Automatic Identification of Load Testing Problems

Zhen Ming Jiang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, ON, Canada
{zmjiang, ahmed}@cs.queensu.ca

Gilbert Hamann and Parminder Flora
Performance Engineering
Research In Motion (RIM)
Waterloo, ON, Canada

Abstract

Many software applications must provide services to hundreds or thousands of users concurrently. These applications must be load tested to ensure that they can function correctly under high load. Problems in load testing are due to problems in the load environment, the load generators, and the application under test. It is important to identify and address these problems to ensure that load testing results are correct and these problems are resolved. It is difficult to detect problems in a load test due to the large amount of data which must be examined. Current industrial practice mainly involves time-consuming manual checks which, for example, grep the logs of the application for error messages.

In this paper, we present an approach which mines the execution logs of an application to uncover the dominant behavior (i.e., execution sequences) for the application and flags anomalies (i.e., deviations) from the dominant behavior. Using a case study of two open source and two large enterprise software applications, we show that our approach can automatically identify problems in a load test. Our approach flags $< 0.01\%$ of the log lines for closer analysis by domain experts. The flagged lines indicate load testing problems with a relatively small number of false alarms. Our approach scales well for large applications and is currently used daily in practice.

1. Introduction

Many systems ranging from e-commerce websites to telecommunications must support concurrent access by hundreds or thousands of users. To assure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, such as unit and integration testing.

Load testing, in general, refers to the practice of accessing the system behavior under *load* [15]. *Load* refers to the rate of the incoming requests to the system. A load test usually lasts for several hours or even a few days. Load testing requires one or more load generators which mimic clients sending thousands or millions of concurrent requests to the application under test. During the course of a load test, the application is monitored and performance data along with execution logs are stored. Performance data record resource usage information such as CPU utilization, memory, disk I/O and network traffic. Execution logs record the run time behavior of the application under test.

Load testing is an area of research that has not been explored much. Most work focuses on the automatic generation of load test suites [11, 12, 13, 14, 16, 20, 29]. However, to the best of our knowledge, there is no previous work, which proposes the systematic analysis of the results of a load test to uncover potential problems.

Load testing is a difficult task requiring a great understanding of the application under test as observed by Visser [9]. Problems in the application under test (e.g., bugs), the load generator or the load environment are usually the sources of load testing problems. However, looking for problems in a load testing is a time-consuming and difficult task, due to the following challenges:

No Documented System Behavior: Correct and up-to-date documentation of the behavior of an application rarely exists [26].

Time Constraints: Load tests last for hours or days, and the time allocated for analyzing their results is limited. Load testing is usually the last step in an already delayed release schedule which managers must speed up.

Monitoring Overhead: Approaches which monitor or profile an application affect the validity of performance measurements and are not practical for load testing.

Large Volume of Data: A load test, running for a few hours, generates performance data and logs that are usually several hundred megabytes. This data must be analyzed to uncover any problems in the load test.

Due to the above challenges, most practitioners check the results of a load test in an ad-hoc manner. The following checks are commonly used in practice:

978-1-4244-2614-0/08/$25.00 © 2008 IEEE

Crash check Load testers check whether the application has crashed, restarted or hung during the load test.

Performance check Load testers examine performance metrics to see if they exhibit large fluctuations. For example, an up-trend for the memory usage throughout a load test is a good indicator of a memory leak.

Basic error check Load testers perform a more in-depth analysis by grepping through the log files for specific keywords like "failure", or "error". Load testers then analyze the context of the matched log lines to determine whether they indicate problems or not.

Depending on the length of a load test and the volume of generated data, it takes load testers several hours to perform these checks. We believe this current practice is not efficient since it takes hours of manual analysis, nor is it sufficient since it may miss possible problems. On one hand, not all log lines containing terms like "error" or "failure" are worth investigating. A log such as "Failure to locate item in the cache" is likely not a bug. On the other hand, not all errors are indicated in the log file using the terms "error" or "failure". For example, even though the log line "Internal queue is full" does not contain the words "error" or "failure", it might also be worthwhile investigating it, since newly arriving items are possibly being dropped.

In this paper, we introduce an approach for automatically uncovering the dominant behavior of an application by mining logs generated during a load test. We use the recovered dominant behavior to flag any deviation, i.e., anomalies, from the dominant behavior. The main intuition behind our work is that a load test repeatedly executes a set of scenarios over a period of time. Therefore, the applications should follow the same behavior (e.g. generate the same logs) each time the scenario is executed. Therefore, the dominant behavior is probably the normal (i.e., correct) behavior and the minority (i.e. deviated) behaviors are probably troublesome and worth investigating. The main contributions of our work is as follows:

1. This work is the first work to propose a systematic approach to detect problems in a load test;
2. Unlike other existing log analysis work which require a user specified model (e.g., [10]), our approach is self-learning, requiring little domain knowledge about an application and little maintenance to update the models over releases. The model for the dominant behavior is created automatically;
3. Case studies show that our approach flags a very small percentage of log lines that are worth investigating. The approach produces very few false alarms (precision $> 50\%$) with many of the flagged lines indicating load testing problems;
4. Our proposed approach is easy to adopt and scales well to large scale enterprise applications. Our approach is currently used daily in practice for analyzing the results of load tests of large enterprise applications.

Organization of the Paper

The paper is organized as follows: Section 2 explains the types of problems that can occur during a load test. Section 3 describes our anomaly detection approach. Section 4 presents a case study of our anomaly detection approach. We have applied our approach to uncover problems in load tests for two open source and two large enterprise applications. Section 5 discusses our current approach and present some of its limitations. Section 6 describes related work. Section 7 concludes this paper and lists some future work.

2. Load Testing Problems

Load testing involves the setup of a complex load environment. The application under test should be setup and configured correctly. Similarly, the load generators must be configured correctly to ensure the validity of the load test. The results of a load test must be analyzed closely to discover any problems in the application under test (i.e., load related problems), in the load environment, or in the load generation. We detail the various types of problems that occur during load testing.

Bugs in the Application Under Test

The main purpose of a load test is to uncover *load sensitive* errors. Load sensitive errors are problems which only appear under load or extended execution. For example, memory leaks are not easy to spot under light-load with one or two clients, or during a short-run. However, memory leaks usually exhibit a clear trend during extended runs. Another example of load sensitive errors are deadlock or synchronization errors which show up due to the timing of concurrent requests.

Problems with the Load Environment

Problems with the load testing environment can lead to invalid test results. These problems should be identified and addressed to ensure that the load test is valid. Examples of load environment problems are:

Mis-configuration The application under test or its runtime environment, e.g., databases or web servers, may be mis-configured. For example, the number of concurrent connections allowed for a database may be incorrect. A small number of allowed connections may prevent the login of several users and would lead to a lower load being applied on the application under test.

Hardware Failures The hardware running the application and the load test may fail. For example, the hard disks may fill up due to the tester forgetting to clean up the data from an older run. Once the disk is full, the application under test may turn-off specific features. This would lead to an incomplete load test since some of the

functionalities of the application have not been fully load tested.

Software Interactions A load test may exhibit problems due to intervention from other applications. For example, during a long running load test, an anti-virus software may start up and and intervene with the running load test. Or the operating system may apply updates and reboot itself.

Problems with the Load Generation
Load generators are used to generate hundreds or thousands of concurrent requests trying to access the application. Problems in the load generators can invalidate the results of a load test. Examples of possible load generation problems.

Incorrect Use of Load Generation Tools Some of the generic load testing tools [8] require load testers to first record the scenarios, edit the recordings and replay them. This is an error-prone process. Edited recordings may not trigger the same execution paths as expected. For example, in a web application, the recorded URLs have a session ID which must be consistent for each request by the same user otherwise the application would simply return an error page instead of performing the expected operations.

Buggy Load Generators The load generation tools are software applications which may themselves have *load sensitive* problems or bugs. For example, rather than sending requests to the application under test in a uniform rate, many load generation tools allow load testers to specify different distributions. However, the requests may not follow that distribution during a short run.

It is important to identify and remedy these problems. However, identifying these problems is a challenging and time-consuming task due to the large amount of generated data and the long running time of load tests. The motivation of our work is to help practitioners identify these problems.

3. Our Anomaly Detection Approach

The intuition behind our approach is that load testing involves the execution of the same operations a large number of times. Therefore, we would expect that the application under test would generate similar sequences of events a large number of times. These highly repeated sequences of events are the dominant behavior of the application. Variations from this behavior are anomalies which should be closely investigated since they are likely to reveal load testing problems.

We cannot instrument the application to derive the dominant behavior of the application, as instrumentation may affect the performance of the application and the software behavior won't be comparable with the deployed application. Fortunately, most large enterprise applications have some form of logging enabled for the following reasons:

1. to support remote issue resolution when problems occurs and
2. to cope with recent legal acts such as the "Sarbanes-Oxley Act of 2002" [7] which stipulate that the execution of telecommunication and financial applications must be logged.

Such logs record software activities (e.g. "User authentication successful") and errors (e.g. "Fail to retrieve customer profile"). We can mine the dominant behavior of the application from these commonly available logs. In this section we present an approach to detect anomalies in these logs. These anomalies are good indicators of problems in a load test.

As shown in Figure 1, our anomaly detection approach takes a log file as input and goes through four steps: Log Decomposition, Log Abstraction, Dominant Behavior Identification, and Anomaly Detection. Our approach produces an HTML anomaly report. We explain each step in detail in the following subsections.

3.1. Log Decomposition

Most modern enterprise applications are multi-threaded applications which process thousands of transactions concurrently. The processing of all these transactions is logged to a log file. Related log lines do not show up continuously in the log file, instead they may be far apart. The log decomposition step processes the log file and groups related log lines together. Log lines could be related because they are processed by the same thread or because they are related to the same transaction. Most of the enterprise applications have a standard format for logging the transaction information (e.g. header part of a log line), as this information is important for remote issue resolution. For example, in a web application, each log line contains a session or customer ID. Or in a multi-threaded application, each log line contains a thread ID. Or in a database application, each log line might contain the transaction ID. Sometimes, a log line might contain multiple types of IDs. For example, in an e-commerce application, a log line can contain both the session and customer IDs. Depending on the granularity of the analysis, one or multiple of these IDs are used to group related lines together.

Table 1 shows a log file with 7 log lines. If the log file is decomposed using the accountId field, the log decomposition step would produce 3 groups (Tom, Jerry and John). This step requires domain knowledge by the load tester to decide which field to use to decompose the logs.

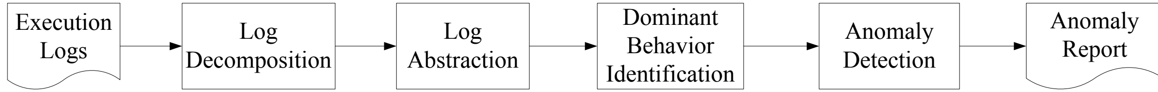

Figure 1. Our anomaly detection approach

#	Log lines	Group
1.	accountId(*Tom*) User purchase, item=*100*	Tom
2.	accountId(*Jerry*) User purchase, item=*100*	Jerry
3.	accountId(*Tom*) Update shopping cart, item=*100*	Tom
4.	accountId(*John*) User purchase, item=*103*	John
5.	accountId(*Tom*) User checkout	Tom
6.	accountId(*Jerry*) Update shopping cart, item=*100*	Jerry
7.	accountId(*John*) User purchase, item=*105*	John

Table 1. Example log lines

Event ID	Event Template
E_1	User purchase, item=$v
E_2	Update shopping cart, item=$v
E_3	User checkout

Table 2. Example execution events

3.2. Log Abstraction

Each log line is a mixture of dynamic and static information. Log lines containing the same static information belong to the same execution event. We want a technique that would recognize that two log lines are due to the same event. We call this process the log abstraction problem. We have developed a technique which can uniquely map each log line to an execution event [22]. The technique parameterizes log lines using a similar process as token-based code cloning techniques. The log lines in Table 1 would be abstracted to only 3 execution events as shown in Table 2. The "$v" sign indicates a runtime generated parameter value.

Based on the log decomposition and abstraction steps, the sample log file in Table 1 would result in the grouping of events shown in Table 3.

Group	Event ID	Log line #
Tom	E_1	1
	E_2	3
	E_3	5
Jerry	E_1	2
	E_2	6
John	E_1	4
	E_1	7

Table 3. The sample log file after the log decomposition and the log abstraction steps

$(E, *)$	Event Pair	Occurrences	Sample Line #
$(E_1, *)$	$(\boldsymbol{E_1}, \boldsymbol{E_2})$	2	1, 3
	(E_1, E_1)	1	4, 7
$(E_2, *)$	$(\boldsymbol{E_2}, \boldsymbol{E_3})$	1	3, 6

Table 4. Log file after the dominant behavior identification step

3.3. Dominant Behavior Identification

In this step, we identify the dominant behavior in the logs. We achieve this by analyzing the *execute-after* relations for each event E. The **execute-after** relation for an event E, denote by (E, *), refers to the occurrences of all the event pairs with the leading event E. Two events E_1 and E_2 form an *event pair*, if

1. E_1 and E_2 belong to the same group; and
2. E_2 is the next event that directly follows E_1.

In the event pair (E_1, E_2), E_1 is referred to as the leading event. The execute-after pair for event E_1 is formed by aggregating all the event pairs which have E_1 as the leading event. Table 4 shows all the execute-after pairs in Table 3. There are two execute-after pairs: one for E_1 and one for E_2. For each execute-after pair, the table shows the event pairs, the number of occurrences for each event pair, and a sample log lines corresponding to the first occurrence of each event pair. There are two types of events which are executed after the *User purchase* event (E_1). E_1 could be followed with another E_1. This is generated by John's session from log lines 4 and 7. Or E_1 could be followed with *Update shopping cart* (E_2). There are two occurrences of (E_1,E_2) which are attributed to Tom's and Jerry's sessions. The first and third log lines correspond to the first occurrence of the event pair (E_1,E_2). Event pairs are grouped by the *execute-after* relations. For example, (User purchase, *) includes all the event pairs which start with the *User purchase* event. Thus, the event pairs (User purchase, Update shopping cart) and (User purchase, User purchase) are grouped under the *execute-after* relations for the *User purchase* event, (User purchase, *).

The dominant behavior for (E, *) refers to the largest event pair(s) which starts with E. The dominant behavior pairs for each *execute-after* relation are shown in bold in Table 4. The dominant behavior for (E_1, *) is (E_1, E_2). The dominant behavior for (E_2, *) is (E_2, E_3). Sample line

(**E**, $*$)	Event Pair	Frequency
(User purchase, *)	**(User purchase, Update cart)**	$1,000$
	(User purchase, User purchase)	1
(User signin, *)	**(User signin, Browse catalog)**	100
	(User signin, Update account)	20
	(User signin, Browse history)	10
(Browse catalog, *)	**(Browse catalog, User purchase)**	500
	(Browse catalog, Update account)	500
	(Browse catalog, Search item)	100

Table 5. Summary of execute-after pairs

#	Z-Stat	Kinds	Min	Max	Total	Event
E_1	10.44	2	1	1,000	1,001	accountId(Tom) User purchase, item=100
E_3	-4.97	3	10	100	130	accountId(Tim) User signin, user=Tim
E_4	-49.25	3	100	500	1,100	accountId(John) Browse catalog, catalog=book

Figure 2. An example anomaly report

#	Z-Stat	Kinds	Min	Max	Total	Event
E_1	10.44	2	1	1,000	1,001	accountId(Tom) User purchase, item=100
E_3	-4.97	3	10	100	130	accountId(Tim) User signin, user=Tim
E_4	-49.25	3	100	500	1,100	accountId(John) Browse catalog, catalog=book

Freq	Sample	Details (Sort by Freq)
1,000 (99%)	log.txt, line 20 log.txt, line 23	E_1 --> accountId(Tom) User purchase, item=100 E_2 --> accountId(Tom) Update shopping cart, item=100
1 (<1%)	log.txt, line 104 log.txt, line 108	E_1 --> accountId(John) User purchase, item=103 E_1 --> accountId(John) User purchase, item=105

Figure 3. An expanded anomaly report

numbers show the first occurrences of the event-pair in the log file. The sample log line numbers are displayed later in the anomaly report.

3.4. Anomaly Detection

The previous step identifies the dominant behavior in the logs. In this step, we mark any deviations, i.e., anomalies, from the domination behavior for closer investigation. As load testers have limited time, we need a way to rank anomalies to help load testers prioritize their investigation. We use a statistical metric called *z-stats*. Recent work by Kremenek and Engler [23] shows that the z-stats metric performs well in ranking deviation from dominant behaviors when performing static analysis of source code. The z-stats metric measures the amount of deviation of an anomaly from the dominant behavior. The higher the z-stats is, the stronger the probability that the majority behavior is the expected behavior. Therefore the higher the z-stats value, the higher the chance that a deviation, i.e. low frequency pairs, are anomalies that are worth investigating. The formula to calculate z-stats is as follows: $z(n,m) = \frac{(\frac{m}{n} - p_0)}{\sqrt{\frac{p_0 \times (1-p_0)}{n}}}$, where n is total number of occurrences of event E, m is the occurrences of the dominant event pairs which starts with E, and p_0 is the probability of the errors. p_0 is normally assigned a value of 0.9 [19] for error ranking.

We illustrate the use of z-stats using the example shown in Table 5 with dominant behavior marked in bold. The dominant behavior for (User purchase, *) is (User purchase, Update cart). Thus the z-stats for (User purchase, *) is calculated as follows ($m = 1000$, $n = 1001$): $z(1001, 1000) = \frac{(\frac{1000}{1001} - 0.9)}{\sqrt{\frac{0.9 \times (1-0.9)}{1001}}} = 10.44$, and the z-stats for (User signin, *) is z($m = 100$, $n = 130$) = -4.97. The dominant behavior for (Browse catalog, *) is (Browse catalog, Purchase item) or (Browse catalog, Update account). Thus the z-stats for (Browse catalog, *) is z($m = 500$, $n = 1100$) = -49.25. (User purchase, *) has a higher z-stats score than (User signin, *) and (Browse catalog, *). This indicates that low frequency event pairs in the group of (User purchase, *) are likely anomalies that should be investigated closely. Normally, each purchase is followed by an update. The missing *Update cart* event suggests that the system might miss information about items selected by a customer.

3.5. Anomaly Report

To help a load tester examine the anomalies, we generate an anomaly report. The report is generated in dynamic-HTML so testers can easily attach it to emails that are sent out while investigating a particular anomaly.

Figure 2 shows the generated report for our running example. Our anomaly report is a table with each row corresponding to one *execute-after* relation. Rows are sorted by decreasing z-stats score. *Execute-after* relations with high z-stats value are more likely to contain anomalies that are worth investigating. The first row in Figure 2 corresponds to the execute-after pair for the *User purchase* event (E_1). There are in total two types of event pairs with *User purchase* as the leading event. One event pair occurs $1,000$ times and the other event pair occurs just once. In total, all the event pairs, with *User purchase* as the leading event, appear $1,001$ times during the course of this load test. A sample line for this event (E_1) is also shown.

Each sample line is a clickable hyperlink. Once a user clicks the hyperlink, the report shows detailed information about the execute-after pairs for that event. Figure 3 shows the screenshot of the anomaly report after clicking the sample line for Event E_1. Event pairs for (User purchase, *) are sorted with decreasing frequency. The topmost event pair (User purchase, Update cart(E_2)) is the dominant behavior. (User purchase, Update cart) occurs 99% $(1,000)$ of

the time. The first occurrence of this event pair is in log file *log.txt* lines 20 and 23. The other event pair (User purchase, User purchase) is a deviated behavior. It occurs only once ($< 1\%$). It is recorded in log file *log.txt* lines at 104 and 108.

4. Case Studies

We have conducted three case studies on four different applications. The application are: the Dell DVD Store (DS2), the JPetStore application (JPetStore), and two large enterprise software applications. Table 6 gives an overview of these four case studies. Based on our experience, z-stats lower than 10 are likely noise. Thus, we only output event pairs with z-stats score larger than 10 in these four experiments. The table summarizes the types of the applications, the duration of the load test, size of logs, and our anomaly detection results. For example, Dell DVD Store is an open source web applications implemented using JSP. The load test was 5 hours long and generated a log file with $147,005$ log lines. Our anomaly detection approach takes less than 5 minutes to process the log file. We have discovered 23 abstract event types. There are 4 anomalies detected and 18 log lines are flagged, that is less than 0.01% of the whole log file. Among these four anomalies, two of them are actual problems in the load test. Our precision is 50%. Among these two problems: one is a bug in the application under test, the other is a bug in the load generator. We did not detect any problems with the load environment. The percentage of flagged lines is the total number of log lines shown in the anomaly report. As shown in Figure 4, there are total 9 event pairs ($3+2+2+2$). Thus our approach has flagged $9 \times 2 = 18$ lines. The processing time for our approach is measured using a laptop with $2G$ memory, $7,200$ RPM hard-drive and a Dual Core 2.0 GHz processor.

The rest of this section covers the details of our case studies. For each application, we present the setup of the load test then we discuss the results of applying our approach to identify problems. The goal of the studies is to measure the number of false positive (i.e. precision) reported by our approach. A false positive is a flagged anomaly that did not point to a load testing problem. We cannot measure the recall of our approach since we do not know the actual number of problems.

4.1. DELL DVD Store

The DVD Store (DS2) application is an online web application [1]. DS2 provides basic e-commerce functionality, including: user registration, user login, product search, and item purchase. DS2 is an open source application and is used to benchmark Dell hardware, and for database performance comparisons [6]. DS2 comes in different distribution package to support various web platforms (e.g. Apache Tomcat, or ASP .NET) and database vendors (MySQL, Microsoft SQL Server, and Oracle).

Parameter	Value
Duration	5 hours
Number of driver threads	50
Startup request rate	5
Think time	50 sec
Database size	Small
Percentage of new customers	20%
Average number of searches per order	3*
Average number of items returned in each search	5*
Average number of items per order	5*

Table 7. Workload configuration for DS2

Experiment Setup

DS2 contains a database, a load generator and a web application. For a load test, the database is populated with entries using provided scripts. The web application consists of four JSP pages which interact with the database and display dynamic content. The DS2 load generator supports a range of configuration parameters to specify the workload. Table 7 shows the parameters used in our experiment. Note that "Think Time" refers to the time the user takes between different requests. We use a small database, which by default contains $20,000$ users. Parameter values marked with a "*" indicate that we use the default value. In this experiment, we use MySQL as the backend database and the Apache Tomcat as our web server engine. We increase the number of allowed concurrent connections in MySQL to enable a large number of concurrent access. For this configuration, The web application layer is implemented in JSP and the load generator is implemented in C#.

Each action from the user (login, registration, browse, purchase) results in a separate database connection and transaction. Since DS2 has no logs, we manually instrument its four JSP pages so that logs are output for each database transaction. Each log line also contains the session ID and customer ID.

Analysis of the Results of the Load Test

The load test generated a log file with $147,005$ log lines for 23 execution events. Our approach takes about 2 minutes to process the logs.

Figure 4 shows the screenshot of the anomaly report for the DS2 application. The report shows 4 anomalies. We cross examine the logs with the source code to determine whether the flagged anomalies are actual load testing problems. The precision of this report is 50%. Two out of four anomalies are actual problems in the load tests. We briefly explain these two problems.

Figure 4 shows the details of the first anomaly. The first execute-after pair is about a customer trying to add item(s) into their shopping cart (E_{13}). About 99% ($87,528$) of the time, the customer's shopping cart is empty. Therefore, a

Applications	DS2	JPetStore	App 1	App 2
Application Domain	Web	Web	Telecom	Telecom
License	Open Source	Open Source	Enterprise	Enterprise
Source Code	JSP, C#	J2EE	C++, Java	C++, Java
Load Test Durations	5 hr	5 hr	8 hr	8 hr
Number of Log lines	$147,005$	$118,640$	$2,100,762$	$3,811,771$
% Flagged	$\frac{18}{147005}(< 0.01)\%$	$\frac{8}{118640} < 0.01\%$	$< 0.01\%$	$< 0.01\%$
Number of Events	23	22	> 400	> 400
Application Size	$2.3M$	$4.9M$	$> 300M$	$> 400M$
Precision	$\frac{2}{4}(or\ 50\%)$	$\frac{2}{2}(or\ 100\%)$	56%	100%
Processing Time	< 5 min	< 5 min	< 15 min	< 15 min
Break Down of Problems (Application/Environment/Load)	1/0/1	2/0/0	Y/Y/N	Y/N/N

Table 6. Overview of our case studies

shopping cart is created in the database along with the purchased item information (E_{14}). Less than 1% $(1,436)$ of the time, the customer adds more item(s) into their existing shopping cart (E_{13}). For the other 358 $(< 1\%)$ cases, the customer directly exits this process without updating the order information (E_{15}).

The first event pair is the dominant behavior. The second event pair refers to the cases that customers purchases multiple items in one session. However, the last event pair (E_{13}, E_{15}) looks suspicious. A closer analysis of the DS2 source code reveals that this is a bug in the web application code. DS2 pretty prints any number if it is larger than 999. For example, 1000 would be outputted as $1,000$. However, the pretty printed numbers are concatenated into the SQL statement which are used for updating (or inserting) the customer's information. The additional comma results in incorrect SQL code since a comma in the SQL statements means different columns. For example, a SQL statement like: "INSERT into DS2.ORDERS (ORDERDATE, CUSTOMERID, NETAMOUNT, TAX, TOTALAMOUNT) ('2004-01-27', 24, 888, 313.24, 1,200)" will cause an SQL error, since SQL treats a value of $1,200$ for TOTALAMOUNT as two values: 1 and 200.

The second and third anomalies are not problems. They are both due to the nature of the applied load. For example, the second anomaly is because we only have a few new customers in our experiment (20% new customers in Table 7). The expected behavior after each customer login is to show their previous purchases. There are a few occurrences where DS2 does not show any previous purchase history. These occurrences are due to newly registered customers who do not have any purchase history.

The fourth anomaly is due to a problem with the load generator. The load generator randomly generates a unique ID for a customer. However, the driver does not check whether this random number is unique across all concurrent executing sessions. The shown anomaly is due to one occurrence, in a 5 hour experiment, where two customers were given the same customer ID.

4.2. PetStore

We used our approach to verify the results of a load test of another open source web application software called JPetStore [2]. Unlike Sun's original version of Pet Store [5] which is more focused on demonstrating the capability of the J2EE platform, JPetStore is a re-implementation with a more efficient design [3] and is targeted for benchmarking the J2EE platform against other web platforms such as .Net.

JPetStore is a larger and more complex application relative to DS2. Unlike DS2 which embeds all the application logic into the JSP code, JPetStore uses the "Model-View-Controller" framework [4] and XML files for object/relational mappings.

Experiment Setup

We deployed JPetStore application on Apache Tomcat and use MySQL as the database backend. As JPetStore does not come with a load generator, we use Webload [8], an open source web load testing tool, to load test the application. Using webload we recorded four different customer scenarios for replay during load testing. In Scenario 1, a customer only browses the catalog without purchasing. In Scenario 2, a new customer first registers for an account, then purchase one item. In Scenario 3, a new customer first purchases an item, then register for an account then checkout. In Scenario 4, an existing customer purchases multiple items.

In this load test, we ran two WebLoad instances from two different machines sending requests to the JPetStore web application. For each WebLoad instances, we added in $5,000$ users. Table 8 shows the workload configuration parameters for JPetStore load test. Note that WebLoad can specify the distribution of the generated request rate. In this experiment, we specify a random distribution for the user's requests with minimum rate 5 requests/sec and maximum rate 150 requests/sec.

Analysis of the Results of the Load Test

The load test generated a log file with 118,640 log lines. It takes our approach around 2 minutes to process the logs.

#	Z-Stat	Kinds	Min	Max	Total	Event
E_{13}	79.61	3	358	87,528	89,322	SessionID=19420, Entering purchase for simple quantity queries (details below)
E_{6}	39.96	2	1	14,393	14,394	SessionID=11771, Login finish for existing user
E_{19}	34.73	2	317	16,273	16,590	SessionID=14128, End of purchase process
E_{22}	20.65	2	1	3,857	3,858	SessionID=12067, Purchase complete

Details for E_{13}:

Freq	Sample	Details (Sort by Freq)
87,528 (98%)	ds2logs.txt 688 ds2logs.txt 689	E_{13} --> SessionID=19420, Entering purchase for simple quantity queries E_{14} --> SessionID=19420, Initial purchase, update cart
1,436 (<1%)	ds2logs.txt 2,484 ds2logs.txt 2,488	E_{13} --> SessionID=16242, Entering purchase for simple quantity queries E_{13} --> SessionID=16242, Entering purchase for simple quantity queries
358 (<1%)	ds2logs.txt 10,020 ds2logs.txt 10,021	E_{13} --> SessionID=13496, Entering purchase for simple quantity queries E_{15} --> SessionID=13496, Finish purchase before commit

Figure 4. DS2 Anomaly Report

Parameter	Value
Duration	300 minutes (5 hours)
Request rate	5 - 150 (random distribution)
Think time	50 sec
Scenario 1/2/3/4	25% / 25% / 25% / 25%

Table 8. Workload configuration for JPetStore

Two anomalies are reported and they are both application problems.

The first problem is a bug in the registration of new users. We have two load generators running concurrently. Each load generator has an input file with randomly generated customer IDs. These customer IDs are used to generate web requests for scenarios (2 and 3). There are a some user IDs which are common to both WebLoad instances. If a user tries to register an ID which already exists in the database, PetStore does not gracefully report a failure. Rather, PetStore will output a stack of JSP and SQL errors.

The second problem reveals that JPetStore does not process page requests when it is under a heavy load. There is one instance out of $22,330$ instances where the header JSP page is not displayed. The error logs for the WebLoad tool indicate that the PetStore application timed out and could not process the request for the header JSP page on time.

4.3. Large Enterprise Applications

We applied our approach on two large enterprise applications, which can handle thousands of user requests concurrently. Two applications are both tested for 8 hours. It takes our approach about 15 minutes to process the log files. Table 6 shows the precision of our approach (56% - 100%). We have found bugs in development versions of the applications (App 1 and App 2). One of the bugs in the applications shows the SQL statement was corrupted due to a memory corruption. Further investigation leads to a memory corruption problems in the systems. In addition, our approach detected problems with the load environment due to the complexity of the load environment for the enterprise applications. The false positives in App 1 are mainly due to some rare events at the start of the application. When using our approach in practice, load testers commented that:

1. Our approach considerably speeds up the analysis work for a load test from several hours down to a few minutes.
2. Our approach helps uncover load testing problems by flagging lines that do not simply contain keywords like “error” or “failure”.
3. Our approach helps load testers communicate more effectively with developers when a problem is discovered. The generated HTML report can be emailed to developers for feedback instead of emailing a large log file. Moreover the report gives detailed examples of the dominant and the deviated behaviors. These simple examples are essential in easing the communication between the testers and the development team.

5. Discussions and Limitations

Our approach assumes that load testing is performed after the functionality of the application is well tested. Thus, the dominant behavior is the expected behavior and the minority deviated behavior is the anomalies. However, this might not be a valid assumption. For example, if the disk of an application fills up one hour into a ten hour load test, then

the majority of the logs will be the error behavior. That said, our approach would still flag this problem and the expert analyzing the logs would recognize that dominant behavior is the problematic case.

Our approach processes the logs for the whole load test at once. This whole processing might cause our approach to miss problems. For instance, if the disk for the database fills up halfway during a load test, the application under test will report errors for all the incoming requests which arrives afterwards. Normal and erroneous behavior may have equal frequencies. Our statistical analysis would not flag such a problem. However, if we segment the log files into various chunks and process each individual chunk separately, we can detect these types of anomalies by comparing frequencies across chunks.

Finally, our anomaly report contains false positives. Anomalies can be flagged due to the workload setup. For example, our report for the DS2 case study contains two false positives which are due to the workload. Also in a threaded application when a thread is done processing a particular request and starts processing a new request, the pair of events: event at end of a request and event at start of a request may be incorrectly flagged as an anomaly. We plan on exploring techniques to reduce with these false positives. For now, load testers are able to specify a false positive pair in a separate exclusion file. These pairs are used to clean up the results of future log file analysis.

6. Related Work

Much of the work in literature focuses on identifying bugs in software applications. Our work is the first, to our knowledge, that tackles the issue of identifying problems in load tests. These problems may be due to problems in the application, load generation or load environment.

The work in the literature closest to our approach is all the work related to inferring dominant properties in a software application and flagging deviations from these properties as possible bugs. Such work can be divided into three types of approaches: 1) **Static approaches** which infer program properties from the source code and report code segments which violate the inferred properties; 2) **Dynamic approaches** infer program properties from program execution traces; 3) **Hybrid approaches** combine both approaches. Table 9 summarizes the related work. For each work, the table shows the main idea of the approach, the used techniques, and the challenge of directly adopting this approach to load testing.

Dawson et al. [19] gather statistics about the frequency of occurrence for coding patterns such as: pointer deference and lock/unlock patterns. They then use z-stats to detect and rank the errors. Li et. al. [24] use frequent itemset mining techniques to mine the call graph for anomalies. However, this approach produces many violations and the authors only evaluate a few violations thus the overall precision is unknown.

Hangal et al. [21] use dynamically inferred invariants to detect programming errors. Csallner et. al. [18] further improve the precision of bug detection techniques by executing the program using automatically generated test cases that are derived from the inferred invariants. Liu et. al. [25] use hypothesis testing on code branches to detect programming errors. The above three techniques cannot be applied to load testing since the detailed instrumentations would have an impractical performance overhead.

Weimer et al. [27] detect bugs by mining the error handling behavior using statistical ranking techniques. Their approach only works for Java applications which have try-catch blocks and requires good knowledge of the source code which is not applicable for load testers..

Yang et al. [28] instrument the source code and mine the sequences of call graphs (pairs) to infer various programme properties. They look at function calls which are directly adjacent to each other as well as gapped function pairs. Due to the large size of inferred explicit properties, they use heuristics to select interesting patterns (e.g. lock/unlock). Their approach requires a great deal of manual work and the instrumentations has a high performance overhead.

Cotroneo et al. [17] produce a finite state machine based on profiled data. Then a failed workload is compared against the finite state machine to infer the failure causes. Profiling during a load test is infeasible due to inability to collect performance data. In addition, inferring a deterministic finite machine is not possible in a complex workload due to the large number of events that are generated using random load generators.

7. Conclusions and Future Work

In this paper, we present an approach to automatically identify load testing problems. Our approach mines the logs of an application to infer the dominant behavior of the application. The inferred dominant behavior is used to flag anomalies. These anomalies are good indicator of load testing problems. Our case study on four applications shows that our approach performs with high precision and scales well to large systems. In the future work, we plan to estimate the recall value by injection load testing problems and determining whether our approach would identify them.

Acknowledgement

We are grateful to Research In Motion (RIM) for providing access to the enterprise applications used in our case study. The findings and opinions expressed in this paper

	Category	Key Idea	Technique	Challenges
[19]	Static	Inferring source code deviations	Statistics (z-stats)	Requires templates
[24]	Static	Inferring call sequences inconsistency	Frequent Item-Set mining	Too many reported violations, precision unknown
[25]	Dynamic	Inferring control flow abnormality	Hypothesis Testing	Scalability
[21]	Dynamic	Inferring invariant violations	Invariants Confidence	Scalability
[18]	Hybrid	Inferring invariant violations	Testing	Scalability
[27]	Dynamic	Inferring error handling policy	Statistics	Coverage
[28]	Dynamic	Inferring programme properties	Heuristics	Requires heuristics for interesting properties
[17]	Dynamic	Comparing pass and failure runs	Finite State Machine	Scalability

Table 9. Summary of Related Work

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 software products.

References

[1] Dell DVD Store. http://linux.dell.com/dvdstore/.
[2] iBATIS JPetStore. http://sourceforge.net/projects/ibatisjpetstore/.
[3] Implementing the Microsoft .NET Pet Shop using Java. http://www.clintonbegin.com/JPetStore-1-2-0.pdf.
[4] Jakarta Struts. http://struts.apache.org/.
[5] Java Pet Store. http://java.sun.com/developer/releases/petstore/.
[6] MySQL Wins Prestigious International Database Contest. http://www.mysql.com/news-and-events/press-release/release_2006_35.html.
[7] Summary of Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/.
[8] WebLoad. http://www.webload.org/.
[9] Willem Visser's Research. http://www.visserhome.com/willem/.
[10] J. Andrews. Testing using log file analysis: Tools, methods, and issues. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, 1998.
[11] A. Avritzer and B. Larson. Load testing software using deterministic state testing. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, 1993.
[12] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, 1994.
[13] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.*, 21(9), 1995.
[14] M. S. Bayan and J. W. Cangussu. Automatic stress and load testing for embedded systems. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, 2006.
[15] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, March 1984.
[16] L. C. Briand, Y. Labiche, and M. Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2), 2006.
[17] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *SOQUA '07: Fourth international workshop on Software quality assurance*, 2007.
[18] C. Csallner and Y. Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, 2006.
[19] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.
[20] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on uml models. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.
[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.
[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An Automated Approach for Abstracting Execution Logs to Execution Events, 2008. To appear at September/October 2008 issue of the Journal on Software Maintenance and Evolution: Research and Practice (JSME).
[23] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, 2003.
[24] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 2005.
[25] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *SDM*, 2006.
[26] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Los Alamitos, CA, USA, 1994.
[27] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.
[28] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.
[29] J. Zhang and S. C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Softw. Pract. Exper.*, 32(15), 2002.