such difference between two performance counter values corresponds to an actual performance difference or is due to noise [30]. Statistical tests have been used in prior research and in practice to detect whether performance counter values from two tests reveal performance regressions [21], [43]. Therefore, we leverage statistical tests (e.g., Wilcoxon test [23]) to determine whether the performance counter values are repetitive between two time periods. We choose Wilcoxon test because it does not have any assumption on the distribution of the data.

1) Selecting the First Random Time Period ($Period_i$): In order to measure the repetitiveness of performance counters, 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 I has 30 observations. With a time period with 5 observations ($len_\tau = 5$), we may select time period $Period_i$ from time stamp 317 to 321.

2) 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 I, 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 performance test for a long time, leads to a substantial number of possible time periods. For example, a typical performance test that runs for 48 hours and collects performance counters every 30 seconds, would contain over 5,700 possible time periods of 30 minutes. Thus, our approach would likely take a long time to process. 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 [31].

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 [31]. The confidence interval is the error margin that is included in reporting the results of the representative samples.

For example, if we choose the confidence interval of 5, and based on the 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 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 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.

b) Selecting Another Random Time Period ($Period_j$): We select a 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 counter values during these two time periods using a two-tailed unpaired Wilcoxon test [23]. Wilcoxon tests do not assume a particular distribution of the population. A $p - value > 0.05$ means that the difference between the counter 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 counter values between $Period_i$ and $Period_j$). Failure to reject the null hypothesis means that the difference between the counter 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 counters from both time periods. The p-values of *Wilcoxon tests* for the counters of our working example are shown in Table II.

TABLE II: Wilcoxon test results for our working example

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

The two time periods ($Period_i$ and $Period_j$) are considered **repetitive** if all the differences of all the counters 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 repetitive. In this case, another random time period $Period_j$ would be selected if the number of searches 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 counters are greater than 0.05). Thus, $Period_i$ is reported as **repetitive**.

3) 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 counters. Efron et al., [18] state that 1,000 replications can make an inference of the data. 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**.

4) 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.

D. 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 to 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 [20] to fit the smoothing spline.

E. Calculating the First Derivative of Repetitiveness

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

$$Derivative = \frac{likelihood_{current} - likelihood_{previous}}{Periodically_{diff}} \quad (1)$$

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.

F. 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 counter does not change, i.e., every counter 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 in 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.

V. Experiment Setup

To evaluate our approach for determining when to stop a performance test, we perform experiments with three different large systems, i.e., CloudStore [40], PetClinic [41] and Dell DVD Store (DS2) [16]. In this section, we present the subject systems, the workloads that are applied to them, and our experimental environment.

A. Subject Systems

CloudStore [40] is an open-source e-commerce system that is built for performance benchmarking. Cloud Store follows the TPC-W performance benchmark standard [49].

PetClinic [41] is an open-source web system that aims to provide a simple yet realistic design of a web system. PetClinic was used in prior studies for performance engineering research [12], [22], [28].

The Dell DVD store (DS2) is an open-source web system [16] that simulates an electronic commerce system to benchmark new hardware system installations. DS2 has been used in prior performance engineering research [36], [43].

B. 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 [35] 5.6.21 as our database.

C. Performance Tests

We use the performance test suites, for each system. Our aim is to mimic the real-life usage of the system and ensure that all of the common features are covered during the test [7]. These suites exercise the subject systems when evaluating our approach. For CloudStore and PetClinic, their performance test suites are based on Apache JMeter. Apache JMeter is a performance testing software that is designed to generate load against online systems [29]. DS2 uses its own load generator that conducts a performance test by pushing a workload against the system. We run performance tests for 24 hours for all three subject systems. The workload scenarios for our test suites are preset by the JMeter and DS2 load drivers. To avoid intentionally generating a repetitive load, we randomly change load intensity during our performance tests.

D. Data Collection

We collected both physical level and domain level performance counters as the results of the performance tests. We leverage a performance monitoring system named PerfMon [39] to collect physical level performance counters. We record CPU usage, memory usage and I/O traffic usage that are associated with the process of our systems during the performance tests. We also collect response time as domain level performance counters. Response time is used in prior performance engineering research to measure the performance of a system [13], [51]. Response time measures when a user sends a request to the system, how fast the system responses to the user. In our case study, we leverage JMeter to generate requests to the systems. As simulated users, JMeter tracks the response time of each request. Therefore, we leverage JMeter to measure the average response time for every ten seconds.

One of the challenges of leveraging performance counters that are generated from different sources is the clock skew [21]. In our case study, the PerfMon and the load generator (i.e., JMeter or DS2's driver) would not generate performance counters at exactly the same time. To address this challenge, we record performance counters every 10 seconds and use the average value of every three consecutive records that belong to the same half of a minute to generate a new value of the performance counter for every 30 seconds. For example, the CPU usage may be recorded three times at 12:01, 12:11 and 12:21, while the response time is recorded at 12:05, 12:15 and 12:25. We calculate the average values of CPU usage and response time as their values for the 12:00 to 12:30.

E. Parameters of Our Approach

To determine when to stop a performance test, we apply our approach on the performance counters 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 first derivatives 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 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 counters.

F. Levering Existing (Jian's) Approach to Stop Performance Tests

Jain also propose an approach that recommends when to stop a performance test [9]. Their approach recommends when to stop a performance test by measuring the variances between response time observations. 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 [9]. 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 Jian'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.

G. 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 counters 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 counters from a 15 or 30 minutes len_T 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 counter 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.

TABLE III: 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

VI. Case Study Results

In this section, we present the results of evaluating our approach. Table III 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 data that brings new information after the stopping time. However, stopping the tests late is against our purpose of reducing the length of a performance test.

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 2) and data generated after the stopping time (i.e., *post-stopping data* and (b) in Figure 2).

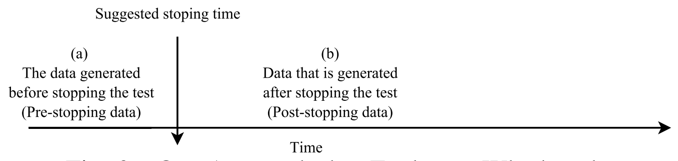

Fig. 2: Our Approach that Evaluates Whether the Recommended Stopping Times is too Early

We follow a similar approach as described in Section IV 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 counter data that is not statistically significantly different from $Period_i$. Third, we repeat this process, i.e., selecting random $Period_i$ from (b) in Figure 2),

TABLE IV: 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	97.6	97.1	100	97.8	98	100	98.6	97.7	99.9
	30 minutes	91.9	98.1	99.9	94.1	97.8	99.9	92.9	98.4	100
PetClinic	15 minutes	100	100	100	100	100	100	100	100	100
	30 minutes	100	100	100	100	100	100	100	100	100

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 good 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:

$$EffectivenessScore = (R_h - R_{h-1}) - (R_{h+1} - R_h) \quad (2)$$

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-effectiveness before finishing the test to gather the complete dataset. We use the most cost-effectiveness stopping time to evaluate whether our approach can recommend stopping 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 IV 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 parameters.

TABLE V: The delay between the 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

There is a short delay between the most cost-effective stopping time and the stopping time recommended by our approach. Table V shows the delay of our approach to find a cost-effective stopping times of the tests. Out of 54 stopping times in Table V, 27 are under three hours away from the most cost-effective stopping times. Whereas, only six stopping times that are recommended from our approach are more than six hours away from the cost-effective times, i.e., all when setting *Threshold* as 0.01.

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 counters, during the performance test. Such measurements quantify the risk that is associated with stopping the test early. Performance analysts 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 to stop the test (i.e., comfort level).

The post-stopping data is highly repetitive (91.9% to 100%) 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.

VII. Threats to Validity

A. Threats to Internal Validity

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 three values for each threshold. We evaluate our approach with the combinations of different thresholds. We find that the choice of these parameters impacts our recommended stopping time. Further studies may consider automatically optimizing our approach by choosing optimized thresholds.

Randomness in the Experiments and the Approach. To avoid intentionally generating a repetitive data in our experiments, we randomized the workload of the performance tests. Furthermore, 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 replicating the experiments and running our approach multiple times.

B. Threats to External Validity

Our Subject Systems. We used three open-source e-commence systems (i.e., CloudStore, PetCinic, and DS2) to evaluate our approach. The programming language used for DS2 is PHP, while Java language is used for both CloudStore and PetCinic. Our approach may not have similar results when applied to other systems. However, the goal of this paper is not to recommend a universal "stopping time", but to propose an approach that helps performance analysts determine whether to stop their performance tests. More case studies on additional software system (e.g., commercial systems) in additional domains and additional tests are needed to 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 system's is performing well, experiencing performance issues or encountering a performance bottleneck. Moreover, in the performance tests of our case study, the load drivers periodically send pre-defined combinations of requests to the systems to evaluate performance. However, large software system, especially commercial systems, may leverage more complex workload in performance tests. Evaluating our approach with more complex performance tests is needed in our future work.

C. Threats to Construct Validity

The Quality of Performance Counters. In our case study, we leverage PerfMon and load drivers (i.e., JMeter and DS2 driver) to provide performance counters. 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. Using different time periods to measure performance counters to evaluate our approach will address this threat.

Determining Repetitiveness. In our approach, we consider two time periods to be repetitive only if none of the performance counters are statistically significantly different between two time periods. In practice, domain experts may consider two time periods repetitive even with some performance counters that 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 analysts may choose to use a less strict rule for repetitiveness based on the subject systems.

The Choice of Our Performance Counters. We use four performance counters (i.e., CPU usage, memory usage, I/O traffic and response time) in the evaluation of our approach. All of these performance counters are widely used in prior research (e.g., [36], [43], [46]), and by developers [15]. In a typical performance test, hundreds of different performance counters are collected. However, comparing every performance counter to identify repetitiveness would be time consuming. Moreover, many of the performance counters are highly correlated [32]. Future studies may consider more and different type performance counters. Considering to use a workload counter (e.g., throughputs), might be more beneficial for our approach.

Measuring Trends in Performance Counters. We use the Mann-Kendall Test to determine whether a performance counter has a monotonic trend since the beginning of the test and we use delta values of every two consecutive observations a performance counters to measure the trend. However, a performance counter may exist more complex (e.g., sinusoid trends) trends or the trends may not start at the beginning of the test. In our future work, we will extend our approaches in order to identify other trends in performance counters.

The Length of the Performance Test. We evaluate our approach by examining likelihood of missing unique (i.e., non-repetitive) data if we stop the test early. However, it is impossible to run the test forever to know all the possible data that may generated from the test. Therefore, we run the test for 24 hours as an initial test length. The evaluation of our approach may have different results if a different stopping point is chosen. However, our approach not only recommends when to stop the test, but calculates the likelihood that future performance counters will be repetitive. Therefore, performance analysts should have more confidence in our approach. Future evaluations of our approach against different initial test lengths can address this threat.

VIII. Conclusion

Performance testing is critical to ensuring the performance of large-scale software systems. Determining the length of performance tests is a 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 data that is generated since the start of the performance test. Repetitiveness of performance counters is due to repeating values or repeating trends in the counters. If the repetitiveness of performance counters stabilizes, then our approach recommends that the test be stopped.

The highlights of this paper are:

- We propose an approach to measure the repetitiveness of performance counters.
- We propose an approach to automatically recommend when to stop a performance test based on the repetitiveness of performance counters.
- 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).

References

[1] Yahoo outage and amazon loss. http://blog.smartbear.com/performance/top-10-web-outages-of-2013/.

[2] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Software*, (5):76–82, 1996.

[3] Azure outage 1. www.entrepreneur.com/article/240029.

[4] Azure outage 2. http://www.gallop.net/blog/top-10-mega-software-failures-of-2014/.

[5] C. Barna, M. Litoiu, and H. Ghanbari. Autonomic load-testing framework. In *Proceedings of the International Conference on Autonomic Computing*, pages 91–100. ACM, 2011.

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

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

[8] A. 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.

[9] P. N. D. Bukh and R. Jain. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling, 1992.

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

[11] J. W. Cangussu, K. Cooper, and W. E. 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. Citeseer, 2007.

[12] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. 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.

[13] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Symposium on Operating systems principles*, volume 39, pages 105–118. ACM, 2005.

[14] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *Proceedings of the International Workshop on Software and Performance*, volume 29, pages 94–103. ACM, 2004.

[15] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proceedings of the International Symposium on Parallel and Distributed Processing*. IEEE, 2003.

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

[17] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999.

[18] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994.

[19] S. Elbaum, G. Rothermel, and J. 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.

[20] Fitting function. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/loess.html.

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

[22] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments.

[23] E. A. Gehan. A generalized two-sample wilcoxon test for doubly censored data. *Biometrika*, pages 650–653, 1965.

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

[25] H. Hemmati, Z. Fang, and M. 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.

[26] Z. M. Jiang, A. Avritzer, E. Shihab, A. E. Hassan, and P. 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.

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of the International Conference on Software Maintenance*, pages 307–316. IEEE, 2008.

[28] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the International Conference on Software Maintenance*, pages 125–134. IEEE, 2009.

[29] Apache jmeter. jmeter.apache.org.

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

[31] J. Kotrlik and C. Higgins. Organizational research: Determining appropriate sample size in survey research appropriate sample size in survey research. *Information technology, learning, and performance journal*, 19(1):43, 2001.

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

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

[34] D. Mondal, H. Hemmati, and S. 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.

[35] Mysql. www.mysql.com.

[36] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the International Conference on Performance Engineering*, pages 299–310. ACM, 2012.

[37] T. Noor and H. 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.

[38] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data.

[39] Perfmon. technet.microsoft.com/en-us/library/bb490957.aspx.

[40] Spring cloudstore. github.com/cloudstore/cloudstore.

[41] Spring petclinic. docs.spring.io/docs/petclinic.html.

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

[43] W. Shang, A. E. Hassan, M. Nasser, and P. 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.

[44] E. Shihab, Z. M. Jiang, B. Adams, A. E. Hassan, and R. Bowerman. Prioritizing the creation of unit tests in legacy software systems. *Software: Practice and Experience*, 41(10):1027–1048, 2011.

[45] N. Snellma, A. Ashraf, and I. 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.

[46] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. 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.

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

[48] Apache tomcat. tomcat.apache.org.

[49] Tpc. http://www.tpc.org/tpcw/.

[50] E. J. Weyuker and F. 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.

[51] P. Xiong, C. Pu, X. Zhu, and R. 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.