An Industrial Case Study of Customizing Operational Profiles Using Log Compression

Ahmed E. Hassan
School Of Computing
Queen's University
Kingston, Canada
ahmed@cs.queensu.ca

Daryl J. Martin, Parminder Flora,
Paul Mansfield, Dave Dietz
Research In Motion (RIM)
Waterloo, Canada

ABSTRACT

Large customers commonly request on-site capacity testing before upgrading to a new version of a mission critical telecom application. Customers fear that the new version cannot handle their current workload. These on-site engagements are costly and time consuming. These engagements prolong the upgrade cycle for products and reduce the revenue stream of rapidly growing companies.

We present an industrial case study for a lightweight simple approach for customizing the operational profile for a particular deployment. The approach flags sequences of repeated events out of millions of events in execution logs. A performance engineer can identify noteworthy usage scenarios using these flagged sequences. The identified scenarios are used to customize the operational profile. Using a customized profile for performance testing alleviates customer's concerns about the performance of a new version of an application, and results in more realistic performance and reliability estimates. The simplicity of our approach ensures that customers can easily grasp the results of our analysis over other more complex analysis approaches. We demonstrate the feasibility and applicability of our approach by customizing the operational profile of an enterprise telecom application.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*; D.2.8 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Performance, Reliability ,Verification

Keywords

Capacity planning, Execution logs, Load testing, Operational profile

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

Modern telecom products, such as VOIP and wireless email systems like the Blackberry email service, have a shorter release cycle and a larger and more varied customer base than traditional telecom products. These new realities bring many challenges for software development activities in the telecom sector. Companies and their support staff must interface with hundreds or thousands of customers instead of a few large carriers.

Companies commonly limit their testing efforts to a few standard configurations and benchmark usage patterns, instead of adopting more comprehensive testing efforts. However, standardized benchmark workloads rarely capture the variances across deployments [2, 17]. Weyuker and Vokolos from AT&T note that a large number of post-release problems for telecom systems are attributed to performance degradation and the failure of applications in handling their field workload instead of feature bugs [20]. Performance testing and capacity planning activities are rapidly becoming an integral part of any deployment of mission critical applications worldwide [12].

The telecom sector has used operational profiles with great success, to ensure that testing efforts offer a realistic view of the expected performance of a particular application in the field [14]. However, the large variances between modern deployments lead to mismatches between the tested operational profile (*expected usage*) and the actual usage scenarios. These mismatches lead to customer dissatisfaction with the reliability and scalability of telecom applications in large deployments with thousands of users. Such large deployments usually reside in the data centers of the largest customers of a company. The satisfaction of these high value customers is a major concern and a high priority. Many large customers request on-site capacity testing before upgrading to a new version of a mission critical telecom applications. Customers fear that the new version cannot handle their current workload. These on-site capacity engagements require the creation of a customized operational profile, a lengthy and labor intensive task [19]. In short, these engagements prolong the upgrade cycle for products and reduce the revenue stream of rapidly growing companies.

We developed a lightweight approach to customize an operational profile for large deployments. The approach reduces the time needed for creating customized operational profiles from days to hours. The approach makes use of execution logs which are readily available and which are representative of the field usage of an application. Instead of recovering all usage scenario of an application, our approach

uncovers a limited set of high workload scenarios. Such scenarios exert a high workload on an application and are usually critical to the performance of an application due to the high event repetition in them. For example, a "Check Email" scenario would contain the following logged events in the execution log of a wireless email router: `connect_server`, `read_server_msg`, `get_new_email`, `disconnect_from_server`. Another instance of the "Check Email" scenario may have the `get_new_email` event repeated a 100 times, indicating that a user has received one hundred emails at once. Our approach would flag such large sequence of repeated `get_new_email` events. A performance engineer would then determine the scenario corresponding to the repeated events, consult the operational profile, and add the scenario to it, if needed. This process would result in a customized profile.

In our experience, the approach identifies many large and high workload scenarios within a few hours without having to dispatch performance engineers to the site of a deployment. The simplicity of the approach ensures that customers can easily grasp the results of our analysis over other more complex analysis approaches. The gained understanding through our approach is valuable in ensuring that high value customers with large deployments are satisfied with the performance of an application.

1.1 Organization of the Paper

The paper is organized as follows. Section 2 presents background information needed to understand the motivation for our approach: operational profiles and redundancy in logs. Section 3 presents our approach. We elaborate on our use of redundancy in execution logs to reveal high workload scenarios. Section 4 showcases our proposed approach on the execution logs of a large deployment of a multi-threaded multi-user telecom application that is deployed in thousands of enterprises worldwide. Section 5 explores our results and discusses the limitation of our approach. Section 6 presents related work. Finally Section 7 concludes the paper and briefly outlines future work.

2. BACKGROUND

2.1 Operational Profile

An operational profile captures the most common usage scenarios (e.g., sending email, receiving email, and deleting email) of a particular application (e.g. wireless email client) and their rate of occurrence (e.g. frequency of sending email) [14]. For example, the profile of an email client would contain information such as: {send email (60%), receive email (30%), delete email (10%)}.

Musa informally measures the cost-to-benefit ratio of creating a profile to be as high as 1 to 10 [15]. Many success stories of using operational profiles for guiding software testing are reported in literature. AT&T reports that operational-profile-driven testing helped significantly reduce the number of reported customer problems. Similarly, HP reported that operational profiles helped reduce the time and cost for testing a multiprocessor operation system by as much as 50% [14, 15].

Deriving an operational profile is time consuming and challenging. Creating a profile requires the participation of many system experts and the analysis of large amounts of data across various deployments of an application [19]. Musa reports that creating a profile for a typical 100 KLOC application, that has been developed by 10 developers over 18 months, requires around one staff month [15]. Field data should be collected for a significant period of time (two to twelve months) [1]. Performing extensive data collection on actual deployments is usually not feasible, due to the needed on-site resources and time commitment. Analyzing the collected data requires a good understanding of the features and usage scenarios of a software system. Moreover, examining data collected from large deployments is challenging due to the large amount of data which should be analyzed. For example, many telecom applications produce several hundred megabytes of execution logs each day.

2.2 Redundancy in Logs

Logs record the execution of an application and could be used to build operational profiles. Tracing logs are generated by instrumenting or monitoring of an application using a variety of technologies such as the Java Virtual Machine Profiler Interface (JVMPI). In contrast, execution logs are produced without any special instrumentation. Execution logs are primarily used for issue resolution and for remote debugging. For instance, customer support agents use execution logs to remotely determine field problems. There is an abundance of execution logs in the field, whereas tracing logs must be produced for specific scenarios.

Both execution and tracing logs have similar characteristics. They both contain valuable information which could benefit program understanding activities [3]. The main differences between both types of logs are: (1) Tracing logs are more detailed. (2) More time and effort is needed to generate tracing logs. Tracing logs are more detailed since they track the execution of an application at a much lower level of detail (e.g., function call and return events). It is undesirable to produce tracing logs while a telecom application is running in the field. Instead the application needs to be taken offline due to the large resource requirements for producing such traces. Nevertheless, redundancy in both types of logs is a widely recognized attribute [6]. Redundancy is usually due to either the repetition of sequences of events (e.g. syncing all messages to a wireless email device) or due to repeated scenarios (e.g. sending multiple emails to a wireless device).

Redundancy is considered as an undesirable feature of logs since it hampers program understanding. Researchers have proposed approaches to automatically detect, and remove or abstract redundancies in execution logs [7, 8, 9, 10, 16, 18]. Many of these approaches summarize or visualize log information to assist analysts in investigating problems.

For the approach presented in this paper, we use execution logs. In contrast to previous approaches, we do not execute a particular set of test cases on an application, since we recognize that shutting off a mission critical system to execute a few test scenarios at a customer's site is usually not feasible nor realistic. Moreover, we see redundancy in execution logs as a desirable feature which our approach exploits to uncover the characteristics of a workload. Finally we analyze logs that are much larger in size in comparison to other approaches, since we deal with actual execution logs instead of logs for specific scenarios. Our approach is designed to process and study log files which contain millions of events in contrast to previous approaches which deal with logs containing less than 100K events [7].

3. OUR LOG COMPRESSION APPROACH

Creating an operational profile is a labor and time consuming process. Menascé et al. [13] suggest to begin by gathering a list of frequent usage scenarios of an application. For each identified scenario, a scenario signature is created. The scenario signature is a regular expression pattern which lists the sequence of function calls or events which occur in that scenario. For example, the signature for the "Check Email" scenario may look as follows: `connect_to_server`, `read_server_msg`, `get_new_email?`, `disconnect_from_server`. The signature indicates that the `get_new_email?` function is an optional function which is executed based on the server's reply. The execution log records the execution of most of the functions in an application. For instance, a "Send Email" scenario is implemented by calling several functions such as `connect_to_outgoing_server`, `send_message_to_server`, `open_outbox`, `store_sent_message`, and `close_outbox`. Analyzing the execution logs using these scenarios signatures, one can determine how frequently do users "Check Email" or "Send Email". This process must be repeated for every identified usage scenario for a particular application. A good understanding of the different usage scenarios and their signatures is needed.

A good knowledge of all the scenarios and their signatures is not realistic given that many software systems are not well documented and even if they are documented the documentations are rarely kept up to date [11]. At first glance it may appear that determining the various usage scenarios is a simple task. But one must consider that different alternatives for a simple scenario are considered as different scenarios. For instance, the "send email" scenario may have various alternatives such as "send email to invalid account" or "send email to self". The resource requirements for every scenario varies and should be considered. For example, sending an email to an invalid account would require less resources since the email client won't contact the server.

Whereas traditional techniques for creating an operational profile focus on capturing frequently occurring scenarios, we are interested in uncovering large scenarios which contain several repeated events in them. Although the scenarios may not occur often, the high repetition of events in them is likely to affect the performance of a software application. These scenarios are often missing from basic operational profiles since they do not usually occur is most deployments of an application.

To cope with the large amount of data (millions of events) present in the execution logs of telecom applications, our approach converts the execution log into a *log signal*. The log signal is generated by breaking the data in the execution log into equal sized periods, compressing each period, and measuring the compression ratio of the period. Periods with a large number of repeated sequences of events can then be easily identified since they would compress very well compared to other sequences. A performance engineer examines every identified sequence of repeated events (less than 1% of the log file) and updates the operational profile with new scenarios. For each newly created scenario, a filtering rule is created. The filtering rules for the new scenarios are applied to the execution log to remove the events for that scenario from the log. Once the events are removed, a new log signal is created and the performance engineer examines the signal again to identify other interesting sequences. After a few iterations of this process, the performance engineer should be able to recognize most of the high load scenarios peculiar to the studied deployment. These scenarios are used to augment the operational profile in order to better match deployment characteristics. This process takes 2-3 hours in practice.

We note that our approach can be used on any type of data produced by a software application such as disk access, database operations, or memory usage, as long as a scenario signature could be described using patterns in the logged data. For example, the signature for the "Check Email" scenario can be described as one large disk write, followed by several repeated small disk writes, and followed by one large disk write. For this paper, we use execution logs since they are readily available for most applications, and no additional effort, like the instrumentation of the application, is required to collect them. Moreover creating scenario signatures using log events is much simpler and straightforward in comparison to using disk or memory access patterns.

3.1 Approach Details

We now present our approach. Given a log file, a performance engineer picks an analysis window (e.g., 400 events). The analysis window is used to break the log file into periods of equal size. Each analysis period is compressed and its compression ratio is recorded and visualized. Any type of compression algorithm could be used. For our purposes, we use the Gzip compression algorithm since the `gzip` tool is readily available on all platforms. We define compression ratio (CR) for a period P as:

$$CR(P) = 1 - \frac{size\ of\ P\ after\ compression}{size\ of\ P\ before\ compression}$$

The compression ratio will be high for periods with high redundancy in event sequences and low for periods with low redundancy. Given the two strings "`BAAAAAAAC`" and "`ABCXEWT`" both representing a sequence of events. Each character in the sequence represents an event. For example, "A":Open_Inbox and "B": Close_Inbox are events. The sequence "`BAAAAAAAC`" contains a large amount of redundancy in it and will compress very well. On the other hand the other sequence "`ABCXEWT`" contains no redundancy and will compress poorly.

Instead of using the proposed compression ratio technique, one could count the number of consecutive events in the execution log and highlight sequences of repeated events. Such a counting technique can recognize simple event sequences constituting of the repetition of a single event (e.g. "`AAAAAAA`"). However, a counting technique cannot easily recognize more complex sequences of repeated events. Such complex sequences are found often in the execution logs of applications. Examples of complex sequences:

1. **Event groups ("`ABCABCABC`"):** We often find a group of events, e.g."`ABC`", repeated instead of a single event. An example of a group of events would be: Open Inbox, Verify Password, and Contact Server.
2. **Approximate repetition ("`ABCXDABCTWABC`"):** When searching for groups of repeated events, we commonly find repeated groups of events that are interrupted with other events. The interruption by other events (e.g. "`X,D,T,W`") occurs for various reasons. For example, due to the multi-threaded nature of many telecom applications, the log of sequences of events may be interrupted by events produced by other threads. Or,

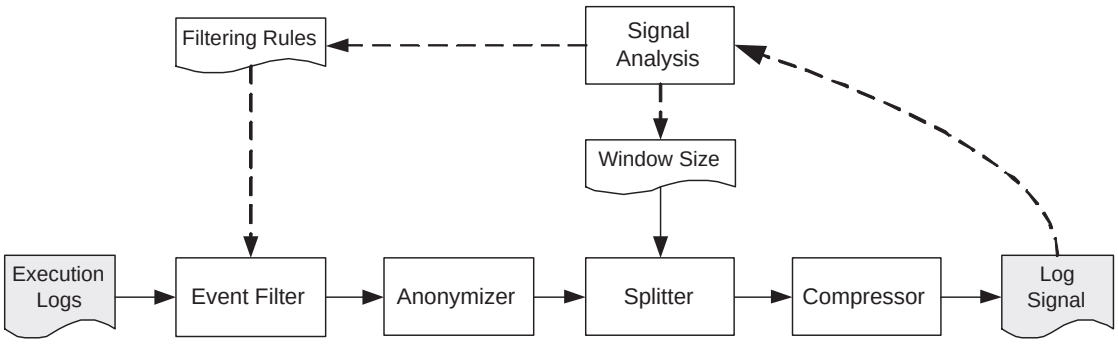

Figure 1: Our Approach for Studying Repeated Sequences of Events in Execution Logs

repeated groups of events may differ slightly from one instance to another. For example, when receiving a 100 emails a few of the emails may have a high priority flag set on them. This flag may cause the appearance of a different event in the log.

The compression ratio technique recognizes such complex sequences since the redundancy within these sequences results in high compression ratios even if the repeated groups of events are not identical.

Figure 1 presents an overview of our iterative approach. The solid arrows indicate the basic flow of our approach. The dotted lines indicate the steps performed at the end of each iteration of the approach. Based on the Signal Analysis step, a performance engineer determines whether a new iteration is needed or the analysis should terminate. We explain below the steps in our approach:

3.1.1 Anonymizer

The Anonymizer step is responsible for removing all references to specific accounts and user data. For example, execution logs such as `10:15AM-Handle Email for User='User1'` and `11:40PM-Handle Email for User='User22'` are converted to `-Handle Email for User=`. The anonymization of the log lines converts the log file into an event listing. This conversion ensures that similar events have similar compression ratios independent of the actual data outputted in the log line during the execution of a specific instance of an event. In order to ensure the generality of our approach, we use several general heuristics coded in a Perl script to anonymize log lines. For example, we remove all characters surrounded by single or double quotes. We also remove all numbers in a log line. Both heuristics assume that these values are likely parameters of an event and should be removed.

3.1.2 Splitter

The Splitter step breaks the log events into periods of equal size using a defined window size. A performance engineer would start with a large window size and keep on reducing the window size as the analysis progresses. In our experiments, we started with windows of size 400 events (larger sizes are possible) and continued reducing the size of the window till it reached 50 events. Larger window sizes will result in the identification of larger sequences of repeated events.

3.1.3 Compressor

The Compressor step simply compresses the events in each analysis period, measures the compression ratio of these events, and produces a graph of the log signal.

3.1.4 Signal Analysis

The Signal Analysis step is a manual step where the performance engineer examines the produced graph of the log signal and studies the spikes in the graph. We chose to examine the top ten spikes whose value is four times the average, other values may be used. For each examined spike, the performance engineer consults the corresponding log events and log lines. Since the spike is due to redundancy in the log, the log lines causing the spike can be easily recognized when the log is inspected. Using a regular expression, the performance engineer manually specifies *Filtering Rules* for each identified scenario. These rules are equivalent to a scenario signature. The name of a scenario is derived by consulting a system expert or based on the events in the scenario.

Based on our experience using this approach, it is preferable to focus on one spike at a time. Once the filtering rule corresponding to that spike is created, then the filtering rule is applied to the log file and the log signal is re-examined. By removing the events corresponding to the just identified spike, other spikes, which are due to the just removed scenario, are removed as well. This technique reduces the number of spikes which must be examined by hand.

To decide whether to perform an additional iteration or to stop the process, we used the following heuristics during our analysis:

- If no spikes are present in the log signal, then the window size is reduced, usually by half. By reducing the window size then the log signal is likely to show additional spikes. These new spikes are due to the existence of shorter sequences of repeated events. The engineer can keep on reducing the window size until the compression ratio is negative. The compression ratio becomes negative once the overhead of the compression is too high (i.e., the compressed events take more space that the uncompressed events).
- If there are too many spikes in the signal, the engineer can decide to identify 10-20 filtering rules and stop. The decision to continue further or to stop is based on the time allocated to the customization of the operational profile. In our experience, the first ten filtering rules are very valuable in highlighting large and important peculiarities of a particular deployment.

Using the filtering rules, we can calculate the frequency of occurrence of their corresponding scenarios by examining the initial log file. The scenarios along with their calculated frequencies are used to update the operation profile. The simplicity of the approach permits a customer to be more involved in the analysis. A customer looking at the signal may decide to have the engineer continue the spike analysis for a longer period of time.

3.1.5 Event Filter

The Event Filter step applies the filtering rules identified during the Signal Analysis step. Each filter is specified as a regular expression. The expression specifies the starting, ending, and repetitive events in a particular scenario. For example, a filter to remove the execution logs corresponding to the reception of a batch of email messages (i.e. "receive email" scenario), would be expressed as follows: A starting event of the form "`<TIME>-Handle Email for User='##USER##'`" and an ending event of the form "`<TIME>-Handled Email for User='##USER##'`". To remove email reception scenarios with more than 10 repetitive events, the repetitive event is expressed as the occurrence of the following expression more than 10 times: "`<TIME>-Handle Email Message for User='##USER##', MsgId=`". The `##USER##` expression back-references earlier matches in the filter to ensure that only the appropriate log lines, matching the correct user, are removed. Since the filtering rules reference particular entries in the data (e.g., the user name), the filtering should occur before the Anonymizer step where such information is removed from the logs.

4. INDUSTRIAL CASE STUDY

To demonstrate the applicability of our approach, we used it to study the execution logs of five large deployment of a multi-threaded telecom application. The studied application processes data for hundreds of users within large enterprises. Due to the critical nature of the application, the presentation of our analysis does not name the specific events nor scenarios identified by our approach instead each event and scenarios is assigned a unique identifier.

In our case study, we want to customize an operational profile for a particular deployment. We first used a current operational profile to conduct a load test on the application in a lab setting. The load test contained the same number of users as the deployment but was conducted for just 6 hours. The generated execution log is stored for further analysis. We then acquired the execution log for the deployment that we wanted to study. The deployment log is for a full day. We tested our approach using the deployment log. In order to demonstrate the benefit of our approach, we compared the content of two log files (the load log and deployment log) and uncovered missing scenarios in the load log. These missing scenarios demonstrate the value and importance of customizing the operational profile.

Figure 2 shows the log signal produced by applying our approach on the load (black) and deployment (grey) logs using a window of 400 events. The load log is for 500 users, it contain 1,152,049 log lines and it is 137MB in size. The deployment log is for a 24 hour period, it contains 953,802 events and it is 110MB in size. We note that the deployment log contains less events than the load log even though the deployment log captures events over a longer period (24 hours vs. 6 hours). The variance in size indicates that the

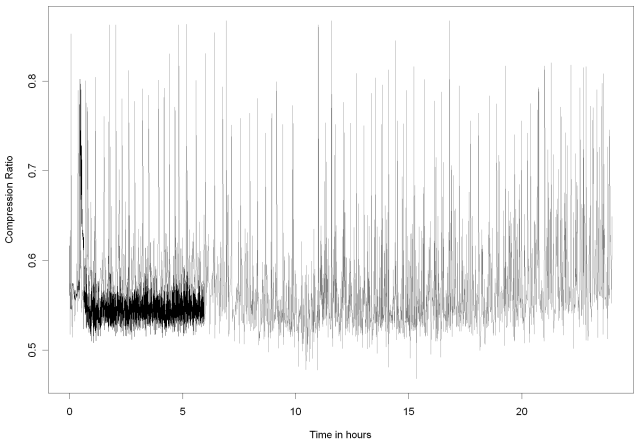

Figure 2: Deployment Logs in Grey, Load Logs in Black

operational profile used for load testing is exerting a much higher workload than the actual deployment workload. After closer analysis of the graphs shown in Figures 2 and 3, we notice the following:

1. The deployment log has a slightly higher compression ratio in contrast to the load log. The load signal, in Figure 2, has a lower range of values in comparison to the deployment signal. The higher compression ratio indicates that the deployment log has more repetitive sequences of events in contrast to the load logs. The operational profile should be updated since the load logs are generated due to load testing of the application using the operational profile.
2. The deployment signal exhibits spikes in compression ratio. These spikes indicate the presence of highly repetitive sequences. These sequences likely correspond to high load usage scenarios. These scenarios should be integrated into the operational profile to make the results of load testing efforts more realistic.
3. The spikes at the beginning of both log signals, in Figure 2, are due to the initialization of the studied application. During initialization similar events are executed for each user (e.g. the resetting of each user account). Once these similar events are anonymized, they compress very well.
4. The detailed signal breakdown, shown in Figure 3, indicates that the deployment signal has a higher median value in comparison to the load signal. The median value is marked with a red vertical bar in both graphs shown in Figure 3.

Using our approach, a performance engineer can examine the spikes in Figure 2 and create *Filtering Rules*. These rules remove all the log lines which correspond to the identified spikes. Once the lines are removed, other spikes due to the recurrence of the same scenario will be removed. This process (finding a spike, examining the logs for its corresponding period, expressing the repetition as a scenario signature, and writing a filtering rule to remove such a scenario from the logs) is repeated several times. The final filtered signal is shown in Figure 4. In our experiment, we started

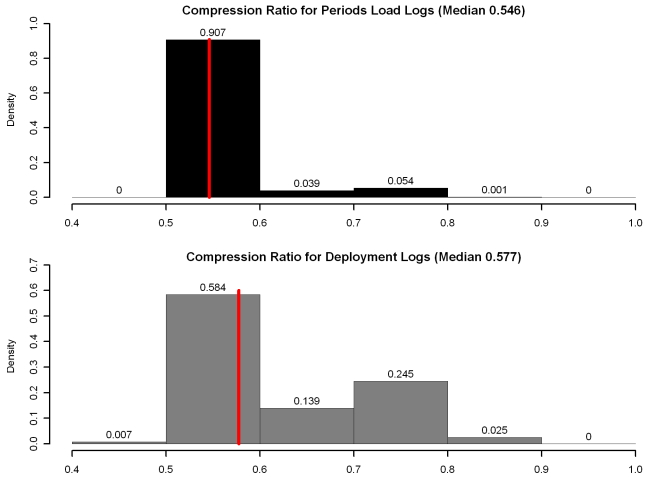

Figure 3: Histogram for the Load (Top-Black) and Deployment (Bottom-Grey) Log Signals

from a benchmark operational profile which was considered to be representative of most field deployments. We repeated the aforementioned process seven times and we recognized seven usage scenarios which cause high redundancy in the deployment logs. For each of the identified scenarios, the performance engineer must determine whether:

1. The identified scenario is valid. For valid scenarios, the operational profile should be updated.
2. The identified scenario is invalid. For invalid scenarios, the engineer needs to report these problematic cases to the development team. The development team would in turn decide if these cases are due to bugs or misconfiguration of the application.

4.1 Discovered Scenarios

Table 1 summarizes statistics for the seven discovered scenarios in the deployment log. The table also highlights in grey the statistics for these seven scenarios in the load testing log. From the table, we notice that:

1. The RE, OT and CI scenarios do not exist in the load testing workload. They are missing from the operational profile and should be added.
2. The other four scenarios have similar minimum and median number of events in the load and deployment logs. However, the maximum number of events varies considerably between both log files. It appears that the load testing workload assumes a gaussian like distribution whereas the deployment log suggests that a *zipf* [21] distribution of events may be more appropriate. The operational profile should be adjusted to better simulate the frequency of occurrence of such scenarios. This finding was the most interesting for the development team since they did not expect such high repetition levels for some of the scenarios. For example, the RE scenario has instances with over thirty three thousand repeated sequences.
3. The GF scenario in the load testing log occurs at a much higher frequency than in the deployment workload. It may be desirable to reduce the frequency of the GF scenario in the operational profile.

Based on our aforementioned findings, we can define the following filtering rules:

1. We filter out all instances of the RE, OT and CI scenarios since they have a large number of repeated event sequences in each instance of these scenarios. The large number of repeated sequences causes spikes in the compression ratio of the generated log signal.
2. For all scenarios other than RE, OT and CI, we filter out any instance of the scenario with more than the median number of events in it.

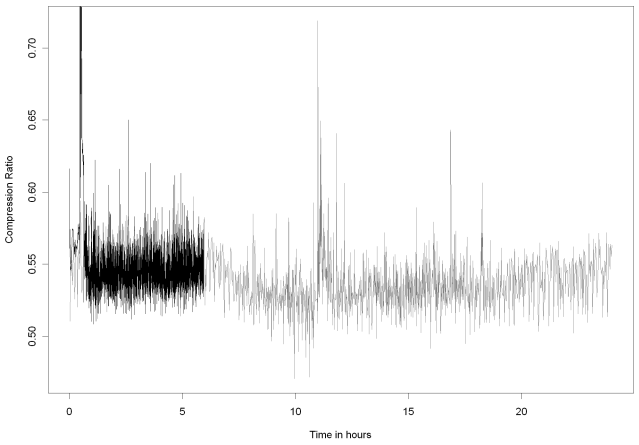

Figure 4: Final Deployment (Grey) and Load (Black) Signal (400 events-window)

Figure 4 shows the filtered log signal for the studied log. The Figure also shows for reference the log signal for the load testing log. All spikes have been removed except for a spike around 11AM. A closer investigation of the spike at 11AM reveals that a process monitoring the application had detected a possible deadlock situation and restarted the application automatically. The redundancy in the log file is due to the application performing similar activities for each user at startup.

5. DISCUSSION OF RESULTS AND LIMITATIONS

Picking the appropriate window size. Our approach depends on picking an appropriate window size. As we reduce the size of the window, new spikes are likely to show up. These spikes exist for larger analysis windows however they are not as visible since the larger windows contain a large amount of non-repeated events which in turn reduces the compression ratio. Figure 5 shows the effects of varying the window size (200,100 and 10) for the filtered signal for 400 events window. The signal for 400 events window is shown at the top of Figure 5 for convenience. We note that the restart event is visible at window sizes equal to 400, 200 and 100. The restart event is marked on the Figure using a vertical grey box. We highlight a few spikes using grey circles in Figure 5. The signals for the 200 and 400 events windows have the same number of large spikes. However, for the signal for the 100 events window, additional large

Scenario Name	Frequency	Filtered (%)	Number of Events In A Scenario		
			Min	Median	Max
MA	7,837	1,856(23.68%)	1	1	1,987
	8,079		1	1	3
RE	27	27(100%)	33	6,244	33,575
			-	-	-
MR	7,837	2,247(28.67%)	1	1	1,253
	8,050		1	1	57
GF	27	545(6.95%)	8	16	2,126
	7,569		16	16	17
CA	7,405	588(7.94%)	2	2	707
	7,977		2	3	10
OT	1,788	842(47.08%)	2	3	343
			-	-	-
CI	5	5(100%)	41	79	572
			-	-	-

Table 1: Statistics for the Seven Scenarios in the Deployment and Load (in grey) Logs

spikes are more prominent. Looking back at the 200 and 400 signals, we note that a few of the large spikes in the 100 events signal existed but were not as prominent. Using our approach on this log file, an analyst would reduce the analysis window to 200 events and discover that there are no large spikes. He or she would then reduce the window to 100 events and discover a few large spikes. The new spikes are studied and additional scenarios are added to the operational profile. This process continues until the window size is too small or the engineer runs out of allocated time.

In practice, the time needed to find an appropriate window size was not a major concern since the approach already reduces the time needed for profile customization from days to hours. We have explored the use of a brute force technique to determine the optimal window size. The optimal window size is the one that results in the largest number of high spikes (i.e., four times above average) with no negative values in the signal. We continuously adjusted the window size and recorded the number of observed spikes for a log file. Our analysis shows that the number of spikes is maximized at window sizes of 400-550 events. This optimal window size is application and deployment specific. Once the optimal size is found, it can be reused when analyzing log files in the same deployment. Others interested in using our approach can perform similar analysis on their log files.

Approach in practice. The presented approach is semi-automated. The semi-automated nature of the approach is very valuable in speeding on-site capacity engagements. The approach directs the attention of a performance engineer to a few spots, less than 1% of the lines in millions of log events. These spots are examined to identify scenarios. The identified scenarios are expressed as filtering rules. The performance engineer has to perform this task a limited number of times. In our experiment, this manual step was performed seven times on a file comprising a full 24 hour execution of a large deployment. This step required overall around 30 minutes of work and around 2 hours of automated processing. Current methods used on site would require around 4-5 days of analysis to create a customized profile. An engineer may need to repeat this analysis several times when examining log files for other deployments. In later experiments on the logs for four other deployments, we recognized another three additional filtering rules within a total of one hour of analysis. We are currently investigating techniques to automate the Signal Analysis step by using text analysis techniques. These techniques can propose filtering rules with minimal manual intervention in a wizard like fashion.

In practice, we observed that our approach located an unexpected scenario (the restart of the analyzed application). The proposed approach helps performance engineers recognize noteworthy normal and abnormal large scenarios with repeated sequences of events.

Measuring approach accuracy. To measure the accuracy of our approach, we measured its precision. Precision measures the number of flagged events that resulted in updating the operational profile versus all flagged events. An approach with low precision is not desirable since it would waste an engineer's time. In our first experiment, we examined nine flagged event sequences (i.e. spikes) and seven of them resulted in updating the operational profile. The other two spikes were due to the application startup events. Therefore the precision of the approach is 77%. We examined the logs for another four deployments and the overall average precision of our approach is 80%. We cannot measure the recall of our approach due to the very large number of events in each log file (around one million events per file).

Limitations. For very small window sizes (see window size = 10 at bottom of Figure 5), the overhead of compressing the small number of events is too large. This high overhead results in negative compression ratios because of the size of the compressed period being larger than its original size. Currently our approach cannot recognize small scenarios (i.e. scenarios with a small group of repeated events). Some of these small scenarios may affect the overall performance of an application and should be tested. We are currently exploring data mining techniques to uncover such small scenarios automatically.

The anonymization step in our approach hinders the recovery of concurrent interactions. For example, two log lines, which indicate two received email messages, may refer to a single user receiving two emails or two users each receiving a single email. In our experiments these interactions showed up at application startup. During startup, many users are initialized concurrently. Following the anonymization step the data simply indicates many initializations without reference to the fact that these initializations happened to dif-

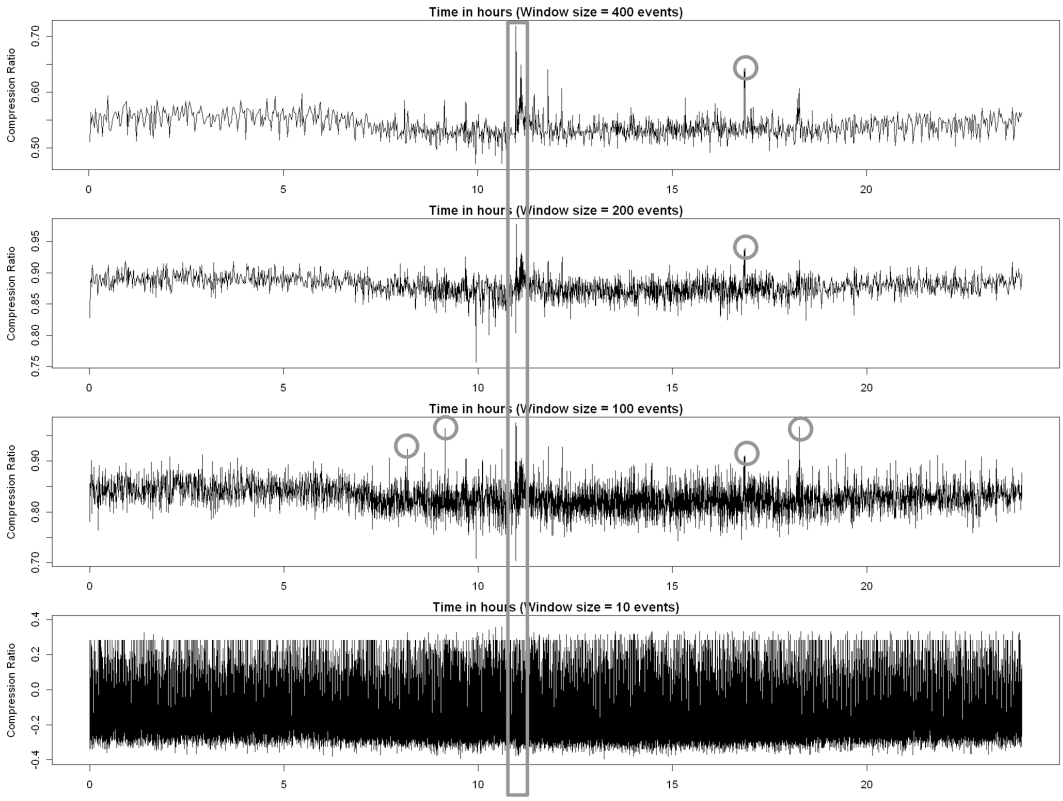

Figure 5: Changing the Window Size for the Deployment Log (Black)

ferent users. We don't consider this as a limitation of our approach since we are interesting in finding large general repetitive sequences of events. Manual analysis of the events and logs will highlight the rationale for these concurrent interactions. Timing during load testing could be adjusted to generate such interactions.

6. RELATED WORK

Traditional approaches to derive operational profiles are semi-automated and time consuming, they depend on user surveys or data mining techniques (such as frequent item set or clustering algorithms) [13, 14, 15]. In contrast, our approach requires 2-3 hours and uncovers large variations between a derived profile and the actual usage scenarios. The approach flags less than 1% of the events in large log files, containing around one million events.

The use of a compression algorithm simplifies the implementation and adoption of our approach in an industrial setting since no specialized algorithms are required. Moreover, our approach recognizes high load scenarios that are intermixed with other scenarios due to threading issues (i.e. outputs from different threads) or simple variations in the repetition of a scenario. Advanced enhancement to data mining algorithms, such as gapped frequent sets, may be adopted to detect intermixing of scenarios. However our attempts to use such algorithms on the logs have resulted in large number of false positives due to the algorithms recognizing repetitive patterns across usage scenarios not within the same scenario.

Work in program understanding (e.g., [5]) maps particular features to their source code implementation using technique such as concept analysis. Such approaches require the instrumentation of an application and the execution of a limited number of tests. The results of these tests are presented in a large concept lattice. The lattice is manually studied by a domain expert to locate the functions implementing each feature. Our approach does not require any type of preprocessing or execution of specific tests, instead we use logs that are readily available.

The work most closely related to our approach is work on network intrusion detection (e.g., [4]). Compression techniques are used to detect abnormal behavior patterns. For example, an attacker attempting to locate an open network port would probe every port, in contrast most users are likely to connect to a limited number of network ports. Therefore, during an attack the variation in the port numbers will be too high and would results in a low compression ratio (in contrast we are interested in high compression ratio). Areas with low compression ratios are flagged and shown to a network security analyst for further investigation.

7. CONCLUSION

An accurate operational profile is needed to ensure that an application can handle the needs of users, otherwise users will be disappointed with the performance of an application. We present an approach to customize operational profiles. The approach analyzes the execution logs of deployed applications and highlights noteworthy sequences of events. The highlighted sequences represent less than 1% of a large execution log. Performance engineers can identify perfor-

mance critical usage scenarios by studying the highlighted sequences. The scenarios are added to the operational profile of the application. The customized profile is used during load testing of the application. The results of performance tests using the customized profile are more representative of the performance of an application for a particular deployment.

The presented approach uses execution log files which are commonly and widely available for most telecom applications. The approach requires no code changes nor does it require instrumenting the analyzed application. The approach can be applied immediately on an application since the logs of many applications are readily available and are usually archived. As no code changes, instrumentation, or data collection is needed, the proposed approach can be easily adopted by companies and does not depend on a particular software version, build, or platform. Nevertheless, the approach has limitations. It is a semi-automated approach and it cannot recognize small execution scenarios. In future work we plan on addressing these two limitations. Moreover, we plan on applying the approach on other software applications so we can generalize our findings across different types of software applications.

Acknowledgments

We are grateful to Research In Motion (RIM) for providing access to the execution logs of the large telecom applications used in 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 software products.

8. REFERENCES

[1] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *International Workshop on Software and Performance*, pages 17–24, 2002.

[2] L. Bertolotti and M. Cazarossa. Models of mail server workloads. *Performance Evaluation*, 46(2):65–76, 2001.

[3] W. Citrin, A. Cockburn, J. von Känel, and R. Hauser. Using formalized temporal message-flow diagrams. *Software Practice and Experience*, 25(12):1367–1401, 1995.

[4] R. Eimann, U. Speidel, N. Brownlee, and J. Yang. Network Event Detection with T-Entropy. CDMTCS-266, University of Auckland, May 2005.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):195–209, Mar. 2003.

[6] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55, Toronto, Canada, Oct 2004.

[7] A. Hamou-Lhadj and T. Lethbridge. Measuring various properties of execution traces to help build better trace analysis tools. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05) - Volume 00*, pages 559–568, Shanghai, China, Jun 2005.

[8] D. F. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. pages 56–, Amsterdam, The Netherlands, Oct 1997.

[9] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. Sced: A tool for dynamic modeling of object systems. University of Tampere, Dept. of Computer Science, Report A-1996-4, 1996.

[10] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings of the 22nd International Conference on Software Maintenance*, Philadelphia, PA, Sep 2006.

[11] L. D. Landis, P. M. Hyland, A. L. Gilbert, and A. J. Fine. Documentation in a software maintenance environment. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 66–73, Oct. 1998.

[12] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall PTR, 2004.

[13] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of E-commerce sites. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 119–128, Denver, Colorado, United States, Nov 1999.

[14] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

[15] J. D. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin. The operational profile. pages 167–216, 1996.

[16] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, May 2001.

[17] J. M. Voas. Quality time - will the real operational profile please stand up? *IEEE Software*, 17(2), 2000.

[18] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–283, Vancouver, British Columbia, Canada, Oct 1998.

[19] E. J. Weyuker and A. Avritzer. A metric for predicting the performance of an application under a growing workload. *IBM Systems Journal*, 41(1):45–54, 2002.

[20] 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.

[21] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.