2011 18th Asia-Pacific Software Engineering Conference

Automated Verification of Load Tests Using Control Charts

Thanh H. D. Nguyen, Bram Adams, ZhenMing Jiang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: {thanhnguyen,bram,zmjiang,ahmed}@cs.queensu.ca

Mohamed Nasser, Parminder Flora
Performance Engineering
Research In Motion (RIM)
Waterloo, Ontario, Canada

***Abstract*—Load testing is an important phase in the software development process. It is very time consuming but there is usually little time for it. As a solution to the tight testing schedule, software companies automate their testing procedures. However, existing automation only reduces the time required to run load tests. The analysis of the test results is still performed manually. A typical load test outputs thousands of performance counters. Analyzing these counters manually requires time and tacit knowledge of the system-under-test from the performance engineers.**

The goal of this study is to derive an approach to automatically verify load tests results. We propose an approach based on a statistical quality control technique called control charts. Our approach can a) automatically determine if a test run passes or fails and b) identify the subsystem where performance problem originated. We conduct two case studies on a large commercial telecommunication software and an open-source software system to evaluate our approach. Our results warrant further development of control chart based techniques in performance verification.

***Keywords*-Performance Testing, Load Testing, Mining software repository**

I. Introduction

Performance is an importance aspect of software quality. For example, an online store should respond to a client's page request within a minimum/average/maximum of 1/5/9 seconds when the server is servicing a standard work load. Web users usually lose patience after eight seconds [1]. The user will most likely leave the site after a ten seconds wait. To make the matter worst, the more familiar the users are with the site, the less tolerant they are to wait time. Keeping user is a very importance requirement for an e-commerce company. In such situation, the performance requirements are sometimes more important than functional requirements [2]. Similar can be said for other mission critical systems such as email servers or application servers where availability under high load is crucial to the business operation.

Load testing determines if a software meets its performance requirements. After a development iteration of fixing bugs and implementing new features, the changes to the code might degrade the performance of the software. Performance engineers must perform load tests to ensure that there is no performance regression. A typical load test puts the system under a designed load and collect performance counters such as response time or CPU usage. If the performance counter is within an acceptable range compared to previous versions of the software, the test passes and the software can be released. Otherwise, testers have to determine which subsystem causes the performance problem and alert the developers so the defect can be fixed.

A load test typically outputs thousands of performance counters which record the system's and its subsystems's performance during a load test. Testers need to analyze the counters after each test run. Analyzing these counters are usually done manually [3]. It is a very time consuming process. In order to assist load testers, an automated approach to load test verification is needed. Such system should analyze the load counters and report if and where anomalies occur. This would reduce the time required to analyze load test results.

In this paper, we investigate the application of a statistical quality control technique called control chart to automate load tests verification. Control chart has been widely used in manufacturing to detect anomalies in manufacturing process [4]. For each unit of output, a control chart is used to monitor the degree to which the unit is deviated from the normal expected behaviour. If the deviation is too high, the unit is marked for further inspection. We believe that control chart can also be used on performance counters.

We derive two control chart based approaches to a) automatically mark a load test run as good or problematic and b) locate the subsystem with the performance anomaly. We develop and evaluate the approaches on a large telecommunication software system and an open-source web store [5]. Both approaches show promising results.

Our contributions are:

- We show that control chart based approach can automatically determine if a load test run is good or problematic.
- We find that control chart based approach can locate subsystems with performance anomaly.

The paper is organized as follow. Section II introduces the background on performance load tests, our motivation, and our research questions. Section III explains our study setting and experiment design. Section IV and Section V tackle each research question separately. We discuss about the findings in Section VI. Section VII discusses the related works. We conclude in Section VIII.

1530-1362/11 $26.00 © 2011 IEEE
DOI 10.1109/APSEC.2011.59

IEEE computer society

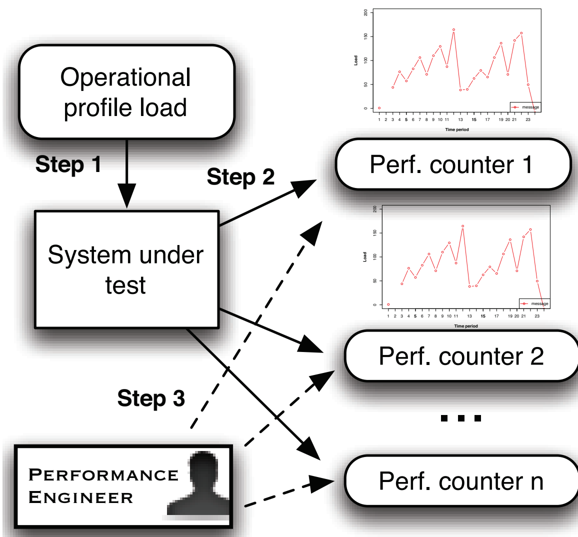

Fig. 1. A typical load test verification process.

II. Background and Motivation

In this section, we introduce the background on performance load testing and the motivation for an automatic approach to load test verification. We will also explain what control chart technique is and how we want to utilize it for load tests verification.

A. Performance load testing

Load testing, in general, refers to the practice of assessing a systems behaviour under load [6]. A load is typically based on an operational profile, which describes the expected workload of the system once it is operational in the field [7]. A load profile consists of the use case scenarios and the rate of these scenarios. For example, the load of a commercial website would contain information such as: browsing (40%) with a min/average/max rate of 5/10/20 requests/sec, and purchasing (40%) with a min/average/max rate of 2/3/5 requests/sec.

Figure 1 shows a typical workflow of a performance engineer. The first step is to execute the test with an operational profile. The second step is to monitor and record the performance counters during the test run. The third step is to analyze the performance counters and determine if the test run passes or fails. If it fails, the tester needs to determine which subsystem causes the problem so they can alert the appropriate team to investigate further.

Load testing is usually very time consuming. A test run typically lasts for several hours or even a few days. A memory leak, for example, might take hours or days to fully manifest itself. Therefore, the load tests have to be long enough. Also, load tests are usually performed at the end of the testing phase after all the functional tests. Testing in general is performed at the end of a development cycle. So load testers are usually under a tremendous pressure to finish testing and certify the software for release.

Because of the time sensitive nature of load testing. Companies try to automate as much as possible. The first step is usually automated using load generators which mimic the use case scenarios at the specified rate on the system under test. Mozilla, for example, has an automated load testing application [8] for testing their web browser suite. HP has the LoadRunner software [9] that can automatically simulate the work load of many network protocols. Microsoft also has a load testing tool that can drive traffic on web sites or web services. It is offered as part of the Visual Studio suite [10]. The second step is usually done automatically by instrumenting the operating system's performance components such as the Windows Reliability and Performance Monitor system [11] or Linux's `proc` [12]. Most software companies also implement their own performance counters to assist in the load testing process.

However, analyzing the load counters are usually done manually by the tester. This is a very overwhelming task. There are four main types of performance counters: CPU usage, Memory usage, Network IO, Disk IO. CPU usage is the percentage of CPU used. Memory usage is the number of memory used. Network IO is the amount of network input and output of the server. Disk IO is the amount of disk input and output. Each of these types has a few specific counters. For example, the CPU usage of a process can be measured by percentage of processor time, user time, privilege time, or interrupt rate. In large software system, there are hundreds of processes across different servers. On top of that, we also have customer-facing performance counters which directly influence users' interaction such as response time or processing rate. In total, each test run can produce thousands of performance counters.

Testers normally have to compare each counter of the new run, which is called target run, with older runs, which are called baselines. If most of the counters are similar to the counters in the baseline and are within the requirements, the test run passes. For example, the old run uses 40% of CPU on average. The new run uses 39% of CPU on average. This should be acceptable. If the new run uses 55% of CPU on average, there must be a performance problem with the new build.

Since there are too many counters, testers usually have to rely on tacit knowledge of the system to reduce the number of counters they have to investigate. For example, in an online store system, if the new changes are mostly in the credit card processing subsystem, the testers can just look at the counters of the processes belonging to that subsystem. However, tacit knowledge requires experience which takes time to accumulate. Also, unexpected behaviours might emerge differently from the testers' past knowledge. For example, the changes to the credit card processing subsystem might not influence its performance but incur additional processing on other subsystems. The replies of the credit card processing system can be in different format which causes more processing time for the check out subsystem.

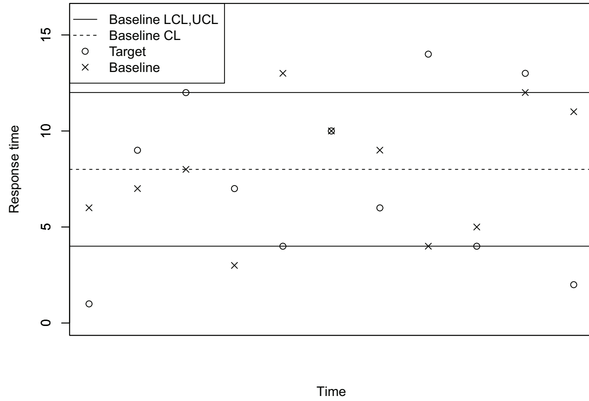

Fig. 2. An example of control chart built on the old run (baseline). The dashed line in the middle is the control line (CL). The solid lines are the the upper control limit (UCL) and the lower control limit (LCL). The × dots are the baseline run's readings. The ∘ dots are the target run's readings.

B. Control Charts

In this project, we want to investigate new approaches that can automatically compare the performance counters between the baseline test and the new target test based on a statistical quality control technique called control chart.

Control charts were first introduced by Shewhart [13] at Bell Labs, formerly known as Western Electric, in the early 1920s. The goal is to automatically determine if fluctuation in performance is caused by common causes, e.g. fluctuation of the input, or by special causes, e.g. defects. Control charts were originally used to monitor problems on telephone switches. Control chart has since became a common tool in statistical quality control. We note that, despise the name, control chart is not just a visualization technique. It is a technique to analyze process data.

Figure 2 shows an example of a control chart. The x-axis show the readings ordered, usually, in a unit of time, e.g. every minute. The y-axis shows the performance counter, e.g., the response time. A control chart is typically built by two sets of data: baseline test's counter readings and target test's counter readings.

The baseline readings determine the control limits. In our load testing world, the baseline would be an previously passed test run. The Centre Limit (CL) is median of all readings in the baseline set. The Lower Control Limit (LCL) is the lower limit of the normal behaviour range. The Upper Control Limit (UCL) is the upper limit. The LCL and UCL can be defined in several ways. A common choice is three standard deviations from the CL. Other choices would be the 1th, 5th, or 10th percentiles for the LCL and the 90th, 95th, or 99th percentiles for the UCL. Figure 2 shows an example where the baseline run has eleven response time readings consist of 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13 (not in that order). The LCL (10th), CL (median), and UCL (90th) for the baseline would be 4, 8, and 12 respectively. The baseline readings are shown with the × dots. The LCL and UCL are the solid lines. The CL is the dashed line in the middle.

TABLE I
DELL DVD STORE LOAD GENERATOR CONFIGURATION

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

We calculate violation ratio of the target readings data on the control chart of the baseline. In load testing, a target run is a new test run of the software which is being verified. In Figure 2, the target run's response time readings are the ∘ dots in the graph. These readings are 1, 2, 4, 4, 6, 7, 9, 10, 12, 13, and 14 (not in that order). The violation ratio indicates the degree to which the target run behaviour is deviated from the baseline. It is calculated as the percentage of the readings that are greater than the UCL or smaller than the LCL of the baseline. In this example, the readings of 1, 2, 13, and 14 are out side the limits set by the baseline. So the violation ratio would be $4/11 = 36\%$.

C. Control Chart Based Approaches for Load Test Verification

We propose to use control charts on performance counters to determine if the target test is similar to the baseline test. We can build a control chart for each counter of the baseline test. Then we score the violation ratio of the equivalent counter of the target test. A high violation ratio indicates that the target counters behave differently from the baseline counters. This signals that the target test might have performance problems.

To guide our inquiry, we derive the following research questions:

- **RQ1:** Can control charts automatically verify load tests?
- **RQ2:** Can control charts indicate subsystems with performance anomaly?

III. CASE STUDIES

In order to investigate the use of control chart as an approach to verify load test results, we conduct two case studies.

A. Study setting

1) Telecom: Our first case study is a large software system in the telecommunication industry. We call it Telecom for short. The software is developed in a tight agile development process. Performance engineers have to perform load tests at the end of each development iteration to determine if the software can be released. The software has a multi-tiers server client architecture. A typical load test exercises load on multiple subsystems which reside on different servers. The

behaviour of the subsystems and the hardware servers are recorded during the test run.

We cannot disclose any details about the software system due to a non disclosure agreement (NDA). We cannot disclose any real performance indicator either. Thus, we must scale our axes if they represent the system performance. This should not effect the validity of our analyses as we only compare the relative differences between the baseline and target runs' counters.

2) DVD Store: We investigate and build our approach using the Telecom case study. However, since we cannot discuss the technical details of Telecom, we apply our approach on the Dell DVD Store 2 which is a popular open-source web store for benchmarking hardware servers [5]. We call it DS2 for sort.

DS2 implements a sample web store which a user can browse, search, and purchase DVD titles. DS2 has a three-tier architecture. Clients access the store using a web browser. For testing purposes, a load generator is provided which mimics the behaviour of many users simultaneously. Table I shows our configuration for the load generator so others can replicate our analyses if needed. The front end web server is an Apache Tomcat application server [14]. The backend database is a MySQL database server [15]. We host each component on a separate server with a single-core Pentium 4 processor at 3.2 Ghz with 4 GB of RAM.

B. Experiment Design

In order to evaluate if a control chart approach can be used to automatically verify load test runs, we run controlled test runs on both the Telecom and the DS2 system.

We perform two types of test runs. The first type is the good runs. We pick a very stable version of the Telecom software and push it through a stable load profile. For DS2, we perform the test runs using the profile specified in Table I. Both systems perform comfortably under these load profiles. For DS2, the CPU usage of the web server is about 40% on average. The CPU usage of the database server is about 50% on average. Neither system is sensitive to memory or disk IO.

The second type of test runs is the problematic test runs which are test runs with simulated performance anomalies. We apply the same load on both systems as in the good runs. However, we simulate different anomalies to either the front end server or one of the back end servers. Table II shows the four anomalies we try to simulate. Since we cannot alter the code of Telecom, we simulate the problem in the front end's code (P1) in both systems by starting a separate process with realtime priority that uses about 50% (P1a) and 75% (P1b) of the CPU time. This process will compete with the actual service processes for CPU cycles. This effectively means that the front end code performs about 10% and 25% worst on DS2. We apply the same technique on one of the back send servers of Telecom and the MySQL database server of DS2 (P2). This simulates slower performance of the backend subsystem. Since we can modify the DS2 code, we also introduce two changes to the code. The first change is

TABLE II
PROBLEMS INTRODUCED TO SIMULATE PROBLEMATIC TEST RUNS

Problem ID	System	Description
P1a P1b	Both	CPU hog a) 50% and b) 75% of CPU on the front end server
P2a P2b	Both	CPU hog a) 50% and b) 75% of CPU on one of the back end servers
P3	DS2	Busy waiting in the front end server code
P4	DS2	Extra query to the database back end

a busy waiting loop in the credit card processing function (P3). This slows down the Tomcat front end process about 15 milliseconds per item ordered. The second change is to issue extra calls to the MySQL server (P4). This slows the MySQL server down to about 50% of its original capacity.

We use combinations of the good and problematic runs to answer RQ1 and RQ2 in the next two sections.

IV. RQ1: CAN CONTROL CHARTS AUTOMATICALLY VERIFY LOAD TESTS?

A. Motivation

The first step of load test analysis is to verify if the target test run passes or fails. So first goal of our control charts approach is to automatically classify if a test run is problematic or good using the performance counters. We use the control test runs as explained in Section III-B in different configurations to determine if this is feasible for both Telecom and DS2 system load tests. A configuration is the way we call a pair of baseline and target run. For example, Normal $\rightarrow$ Normal means that we build control chart using a normal run and test another normal run on it.

B. Approach

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

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

For both systems we use the main customer-facing counter response time to build our control charts. In both Telecom and DS2, the other performance counters such as processing rate are highly correlated to response time. So these counters should work the same way. For the control chart limits, we use 1th percentile and 99th percentile for the lower and upper

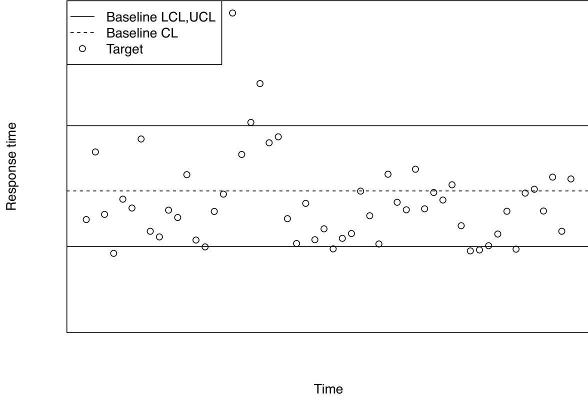

Fig. 3. Control chart of a good baseline run with a good target run in Telecom (Normal → Normal)

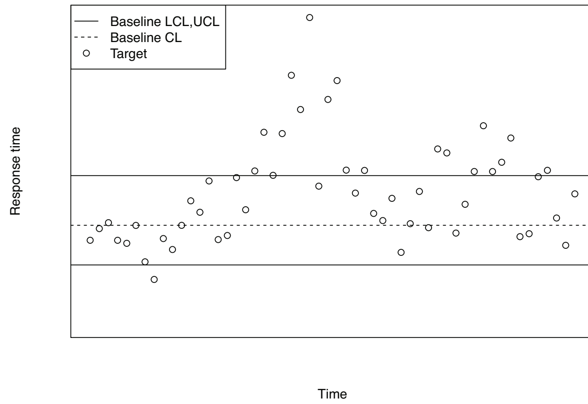

Fig. 4. Control chart of a good baseline run with a problematic target run (P2a) in Telecom (Normal → P2a)

control limits respectively. This is equivalent to about ± 3 standard deviations from the mean response time.

C. Result

Table III shows the violation ratio for different configurations on both systems. As we can observe, when we use a control chart to verify a good run against another good run (Normal → Normal), the average violation ratios are 4.67% and 12.37% for Telecom and DS2 respectively. Figure 3 shows an example of a control chart with a good baseline run and a good target run in Telecom. The x-axis shows the timeline of the test run. The y-axis shows the response time. As we can see, most of the target run's response times are within the control limits set by the baseline. This indicates that control chart approach can actually mark a good run as a good run.

On the other hand, the violation ratio is high when the target runs are problematic. On the Telecom system, if we mark all runs with violation ratio higher than 10% as problematic runs, we should be able to mark all problematic runs for further inspection. On the DS2 system, if we mark all runs with violation ratio higher 15%, we can catch all the problematic runs. Figure 4 shows an example of a control chart with a good baseline run and a problematic target run in Telecom (Normal → P2a). This target run has the P2a anomaly. As we can observe, there are more data points (response time samples) on the outside of the control limits set by the normal run. For P3 and P4, both runs on DS2 return a high violation ratios when compared to a normal DS2 run. These results indicate that control chart based approach can automatically verify load test run's result.

TABLE III
RQ1 EXPERIMENT RESULTS

Configuration	Violation Ratio	
	Telecom	**DS2**
Normal → Normal	4.67%	12.37%
Normal → P1a	18.51%	100%
Normal → P1b	94.33%	100%
Normal → P2a	11.11%	100%
Normal → P2b	15.09%	96.66%
Normal → P3	NA	28%
Normal → P4	NA	100%

Interestingly, despise being much simpler and smaller, the fluctuation in DS2 is much higher. When the target runs are normal, the average violation ratio is 12.37% compared to 4.67% on the Telecom system. Telecom is a commercial product which undergoes constant performance engineering tests. DS2, on the other hand, is a sample application that is not as throughly tested. This shows that careful attention to performance engineering can increase the stability of the system performance under adversed conditions.

Control chart based approach can automatically flag problematic test runs.

V. RQ2: CAN CONTROL CHARTS INDICATE SUBSYSTEMS WITH PERFORMANCE PROBLEM?

A. Motivation

Determining if a load test run is problematic is the first step. The next step is determining the subsystem responsible for the performance problem so the appropriate development team can be informed. If done manually, this step is more difficult and time consuming since there are many more counters to investigate. We want to derive a control chart based approach that shows the problematic subsystems using their performance counters e.g., CPU, memory, network IO, or disk IO.

B. Approach

The intuition is that if a subsystem is responsible for the performance issue, that subsystem's counters should have more violations compared to other subsystems. So we first build a control chart for each counter of a good baseline test.

Then for each equivalent counter of the problematic target test, we determine the violation ratio on the control chart. We rank the counters based on the violation ratio. The problematic subsystem should have the counters with the highest control chart violation ratio.

Since the DS2 system is very simple, we can only evaluate this approach on the Telecom system. We use the test runs with the P1 and P2 anomalies. As explained in Section III-B, the P1 test runs contain anomaly in the front end subsystem. The P2 test runs contain problems with one of the back end subsystem. We call this back end subsystem the problematic back end to distinguish it from other back end subsystems. If this approach works, the counters of the front end should have the most violation in the P1 test runs. The counters of the problematic back end should have the most violation in the P2 test runs.

C. Result

Table IV shows the top ten counters with the highest violation ratios for five configurations. In all five cases, the baseline is a good run. The target test run of the first configuration is also a good run (Normal → Normal). This is the case, the majority of the counters should have low violations. The next two configurations use target runs with P1a and P2a anomalies. P1a has performance problem in the front end. So the front end counters should have the most violations. P2a test has a performance problem in the problematic back end. So the backend counters for that subsystem should have the most violations. Similarly, the next two configuration use target runs with P1b and P2b problems. The expect results are similar to those of P1a and P2b.

We anonymize the counter names due to NDA. "Front end" signifies a performance counter that belongs to the front end server's processes. "Problematic back end" signifies a performance counter that belongs to the back end server's processes where we injected performance problems. "Other back end" signifies a performance counter that belongs to other back end servers's processes.

We can observe that, in the Normal → Normal configuration, the violation ratios are relatively low compared to the other four configurations. Except for the first four counters, most counters have less than 8% violation ratio. In the Normal → P1a and the Normal → P1b configurations, most of the top violated counters belongs to the front end. We can see this as an indicator that the anomaly most likely originates from the front end subsystem. This would be the correct situation. In the Normal → P2a and the Normal → P2b configurations, four of the problematic back end's counters are among the top five counters. This can be interpret as the problem most likely originates from the problematic backend which would also be the correct situation.

In all five test configurations, our approach correctly identifies the problematic subsystems. However, we can improve the technique by applying the following heuristics: in the Normal → Normal configuration, we mark the counters that have more than 10% violations. These counters change even though the test performance does not change. This means that these counters already fluctuate under normal conditions. For example, pruning processes are invoked once in a while to re-index the system internal data storage. The performance of such processes depend on a timer instead of the system's input load. So we can safely remove such counters from the counters pool that should be considered in future tests. In Table IV we mark these counters with a "*". As we can observe, once we apply this heuristic, the results are clearer. For example, in the Normal → P2a configuration, the top five counters are all from the problematic back end server.

> *Control chart can accurately identify the subsystem with the performance anomaly.*

VI. Discussion

A. Using control chart to automatically verify of the results of load tests

The result in RQ1 shows that a control chart approach can automatically determine if a test run is normal or problematic. This has a large potential application in practice because it removes the human intervention out of the load test verification process (Step 3 in Figure 1). Since Step 1 and Step 2 have been automated, the entire process can be automated now. For example, the development team can set up their build system to automatically perform load tests the same way JUnit tests are automated so far. When a developer checks in a new code, the build system will perform the integration build. Then it starts the load test and records the counters. Then the build system can count the violation ratio of the new run on control charts of past test runs. If the violation ratio is too high, the developer can be notified immediately. Being able to understand the performance impact of their code right away would make developers more conscious about performance. This can have a strong potential impact on the performance of software products like what JUnit tests have done for products quality.

B. Can this approach be generalized for other software system?

We believe that the results in RQ1 can be generalized. Customer-facing performance counters such as response time are relatively similar across software systems. Using control chart to classify a test run can be done similarly to our case study. The threshold of violation ratio, over which a run is deemed as a failure, can be determined automatically by running a few good runs to good runs comparison.

On the other hand, the results in RQ2 shows that some tacit knowledge is still required to automatically locate problematic subsystems. For example, we derived the heuristic to remove the marked counters, which always have high violation ratio, because we know their functionality. Such tacit knowledge is still required. However, we believe further studies into how different counters behave during a load test might be able to identify those counters automatically.

TABLE IV
THE TEN MOST OUT-OF-CONTROL COUNTERS IN TELECOM'S TEST RUNS

Normal → Normal	**Vio%**	**Normal → P1a**	**Vio%**	**Normal → P2a**	**Vio%**	**Normal → P1b**	**Vio%**	**Normal → P2b**	**Vio%**
Front end*	98%	Front end	100%	Front end*	100%	Front end	100%	Front end*	95%
Front end*	83%	Front end	98%	Problematic back end	98%	Front end	100%	Problematic back end	93%
Back end*	58%	Front end*	93%	Problematic back end	98%	Front end	100%	Problematic back end	93%
Front end*	23%	Front end	78%	Problematic back end	98%	Front end	100%	Problematic back end	93%
Back end	8%	Front end	75%	Problematic back end	98%	Front end	100%	Problematic back end	91%
Back end	8%	Front end	75%	Front end*	81%	Front end	100%	Front end*	83%
Back end	6%	Front end	73%	Other back end*	70%	Front end	100%	Other back end*	53%
Back end	6%	Other back end*	61%	Problematic back end	25%	Front end*	98%	Front end*	41%
Back end	5%	Other back end	46%	Front end*	25%	Front end*	71%	Front end	20%
Back end	5%	Front end*	43%	Other back end	15%	Front end*	66%	Other back end	10%

* Counters which are filtered out using heuristic.

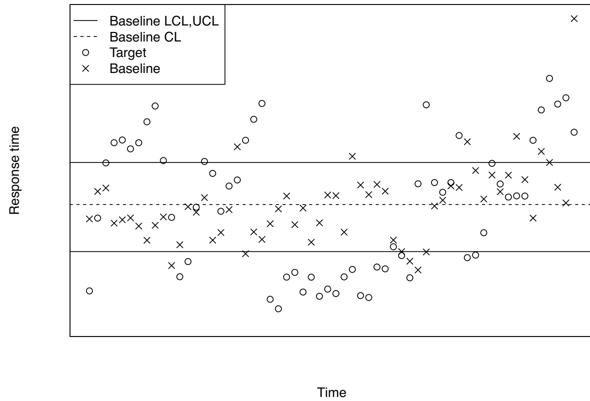

Fig. 5. Control chart of the same performance counter on two different test runs. Both runs have the same average. However, one fluctuates more than the other. Control chart would be able to distinguish between the two runs while comparing averages would miss.

C. Can we just compare the averages instead of the violation ratio?

A control chart is built around the average value of the readings. The average value is the CL of the control chart. Based on our results, a naturally question to ask is: Can we just compare the averages instead of the violation ratio? Comparing the averages, i.e., using the CLs alone, would give similar results in most cases except for CPU usage counters. CPU usage counters such as processor time, user time, or queue length usually have the same average but fluctuate differently under different conditions.

Figure 5 shows the processor time counter of a Telecom's subsystem in two different test runs. As we can see, both the baseline and the target readings fluctuate around the same CL. The plot of the target run's readings average would overlay the CL line of the baseline. However, the target readings fluctuate more than the baseline readings. In the target test, this subsystem seems to process all the queued inputs at quickly. Then it sleeps for a short period of time. Then it pick up all the queued inputs again and quickly process through. On the contrary, the same subsystem in the baseline run would pick up the input and process them slowly but steadily at the same average rate. A control chart would mark the target run as problematic while comparing the averages alone would not even though the performance characteristics are clearly different. This is why control chart based approaches are more accurate than comparing the averages.

VII. RELATED WORK

A. Load testing

There are other proposed approaches to automate the analysis of load test results. Foo et al. [3], [16] detected the change in behaviour among the performance counters using association rules. If the differences are higher than a threshold, the run is marked as problematic. For example, in the good run the front end CPU usage is low and the back end CPU usage is low. If a target run has low CPU front end but high CPU backend, then there might be a problem. The limitation of this approach is the amount of learning required. The verification system needs a large set of test runs to extract these rules. Control chart, while not as novel, can give results using only a limited number of baseline runs.

Malik et al. [17] used a factor analysis technique called principle component analysis to transform all the counters into a small set of more distinct vectors. Then they compare the pairwise correlations between the vectors in the target run with those of the baseline run. They were able to identify possible problems in the target run. Compared to this approach, the strength of control charts based approaches is their simplicity and intuitiveness. We find that practitioners rarely adopt approaches that rely heavily on statistical inferences as it is difficult to understand, rationalize, and defend in critical situations.

Performance load test also produce execution logs. Logs are text files which describe the events during the test run. For example, when and where web pages were requested from the server. Logs allow testers to trace the exact sequences of events if something wrong happens. Jiang et al. [18], [19] introduced approaches to automatically detect anomaly in performance load test. Their approaches automatically detect out-of-order

sequences in the system log. If the frequency of the out-of-order sequences is higher in a test run, the run is marked as problematic.

B. Other fields

There are also other approaches to detect anomaly in performance of production systems. These approaches can potentially be applied to load test analysis. For example, Cohen et al. [20] proposed the use of a supervised machine learning technique called Tree-Augmented Bayesian Networks to identify combinations of related metrics that are highly correlated with faults. This technique might be able to identify counters that are highly correlated with problematic runs. Jiang et al. [21] used Normalized Mutual Information to cluster correlated metrics. Then they used Wilcoxon Rank-Sum test on the metrics to identify faulty components. This approach can be used to identify problematic subsystems during a load test. Chen et al. [22] also suggest an approach which analyzes the execution logs to identify problematic subsystems. Cherkasova et al. [23] developed regression-based transaction models from performance counters. Then they use the model to identify runtime problems. These works can potentially be adapted to load test verification.

Researchers also suggested the use of control charts in monitoring production systems. Trubin et al. [24] used control charts to detect problems during the capacity management process at a large financial institute. In a production system, the performance counters fluctuate according to the field load. For example, a web server can be serving web pages in one minute and not serving any page in the next minute depending on user demand. The CPU utility will fluctuate accordingly. Control charts can automatically learn if the fluctuation is out of a control limit. At which time it can alert the operator. The main difference between monitoring and load testing is that, in monitoring, the fluctuation in performance are caused by variance in the field load. While in load testing, the load is consistent across the runs. Therefore, the fluctuation are most likely caused by the properties of the software, i.e. performance defects.

VIII. Conclusion

Our goal in this study is to derive an approach to automatically verify load tests results. We propose the use of control charts to a) automatically determine if a test passes or fails and to b) identify the subsystem where the performance problem originated. We conduct two case studies on a large commercial telecommunication software system and the open-source Dell DVD Store. We believe that our results warrant further studies to apply statistical quality control techniques such as control charts, into software testing. Statistical process control has been used in many fields such as business and manufacturing. Hence, researchers in those fields already have a broad and solid knowledge on how to leverage these techniques in their operation. If we can leverage these statistical quality control techniques into software testing, we might be able to reduce the cost of running load tests and improve software quality overall.

Acknowledgment

We would like to thanks Research in Motion for providing support and data access for this study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's products.

References

[1] J. Palmer, "Designing for web site usability," *Computer*, vol. 35, no. 7, pp. 102–103, 2002.
[2] 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 (TSE)*, vol. 26, no. 12, pp. 1147–1156, 2000.
[3] K. C. Foo, J. Zhen Ming, B. Adams, A. E. Hassan, Z. Ying, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *International Conference on Quality Software (QSIC)*, 2010, pp. 32–41.
[4] D. Montgomery, *Introduction to statistical quality control*. John Wiley, 2005.
[5] Dell Inc., "Dvd store test application," 2010, ver. 2.1.
[6] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold, 1984.
[7] A. Avritzer and E. R. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Transactions on Software Engineering (TSE)*, vol. 21, no. 9, pp. 705–716, 1995.
[8] Mozilla Project, "Talos," 2010.
[9] Hewlett Packard, "Loadrunner," 2010.
[10] Microsoft Corp., "Visual studio," 2011.
[11] ——, "Windows reliability and performance monitor," 2011.
[12] Linux Kernel Organization Inc., "Linux kernel," 2011.
[13] W. Shewhart, *Economic control of quality of manufactured product*. American Society for Quality Control, 1931.
[14] The Apache Software Foundation, "Tomcat," 2010, ver. 5.5.
[15] MySQL AB, "Mysql community server," 2011, ver. 5.5.
[16] K. C. Foo, "Automated discovery of performance regressions in enterprise applications," Master's thesis, 2011.
[17] H. Malik, "A methodology to support load test analysis," in *International Conference on Software Engineering (ICSE)*. Cape Town, South Africa: ACM, 2010, pp. 421–424.
[18] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *International Conference on Software Maintenance (ICSM)*, 2008, pp. 307–316.
[19] ——, "Automatic performance analysis of load tests," in *International Conference in Software Maintenance (ICSM)*, Edmonton, 2009, pp. 125–134.
[20] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *Symposium on Opearting Systems Design Implementation*. San Francisco, CA: USENIX Association, 2004, pp. 231–244.
[21] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *International Conference on Dependable Systems Networks (DSN)*, 2009, pp. 285–294.
[22] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 595–604.
[23] L. Cherkasova, K. Ozonat, M. Ningfang, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 452–461.
[24] I. Trubin, "Capturing workload pathology by statistical exception detection system," in *Computer Measurement Group (CMG)*, 2005.