NAGAPPAN, MEIYAPPAN. A Framework for Analyzing Software System Log Files. (Under the direction of Mladen A. Vouk.)

Ability to perform large scale data analytics is fueling collection of a variety of data streams from complex production systems, like the provenance of data, performance statistics, and user statistics, in order to analyze them for specific purposes. Log files are a typical example of such a data stream. Logs are often a record of the execution activities in production systems where the data recorded is intentionally chosen by the developer as useful information. Analyzing log files is an important activity in software engineering, and now in cloud engineering. The data from the log files can be analyzed to guide product related decisions. For example logs can be used to find ordinary and security related problems, define operational profiles, and even pro-actively prevent issues (in support of fault tolerance).

It is the premise of this work that current log analysis methods are too ad hoc and do not scale well enough to be effective in the domain of large logs (such as those we might expect in a computational cloud system). The more complex the system, the more complex and voluminous its logs are. In this dissertation we investigate, identify and develop components needed for an adaptable end-to-end framework for the analysis of logs. The framework needs to take into consideration that different users look for different kinds of information in the same log files. Required are adaptable techniques and algorithms for efficient and accurate log data collection, log abstraction and log transformations. The techniques or algorithms that are used in each component of the framework will vary according to the application, its logging mechanisms and the information that the stake holder needs to make decisions.

Some of the questions of interest are:
What information is required by the developers to make the decisions they want to, and which analysis technique can build that information? To what form can the logs be transformed, so that we can get this information (required in the previous question) efficiently and accurately? How to abstract the semi structured log lines to log events so that the above transformations can be made efficiently and accurately? How can we collect a rich set of data in the log files to begin with so that we can provide better results to the developers?

We discuss a concrete instantiation of the framework (i.e. specific solutions for each component) and explain the context in which it is used. Throughout the dissertation we have used the log files from the cloud computing infrastructure at North Carolina State University, called the ‘Virtual Computing Laboratory’ (VCL) (see Appendix A for more details).

We built the operational profile of VCL from its log files, and the developers used this for regression testing, feature selection, and performance improvement. We were successful in
identifying the most frequent and least frequent set of events (operational profile) by using two different analysis techniques.

Our approach involves transforming the log files using either suffix array data structure or a weighted directed cyclic graph data structure. The current state of the art in building operational profiles from log files offers only semi automated or manual approaches. Using these two transformations, we were able to get the actual usage frequency of VCL use cases, in a automated, linearly scaling approach.

In order to build these transformations we had to abstract the semi structured log messages to specific log events. The current state of the art in abstracting semi structured logs involves either using regular expressions, which takes a long time, but provides accurate results, or using heuristic approaches that are much faster but provided poorer accuracy. We propose an empirical approach, in addition to the heuristic approach, to maintain the speed but improve the accuracy. We were able to reduce the inaccuracies in abstraction by half (down from 5% to 3%), as compared to the pure heuristic based approaches, when we used a hybrid approach. In the case of large logs, this can represent a considerable improvement. For example, in the VCL case this improved the abstraction accuracy of about 16,000 events in a daily log file.

On one hand, we need more data to be more accurate, on the other hand we need less data to make analysis faster. Unfortunately, common current log collection mechanisms can either collect a detailed level of logs at all times or basic level of logs at all times. Ideally, one would collect more data when necessary and less when not required. We propose such a technique. It uses a trained decision engine to intelligently collect information from the application to the log file. By using this technique we were able to reduce the size of the log by almost 29% and still retain the data that the developers want to inspect. This represents a major contribution to the state-of-the-art in accurate but efficient log processing.
A Framework for Analyzing Software System Log Files

by

Meiyappan Nagappan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Laurie Williams                  Matthias Stallmann

Michael Devetsikiotis           Mladen A. Vouk
                                  Chair of Advisory Committee
DEDICATION

To the people whose works have inspired me along the way to be a better person,
Mohandas Karamchand Gandhi,
George Denis Patrick Carlin,
Shivaji Rao Gaekwad, and
Stephen Tyrone Colbert
BIOGRAPHY

The author was born in a small village called Kottaiyur, but grew up in the coastal city of Chennai, in India. It was in high school, that he fell in love with Computer Science and Economics. Since the former had more career prospects than the latter, his dad convinced him to major in Computer Science. He graduated from Vidya Mandir Senior Secondary School in 2002 and went on to get his B.E. degree in Computer Science from Anna University in 2006. In 2006 he left for the U.S. to pursue his graduate studies and got his M.S degree in Computer Science from North Carolina State University in 2008. In his free time he enjoys spending time with his friends, watching movies, playing squash and of late cooking for other people!
ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of a lot of people with whom I have had the good fortune to interact with. I would like to take this opportunity to thank them:

- Appa and Amma (my parents), for constantly encouraging me in all of my endeavors. Thank you for providing me with the best at every stage in my life!

- Dr. Vouk (my academic parent) for advising me on not only this dissertation topic, but also on guiding me to identify and do good research.

- Dr. Shoshani, Dr. Wu, and Dr. Sim of Lawrence Berkeley National Labs got me interested in this field of research. For that and their invaluable advice and help during the initial stages of this research, I thank them.

- Everybody at the SDM center for supporting me on this research.

- Dr. Robinson, Andrew Cordes, Will Snipes, Patrick Francis, Dr. Augustine, and Dr. Acharya at ABB Research and Dr. Murphy, Dr. Nori, and Dr. Zimmermann at Microsoft Research, for giving me some fresh ideas in this research topic.

- Aaron Peeler, Andy Kurth, Josh Thompson and the rest of the VCL team at NCSU for providing me with the log files necessary for this research as well as helping me out with all things VCL related.

- Dr. Williams, for her support throughout my time at NCSU and for being a wonderful person to work with.

- Dr. Stallmann and Dr. Devetsikiotis for agreeing to be my committee members and keeping me on track.

- Dr. Xie, Dr. Thuente, Dr. Samatova, Dr. Reeves, Dr. Honeycutt, Dr. Hill, Dr. Heber, and Dr. Bitzer, for their insightful discussions on both research and career paths.

- The wonderful staff in the CSC department, namely, Ms. Allen, Ms. Coates, Ms. Cook, Mr. Fullmer, Mr. Hartis, Ms. Honeycutt, Ms. Hunt, Ms. Page, Ms. Peaslee, Mr. Sleeth, and Ms. Watkins for making every administrative activity so smooth and easy.

- Laurie and Annie for giving me desk space in their lab, with the other software engineering students.
• Lucas for being the Senior Researcher of our lab by answering any research related questions we had and being constructively critical of my work and for making very hard puzzles for us to figure out where we were headed to for lunch.

• Andy for being the ideal student and constantly setting the milestones for me.

• Aaron, Ben and Prasanth for the many insightful discussions I have had with them.

• Ayla, Dright, Gurleen, Jeff, Jeremy, Jessica, Jiang, Kristy, Kunal, Lauren, Mark, Paul, Pierre, Ramya, Raza, Roselyne, Sarah, Stephen, Suresh, and Yonghee, for all the research and non research discussions we have had in and out of the lab.

• All the 326 students who I TA’ed. They graduated before me, took up great jobs and constantly reminded me that you don’t have to be a graduate student to succeed.

• M. Kothandaraman, Harifa Begum, and Sankari Srinivasan, my teachers in primary and high school, for making math and computer science cool and easy to learn.

• This research was funded in part by the DOE grants DE-FC02-ER25809, and DE-AC02-05CH11231.

I would also like to take this opportunity thank my friends (outside the Software Engineering Research community) who made sure that I never missed home

• Roommates: Srini, Shalaka, Sayantan, Sami, Ravi, Pratheek, Pakshi, Pachu, Kat, Dee, Akshay, Adi.

• Neighbors: Vinod, Swanand, Suresh, Sid, Rajeev, Pradeep, Krish, Karthik, Jay, Dinesh, Apu, Abhi.


• STARS: Mark, Lucy, Kamar, Jen, Jasmine, Anne, Don, Veronica and other STARS members.

• Undergrad: Udayshankar, Ram, Rajesh, Pratap, Koushik, Dhiku, Shruti, Mithra.

• High School: Sivaram, Ramsharan, Prithvi, Pavan, Ganesh, Anirudh.
I would like to thank Sowmya and Pradeep who were my mom and dad’s representatives here in Raleigh making sure I was doing OK. I would also like to thank Sekar and Suja for being wonderful hosts, when I was stranded in Berkeley. Likewise I would like to thank Akshay and Amrita for being wonderful hosts while I was at Cambridge. I would also like to thank Chidambaram, Priya and their two kids for all the help they have extended to me in Raleigh. A lot of family and friends, namely Ashok, Jaya, Nachu, Radha, Palani and Divya, have hosted me in their homes over the last 5 years and made me feel very welcome in a new city. I would like to take this opportunity to thank them too.

I would like to thank Anna, Meena Akka, Nacha, and Rachana who regularly fed me “great home cooked” food.

I would also like to thank Prem and Divya who made me feel a part of their family. I can’t thank you enough for how welcome you have made me feel.

I would also like to thank Aiyya(my grandpa) for all the (snail mail) letters that you have written over the last 5 years. Every letter I got from you made my day better than it already was.

None of this would have been possible without the constant encouragement of Krishna. Thank you for always believing that I had great potential (and constantly reminding me I was not working hard enough towards it). I can’t thank you enough for everything that you have done for me, Krishna(and Janaki)!
# TABLE OF CONTENTS

List of Tables ................................................. ix

List of Figures ................................................. x

Chapter 1 Introduction ........................................ 1
  1.1 Framework - Log Collection ................................. 3
  1.2 Framework - Log Abstraction ............................... 4
  1.3 Framework - Log Transformations ......................... 5
    1.3.1 Suffix Array .................................. 6
    1.3.2 Log Visualization .............................. 6
  1.4 Framework - Analysis Techniques ......................... 7

Chapter 2 Related Work ....................................... 9
  2.1 Log Collection ......................................... 9
  2.2 Log Abstraction ....................................... 10
    2.2.1 Regular Expression Based Abstraction ............... 11
    2.2.2 Clustering Based Abstraction ..................... 12
    2.2.3 Heuristic based Abstraction ..................... 12
  2.3 Deriving Operational Profiles ......................... 13
    2.3.1 Manual Operational Profiling .................... 13
    2.3.2 Commercial Pattern Recognition Tools ............. 14
    2.3.3 Compression Algorithm Based Operational Profiling 15
    2.3.4 Clustering Based Operational Profiling .......... 15
    2.3.5 Log Visualization ................................ 15

Chapter 3 Log Collection using Adaptive Logging ....... 17
  3.1 Introduction .......................................... 17
  3.1.1 Contributions .................................. 18
  3.2 Our Approach ........................................ 18
    3.2.1 Logging Information from the Source Code .......... 19
    3.2.2 Decision Engine ................................ 20
    3.2.3 Adaptive Logging by Inspection .................. 21
  3.3 Experiments and Results ................................ 22
  3.4 Complexity Analysis .................................. 25
  3.5 Conclusion ............................................ 26

Chapter 4 Log Abstraction using Empirical Frequency Analysis .... 27
  4.1 Introduction .......................................... 27
  4.1.1 Contributions .................................. 28
  4.2 Our Approach ........................................ 28
  4.3 Results and Discussion ............................... 30
  4.4 Complexity Analysis .................................. 35
Chapter 5 Operational Profiling Using Suffix Arrays

5.1 Introduction

5.1.1 Contributions

5.2 Suffix Arrays

5.3 Our Approach

5.3.1 Log Abstraction

5.3.2 Construct SA and LCP

5.3.3 Find Patterns

5.4 Algorithm Analysis

5.4.1 Complexity Analysis

5.4.2 Correctness

5.5 Results

5.5.1 VCL Logs

5.5.2 Discussion

5.6 Conclusion

Chapter 6 Operational Profiling using Weighted Directed Graphs

6.1 Introduction

6.1.1 Contributions

6.2 Our Approach

6.2.1 Example

6.2.2 Operational Profiling

6.3 Results and Discussion

6.4 Complexity Analysis

6.5 Conclusion

Chapter 7 Conclusions and Future Work

References

Appendix

Appendix A Virtual Computing Laboratory

A.0.1 VCL Management Node Log Format

A.0.2 VCL Database Logs
### LIST OF TABLES

Table 1.1 Analysis Techniques and Corresponding Users ......................... 7
Table 3.1 Event Statistics of the 9 VCL log Files .............................. 23
Table 3.2 Adaptive Logging Simulation Results ................................. 25
Table 4.1 Frequency table of the words after Data Summary Step .......... 29
Table 4.2 Frequency of the words in the log line ‘Start processing for Jen user’ after Frequency Lookup Step ........................................... 30
Table 4.3 Log Files used in the Analysis ........................................ 31
Table 4.4 Accuracy of Heuristic Abstraction Technique .................... 32
Table 4.5 Accuracy of Empirical Abstraction Technique .................... 33
Table 4.6 Accuracy of Two Step Hybrid Abstraction Technique .......... 33
Table 4.7 Summary of Heuristic, Empirical, and Hybrid Approaches .... 34
Table 5.1 Example: SA of the text "abracadabra" and the LCP .............. 40
Table 5.2 VCL Log Data. Total number of events covered is 5,500,000. .... 49
Table 5.3 The Operational Profile Collected from VCL Logs ............... 50
Table 6.1 Adjacency Matrix $G[M][M]$ ........................................ 59
LIST OF FIGURES

Figure 1.1 A Framework for Analyzing Log Files ........................................ 3
Figure 3.1 Adaptive logging using a Prediction Engine ................................. 21
Figure 5.1 Main steps in our approach ....................................................... 42
Figure 5.2 Operational Profile from LCP and SA ....................................... 44
Figure 5.3 Examples of Log Entries from VCL ........................................... 48
Figure 5.4 Execution Time vs. Number of events in VCL log. \( k \) is shown next to each data point. .......................................................... 50
Figure 6.1 Weighted Directed Graph of the Log File .................................. 58
Figure 6.2 Results (Use Case 1) from Operational Profiling the VCL system by building the WDG of its Log File ........................................ 61
Figure 6.3 Results (Use Case 2) from Operational Profiling the VCL system by building the WDG of its Log File ........................................ 62
Figure 6.4 Results (Use Case 3) from Operational Profiling the VCL system by building the WDG of its Log File ........................................ 63
Figure 6.5 Results (Use Case 4) from Operational Profiling the VCL system by building the WDG of its Log File ........................................ 65
Figure A.1 Cloud Computing Architecture ................................................. 75
Figure A.2 VCL Architecture ................................................................. 75
Figure A.3 Partial Structure of VCLd (or MN) Log Record ........................ 76
Figure A.4 Dashboard for VCL Transactions ............................................... 78
Chapter 1

Introduction

Modern software-based systems collect information about their activity and history in logs. The information in the logs can then be used to analyze system’s activities. Information can come from single files, collections of files, databases, or streams of data from sensors, network interfaces, and continuous monitoring devices and software solutions. The term ‘to log’ is analogous to making entries in a logbook and may involve recording of only a portion of the data deemed needed or useful. The information in the logs typically consists of timestamps, descriptions of events or actions, system state information and/or error information. Each log record or entry may also contain user information, and application information. Logs are most often collected for the purpose of system monitoring, system debugging, identifying security and privacy violations and fault diagnostics. Examples of latter are: fault diagnosis, fault detection by monitoring [7, 18], fault isolation [26], fault prediction [37], operational profiling [19, 33] etc. Numerous tools and techniques are available to carry out log analytics [17]. The potential applications are constantly increasing as noted by Oliner and Stearley [35].

In this dissertation research, we explore log management and analysis issues and formulate solution components and options. On the empirical side, this work focuses on log files generated by the NC State’s Virtual Computing Laboratory (VCL) [10, 38]. This is a university-wide 24x7 private cloud computing solution serving 40,000+ users. Log files from these systems are large, complex, and dynamic. The system produces hundreds of MBs of log data with millions of events in them. Full manual analysis of these logs is not a realistic option, yet they need to be used daily by the VCL operational staff, developers and managers to monitor and resolve problems. Unfortunately, existing log analysis tools leave a lot to be desired. They have issues with varying formats, varying granularity of the collected information, the noise inherently present in logs files, etc. Scalability of the analysis tools and techniques is of utmost importance. When analysis rate is slower than the rate at which information is collected in the log files, usefulness is reduced.
The goals of this research are:

- To better understand large-scale logging processes (provenance, meta-data), and identify the various steps required to analyze log files to provide useful information to developers and decision makers.

- Build an adaptable end-to-end framework for facilitation of log data collection, mining and analysis.

- Explore and provide a set of solutions (tools, techniques, and algorithms) for each component/subsystem of the framework.

- Discuss other solutions available in the literature and perform comparative analysis with the solution proposed in this dissertation.

A significant amount of activity needs to precede successful data mining of log files. This includes log data collection, log abstraction (or reduction of information to more canonical or generic structures and concepts), and formatting of the data for use by analytics engines. Currently, most of these activities are specific to a particular system or application, and often they are conducted in a very ad hoc manner.

We propose to decompose the principal log management and analysis task groups and build a log analysis framework, (see Fig. 1.1) into which individual tools and techniques from each of these phases can be plugged into and allowed to interact through common application interfaces.

Of course, the concept of collection or abstraction or analysis of log files is not a new one. Sophisticated analytics - simple abstraction: In the literature there are numerous log analysis techniques, each one with their own abstraction approach [33, 37, 45]. While the analyzes techniques tend to be highly sophisticated, the abstraction processes are usually very simple. Unfortunately, often simple abstraction techniques may not work beyond the specific application they have been designed for. Sophisticated abstraction - simplistic analytics: On the other end of the spectrum are tools like Spunk [7], and Swatch [18] that have highly sophisticated algorithms to extract and abstract the information from the log files, but the analytics they offer are simplistic. In addition to developing new and more efficient algorithms for each component, we intend to document the best combination of new and existing tools for each component of our framework for a specific software engineering decision. In this dissertation, as an example and case study, we specifically discuss the Operational Profiling of the VCL system from its log files.

To construct operational profile of a VCL installation using its log files, we first need to collect the execution footprint of the system. For fast and efficient analysis we need a small log file and for accurate and descriptive results we may need a more detailed log file. Therefore
we need to collect information in the log file intelligently. Hence the first component of our framework is the ‘log collection’ subsystem. Then we need to abstract the log message (a semi structured string), to its corresponding event type. That is, identify that two or more instances of an event in the log file are recognized and coded to the same event type in the code. Abstracting log messages to event types reduces the noise in the log file. Thus the second component of the framework is the ‘log abstraction’ component. Now that we have abstracted the messages to events, we transform the sequence of events in the log file to a suitable data structure for analysis. In our case we transform the log file to either a ‘suffix array’ data structure or a ‘weighted directed cyclic graph’ data structure so that we can extract the operational profile of VCL.

The following subsections give a brief introduction to each component of the framework in Fig 1.1.

1.1 Framework - Log Collection

Developers, testers, and system administrators typically search log file for events relevant to a particular action and analyze them. They look for events from a particular section of the code when a specific instance of an action took place. In such scenarios they would like more information in the log file which is about the action in question. This can be achieved, for example, by instrumenting the code to collect more information in that section of the code. But if the specific instance of the action is rare compared to the total number of instances of that action type, then there would be an explosion of data in the log file as we would be collecting this extra information during instances of the action that the users of log files are not going to analyze. A logging technique that collects different granularities of information would avoid this data explosion problem. This is because, depending on the context we would be
collecting more or less log lines, when the same piece of code is executed. We call this adaptive logging and it is discussed in Chapter 3.

Related research goals are:

- To develop an adaptive logging technique that collects information in the log files based on the context of the execution.
- To perform an evaluation of our adaptive logging technique using simulations and data from real logs.
- To evaluate the performance of our approach.

1.2 Framework - Log Abstraction

The second component in our framework is the log abstraction subsystem. This is where we have tools and techniques to abstract the data in the given log file to a common format or event type. The motivation for log abstraction is that, often a log file contains a lot of information that comes from dynamic run time changes. Some examples are IP-addresses, port numbers, process identifiers etc. Also in this category are timestamp fields, static message type fields, and variable parameter fields. While that information, of course, is useful it also makes it very diverse. It is possible to make the log structure more canonical by abstracting some of the specificity and variability. The presence of the variable parameter fields makes it difficult to reason about the data in an automatic fashion. So construction of a higher level conceptual data element may be helpful. An example in the network traffic space is 'flows’. A TCP flow log consists of source and destination IP numbers, port numbers, time stamps, direction, payload size, etc. Talking about such flows may be more useful for identification of security issues than talking about individual IP numbers or port numbers.

Data mining algorithms perform poorly if there is too much noise. Therefore, tools and techniques may be required to smooth the ‘noise’ and abstract each line in a log file to a unique message type. One can recognize two types of ‘noise’. The first one is variable parameter information mentioned above. The second type is a sequence of events that always happens together (e.g., a “flow” always has source followed by destination). For analysis, such event-based clusters can be more useful and more efficient, than individual events, as the log files can often have a great deal of information.

Examples of the log abstraction techniques found in the literature include the Simple Log File Clustering Tool [41], the Levenshtein’s edit distance technique [37], and Xu’s approach of leveraging the source code [45]. These techniques are used for abstracting the lines in the log file to an integer or a unique message type. Clustering algorithms are used to cluster very
frequently occurring set of events. These algorithms are used to smooth the data, i.e., reduce the ‘noise’ in the log files. This component of the framework translates each log message to a unique message type so that different analysis algorithms can operate on it.

In this dissertation we observed that, Vaarandi [41] and Jiang et al [22] come to the same conclusions about the properties of messages in each log line. We agree with their conclusions and exploit these properties to abstract free form log lines. The key property we exploit is that if a particular event occurs in multiple places in a log file with different values for its parameter field, then the static parts of the log line, i.e the words in the message type field will occur many times whereas the variable values will occur fewer times as compared to the static words.

Related research goals are:

- To take an extensive look at the research related to log file abstraction to identify the assumptions behind each of those solutions. We intend to clearly define the circumstances under which our solution would out-perform other solutions, for example in the case of semi-structured logs with no access to source code.
- To propose a new solution to abstract semi structured log files in linear time, by exploiting the frequency of occurrence property in log files.
- To derive the time complexity of the algorithm.
- To calculate the accuracy of our solution in comparison with other solutions.

1.3 Framework - Log Transformations

Once the log lines in the log files are abstracted to log events, we transform them to a specific data structure that would make the analysis of this data more efficient and accurate. In this dissertation we transform the events to a suffix Array and a weighted directed graph data structure.

Related research goals are:

- To find a solution that determines the occurrence probability of sequences of events - high occurrence probability events and low occurrence probability events. Most other solutions provide only the probability of the most frequently occurring events [19].
- To build an automated solution that does not require users to tune any parameters.
- To estimate the performance and derive the complexity of our approach.
- To interview the developers of the system whose logs we analyze to determine if the operational profile we got matched their expectations.
In the following two subsections we briefly discuss the two transformations we perform in this dissertation research.

### 1.3.1 Suffix Array

In Chapter 5 we discuss how we built the Suffix Array (SA) and Longest Common Prefix (LCP) array of the abstracted log file. Using these data structures we are able to very efficiently calculate the frequency of every sequence of event patterns. Theoretical analysis of the space and time complexity of our algorithm shows an almost linear scaling. We experimentally evaluated this claim using VCL logs. We conducted an interview with the VCL Program Manager to evaluate the accuracy and utility of the technique. Some of the sequence of events that our algorithm produced (in the most frequently occurring sequences category) was expected to be frequently occurring sequences. However to the surprise of the manager some other tasks featured in the top ten frequent sequences were not. They were able to optimize the system based on our results. Our algorithm also picked out the least frequently occurring events. The program manager used these results to test system scheduling algorithms further.

### 1.3.2 Log Visualization

In Chapter 6 we discuss how we transformed the sequence of events in flat log files to a graph. Every line in the log file is first abstracted to a static message type and a variable parameter type. In our graph each unique event is a node i.e., each static message type will be represented as a node. A directed edge exists from one node to another if the latter occurs immediately after the former in the log file. The edge is labeled with the parameter value. If we don’t consider the parameter value then an unlabeled but directed edge will exist from one node to another. We also keep track of the number of times an edge between two nodes is created as a weight on the edge. In reality we don’t create an edge if it already exists, we just increment its weight. We used the adjacency matrix to represent the resulting graph. The transformation can be done in $O(N\log(M))$ time, where $M = \text{No of unique events}$, and $N = \text{No of log lines}$. In a given application $M$ is a constant, albeit a large one.

By translating it into a graph we are able to apply various existing graph algorithms to meet a variety of analysis goals. We are able to find the patterns in the log file that occur very frequently. We are also able to find paths that have been taken only once. Hence we are able to determine the operational profile of the system. We are also able to find all paths between two given events and their frequencies. All these analysis can be done using the weighted adjacency matrix representation of the log file. Also, we use the Graphviz [2] library to visualize the
graph. For this we convert the graph we have in the data structure into a file in *dot* format \(^1\), which can be used to create a very visually intuitive graph.

### 1.4 Framework - Analysis Techniques

In their research Hamou-Lhadj and Lethbridge [17], present a variety of trace exploration tools and techniques that are used to analyze the information in the log files. Different users have different needs, so we first define the classes of users. They are i) high level users of a system who generally do not look into a log file, ii) system administrators who look into the log file to find a workaround for the failure that the high level users have faced, iii) the technical assistance people or developers who want to find the root cause of a problem to either provide a solution for the system administrators or fix the bug in the source code. Table 1.1 gives examples of the analysis techniques, and the kinds of users that would need it.

In this dissertation we look into how we can build an operational profile from the log files. The operational profile provides the developers with information on decisions regarding, regression testing, performance improvement, feature selection etc.

### Organization of the Dissertation

In Chapter 2 we discuss the other research projects that are related to the various components of our research. In Chapter 3, we discuss the log collection component of our framework and the technique we used for intelligent logging. In Chapter 4 we discuss the technique we used in the log abstraction component of the framework. In Chapter 5, we explain the suffix array based operational profiling technique and in Chapter 6 we discuss the weighted directed cyclic

\(^1\)This format is used by several commercial and open source graph drawing programs. It encodes information about vertices and edges, allowing different shapes, thicknesses, colors, labels, etc.
graph based operational profiling technique. Finally in Chapter 7 we present the conclusions of this dissertation research and provide future directions for research.
Chapter 2

Related Work

2.1 Log Collection

Existing systems like the VCL or Apache servers have their own multi-level logging system. The log4j [5] logging service is used to log messages in Apache. The levels of logging can be changed without modifying the application binary. This is the way logging in VCL [10] also works [38]. But in these systems, the level of logging has to be set before the application is started or deployed. The level cannot be changed at run time. We use the tagging that is used in these techniques to differentiate the different levels of the log messages. But we allow for change in the levels of logging at run time dynamically.

The transaction logs in SQL Server 2000 [9] employ a different kind of adaptive logging. These logs are used to keep track of a transaction, so that it can be reverted if required. The transaction logs need not be of a persistent nature. Hence in this solution, they divide the physical log file into virtual log blocks. They employ a cyclic rotation method and truncate the virtual logs as and when the transaction logs are no longer required. In systems like VCL [10] that employ a log rotation policy, the logs are backed up before they are overwritten. But here the logs are not backed up as they don’t need them after a transaction has been completed.

The research by Netzer and Xu [34] is most related to our work. In their paper they describe an adaptive logging technique for message passing programs. These programs are often run in a parallel environment. The logs collected in these programs are used for debugging purposes. Since the execution cycle for these programs can be extremely long, the size of the log file could be very large. This may prohibit developers from collecting these log messages. The authors have come up with an adaptive logging technique as a solution to this problem. In message passing programs log messages are used to replay the execution of the program. The authors exploit this property of such programs. They checkpoint the execution of such programs by saving the state of a process at certain times. The remaining log messages are quickly
recomputed during replay itself. So the only log messages they collect are the messages that would take a significant amount of time to recompute. These are ones that would most likely create a domino effect. They define a domino effect as one that would push back the execution of the program to a state well beyond the last checkpoint. Thus by logging these domino effect inducing log messages they can avoid the possible time loss. The remaining messages are recomputed from these messages by re-execution. So in their algorithm they try to detect such domino effect inducing log messages and save them to a file. In their experiments they were able to reduce the number of messages to just 1-10% of the original log message count. But there solution of reducing log messages would work only on message passing programs and not on all applications. Also we cannot dynamically increase the amount of logs collected. They just intelligently avoid logging most of the information. In our approach we try to provide a generic solution that can scale up and scale down the amount of logs collected.

The research by Brecher and Rudolf [12] and Du, Zhao and Le [14] on adaptive logging and dynamic logging respectively should not be confused with our research event though the names are similar. We express clearly that these research works are not related so as to avoid confusion. We include them none the less, to let the readers know that the same terms could be used in different contexts.

In their work, Brecher and Rudolf [12] collect logs from the monitoring and collision avoidance systems of machine tools. They use these logs to monitor characteristics of these tools in order to adapt the correction models of the collision avoidance systems. Du, Zhao, and Le [14], in their paper, propose a dynamic logging approach for a DBMS on a flash memory. Since there is no seek delay in a flash memory, they contend that the log file need not be a long sequential file. Hence the log and data file need not be physically located in different locations, i.e., they can be co-located. Hence they dynamically allocate log sectors when the data access is frequent. In both these research topics they do not dynamically change the level of logging in a application at run time like we do in our research.

2.2 Log Abstraction

A number of techniques are available for log file abstraction. Traditionally the users of log file analysis tools come up with the regular expressions for the abstraction. This can be based on their knowledge of the system, or mining the source code for it [45], or data mining techniques [41] or hybrid of the set of above techniques. One such hybrid technique is proposed by Jiang et al [22]. In this section we discuss in more detail the research by Xu et al. [45], Vaarandi and his tool, SLCT [41], and Jiang et al [22].
2.2.1 Regular Expression Based Abstraction

Xu et al. [45] in their research on mining console logs for problem detection used a novel technique for the abstraction of the log files. Each unique event type was assigned a unique ID and each line in the log file was abstracted to one of those IDs. Each event type in the set was a regular expression. For example

starting: xact (.*) is (.*),

where the “(.*)” is the parameter and the fixed string, “starting: xact” is the message type. This regular expression is assigned a unique ID. A set of these regular expressions are extracted from the source code. Then each line in the log file is compared against each of the regular expressions to find the best match. The ID corresponding to the best match is assigned to this line. Since it is a regular expression they are also able to extract the value of the variable parameter fields in the log lines. These regular expressions are built from the source code. They search the source code for all the calls to the function that prints the message in the log files. This search is done using any standard text editor. The calls to this logging function contain the static message usually within quotes and have variables for the parameter fields. By replacing the variables with the string “(.*)”, they are able to build the regular expressions.

The drawbacks of this solution are that we first need access to the source code. If the people carrying out the analysis are different from the people who develop the application that is logging this information then they may not have access to the source code. Even when we access to the source code there are other issues. In their application Xu et al. had the number of messages under a hundred [45]. But in our prior research [33], where we used a similar abstraction, the number of unique messages were close to 2,000. When there are so many of them spread across multiple files, then it takes a considerable amount of time and manual inspection to build this set of regular expressions. The other issue is that if the message to be printed in the log file is constructed outside the call to the logging function, then we will need to examine the abstract syntax trees to get the regular expressions. Also in some cases the message in the log file is not from the source code of the application. One example is when the source code executes a command on the machine on which the application is executing and prints the output of this command to log file. This log line cannot be abstracted to any of the regular expressions. But if any of the above issues are not present in the log file that we are abstracting, then this solution is the most accurate one.

\(^1\)\(\text{(.*)} - \text{Where . = any character, and * = repeated any number of times. Hence (.*) means any character repeated any number of times.}\)
2.2.2 Clustering Based Abstraction

Vaarandi [41] proposes a clustering algorithm for finding patterns in log files. His tool is called the Simple Log file Clustering Tool (SLCT). This algorithm is very similar to apriori algorithms for mining frequent item sets. The modifications made to a typical clustering algorithm for finding frequent item sets, are based on the observations he made about log file data. Firstly only a few words in the log file occur very frequently. Secondly there was a high correlation among these high frequency words. This was because of the fact that each line on the log file is formatted according to the message type and parameter information for that event. This is the same reasoning as in the Xu et al. [45] approach. The message is part of a function call. In his approach Vaarandi does three passes over the log file. The first is to build a data summary, i.e. the frequency count of each word in the log file according to its position in each line. In the second step he builds cluster candidates by choosing log line with words that occur more than the threshold, specified by the user. What is to be noted is that the frequency of a word may be higher in certain positions but may not be so in another position in the log line. In the third step we choose the clusters from these candidates that occur at a frequency higher than the user specified threshold. Each of these candidates are a regular expression. The words in each candidate that have a frequency lower than the threshold are considered as the variable part and hence replaced by “(.*)”.

In their research on identifying failure causes, Mariani and Pastore [26], utilize the SLCT to abstract the log lines to log events. But the SLCT was not designed to detect the regular expressions to abstract all lines in a log file. It was designed to detect frequently occurring patterns. A pattern is considered frequently occurring if its frequency is greater than a user specified threshold. All log lines that don’t satisfy this condition are stored in the outliers file. Hence even some log lines that repeat a few times but are still less than the threshold won’t be considered as a pattern. The goal of that research was to find frequently occurring patterns and not log abstraction. In his paper [41] Vaarandi used thresholds of 50%, 25%, 10 %, 5% and 1%. None of these would either abstract all log lines to an event type or produce regular expressions for abstraction. But if mining frequently occuring patterns is the goal, then this solution performs better than the other ones as it is application and log file independant.

2.2.3 Heuristic based Abstraction

In their research Jiang et al. [22], propose a very efficient approach to log file abstraction. They have 4 steps in their approach: anonymize, tokenize, categorize and reconcile. In anonymize they pick words that they think are parameters. In their case study they classified the value that followed an ‘=’ or the value following the words ‘is—are—was—were’. They replaced these values with a variable ‘$v’’. Then in tokenize they bag log lines with similar characteristics into...
bins. All log lines in each bin have the same number of static words and same number of parameter fields. Then in the categorize step they go through each bin and compare the anonymized log lines and group all the lines that are exactly the same. Since the variable parts of the line are replaced with a common variable, a simple string comparison would group the log lines. Finally in the reconcile step they go through the groups and combine groups that are different by just one word. This way they are able to group log lines that have a parameter that was not anonymized.

This is a very efficient technique that makes use of the properties of messages in log files for abstraction. But the assumptions are that the logs have enough structure to be able to find the parameters using heuristics. But not all log files have that kind of a rigid structure. The key hurdle when trying to abstract log files is that we don’t know what the parameters are in each line of the log file. If we did know, then this technique like the Xu et al. [45], would provide highly accurate results. If log files have some structure to it then highly accurate abstraction can be done. Both Xu et al’s [45] and Jiang et al’s [22] approach would provide very accurate results when the assumptions under which they operate hold true. When the log files are of a free form and we don’t know where the parameters occur and we don’t have access to the source code, then only an approximate abstraction solution can be achieved. A clustering algorithm like the SLCT [41] would be such a solution. But as mentioned before, SLCT was not designed to abstract log lines.

2.3 Deriving Operational Profiles

2.3.1 Manual Operational Profiling

Operational profiles are traditionally created by a combination of system engineers, high-level designers, test engineers, product planners and marketing people [28]. They derive the operational profile by manually quantifying the usage of each element in the system, in a manner as close as possible to expected customer usage. However the operational profile calculated from the expected usage differs from that based on the actual usage [19], [28]. The latter is likely to lead to better reliability estimates of the system during field operation. Therefore, it is advantageous to use, in the second and higher releases of a product, operational testing profiles that are closer to the actual operational behavior of the product.

Getting the operational profile based on actual usage is not a simple matter [44]. Code profiling and trace analysis are some of the available techniques.

The Eclipse Test and Performance Tools Platform (TPTP) [8], and Java Virtual Machine Profiler Interface (JVMP) [4], are some of the more popular code profiling tools. They are primarily used during the testing phase and not in a production environment as they can slow
down the system [21]. Trace analysis tools and techniques perform a very similar task. They explore traces from program execution dynamically for a variety of purposes like software optimization. In their survey, Hamou-Lhadj and Lethbridge, discuss the strengths and weaknesses of eight trace exploration tools [17]. They state that the object oriented systems have driven the increase in the number of such tools as polymorphism and dynamic binding greatly limit the use of static analysis. They conclude with the need for a common framework for trace exploration tools and techniques.

Execution logs, code profiles and execution traces are a record of the usage of a system. But code profiles and execution traces could generate information that is orders of magnitude bigger as they may include every function call and branch statement. Since this could affect the performance of the software system, they are often not found in production systems. The execution logs on the other hand are more flexible. It collects only that information which a developer wants. Hence even production systems have logs. These logs can be used to build operational profiles that are based on actual usage of the production system by the user.

2.3.2 Commercial Pattern Recognition Tools

The usefulness of analyzing log files has been long recognized. Tools like SEC [40], Splunk [7], and Swatch [18], are used to monitor log files. SEC is an event correlation tool, Splunk is a log management tool, and Swatch is a log monitoring tool. All the three of them and other similar log analysis tools can only monitor the logs for a particular event or sequence of events. Most of them perform a regular expression match. What is common in them is that the event(s) need to be known in advance. Once known, they can analyze the event(s) in the log file and get the frequency of them to build operational profiles. But we need the expertise of the developers to come up with the event or sequence of events. The developer has to think of all possible cases for a particular action and all the actions for which we need the frequency. Using SEC, Splunk, and Swatch we can only find out the usage probability of the sequences we search for. We cannot find the most frequently used part of the system, unless the developer thought of it earlier. This technique helps verify the developers prediction, but does not itself calculate the operational profile. However these tools can be used to extract the event identifiers from the log files to be used by other automated operational profilers for finding repeated sequence of events.

Hassan et al. [19] and Vaarandi [41] have come up with other solutions to analyzing log files that have overcome this issue. They do not require the developers to come up with sequences of events before the analysis starts.
2.3.3 Compression Algorithm Based Operational Profiling

Hassan et al. use a log compression approach to identify patterns and their densities in the log files. In their approach they exploit the fact that a file with more repetition in it will be compressed more by a tool like gzip. They split the log file into equal sized periods, and compress each of them. They plot the compression ratio as a log signal to find the period with the greater density. This period is likely to have more repetitions. An engineer of the system then identifies the pattern that is repeating and writes filtering rules for it. Therefore it only aids the human in identifying patterns, thus making it a semi-automatic approach which cannot avoid human intervention. The log file is filtered of this pattern and then all the steps are repeated again. In this approach we can only detect the top few sequences and their relative densities, and it takes 2-3 hours to just come up those.

2.3.4 Clustering Based Operational Profiling

Operational profiles are often derived using clustering algorithms [27], [28], [29] which have non linear time complexities. Vaarandi’s tool called Simple Log File Clustering Tool (SLCT) [41] uses a novel clustering algorithm to mine for patterns in log files. The tool has very low execution times that vary from 7-11 minutes for a log file of size 1 GB. He exploits log file properties such as (a) Most words in a log file occur only a few times, and (b) Frequently occurring words have a high correlation. The clustering algorithm itself has three steps, viz. building data summary, building cluster candidates, and selecting clusters from this set. The performance of this clustering algorithm proposed by is highly sensitive to a user-specified parameter called support threshold, which makes the algorithm hard to use by other users. Our algorithm on the other hand does not require the user to tune any parameters. Also the SLCT algorithm finds only single line patterns, i.e. the count of a particular log line only unlike our algorithm that can find patterns that extend across multiple lines.

2.3.5 Log Visualization

There is extensive research on operational profiling [19][33], and debugging of systems [13][15] using information collected as log files or execution traces. Also the concept of viewing the execution of software as state machines is not new. In their literature survey on software model checking [20], Jhala and Majumdar report the different ways to build state models of software to improve testing efforts. Some of these tools are the VeriSoft [16], Java PathFinder [42], and CMC [30]. These models are built statically from the code or from traces collected during testing. In our paper we present a way to transform the log files collected from a production system to the adjacency matrix representation of a graph. We then apply existing graph theory algorithms, through their corresponding software libraries, on the adjacency matrix to
perform operational profiling and anomaly detection and visualize them using existing tools like Graphviz [3].
Chapter 3

Log Collection using Adaptive Logging

3.1 Introduction

In his paper [41], Vaarandi uses log files from applications like Linux mail server, Internet banking server, and file and print server. Each of these systems can produce hundreds of MBs of log data, with millions of events in them.

Most applications running on cloud computers are multi-threaded to accommodate multi-user capabilities. The nodes of the cloud may be located in multiple data centers. All these features of the applications in cloud computing environments, leads to large and complex log files. This is due to the exponentially growing (in terms of size and complexity) information set that is being collected in log files, especially in these cloud environments. In our case study we studied the logs from the cloud environment at North Carolina State University called Virtual Computing Lab(VCL) [10]. We examined the log files, identified areas for improvement and provided solutions for optimizations.

Developers, testers, and system administrators typically search log file for events relevant to a particular action and analyze them. They look for events from a particular section of the code when a specific instance of an action took place. In such scenarios they would like more information in the log file which is about the action in question. This can be achieved, for example, by instrumenting the code to collect more information in that section of the code. But if the specific instance of the action is rare compared to the total number of instances of that action, then there would be an explosion of data in the log file as we would be collecting this extra information during instances of the action that the users of log files are not going to analyze. A logging technique that collects different granularities of information would avoid this data explosion problem. This is because, depending on the context we would be collecting
more or less log lines, when the same piece of code is executed. The adaptive logging approach proposed in this chapter is such a technique.

Note that the technique used in this case study is suitable for collecting information if generated by software systems that are run on dedicated servers and accessed by multiple simultaneous users. Log files from most server side software systems are very large. The cloud computing [10] log files we analyzed collect more than 450,000 events each day and can grow log file size by more than 56 MB per day.

3.1.1 Contributions

- We performed a case study to see what information was being logged, and what information was being used by the developers.

- We developed an adaptive logging technique that collects information in the log files based on the context of the execution.

- We performed a simulation of our adaptive logging technique using real logs we collected to evaluate our approach. We present the results and observations of our approach in this chapter.

3.2 Our Approach

The huge size of the log files affects the performance of machine learning and data mining algorithms. Algorithms with a time complexity $O(n^2)$ are often avoided as they will have to operate on millions of events. But if we need the events associated with an error reported in a log file, we will also have to accept the logging of such events when there is no error. If we were going to reduce the size of the log file by collecting less data, then there is a loss of information. The data mining algorithms will not be able to accurately analyze the system, using the log files, due to the lack of data. The more the training data these algorithms have, the better the analysis. Hence they will need as much data as possible.

Thus the developers have to come to a compromise. Either they get a superfluous amount of data that improves accuracy but affects performance, or they get limited data from the logs that can easily be analyzed but the accuracy of these results may be questionable. There is a huge scope for improvement if we intelligently log information.

Most of the analyzes focuses on fault detection, fault isolation, anomalous event monitoring etc. These contribute to a very small fraction of the log file. Thus we have come up with an adaptive logging technique which will collect information depending on the current context of the system state based on what information is being collected as logs. The core idea in our
approach is that a decision engine will decide whether we need to collect more information or not. Thus the developers can train the decision engine to collect more information in the log files when they need it and filter the information coming to the log file when they don’t need extra information. Thus they already collect the ‘normal’ or at least base line information, but get a more detailed log trace when the situation demands. A trained prediction model or a mere inspection based model could be used in the decision engine to determine when to store to disk more than the ‘normal’ information.

3.2.1 Logging Information from the Source Code

The information in the log file is printed from calls to the logging method in the source code of the application. Hence if we want more information from a particular piece of code during a specific instance of its execution, then we need to add the extra calls to the logging method with the extra information in the source code. But then we do not want this information do be printed to the log file every time this piece of code is executed. Therefore we tag the log message in these extra calls as ‘debug’ log lines. To make it uniform we tag the lines already present in the source code as ‘normal’ log lines. The rationale behind this differentiated tagging is that we will log messages tagged as ‘normal’ at all times. The log lines tagged as ‘debug’ will be saved onto disk only during specific instance of the execution as determined by the developers ahead of time. Thus the logs collected from a particular piece of code in ‘debug’ mode is the super set of the logs collected in the ‘normal’ mode.

In most systems this type of logging already exists as a multi mode logging scheme. Note that in these systems the control of what is to be logged is determined before the system starts up. This is usually controlled by a flag that is set during the start up. The system itself controls how much information has to be logged. Hence the granularity cannot be changed at run time. Even if they can be edited at run time, the response is usually not immediate and hence the information we wanted to be logged is missed. Therefore we either get a lot of information at all times when it is in the ‘debug’ mode or limited information all the time in the ‘normal’ mode. In our approach we change the control over filtering of this information from the system to an external decision engine. Through this we can control the granularity of the information at run time.

Since the control is going to be external to the system, we will collect the logs in the ‘debug’ mode in memory and filter out information when the decision engine says that it is not important.
3.2.2 Decision Engine

The decision engine can be any algorithm or technique ranging from a simple prediction model like the logistic regression model to a very complex one like a Hidden Markov Model or a data mining technique or merely an inspection based technique. This model/technique in the engine will have to be trained first to decide when to collect more information and when to collect the normal amount of information. For this we need a log file with labeled data. The labeling is done according to the developers needs. For example, the developers may need more information when a failure occurs. It could be that the developers want information when the performance of the system slows down. The developers decide if a particular event in the training log file is going to lead to a situation where they may need more data or not. The decision engine will then decide according to the data provided in the training phase.

Since we filter out information based on the decision engine’s decision, we need to make sure that we choose a model/technique and its parameters such that the false negative rate of the decision engine is extremely low. This can be at the cost of having a poor false positive rate, but the better the false positive rate the more efficient the collection becomes. This is because there usually is no harm done if we collect extra information in a situation where we don’t need it, so long as it is not excessive, but it will be unacceptable if we fail to log information when it is required to be logged i.e. a false negative.

If we choose to use a prediction model or data mining technique then the model might need to be retrained under certain circumstances. If the operational profile of the system considerably changes, then the prediction may be affected. Also if the application undergoes significant change, like in a new release, then the events leading up to a failure may significantly differ. Under such circumstances the model will have to be retrained to identify and predict the failures. One of the ways to identify when retraining is required, besides the knowledge of the changes stated above, is discussed below.

We experimented first with a conservative prediction model that is tuned to get more false negatives than false positives. Thus a rise in false positives indicates that the prediction model is not performing as it was intended to. In a real implementation we cannot measure the false positive rate. Therefore if the size of the log file gets bigger and bigger even though it contains a similar number of actions, in this case reservations, and a similar number of failures, it is an indication that the false positive rate is increasing. Thus this would be a good time to analyze the reasons for the changes in the log file and take the necessary retraining steps.

In our experiments with a logistic regression prediction model in the decision engine we observed that even though the false negatives was low (the recall rate was 0.99 - 1.0), the false positive rate was very poor (the precision was between 0.02 - 0.53). So instead of pursuing a
prediction based model for the decision engine we used a simple inspection technique in the decision engine. Details of this technique are explained in the following subsection.

3.2.3 Adaptive Logging by Inspection

Our adaptive logging technique is illustrated in Fig. 3.1.

The method/function that the application calls for logging information to file is edited to collect an extra parameter along with the log message. This parameter is used to tag the log event as to weather it has to be collected at all times or only at specific times. Thus the default set of events that have to be collected at all times would be tagged as ‘normal’ and the events that have to be collected only during, say a failure, would be tagged as ‘debug’. Note that the failure event itself would be tagged as ‘normal’ because we would definitely not like to miss that event in the log file.

This logging method/function, instead of writing the output to the log file will send it to a buffer that is in memory. The decision engine would inspect the tail of the buffer where the log events enter the buffer. At the head or other end of the buffer, is filter which if open would log all messages (both ‘normal’ and ‘debug’ level events) in the log file and if closed would log only the ‘normal’ messages.

We build a database of special events to look out for in the decision engine. It inspects the last event or the tail of the buffer. By this time, then buffer has events till its head which have happened before in time with respect to this event at the tail. As long as the decision engine does not match the event at the tail of the buffer with an event in its database, it keeps the filter closed in the head end. Only the ‘normal’ level events are saved in the log file. If the decision engine matches the event at the tail with a event in its database, then then opens up the filter at the head end to save the ‘debug’ level messages as well as ‘normal’ level messages in the buffer.

If the buffer is long enough and the inspection can happen in constant time then all the ‘debug’ level events corresponding to the inspected event can be captured in the log file. We
would have no false negatives. There would also be no false positives since there is no prediction, but only an actual comparison against a database of special events.

The results and discussions follow in the next section.

### 3.3 Experiments and Results
Table 3.1: Event Statistics of the 9 VCL log Files

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>No:of Log Lines</th>
<th>Total Number of Reservations</th>
<th>Avg Events per Reservation</th>
<th>No:of failed Reservations</th>
<th>Avg Events per Failed Reservation</th>
<th>Avg Events in Failed Reservation</th>
<th>Percent Inspected</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>834471</td>
<td>314</td>
<td>1703</td>
<td>10</td>
<td>585</td>
<td>1047</td>
<td>3</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>838622</td>
<td>255</td>
<td>2078</td>
<td>4</td>
<td>991</td>
<td>1950</td>
<td>2</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>731731</td>
<td>168</td>
<td>2759</td>
<td>4</td>
<td>1587</td>
<td>4179</td>
<td>2</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>710157</td>
<td>234</td>
<td>1917</td>
<td>5</td>
<td>575</td>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>785108</td>
<td>306</td>
<td>1684</td>
<td>9</td>
<td>616</td>
<td>1027</td>
<td>3</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>767965</td>
<td>210</td>
<td>2318</td>
<td>7</td>
<td>773</td>
<td>1031</td>
<td>3</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>791334</td>
<td>253</td>
<td>1982</td>
<td>13</td>
<td>742</td>
<td>1041</td>
<td>5</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>870357</td>
<td>287</td>
<td>1902</td>
<td>2</td>
<td>675</td>
<td>931</td>
<td>1</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>830892</td>
<td>206</td>
<td>2538</td>
<td>3</td>
<td>818</td>
<td>1414</td>
<td>1</td>
</tr>
<tr>
<td>Average</td>
<td>795626</td>
<td>248</td>
<td>2098</td>
<td>6</td>
<td>818</td>
<td>1562</td>
<td>2</td>
</tr>
</tbody>
</table>
We evaluated our approach on the logs from VCL. Since we were not able to edit the logging scheme in the production version of the VCL system, we performed a simulation on logs collected from the VCL system. For this simulation we used the 9 log files collected in ‘debug’ mode. We labeled the data in the log files with the tags ‘normal’ or ‘debug’ so that the decision engine would know which events to filter out and which events to retain. In order to implement our approach we first needed to know what events to add in the database of the decision engine, and what was the optimal size of the buffer that we needed.

We conducted interviews with the developers of VCL to identify the situations where they wanted more information in the log file. Another objective of this interview was to identify the events that signified these specific situations. From our interviews we identified that the developers wanted all the information about the reservations that ended up in a ‘failure’ state in the VCL management nodes. These failures could be identified by the events with messages ‘CRITICAL’ and ‘WARNING’. The VCL developers want more information only when a fault occurs. Thus the database in the decision engine has these two events. From the last column of Table 3.1 we observe that these events corresponded to about 2% of the log file on an average. Hence with a more efficient logging approach we would be able to collect more information about this 2% and a basic amount of information otherwise.

The buffer we used was a ‘Queue’ data structure. To determine the size of the buffer we performed a sensitivity analysis on the log files from VCL to determine how many events were logged per reservation in the debug mode and found the average for a successful reservation. We report the average values for the 9 log files, for which we did this analysis in Table 3.1.

We also report the average and maximum number of events for reservations that failed in Table 3.1. We can see that for log file vcld.log.3, the maximum number of events in a failed reservation was 4179, which was the highest among the 9 log files. The average number of events in the failed reservations was 818, and the average maximum value was 1562. Hence we chose the buffer size to be higher than the average maximum, namely 2000.

The decision engine inspected the tail of the queue and compared it against the events in our database. We used a hash map to implement the database. At the head of the queue depending on what label the event has, and whether the filter was open or closed the events were logged in the file. The results of the simulation with the number of events in the original log file, and the number of events in the optimized log file are reported. Table 3.2 has the results from the simulation carried out on the 9 log files. We also report the percentage savings per file in Table 3.2.

As we can see from the last column in Table 3.2, there can be great savings in terms of size if our approach of data collection is used. The average percentage reduction was 29%. Thus, we can reduce the number of events in the log file by more than a third of the original number of events. Hence the savings will directly impact the performance of analysis algorithms.
### Table 3.2: Adaptive Logging Simulation Results

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>No:of Events in Original</th>
<th>No:of Events in Optimized</th>
<th>Percentage Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>834471</td>
<td>628228</td>
<td>25</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>838622</td>
<td>589735</td>
<td>30</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>731731</td>
<td>507615</td>
<td>31</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>710157</td>
<td>507752</td>
<td>29</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>785108</td>
<td>573130</td>
<td>27</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>767965</td>
<td>524576</td>
<td>32</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>791334</td>
<td>556761</td>
<td>30</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>870357</td>
<td>633450</td>
<td>28</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>830892</td>
<td>571546</td>
<td>32</td>
</tr>
<tr>
<td>Average</td>
<td>795626</td>
<td>565866</td>
<td>29</td>
</tr>
</tbody>
</table>

The critical issue is to avoid a situation when there is a fault in the system, and we fail to collect information relevant to the fault due to a small buffer size. But at the same time a large buffer will be expensive to maintain. So we have to strike a delicate balance where the buffer size is optimal. This can be done by doing a sensitivity analysis on the logs of the required application before deciding the buffer size. What we are looking for in this sensitivity analysis is the number of events before a specific event we want recorded in the log file.

The decision engine is a key component of our approach. Any model requires information to make the decision. That information is typically found in the buffer. Depending on how much information needs to be recorded and how much information is required for making the decision we need to adjust the buffer size.

### 3.4 Complexity Analysis

There are four operations in this approach:

1. Pushing the event into the buffer from the application.
2. Inspecting the event at the tail of the buffer.
3. Editing the status of the filter if required.
4. Popping the event from the head of the buffer into the file after checking the status of the filter.

The complexity analysis of these four steps is explained below. Let the number of events logged by the application be $N$. We choose a ‘Queue’ data structure to implement the buffer.
The push and pop operations in steps 1 and 4 are hence done in constant time. We implement the database of events to look out for in a hash map since the number of these events are fixed for every application and typically very small (two events in VCL). Hence to look up if the current event in the tail is in the table would require a look up in the hash map which is done in constant time. Hence step 2 can also be in constant time. Editing the status of the filter in step 3 and checking the status of the filter in step 4 are assign and conditional operations respectively. hence these too can be done in constant time.

From the above analysis, all the four steps takes a constant amount of time. Thus for each event the time taken is a constant. Hence to collect data for the \( N \) events logged by the application this approach would take an \( O(N) \) time. Thus the time complexity of our approach is linear.

The number of events logged in file is smaller than the number of events logged by the application. hence the number of events in file is \( \lt N \). The buffer size and hash map size is constant for an application, albeit the buffer size may be a large constant. Hence the space complexity of our approach is linear or \( O(N) \) as well.

### 3.5 Conclusion

We examined 9 large log files from the cloud management application, VCL. We observed that the developers used about 2% of the log file. They would have like more information about this 2% of the log file of interest, because it contains information about the faults. Ideally they would have liked all interesting parts of the log file to contain ‘debug’ level information. But to do that for all events in the log file would be too expensive. Therefore we have developed an approach that adaptively collects information about recurring events of interest. This approach is based on collecting data at ‘normal’ level for all events and augmenting them with ‘debug’ level information when the decision engine inspects an interesting event. This would considerably reduce the number of events logged to disk as compared to logging the information at ‘debug’ level all the time. In our case study we observed that adaptive logging collects 29% less events as compared to collecting logs in the ‘debug’ mode all the time.
Chapter 4

Log Abstraction using Empirical Frequency Analysis

4.1 Introduction

Software systems collect information about their activity in log files. The term ‘to log’ comes from making entries in a logbook to keep track of activities completed. The information in the log files, called logs, consists of the start or end of events or actions of the software system, state information and error information. Each log line typically contains date and time information, user information, application information, and event information. Logs are often collected for system monitoring, system debugging and fault diagnosis. Numerous log file analysis tools and techniques are available to carry out a variety of analyses. Insights of varying degrees are achieved by log file analysis. These include but are not limited to, fault detection by monitoring, fault isolation [26], operational profiling [33] etc. Tools like Splunk [7], and Swatch [18], are used to monitor log files. Splunk is a log management tool, and Swatch is a log monitoring tool.

The users of log files either manually look for a specific piece of information in the log file or apply an analysis algorithm to mine information from it. In the latter case the accuracy of the results produced are highly dependent on the variability present in log files. Each line in a log file is a combination of a static message type and variable parameter information. For eg.

Request data from 127.0.0.1 to 127.0.0.2

The static fields of the above example are ‘Request data from’ and ‘to’. The parameters in the above example are ‘127.0.0.1’ and ‘127.0.0.2’. The parameters change at run time and may be different in each instance of the above example in the log file, i.e. the parameter can be any IP address. Hence two instances of the same message will look different because of the different IP address in them. But the same line of code essentially has executed. The separation of the
static field from the dynamically changing parameter field is called log file abstraction. Thus in the above example we would separate the log line into two fields:

Message Type: Request data from * to *

Parameter Fields: 127.0.0.1, 127.0.0.2

Log file analysis techniques like operational profiling [33], fault isolation [26], system problem detection [45] etc, operate on the abstracted log files. In each of the above techniques, each log line in the log file is abstracted to a corresponding integer ID. Then the analysis is carried on the set of integers. In most of the techniques the abstraction itself is done by regular expressions. The abstraction techniques usually are used to build these regular expressions. The result of abstraction techniques are a set of regular expressions and corresponding IDs. The log lines in the log files can then be matched to these regular expressions and the ID for the best match is used for each line in the log file.

4.1.1 Contributions

1. A study of the existing abstraction techniques and their advantages and disadvantages.

2. Our abstraction algorithm which tries to address the issues of the other algorithms.

3. The evaluation of our algorithm.

4.2 Our Approach

Vaarandi [41] and Jiang et al. [22] come to the same conclusions about the properties of messages in each line of a log file: “Only a small fraction of the words occur frequently”. This is the key property that we exploit. Thus the basis for our approach is that if a particular event occurs in multiple places in a log file with different values for its parameter field, then the static parts of the log line, i.e the words in the message type field will occur many times whereas the variable values will occur fewer times as compared to the static words. We will use the following example to illustrate this.

Start processing for Jen user
Start processing for Tom user
Start processing for Henry user
Start processing for Tom user
Start processing for Peter user

The words ‘Start’, ‘processing’, ‘for’, ‘user’ occur 5 times each in the positions 1, 2, 3, and 5 respectively. This is because this is the constant part of the log line. The words ‘Jen’, ‘Henry’, and ‘Peter’ occur once and ‘Tom’ occurs twice. This is much less than the frequency of other
words. From this we can make the inference that the message that created these lines in the log file would be of the following format: ‘Start processing for $username user’. This is indeed the statement that created this log message. The words that belong to the constant message type field occur more often than the words in the variable parameter field. We exploit this property to abstract log lines to event types. For this we carry out two passes over the log file.

In the first pass we build a data summary of the words in the log file. We build a frequency table that has the number of times a particular word occurs in a particular position in the log line. Hence the rows in the table are the words, and the columns are the positions in each log line. In Table 4.1 we show a part of what the frequency table would look like in the above example after we parse through the 5 lines. Filling the values in this table can be done in time O(N), where N is the number of words in the log file. We take one line at a time and split it into individual words. Then we look up if we have a row for that word in the table. If not we create one. Then we go to this row and increment the value in the column that corresponds to the position of the word in this log line. Looking up the word can be done in practically constant time (theoretically not constant) by using a hashing function. In the hashing function each word maps to an integer which corresponds to the row in the frequency table, where we increment the frequency of that word. The position of the word in the log line indicates the column in that row where this increment has to happen. At the end of the first pass we would have completed building this table.

In the second pass we examine each log line again. Here we do a look up of the table for each word in the line. Then we pick the frequency of that word in that position in the log line from the corresponding column. So for example, when we are parsing through the log line ‘Start processing for Jen user’, we split it into individual words. Then we look them up in the frequency table. So here we look up, say, ‘Start’. Then since ‘Start’ is the first word in the log line, we retrieve the frequency of the word ‘Start’ in the first column. Here we would be extracting the value 5. We do this for all the words in the log line. So if were currently
Table 4.2: Frequency of the words in the log line ‘Start processing for Jen user’ after Frequency Lookup Step

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>5</td>
</tr>
<tr>
<td>processing</td>
<td>5</td>
</tr>
<tr>
<td>for</td>
<td>5</td>
</tr>
<tr>
<td>Jen</td>
<td>1</td>
</tr>
<tr>
<td>user</td>
<td>5</td>
</tr>
</tbody>
</table>

abstracting the log line ‘Start processing for Jen user’, this second pass would translate to the values in Table 4.2. A similar translation is done for each of the log lines.

Then in the third pass, for each of the log lines we calculate the average of the frequencies we extracted for the words in the second pass. If the individual frequency of a word in that position in a log line is greater than the average of the frequencies of the words in that line then it is a constant word, else it is a variable parameter. In our example, when we are abstracting the log line ‘Start processing for Jen user’, the average from Table 4.2, is 4.2. Hence the words with frequencies above 4.2 is a constant word and those below it would be a variable parameter. We would replace the variable parameters with the character ‘*’. In this case the words ‘Start’, ‘processing’, ‘for’ and ‘user’ will be constant words since they have a frequency of 5 which is greater than 4.2. But the word ‘Jen’ will be classified as a variable parameter since it has a frequency of 1 which is less than 4.2. Hence the log line ‘Start processing for Jen user’ is abstracted to ‘Start processing for * user’. Hence we have successfully extracted the variable parameter field from the constant message type field. In the case of usernames with two or three words, it might affect the frequency of the word ‘user’ in the example. But in log files we have noticed that there are enough of those (two or three word usernames) log lines too. Hence they too are abstracted the same way to the same message type as well.

In SLCT the author tries to find clusters across log lines [41]. What differentiates our approach from SLCT is that we look for clusters within a log line. Thus to identify the event types in the log file we clustered the frequencies of the words instead of clustering the words themselves.

4.3 Results and Discussion

We implemented our approach and tested it on 9 log files from the Virtual Computing Lab [10], a cloud computing management application at North Carolina State University. Table 4.3 gives some basic information about the log files we used in the analysis. We can see the number of log lines in the file, number of reservations and the average events per reservation. The log lines in the VCL log had the following fields (each separated by the ‘|’ symbol):

- Date and Time
Table 4.3: Log Files used in the Analysis

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>No:of Log Lines</th>
<th>Total Number of reservations</th>
<th>Avg Events/Reservation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>834471</td>
<td>314</td>
<td>1703</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>838622</td>
<td>255</td>
<td>2078</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>731731</td>
<td>168</td>
<td>2759</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>710157</td>
<td>234</td>
<td>1917</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>785108</td>
<td>306</td>
<td>1684</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>767965</td>
<td>210</td>
<td>2318</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>791334</td>
<td>253</td>
<td>1982</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>870357</td>
<td>287</td>
<td>1902</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>830892</td>
<td>206</td>
<td>2538</td>
</tr>
<tr>
<td>Average</td>
<td>795626</td>
<td>248</td>
<td>2098</td>
</tr>
</tbody>
</table>

- Process ID
- Request and Reservation ID
- Module Name
- Function name and line number from source code (that acted as an Event ID)
- Message

Note that each log line does not have these 6 fields. There are some log lines that are overhead information. We are interested only about the log lines that have all these 6 fields. By using a tokenizer that splits the log line, using the separator symbol (‘|’), we can get these fields and drop the log lines that do not have the 6 fields.

The string in the message field was written to the log file in that particular log line by a particular line of code from the executing application. This line of code and the function in which it is present acts an Event ID. As stated in the previous sections each event in the source code is instantiated as a (possibly) different message in the log file. We want to extract this message in the log file to a unique event in the source code. In this particular set of log files since we have the Event ID information, we need not perform the abstraction of the message to the event. But this Event ID information is not available in all log files. In our case study we use the Event ID information to see how well the abstraction techniques perform.

We applied three abstraction techniques on the messages in the log files, namely

1. Heuristics based abstraction technique by Jiang et al. [22]
2. Empirical abstraction technique developed by us.
3. Two step hybrid abstraction (first = heuristic, second = empirical)

The message were abstracted to events by applying each of the above techniques. Then the resulting abstraction was compared against the available Event ID’s (line numbers from the VCL source code that is included in each log line) to check how well the abstraction techniques performed. Thus for our experiments we only took the fifth (Event ID as expressed by the line number and function name) and the sixth (message) field. Thus the message field is abstracted and compared for accuracy with the Event ID field. The static parts of the message were retained in the abstraction and the parameter (dynamic) parts were replaced with the character ‘*’. We then assigned a unique integer to each unique message. Thus each log file was abstracted by each technique to a set of integers. Each log file had the same corresponding number of Event IDs. We assigned unique integers to each of these unique Event IDs. Thus the comparison was between the set of unique integers from the Event IDs and the set of unique integers from the abstracted message.

Table 4.4: Accuracy of Heuristic Abstraction Technique

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>Number of Abstractable Events</th>
<th>% of Misclassified Events</th>
<th>% of Misclassified Event Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>556054</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>551034</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>484740</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>469872</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>536658</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>495611</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>522725</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>567174</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>544308</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Average</td>
<td>525353</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

In the heuristic based approach we first had to come up with a heuristic for the VCL log files that was consistent. The only heuristic we could come up with was that any integer was a parameter. The heuristics that Jiang et al. [22] used in their case studies were words that followed an ‘=’ symbol, email addresses and file names with known extension types. In our case there were integers that didn’t follow an ‘=’ symbol too. So by abstracting all integers we ensured as much abstraction as possible. Hence, we abstracted out the integers from the messages. The messages thus had words and each number in them was replaced by the character ‘*’ to create the abstracted message. The result of this comparison can be see in Table 4.4.
Table 4.5: Accuracy of Empirical Abstraction Technique

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>Number of Abstractable Events</th>
<th>% of Misclassified Events</th>
<th>% of Misclassified Event Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>556054</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>551034</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>484740</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>469872</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>536658</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>495611</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>522725</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>567174</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>544308</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>Average</td>
<td>525353</td>
<td>7</td>
<td>18</td>
</tr>
</tbody>
</table>

In the empirical approach we applied our algorithm to the messages in the log file and abstracted them. The results of how well our approach as a stand alone technique performed can be seen in Table 4.5.

Table 4.6: Accuracy of Two Step Hybrid Abstraction Technique

<table>
<thead>
<tr>
<th>Log File Name</th>
<th>Number of Abstractable Events</th>
<th>% of Misclassified Events</th>
<th>% of Misclassified Event Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcld.log.1</td>
<td>556054</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>vcld.log.2</td>
<td>551034</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>vcld.log.3</td>
<td>484740</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>vcld.log.4</td>
<td>469872</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>vcld.log.5</td>
<td>536658</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>vcld.log.6</td>
<td>495611</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>vcld.log.7</td>
<td>522725</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>vcld.log.8</td>
<td>567174</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>vcld.log.9</td>
<td>544308</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Average</td>
<td>525353</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

Finally in the two step hybrid approach, we first abstract using the heuristic approach. In the second step we take the output of the first step and apply the empirical approach on it to further abstract the message. We compared the Event IDs against the output from the second step. The results of this comparison can be seen in Table 4.6.
In the three tables the first column has the log file name, the second column has the number of messages/events in each log file that we wanted to abstract to an event. The third column has the percentage of these messages/events that were misclassified. Finally the fourth column has the percentage of event categories (or event types) that were misclassified.

We provide the following example to illustrate what we mean by misclassified events and misclassified event categories. Suppose we have following 5 lines in a large log file

Move data from 127.0.0.1 to 127.0.0.2
Move data from 127.0.0.3 to 127.0.0.4
Move data from Server 192.0.0.1 to Server 192.0.0.2
Move data from Server 192.0.0.3 to Server 192.0.0.4
Move data from Server 192.0.0.5 to Server 192.0.0.6

The first two log lines (messages/events) will be abstracted to the event category “Move data from * to *”. The next 3 log lines (messages/events) should get abstracted to the event category “Move data from Server * to Server *”. But instead if these 3 log lines (messages/events) also get abstracted to the event category “Move data from * to *”, then we have three messages/events that have been misclassified whereas only one event category that has been misclassified as another one.

Hence the third column in the table is the ratio of the total messages/events in the log file that were misclassified to the total messages/events in the log file that we wanted to abstract. The fourth column in the table is the ratio of the total event categories of the application that were misclassified to the total event categories that are available in the application. We express both these ratios as percentages in the table. Since the total number of event categories in an application (hundreds/thousands) are much lesser than the number of abstractable messages/events in a log file of that application (from hundreds of thousands to millions), the ratio of misclassified event categories are larger than the ratio of the misclassified messages/events themselves.

Table 4.7: Summary of Heuristic, Empirical, and Hybrid Approaches

<table>
<thead>
<tr>
<th>Abstraction Technique</th>
<th>Number of Abstractable Events</th>
<th>% of Misclassified Events</th>
<th>% of Misclassified Event Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic</td>
<td>525353</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Empirical</td>
<td>525353</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>Hybrid</td>
<td>525353</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
We have summarized the data from Tables 4.4, 4.5, 4.6 into the Table 4.7. As we can see in this table the best accuracy is in the two step hybrid approach. With just a heuristic approach we were able to abstract 95% of the messages to events accurately and about 88% of the event categories were classified correctly. With the empirical approach we see that only 93% of the events in the log file on an average were classified accurately and only 82% of the event categories were identified correctly.

We are able to detect events that occur many hundreds to times to ones that occur just twice. We however cannot detect the message and parameter fields in a log line that has occurred just once. But since it has occurred only once it is unique in itself and hence abstraction is not necessary. Also if the variable value is the same in most of the occurrences of that event, then it will not be detected as one. But the frequency of the variables must be very high for this to happen. In the case of the pure empirical approach the integers in the messages (IP addresses and thread IDs and reservation IDs) biased the frequency counts and hence the abstraction performed poorly. The integer parameter values repeated, but not as often as the string parameter values.

In the two step hybrid approach where we abstracted the integers in the messages and then applied the empirical approach, the accuracy was the best. On an average across the 9 log files, we had a 97% accuracy among the events, and a 93% accuracy among the event types. Thus a hybrid approach of applying the heuristic technique to do the first level of abstraction and perform an empirical analysis to do the second level of abstraction works the best in case of the VCL log files.

4.4 Complexity Analysis

In this section we present the complexity analysis of the heuristic and empirical approaches since both are used in the two step hybrid approach. Let the log file have \( n \) log lines that we want to abstract. Let each of these lines have \( m \) words in them. Therefore the number of words in the log file is, say, \( N = n \cdot m \). Thus at the least we need to inspect each word at least once. Thus a linearly scaling algorithm would have a complexity of \( O(N) \).

In the heuristic approach we have a predetermined constant set of heuristics. Let the number of heuristics be \( k \). In our case study we had only one, namely all integers were to be abstracted as parameters, and hence \( k \) in our example would be equal to one. In this algorithm we would examine each of the \( N \) words against each of the \( k \) heuristics to decide if we wanted to leave it as is or abstract it as a parameter. Thus the complexity of this algorithm in \( O(kN) \). But since \( k \) is a constant for the application, the complexity is \( O(N) \).

In the empirical approach we have to make three passes through the \( N \) words of the log file. In the first pass we build the frequency table. The frequency table has a row for each unique
word in the log file. Let the number of unique words in the log file be $p$. Hence there are $p$ rows in the frequency table. Now we need to be able to store the frequency of each word in each possible position in a given log line. Since we assumed that the number of words in each line was $m$, there are $m$ columns in the frequency table. Hence the frequency table is of size $p \times m$. In order to know which row corresponds to which word, we maintain a dictionary of the unique words. We implement this as a (key,value) hash map. The key here is the word, and the value is the row number in the frequency table.

So in the first we examine each word in each log line. Hence we examine each of the $N$ words once. When we examine each word we increment a single cell in the $p \times m$ frequency table. Let this cell be the one at $(i, j)$. This implies that the word we are examining is the $i^{th}$ row in the frequency table. We get the value of $i$, by searching for the word in the hash map dictionary. Searching for a value that corresponds to a given key in hash map can be done in logarithmic time, i.e. in our case $\log(p)$. Since $p$ is constant for an application and does not vary with the log files, or their size, we can assume that $\log(p) = C_1$, a constant, albeit a large one. In the case of VCL the value of $C_1 = 10$. Once we know the value $i$ and get the position in that log line $j$, we can access the frequency table in constant time, $C_2$, since it is just a double dimensional array of integers. Thus the time to access a value in the table is constant $C_1 + C_2$. Once we have reached this cell, we can increment its current value in constant time too. Since every action for each word can be done in constant time in the first pass, the time complexity of the first pass to examine each of the $N$ words in the log file and build the frequency table, is $O(N)$.

In the second pass, we examine each of the $N$ words and extract their positional frequencies from the frequency table. Previously we have established that accessing a particular cell in the frequency table is accomplished in constant time. Hence getting the frequency of a particular word in a given log line in a position in that log line can be achieved in constant time. Since we need to get the frequency of each of the $N$ words in the log file, the second pass also takes $O(N)$ time.

In the third pass we calculate the average of the frequencies of the words in a given log line, which can be done in constant time. Then we compare each word against this average. This takes a constant time for each word. Since we have to do this comparison for each of the $N$ words, the third pass also takes $O(N)$ time.

Hence since all the three passes take $O(N)$ time, we can conclude that the time complexity of the empirical approach is $O(N)$. Since the hybrid approach involves the heuristic approach ($O(N)$ time) first, and the empirical approach ($O(N)$) next, the time complexity of the hybrid approach is $O(N)$ or linear as well.
4.5 Conclusion

Log files contain a lot of information in them and it is often necessary to use an automated analyses technique to mine this information. But the log files have an inherent variability due to the entangling of constant message types and variable parameter types. Hence it is essential for us to abstract the messages in the log file to event types. In the literature there are numerous techniques for log file abstraction. But each of them have their own assumptions like, access to source code [45] or ability to identify parameters using heuristics [22]. These assumptions hold true in a lot of cases and when they do these techniques provide the best and most accurate results. But in the many cases where these assumptions don’t hold true, a more approximate algorithm is required. In this paper we present an approach to log file abstraction that is similar to the SLCT tool [41]. We cluster similar frequency words in each line and abstract it to event types. Our approach is however an approximate approach. In our case study of 9 VCL log files we observed that a hybrid of the heuristic and empirical approaches performed the best. we were able to abstract on an average 97% of the events in the log files accurately. And unlike the regular expressions from source code based technique this approach has time complexity of $O(N)$ and hence can scale well with large log files.
Chapter 5

Operational Profiling Using Suffix Arrays

5.1 Introduction

In his work, Musa suggests that operational profiles can guide the allocation of resources to improve the software reliability and speed of development [28]. The expected operational profile for an average sized project (10 developers, 100KLoC, 18 months development time) may take about one person month to create from scratch (requirements, end-user interviews, designs, etc.). Typically operational profiles are created from the software requirements and through customer reviews. This usually requires extensive human effort and takes a long time. For an appropriately chosen operational profile, the benefit-to-cost ratio can be 10 or greater [32].

In studies of operational profiles, most existing efforts concentrate on frequently occurring patterns for identifying frequently used functions and frequent errors. Our approach identifies frequent patterns as well as infrequent ones because the rare patterns could indicate important events as well. For example, a rare event may indicate a combination that the designer of software did not anticipate and should be disabled or handled differently. A rare event may also point to a potential security hole or anomalies in design or implementation with severe consequences. Therefore, computing these rare event sequences can be an important part of an operational profile.

Most systems log their actions for a variety of purposes. Often in commercial software systems, they are used to assess Quality of Service for the customers [35, 39]. Some systems in the financial sector, primarily log their actions for auditing purposes. Systems in telecommunications and financial companies are now required by the Sarbanes-Oxley Act of 2002 [6] to log their actions. Similarly HIPAA [1] requires health care systems to log their transactions.
These execution logs, which have to be collected anyway for a combination of reasons, contain the actual usage information, and can be used to construct operational profiles.

The problem with using log files for deriving operational profiles is the large volume of the data that needs to be analyzed. For example, the size of a log file from a telecom application can be over 100MB, and have about one million events collected over a 24 hour period [19]. Hassan et al. report getting few of the most frequently occurring scenarios within 2-3 hours with some human intervention [19]. Vaarandi, on the other hand, used a clustering algorithm to get the patterns from event logs [41].

Finding the frequency of a particular event can be done by simply scanning the log file. But the length of sequences of events could vary from 1 to $N/2$, where $N$ is the total number of events in the log file (The maximum length is $N/2$, because if it longer than that, then it cannot be be a sequence that is repeated). A brute force solution to determine the frequency of all the patterns would require an exponential ($N!$) order of time. In this paper we introduce a new approach for construction of second generation operational profile in an automated way in $O(kN)$ time, where $k$ is the average frequency of the sequence of events. We define second generation operational profiles as, those operational profiles that are derived using the actual usage of a software system in a production environment. We use data structures called Suffix Arrays (SA) and Longest Common Prefix (LCP) introduced in [25] for our solution.

In practical cases it is likely that $k$ has an average value that is much smaller than the value of $N$ and is independent of $N$. This implies that complexity of our algorithm is of the order $O(N)$. Given that every one of the $N$ events needs to be examined by any pattern discovery algorithm, $O(N)$ should be regarded as the optimal complexity.

We evaluated our algorithm using logs from a cloud computing environment in production at North Carolina State University since 2004 [10], [11].

5.1.1 Contributions

Principle contributions of this paper are

1. A novel approach for construction of second generation operational profiles from execution logs. Our approach is based on the use of Suffix Arrays that are extensively used in bioinformatics domain [25] and in analyzing whole program paths for software optimizations [36], but have not been used to construct operational profiles.

2. A solution that finds the occurrence probability of both single events and sequences of events - from those with high occurrence probability to those with a low occurrence probability. Most other solutions provide only the probability of the most frequently occurring events [19].
Table 5.1: Example: SA of the text "abracadabra" and the LCP

<table>
<thead>
<tr>
<th>i</th>
<th>SA(i)</th>
<th>Suffix</th>
<th>LCP(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>abra</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>abracadabra</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>acadabra</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>adabra</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>bra</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>bracadabra</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>cadabra</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>dabra</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>ra</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>racadabra</td>
<td>2</td>
</tr>
</tbody>
</table>

3. An automated solution that does not require users to tune any parameters.

5.2 Suffix Arrays

Manber and Myers invented the suffix array (SA) to identify all possible occurrences of a pattern in a text [25]. A SA was built to improve the space and time efficiency of suffix trees. The SA of a given string A, is defined as the lexicographical ordering of all the suffixes of A. Suffixes of a string, say “abcd” are the strings, “abcd”, “bcd”, “cd”, and “d” and prefixes for it are “abcd”, “abc”, “ab”, and “a”. Table 5.1, illustrates the suffix array data structure with an example. The third column contains all the possible suffixes of the string “abracadabra”, which we will call A. The second column is the suffix array for A. The suffix “a” would be the first string in the lexicographical ordering. The index position at which “a” is found in the string A is the value that is stored in the suffix array, here the value being 11. The LCP is an array that is as big as the SA and stores the length of the longest common prefixes between adjacent strings in the SA. For example in Table 5.1, we see that the second and third suffixes are “abra” and “abracadabra”. The longest common prefix between these two strings is “abra”, the length of which is four. Hence we can see the value of 4 in the LCP array at the position corresponding to “abracadabra”. Thus if the LCP value for a suffix is $k \ (\geq 0)$, then the first $k$ characters in
that particular suffix is a pattern that occurs at least twice in the text. This is the property that we will exploit in our approach. In our approach we define a repeating pattern as a subsequence of events that appears more than once in the given sequence of events. Since some of these repeating patterns may be contained in others, we will concentrate on those longest ones called the maximal-length repeating patterns, or the maximal patterns for short. Given a set of repeating patterns that occur in the same locations in a given sequence, the longest pattern is called the maximal-length repeating pattern.

Manber and Myers state that one of the basic applications of SA and LCP is to search all the instances of a pattern in a text [25]. This is because all similar patterns are clustered together in the lexicographically sorted SA. Therefore we need to search only for the first occurrence of a pattern in the SA. When \( N \), the length of the text, is large as in the case of human genome or as in our case of huge log files and the text remains constant, the search using SA and LCP of the text is better than other techniques. Since a small change to the text \( A \) may require SA and LCP to be constructed from scratch, SA and LCP are generally used for text that does not change. Some of the new algorithms can construct the SA in \( O(N) \) time in the worst case [24, 46], and can construct the LCP in \( O(N) \) time [23]. The search for the pattern is then a simple augmentation to binary search and we can find the position of all the \( z \) occurrences, of a pattern of length \( P \), in the text of length \( N \), in \( O(P + \log N + z) \) time.

5.3 Our Approach

Fig. 5.1 illustrates our approach to get the operational profile from the execution logs. It consists of three steps, viz. Log abstraction, construction of the data structures SA and LCP, and finding the patterns.

5.3.1 Log Abstraction

Abstracting the log lines to an event type is an important preprocessing step. An example of log abstraction procedure was proposed by Jiang et al. recently [21]. Tools like SEC [40], an event correlation tool or Swatch [18], a log monitoring tool can also be used to abstract the event identifiers from the log files. Both of these are rule based, and hence light weight and easy to use. Xu et al. [45] however, used source code analysis is done to identify the statements that print to the log file. In our approach we used an abstraction technique similar to the one used by them.

The first step is to identify the function (method) that is called to print the log messages in the log file. To do this we randomly select some lines, say a hundred, from the log files. We identify the message type in these log lines. Then we search for the message types in each
of these log lines, in the source code using the search function of a text editor. Thus we find
the method used to print these statements into the log file. In our case study, a method called
\texttt{notify($error, $LOG, $data)} was used to print to the log file. Then we extract all the calls
to this particular method. For example

(1) \texttt{notify($ERRORS{\textquoteleft OK\textquoteleft}), 0, "$node ssh port $port open"};
(2) \texttt{notify($ERRORS{\textquoteleft WARNING\textquoteleft}), 0, "failed to run newsid.exe on
$computer_node_name, exit status: $newsid_exit_status,output: $newsid_output"};

The parameter which holds the data printed in the log file is extracted. The data is a string
with constants and variables. In our case study, the third parameter called $data holds this
string value. We build a regular expression with this parameter. In the regular expression
we maintain the constant part of the string, but replace the variable with a ‘(.)+’.
Also we assign a unique integer to each regular expression. These steps have to be done only once for
an application. The regular expressions and the corresponding integers are the same for all the
log files of a particular application. Examples of regular expression from the case study are

1: ssh port (.)+ open$
2: failed to run newsid.exe on (.)+ exit status: (.)+,output:(.)+$
A particular line in the log file will match with at least one of these regular expressions. When more than one match occurs we choose the match with the regular expression of greater length. This matching is done using the boost-regex library available in C++ distributions. We replace each log line by the unique integer corresponding to the matching regular expression. The integer representation of the events in the logs is easier to manipulate using SA and LCP than the actual names of the events. Since the abstraction of a line of log is independent of the abstraction of another line, it is a problem with an embarrassingly parallel solution. In our case study a 100 MB log file was abstracted to its integer equivalent in 99.64 minutes. This was done on an 8 core system running CentOS with each core being a Intel(R) Xeon(R) 2.00 GHz CPU, and 2 GB memory. We used the \#pragma omp parallel for num_threads(6) directive to run the abstraction on 6 parallel threads.

### 5.3.2 Construct SA and LCP

We build the SA and LCP of the integer representation of log events using O(N) algorithms. Many such algorithms are known. In our work we used the suffix array construction algorithm proposed by Zhang and Nong [46] and LCP construction algorithm by Kasai et al. [23] respectively. An example of SA and LCP for a sample string is illustrated in Table 5.1. Note that a suffix array represents a sorted version of all subsequences that appear in the events extracted from the log files; and LCP (longest common prefix) counts how many events are common to two neighboring sequences in the sorted list.

### 5.3.3 Find Patterns

We use the LCP to identify all the possible patterns, including single length ones, in the list of events gathered from the log file. Recall that the LCP array contains information about how many events two neighboring sequences in sorted order have in common, it is straightforward to count how many events are shared among more neighboring sequences as illustrated in Fig. 5.2. In our approach the log events are replaced by integer indexes as explained in Section 4.1. In the example though, the suffixes we consider are letters of the English alphabet for ease of understanding.

Consider row 162, with value \((36, \text{QWETU}, 3)\), which is in bold font in the figure. We use the term LCP value for a particular integer in the LCP array. Each LCP value greater than zero indicates a pattern that has repeated at least once. To calculate the frequency count of the pattern we examine the LCP values after and before the element in the LCP array currently under examination, in that order. We count the number of LCP values after the current element in the LCP array under examination, till we reach a LCP value less than it. Then we count the number of LCP values before the current element in the LCP array under examination, till
we reach a LCP value less than it. In the latter case we compare the absolute values instead of the actual values in the LCP array. The aggregate of these two counts + 2 is the frequency count of the pattern.

In our example, the LCP value at position 162, is three. Since three is greater than zero, we count the number of LCP values after this entry till we reach a value less than three. In this example there are two such entries, namely four, and three. The count now is therefore two. At index 164 when the LCP value is equal to the LCP at the current index 162, namely three, we change the LCP value at index 164 to negative three. This is done to indicate that the pattern at index 164 of length three is a repetition of the current pattern at index 162, namely ‘QWE’ and that we should avoid double counting of the patterns. We now compare the LCP value at current index 162 with the absolute value of the LCP value before the current position. We find two entries, namely four and five before we reach two, a value less than the current LCP value. The count therefore becomes four. Therefore in our example the frequency of the pattern at the current position is four plus two, i.e six. The pattern itself begins at the first character of the current suffix and is of length equal to the current LCP value. In our example the first character at the current position is ‘Q’. The LCP value is three. Therefore length of pattern is three. Hence the pattern in this case would be ‘QWE’. Thus we have found out that the pattern ‘QWE’ occurs six times in the text. In our approach we get a sequence of integers as the pattern, and therefore will have to transform the integers to actual event names using the correspondence between the integers and the regular expressions from the first step explained in Section 4.1.

Thus, by exploiting the unique properties of SA and LCP, originally used to detect all occurrences of the given pattern in text, we detect all the possible clustered sequence of events

---

**Figure 5.2: Operational Profile from LCP and SA**

<table>
<thead>
<tr>
<th>i</th>
<th>SA(i)</th>
<th>Suffix</th>
<th>LCP(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>156</td>
<td>30</td>
<td>MCA...</td>
<td>1</td>
</tr>
<tr>
<td>157</td>
<td>3</td>
<td>QWCA...</td>
<td>0</td>
</tr>
<tr>
<td>158</td>
<td>15</td>
<td>QWBO...</td>
<td>4</td>
</tr>
<tr>
<td>159</td>
<td>94</td>
<td>QWERTY...</td>
<td>2</td>
</tr>
<tr>
<td>160</td>
<td>56</td>
<td>QWERT...</td>
<td>5</td>
</tr>
<tr>
<td>161</td>
<td>72</td>
<td>QWERK...</td>
<td>4</td>
</tr>
<tr>
<td>162</td>
<td>36</td>
<td>QWETU...</td>
<td>3</td>
</tr>
<tr>
<td>163</td>
<td>47</td>
<td>QWETX...</td>
<td>4</td>
</tr>
<tr>
<td>164</td>
<td>24</td>
<td>QWEX...</td>
<td>3</td>
</tr>
<tr>
<td>165</td>
<td>81</td>
<td>QWMBG...</td>
<td>2</td>
</tr>
<tr>
<td>166</td>
<td>65</td>
<td>QWMBE...</td>
<td>4</td>
</tr>
<tr>
<td>167</td>
<td>89</td>
<td>RCA...</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Note: In the Suffix column, we show only the prefixes of the suffix and not the entire suffix.
(patterns) and the number of times each occurs in the log file (the text), and hence build the operational profile.

5.4 Algorithm Analysis

5.4.1 Complexity Analysis

Let $N$ denote the number of lines in a input log file to be processed. We can compute the cost of the each step of our approach as follows.

Log Abstraction: We read the log file line by line and store it in a vector. We replace each log line by an integer finally. Let the number of lines in the given log file be $N$. Let the application have $M$ regular expressions that we will use to abstract the $N$ log lines. We read the log file in chunks of $n$ lines at a time ($n < N$) so that we can optimize on space required, when log files are very large (i.e. $N$ is very large). We also read the regular expressions for the application into vector of size $M$. To read the entire log file and the regular expressions we need $O(N + M)$ in time and $O(n + M)$ in space at any given time. Since $M$ is constant for a particular application and if it is significantly smaller than $N$ (in our case study $N = 6,195,200$, and $M = 2629$), the time and space complexity for the read operation will be $O(N)$. Value of $M$ differs from application to application, but in each application it remains a constant irrespective of the growth of the value $N$ for that application.

Below is the pseudocode for the abstraction algorithm.

1: for each line in log_file:
2:     for each regex in regex_file:
3:         if match(line, regex) != 0
4:             return regex_id
5:     endif
6: end for
7: end for

The match(line, regex) method in line 3 does the matching between the line of text in the log file of length $n$, and the regular expression of length $m$. The time complexity of this step is $O(m + n)$. The value $K = m + n$ is not related to $N$. But the maximum value of $K$ is constant for a particular application. The inner for loop executes $M$ times and the outer for loop executes $N$ times. Hence the time complexity of this step is $O(KMN)$. As $K$ and $M$ are constant for a particular application, the value of $N$ is the only value that grows in an application. Therefore in O-notation, the time complexity of this step is $O(N)$. If $K$ or $M$ is a large constant then performance degradation can occur. In our case study, the value of
\(K\) was never over 700 and the value of \(M\) was 2629 both of which are much smaller than the maximum value of \(N\) which was 6,195,200. The output is an array of integers one for each line on the log file. Thus the space required for this step is \(O(N + M + N)\) for the input vector, regular expression vector and output array respectively. Since \(M\) can be considered a constant with respect to the application, the space complexity is also \(O(N)\).

**Construction of SA and LCP**: In this step we construct the SA and the LCP array using linear time algorithms. Such algorithms require both \(O(N)\) space and time[46, 23]. For example, the suffix array construction algorithm proposed by Zhang and Nong recursively reduces the problem size at least by a half in each iteration and each iteration performs \(O(N)\) work, therefore, it requires \(O(N)\) time overall. In addition, it reuses the same workspace of \(O(N)\) size at different iterations, thus it uses \(O(N)\) space altogether as well. The LCP array construction algorithm by Kasai et al. [23] passes through SA array once without any additional storage. Therefore, both SA and LCP can be constructed with \(O(N)\) storage.

**Finding Patterns**: In this step we count the number of patterns using the LCP array. For each element of LCP array, \(LCP(i)\), our algorithm looks forward and backward as follows.

```plaintext
1: for each i less than N
2: Initialize count to 0
3: if the position at \(lcp[i]\) is not marked for repetition then
4: j = i
5: while \(lcp[i] \leq lcp[j + 1]\)
6: increment count
7: if \((lcp[i] == lcp[j+1])\)
8: mark position \(lcp[j+1]\) to indicate repetition
9: endif
10: increment j
11: end while
12: j = i
13: while \(lcp[i] \leq lcp[j - 1]\)
14: increment count
15: decrement j
16: end while
17: endif
18: end for
```

The ‘for loop’ from lines 1 - 18 is used to find the count of all similar patterns and line 2 initializes the count for each pattern to zero. Lines 3 - 17 is an ‘if block’ that is executed only if the current index position \(i\) in the LCP array is not marked for repetition. Lines 4 - 11 of
the algorithm is a ‘while loop’ that is used to scan lcp values after current index, \(i\). Lines 7 - 9 is used to mark the positions in the lcp array indicating a repetition. Line 12 reinitializes the moving index \(j\) back to the value of the current ‘for loop’ index \(i\). Then the ‘while loop’ in lines 13 - 16 is used to scan lcp values before current index, \(i\).

In this procedure, we need to examine no more than \(k + 1\) LCP values at a cost of \(O(k)\), where \(k\) is the average frequency of the sequence of events. The above procedure identifies \(k\) neighboring suffixes with LCP(\(i\)) leading events in common. Note that LCP(\(i\)) is the maximum number of events the suffix \(i\) has in common with the suffix \((i - 1)\). If the suffix \(i\) has more events in common with any other pattern, that pattern must involve the suffix \((i + 1)\) and such a pattern would be counted in the next step.

Let \(K\) denote the maximum value of \(k\), then the total cost of this step is \(O(KN)\). A more accurate bound on the cost is \(O(\bar{k}N)\), with \(\bar{k}\) being the average number of occurrences of a pattern. For a fixed set of possible events, the value of \(\bar{k}\) of a random sequence of these events will increase as a linear function of \(\log(N)\). However, for practical applications, we postulate that \(\bar{k}\) is approximately constant - it does not vary much with \(N\), and is much smaller than \(N\) for large \(N\). In our case studies we found that \(k\) was on the average equal to 5 whereas \(N\) was as large as 6,195,200. If \(\bar{k}\) is much less than \(N\), then the time complexity of this step is \(O(N)\).

This step produces an array with \(N\) elements, each representing the number of occurrences of the longest repeating pattern starting with the event at \(SA(i)\). Therefore the space requirement is \(O(N)\).

Time and space complexity of our approach is the sum of the time complexity of the steps above. Hence it is \(O(N + N + N)\) which is \(O(N)\).

### 5.4.2 Correctness

Since we identify patterns with LCP, our algorithm identifies all maximal patterns in the log files. Recall that a maximal pattern is the longest repeating pattern. In the example given in Fig. 5.2, the shortest repeating pattern starting with 'Q' is 'QW' which repeats 10 times. The pattern 'QW' is a maximal pattern because it also contains a shorter pattern involving 'Q' only. We say pattern 'Q' is covered by 'QW'. The pattern starting with 'W' will appear later in the suffix array and would include the 10 occurrences already shown but may have other occurrences as well. Therefore, pattern 'W' is not covered by 'QW'. In general, any prefix of a maximal pattern will appear at least as many times as the maximum pattern itself. If it only appears as many times as the maximal pattern, it will not be counted separately in our approach. If it appears more times, then it will be counted separately or counted as part of another maximal pattern. In the same example, the length 2 pattern 'QW' appears more times than any of the length 3 patterns starting with 'QW', therefore, it is counted separately.
To see that our approach actually counts all repeating maximal patterns, we observe that the suffix array sorts all $N$ suffixes of a list of $N$ events. Any subsequence appeared in the log file has a chance to be the prefix of one such suffixes. In this sorted list of suffixes, any repeating patterns will appear next to each other by construction [25]. The LCP array records the length of the longest-common prefixes, i.e., the maximum number of common events in two neighboring sequences in the sorted order. The ‘Find Patterns’ algorithm in step 3 of our approach explained in section 4.3 identifies the maximal repeating patterns as explained. In addition, it also counts the number of occurrences of each maximal pattern.

5.5 Results

We explore further using an implementation of our approach. The machine used for testing was a Intel(R) Xeon(R) 2.00 GHz CPU, with 2 MB of cache and 2GB of RAM, running CentOS 5.2. The implementation was written in C/C++ and compiled with the 4.1.2 release of the GCC compiler with the omit-frame-pointer flag turned on for optimization. We used log files from the Virtual Computing Lab (VCL) [10], [11], a cloud computing environment that reserves resources with the desired set of applications for remote access. VCL is written in several languages including Perl and has about 25 modules and more than 60,000 lines of code. Fig. 5.3 has example entries from VCL logs. The IP-Address was anonymised for security.

We tested our operational profile generation implementation on a sample log file. We manually verified the correctness of the results.

We wrote all the sequences of events, of varying lengths, into an ASCII file in sorted order according to their frequency of occurrence in the log file. For each of the log files we recorded some run time details like number of events in the log file, time taken for execution to complete (in seconds) and the percentage of time spent on input/output. The objectives were:

1. To experimentally demonstrate our claim of $O(N)$.

2. To discuss the operational profile with the technical lead of the VCL application for his interpretation of the operational profile.
Table 5.2: VCL Log Data. Total number of events covered is 5,500,000.

<table>
<thead>
<tr>
<th>No:of Events (100,000’s)</th>
<th>Find Patterns (secs)</th>
<th>$\bar{k}$</th>
<th>Total Time (Including I/O) (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.055805</td>
<td>2</td>
<td>1.14445</td>
</tr>
<tr>
<td>2</td>
<td>0.105715</td>
<td>3</td>
<td>2.34686</td>
</tr>
<tr>
<td>3</td>
<td>0.161791</td>
<td>4</td>
<td>4.31964</td>
</tr>
<tr>
<td>4</td>
<td>0.224572</td>
<td>4</td>
<td>6.66332</td>
</tr>
<tr>
<td>5</td>
<td>0.33063</td>
<td>4</td>
<td>10.0355</td>
</tr>
<tr>
<td>6</td>
<td>0.423281</td>
<td>4</td>
<td>13.8074</td>
</tr>
<tr>
<td>7</td>
<td>0.348127</td>
<td>3</td>
<td>17.0773</td>
</tr>
<tr>
<td>8</td>
<td>0.572854</td>
<td>4</td>
<td>22.7966</td>
</tr>
<tr>
<td>9</td>
<td>0.646797</td>
<td>4</td>
<td>26.073</td>
</tr>
<tr>
<td>10</td>
<td>0.754963</td>
<td>4</td>
<td>31.0628</td>
</tr>
</tbody>
</table>

5.5.1 VCL Logs

In this case study we used five different log files of sizes 332 MB, 306 MB, 273 MB, 192 MB, and 59 MB that spanned 4 weeks of operation of the NC State University Virtual Computing Laboratory (VCL) operation. The first 4 log files had more than a million events and the fifth one had 356,950 events. Table 5.2, shows the number of events in the sample, the time to find patterns, the value of $\bar{k}$, and the total time needed to calculate and output operational profile for that sample. We can see that the total execution time, including I/O is a lot more than the time needed to calculate the patterns, and that it grows rapidly with the number of events $N$. This is because I/O to disk slows down the process.

We considered 10 independent samples of VCL log files. One with 100,000 events, another with 200,000 events and so on, till 1 million log events were reached in increments of 100,000 events. Each of these 10 files were from different parts of the log files with no overlap between them.

In Fig. 5.4, we see the relationship between execution time and the number of events in each run. The $\bar{k}$ value of each data point is shown in the graph. The execution time increases approximately linearly with the number of events. We note that $\bar{k}$ is between 2 and 4. A drop in execution time occurs in the 700,000 events sample because its $\bar{k}$ value drops from 4 to 3.
In order to test if our approach works for very large files, we concatenated all five log files to get one very large log file of size more than 1GB with 6,195,200 events. The time taken to find all patterns in this file was 5.9457 seconds. The corresponding $k$ value was 5 and the total execution time including I/O was 718 seconds.

Second generation operational profiles can be used in a number of ways. One is to optimize the application and to find anomalies. For example, Table 5.3 gives us information on the top two operations in VCL. The values in the table are averages across the 5 files. The entire ASCII file with all the identified event sequences, the lengths of these sequences and frequency of occurrence in the log file, sorted in descending order of their frequencies, was shown to the technical lead of the VCL application. The entire interview, including the identification of sequences and the following discussion on the significance of these event sequences lasted for 30 minutes. He used the ‘less’ and ‘tail’ command, available in the Linux OS, to view the most frequent and least frequent event sequences from the ASCII file. From the list of the most frequent event sequences in the beginning of the file, he identified the following
• ‘Load OS Module’ : This was expected to be one of the most frequent operations and it showed up in the top of the list. Every time a VCL reservation is made the ‘Load OS Module’ operation is performed.

• ‘Database Access’ : It was interesting to the technical lead to know that the ‘Database Access’ was also among the most frequent operations. This operation polls the status of a VCL reservation and updates the database and is non critical. As a result of this study they can reduce the number of queries to the database from 15.73% to a more efficient value.

We also discussed the least frequently occurring events that were found at the end of the results file. These were the events that occurred only once in the log files. These events can be interesting because they may indicate anomalies. Two of these events that the technical lead described as worth additional attention were.

1. neighbor request (.)+state set to deleted$ : This is an event sequence of length one. This occurs when a machine that already has an image loaded is assigned to load another image. Normally that should not happen.

2. rpower status is not on, node needs to be reloaded$ : This is an event sequence of length one. This means that the target machine has been powered off. This message indicates that the loader was not informed that the resource was unavailable prior to attempting to load the image. Perhaps an error in the scheduling algorithm?

It was pleasing to see that this second generation operational profile was found useful as a diagnostic tool that can help solve problems and optimize production codes. We plan to package our tool and make it available to the open source community.

5.5.2 Discussion

 Experimental results shown in Fig. 5.4 are consistent with the Section 5.1 O(N) analysis of the complexity of our operational profile computation algorithm. It is interesting to note that Hassan et al.’s approach requires 2-3 hours to find the first pattern in a log file with 1,152,049 log lines and of size 137 MB. In our approach we can find all the patterns in a log file of size 1GB with 6,195,200 events in about 718 seconds after the log abstraction has been done. A direct comparison can not be made due to lack of information about the computer that Hassan et al. used. However, it is worth noting that our log abstraction method is embarrassingly parallel and a very high speed up can be achieved.

 Automation provides considerable advantage in the case of our solution. For example, in comparison Vaarandi’s approach the support threshold parameter has to be tuned by trial and
error. Also the pattern detected is only a frequency count of individual log lines. Patterns of a combination of log lines are not be detected by Vaarandi’s approach. In Hassan et al.’s approach human intervention is needed to filter out the first pattern before the next pattern in the log file can be detected. The authors claim that for their approach to work, a human has to peruse less than 1% of the log files. But this amounts to almost 10,000 lines when a log file of a million or more events is considered. This can be very difficult for the human. We overcome both these limitations. Every pattern irrespective of its length and frequency is detected. Also there is no human involved until the extracted patterns have to be interpreted. At that stage the human merely looks at less than a few hundred log lines. Also in the case study conducted by Vaarandi[41], 181 clusters were found and in the case study conducted by Hassan et al.[19] we found. These are the most frequently occurring patterns. In our approach we identified and calculated the frequency of all patterns and arranged them in sorted order. Thus the most frequently occurring pattern as well as the least frequently occurring pattern can be analyzed simultaneously. By altering the condition for the sort routine from ‘pattern frequency’ to ‘pattern length’ or from descending to ascending order or any combination thereof, we can ensure that the patterns that we care for the most are at the top of the output.

Limitations: However, even our approach has some remaining limitations. For example, a software system developer still needs to look into the operational profile output to determine what each sequence signifies. This human inspection to identify what the patterns mean, takes time, and cannot be avoided with any method. In other solutions, a human will have to peruse the log file to find the sequence of events, before the operational profile can be derived. In our approach though, the operational profile is available and the human is needed only to identify what a sequence of events means. Our approach (as do all automated approaches based on logs) is typically applicable only to second and higher releases of a product, i.e., after the logs have been obtained. We could also use the log files collected during beta-testing as they will contain user usage information. Thus we could use our approach to derive an operational profile to prioritize the issues in the first release of the product. The operational profile derived by our approach is based solely on the logs. The accuracy of the operational profile is therefore directly related to the accuracy of the logging facility for that application.

5.6 Conclusion

Operational profiles are important software reliability engineering tools. In this paper we have presented an automated approach for creating second generation operational profiles using execution logs of a software product. The tool we built to do that is simple (about 800 lines of code) but very effective. Our algorithm parses the execution logs into sequences of events and produces an ordered list of all possible subsequences by constructing a suffix-array of the
events. The difficulty in using execution logs is that the amount of data that needs to be analyzed is often extremely large (more than a million records per day in many applications). Our approach is very efficient. We show that our approach requires $O(N)$ in space and time to discover all possible patterns in $N$ events. We used an implementation of the algorithm in the context of the logs from a large cloud computing system. We were able to build a log-based operational profile in this case study in under a few seconds. When compared to 2-3 hours [19], or one staff month [28] needed by some other approaches for much smaller logs than the ones we used, this can be a significant savings in time. In our approach we collect the frequency of all possible event sequences, including the least frequent ones. From this we are able to construct comprehensive second generation operational profiles more efficiently. Such profiles can then be used as optimization and anomaly detection diagnostic tool, and for construction of regression testing suites.
Chapter 6

Operational Profiling using Weighted Directed Graphs

6.1 Introduction

Log files keep a record of what happened in a system. Software systems often write details about what event was completed (or not) in the log file. Most log files at least collect the following details: (a) The ‘log message’ which contains information about the particular execution instance of an event. The static part of the message is the same across multiple instances of the same event, whereas the dynamic part of the message may be the same/different between multiple instances of that event; (b) The system also collects the time at which this event was executed. Additionally some systems collect an identifier, to identify which event was executed, and record it in the log file.

The information collected in the log file is often used for diagnostic purposes. If a system failure occurs, the logs for that time period can be inspected to see which events were executed by the system, and what were the values for the dynamic information in those events. Since each log line can be traced back to a particular line of code where the method to log this information was called, we know what events were executed. From the dynamic part of the log line, we can determine values for variables in the code and the branches taken by that particular instance of execution. For these reasons the information in the log file is collected in a serially ordered flat text file. Thus a log file is a collection of log lines, with each of them having information about a single event, its time of execution and the dynamic parameter information. Note that each log line may not physically be in one line in the log file. It may span across multiple lines, but the format in which most systems collect information in a log file allows the distinction of two adjacent log lines. Hence we use the term log line in this paper as a unit of information in a log file.
Log files can also be used for operational profiling and detecting the root causes of anomalous system executions. In operational profiling, we determine how many times each subset of events occur in the log file. Anomaly detection involves, examining the alternate paths of execution, to determine what the root cause of the anomaly might be. The current serial ordering of events in a log file is very unintuitive to perform either of these analyses. When we open the log file in an editor, we can view maybe 60-70 lines at any one point. Repeating patterns might have hundreds or thousands of events between them. Hence it is difficult to compare, two sets of events that are separated by other events. For example if a specific sequence of 3 events occurs 10% of the times in a log file with 100,000 events, then that is 30,000 (3 x 10% of 100,000) events that we have to look through to make that conclusion. Manual inspection when the frequency is this high is inefficient. Also the remaining 70,000 events will be in between these repeating patterns.

In order to solve this issue, we propose a transformation of the serially ordered log file to a Weighted Directed Graph (WDG). This is similar to a finite state machine. The advantages of visualizing the log file as a graph are:

1. Only the information needed for the analysis is maintained and condensed into a compact view of the log file.
2. It is easier to identify patterns or anomalies in the graph.
3. It is easier to build analysis tools on the graph representation as opposed to the serial representation. For example, in operational profiling, all the we need to do is to use an already existing library to traverse the graph to determine which edges have a high or low frequency. We can highlight these edges and bring them to the attention of the decision maker.

6.1.1 Contributions

Our contributions in this research project are as follows:

- Developed the transformation of a log file with serial events to a WDG.
- Built the tools required for this transformation in C++. The graph is represented as an adjacency matrix. We can convert this matrix to a DOT file that can then be rendered as an image by the Graphviz tools [3].
- Built tools to analyze the WDG representation of the log file. We then present the results to the users by highlighting the areas of interest.
- Applied it on real log files from the Virtual Computing Lab [10], a cloud management application at North Carolina State University, to determine its operational profile.
6.2 Our Approach

In this section, we present our transformation of a log file with a set of serial events to a WDG. In the serial order each event is important as a stand alone event. But in the WDG representation, the importance shifts to adjacent pairs of events. We do this type of a transformation because we want to record the order in which events happened. Therefore each unique event in the log file is represented by a unique node in the WDG. An edge exists from one node(head) to another (tail) if there is an occurrence of the event representing the tail node immediately after the event representing the head node in the original log file. For example if event B follows event A in the log file, the there is directed edge from node A to node B in the WDG. The edges are labeled with the weights, namely the number of times this transition has occurred. For example if B occurs a hundred times after A in the log file, then the edge from node A to node B in the WDG is labeled with a weight = 100. Typically we keep track of the actual count when building the graph. When we render the graph we display the percentage value as the label. This percentage is with respect to the total number of transitions. We could also store the dynamic parameter information in each log line as a list along the edges. We now present the steps involved in this transformation.

1. Input: Original log file with $N$ log lines, Output: Adjacency matrix representation of the WDG form of the log file.

2. First pass through the N lines of the log file: Parse the log message in each log line of the log file as static event information (or event identifier) and dynamic parameter information [31]. Assign an identifier (ID) to each unique event (defined by the static information in the log message in that log line) in the log file, and leave the dynamic parameter information as it is. Simultaneously create and update an index file with the (ID, event) pairs as new ones are detected in the log file. Let the number of indexes in total be $M$.

3. Create the adjacency matrix $G$, an $M \times M$ matrix of label objects. Each object has two members (in our implementation, but it could be more than that), namely $count(Integer)$ and $param(ListOfStrings)$

4. Second pass through the N lines of the log file: If $i$ is the current line that we are inspecting, then find the event ID in line $i$ and $i + 1$. Then retrieve the object $O$, at $G[eventID(i)][eventID(i + 1)]$. Increment the integer $O.count$. Add the dynamic parameter information to list $O.param$.

5. Once the entire log file has been transformed into the adjacency matrix, we build the DOT file for the matrix. This can be viewed by the Graphviz tools [3].
Alternately, we could apply analysis algorithms, like the operational profiling algorithm (explained in detail in Section 2.2.1), on the adjacency matrix $G$, and highlight the results when the graph is rendered.

### 6.2.1 Example

We now present a small example to illustrate our transformation algorithm. Let the application from which we are collecting logs be used to move a file from one computer in the network to another computer. Before moving it splits the file into chunks of size 10 MB each in the source computer, moves each of these chunks and recombines these chunks into the original file at the destination computer. The log events for moving a file ‘A’ of size 50 MB is below.

Located File A on 127.0.0.1  
File A is 50 MB in size  
File A is split into 5 chunks  
Move chunk 1 from 127.0.0.1 to 127.0.0.2  
Chunk 1 moved from 127.0.0.1 to 127.0.0.2  
Move chunk 2 from 127.0.0.1 to 127.0.0.2  
Chunk 2 moved from 127.0.0.1 to 127.0.0.2  
Move chunk 3 from 127.0.0.1 to 127.0.0.2  
Error in moving chunk 3  
Retrying: Move chunk 3 from 127.0.0.1 to 127.0.0.2  
Chunk 3 moved from 127.0.0.1 to 127.0.0.2  
Move chunk 4 from 127.0.0.1 to 127.0.0.2  
Chunk 4 moved from 127.0.0.1 to 127.0.0.2  
Move chunk 5 from 127.0.0.1 to 127.0.0.2  
Chunk 5 moved from 127.0.0.1 to 127.0.0.2  
Combining 5 chunks in 127.0.0.2 to form file A  
Moved file A from 127.0.0.1 to 127.0.0.2

The index file for this part of the log file will be as follows:

Event ID: Event  
1: Located File * on *  
2: File * is * MB in size  
3: File * is split into * chunks  
4: Move chunk * from * to *  
5: Chunk * moved from * to *  
6: Error in moving chunk *
The events in the log file are mapped to their corresponding event IDs and abstracted as follows:

1: A, 127.0.0.1
2: A, 50
3: A, 5
4: 1, 127.0.0.1, 127.0.0.2
5: 1, 127.0.0.1, 127.0.0.2
4: 2, 127.0.0.1, 127.0.0.2
5: 2, 127.0.0.1, 127.0.0.2
4: 3, 127.0.0.1, 127.0.0.2
6: 3
7: 3, 127.0.0.1, 127.0.0.2
5: 3, 127.0.0.1, 127.0.0.2
4: 4, 127.0.0.1, 127.0.0.2
5: 4, 127.0.0.1, 127.0.0.2
4: 5, 127.0.0.1, 127.0.0.2
5: 5, 127.0.0.1, 127.0.0.2
8: 5, 127.0.0.2, A
9: A, 127.0.0.1, 127.0.0.2

The dynamic parameter information in each event is masked by the symbol ‘*’. The adjacency matrix $G[M][M]$ of the graph is shown in Table 6.1. The graph generated from this
transformation and rendered by the graphviz tool is shown in Fig. 6.1. Each node is a unique event. An edge between nodes 1 and 2 signifies that the event 2 (File A is 50 MB in size) appears after event 1 (Located File A on 127.0.0.1) in the log file. The labels on the edges have the actual count and the percentage with respect to the total number of transitions, namely 16. Hence the transitions that happen once have 6.25%, and the transition from node 4 to 5 and from node 5 to 4, have a percentage value of 25% since they occur four times each.

### 6.2.2 Operational Profiling

Operational profiling involves determining which sequences of actions are repeated many times (or few times). Operational profiles are often used to prioritize regression testing efforts. Using our adjacency matrix representation of the log file we can determine the operational profile of the system as shown in the steps below.

Suppose the number of lines in the log file is $N$ (implies we have $N - 1$ transitions in the graph), and we want to find the sequence of events that occur at least $X\%$ of the time then:

For $i < M - 1$ to $M$

For $j < M - 1$ to $M$

if $(G[i][j] * 100)/(N - 1) > X$ Then

Highlight the nodes $i$ and $j$ in the graph.

Therefore in our example, if we were to determine the top 20% of the event sequence combinations, then nodes 4 (Move chunk * from * to *) and 5 (Chunk * moved from * to *) are highlighted because they each occur 25% of the times. If we want to find the least frequent sequences then we use the condition, $(G[i][j] * 100)/(N - 1) < X$, in the if block. Similarly if we want to find the frequency of a given set of sequences, we can just iterate through the adjacency matrix and get the percentage of these transitions. Since each of these analyses requires the inspection of each element in the adjacency matrix, the order of time complexity is $O(M^2)$.
Since $M^2 < N$ (as explained in Section 2.2), the analysis is often of the order $O(N)$ (since the order to time complexity to build the graph is $O(N)$).

Hassan et.al. [19], and Nagappan et.al [33] propose other log file analysis approaches that build the operational profile of the system. Hassan et.al.’s [19] approach is not a fully automated approach (requiring manual intervention) and can determine only the most frequent set of events. Nagappan et.al.’s approach builds an operational profile that has the most frequent and least frequent sequences. However in order to do this, it determines the frequency of all set of events. This can get computationally prohibitive if the log file has many events or many non repeating set of events. Our approach on the other hand is fully automated, fast and can produce the results only for the query we asked. Thus we can create an operational profile for the system to get most frequent sequence of events, least frequent sequence of events, and the frequency of a specific sequence of events.

6.3 Results and Discussion

We performed a case study on a log file from Virtual Computing Lab [10], a cloud computing management application at North Carolina State University. VCL is a system used by more than 40,000 students and manages over 3000 processors. It is written in Perl and python and collects execution information in log files by calling a logging method written in Perl, with the log message that contains the event information. We used a log file that was over 800 MB in size having 4,728,176 events. This was log information collected over one week. The index file to this log file had 1002 events (implies there were 1002 unique events in the log file which had over 4.7 million log lines). The analysis was run on an eight core Intel Xeon CPU at 2 GHz, with 2 GB of memory. Our implementation was done in C++ and compiled with g++ version 4.1.2 with no compiler options. We also did not write a parallel implementation to use the multicore facility. We abstracted each log line in the file to one of these 1002 IDs. Then we build the adjacency matrix of the log file in 36 seconds. We then convert the graph to a DOT file and render it using Graphviz [3] tools. We don’t include the image for it in the paper due to space constraints. But this graph is definitely easier to explore than a 802 MB file.
Figure 6.2: Results (Use Case 1) from Operational Profiling the VCL system by building the WDG of its Log File
Figure 6.3: Results (Use Case 2) from Operational Profiling the VCL system by building the WDG of its Log File
Figure 6.4: Results (Use Case 3) from Operational Profiling the VCL system by building the WDG of its Log File
We then applied the operational profiling algorithm to the adjacency matrix with a threshold value of 1.2%. We chose the value of 1.2%, as it focused the analysis on the largest few sequences in the log file. The output of the operational profiling analysis can be see in Figs 6.2, 6.3, 6.4, 6.5. In the figure we have anonymized the individual events for non disclosure reasons. But in the following paragraph we explain what each use as a whole signifies.

We traced these few sequence of events back to where they originate in the source code. Then we determined what each of these use cases signify. Use Case 1 in Fig 6.2 refers to the sequence of events that checks if the current process can take control or if the process has to be forked. Use Case 2 in Fig 6.3 refers to the processing the reservation request and creating the reservation. Use Case 3 in Fig 6.4 refers to the initializations that have to be done before an image reservation request can be processed (constructor phase). Finally Use Case 4 in Fig 6.5 refers to the completion of a reservation request, where the image has already been loaded and the corresponding entries in the database are deleted.

Then we interviewed the developers of VCL to see verify the validity of the results. They confirmed our findings that these were expected to be the most frequent usage scenarios.

Given that there were 4.7 million events in total, this implies that these transitions occurred almost 56,400 times (1.2% of 4.7 million) each. Manually searching for them through the file and getting the frequency is extremely inefficient when the frequency is so high. Using the ‘search’ facility in an text editor can only get the frequency of single events. We detected four clusters of events as the most frequent sequences because in the log file that we analyzed they were the most frequent sequences. As we can see in each of the clusters we detected, they were multi event sequences and hence cannot be detected with just a search utility. Similarly if we want to detect the least frequently occurring events, then we would just extract the edges of the graph that occurred lesser than the given percentage threshold.

We were able to reduce the size of the log file from a 802 MB flat file to a very small adjacency matrix file of size 3.83 MB. This is because we condense and collapse all redundant information to a more compact form and retain only the information that we need for the analysis in question. For operational profiling we needed just the event ID, and so we retained only that part of the log line. The graphical representation of the log file definitely has some loss in information (the time stamp and dynamic parameter information not retained when we build the operational profile) and hence we cannot collect the log file itself in that format.

6.4 Complexity Analysis

The complexity of this transformation is linear in the size of the log file, i.e. $O(N)$, where $N$ is the number of lines in the log file. Step 2 makes a single pass through the log file, and examines each log line. The time taken to search through and build the index simultaneously
Figure 6.5: Results (Use Case 4) from Operational Profiling the VCL system by building the WDG of its Log File
scales logarithmically with respect to the index size. But since the the index size (number of unique log events in the application) is constant for each application and does not depend on the log file, we can assume that this search though the index file is constant. In the VCL log files, the index size was under a 1000, and hence the constant in this case was around 10. Since the time taken for each access is constant and there are $N$ such access for each log file, the time complexity of Step 2 is $O(N)$.

In Step 4 we inspect each line of the log file. For each line we do an array access in the two-dimensional adjacency matrix $G[M][M]$. Since array access takes constant time, this step too is $O(N)$. Step 5 iterates through each element of the adjacency matrix $G[M][M]$. Thus it is of the order $O(M^2)$. Since $M << N$, $M^2 < N$. Hence the order of time complexity for the transformation is $O(N)$, or linear in the size of the log file. The two data structures we create from the log file with $N$ log lines is the index of size $O(M)$, and the adjacency matrix $G[M][M]$ of size $O(M^2)$. Since $M^2 < N$, the space occupied by the output is much smaller than the input log file.

### 6.5 Conclusion

Log files have mostly been a flat text file with a set of execution events written in the log file in a serial order. Each log line in the log file is an execution event. Although this is the format required to archive the execution of the system, it is not the format that is most convenient for analysis, such as operational profiling or anomaly detection. A more intuitive format for such analysis is a Weighted Directed Graph (WDG). In this paper we proposed such a transformation for the log files. We performed a complexity analysis on this transformation algorithm and found the time complexity to be of $O(N)$. The graphical form is more efficient for building analysis (operational profiling) tools, because we can use existing graph theory algorithms and the software libraries that exist for those algorithms. In this paper we demonstrate how operational profiling can be done on the adjacency matrix representation of the WDG. Once this analysis is done we can trace the log events back to the source code to determine which execution path is used frequently. We illustrate the operational profiling with the help of a small example. We also analyzed a large log file (802 MB in size with more than 4.7 million events), from the Virtual Computing Lab system at North Carolina State University to see if our approach scaled. The transformation, rendering, and analysis took merely seconds to finish. We identified 4 sequences of events in the log file, which the developers of VCL agreed were the most frequently executed use cases in the VCL system. Thus we were able to build the WDG in a linear time frame from the log file and extract the operational profile from it in merely seconds.
Chapter 7

Conclusions and Future Work

The ability to perform large scale data analytics has led to the collection of a variety of data streams from software systems. Performance metrics, user statistics, provenance information, and execution information are just a few examples. In this dissertation the focus is on one such data stream, namely the log files, or a record of the execution activities of the software system.

Even though the log files might have all the information we need to make the decision, they are not always readily available, and extraction of the information may be complicated. To enable log-based decision making, we proposed an adaptable end to end framework for the analysis of log files. Components of the framework are: (1) Log collection subsystem, (2) Log abstraction subsystem, (3) Log transformations subsystem, and (4) Analysis subsystem. In presenting and discussing this framework, the specific example of operational profiling to illustrate the flexibility of the framework. Throughout this dissertation we used data (log files) from the Virtual Computing Laboratory (VCL), Cloud Computing solution.

- **Log Collection**: We developed an adaptive logging technique that collects information in the log files based on the context of the execution. We performed a simulation-based assessment of our adaptive logging technique using the VCL logs and observed that we were able to collect more information about the reservations that the developers cared about, and less information otherwise. On the average we collected 29% lesser information but retained the detailed log information about the events of interest.

- **Log Abstraction**: We took an extensive look at the research related to log file abstraction to identify the assumptions behind each of those solutions. We found that in published cases log messages are typically highly structured and a heuristic or regular expression based approach is often sufficient. But this was not the case in VCL and some other applications that we inspected. Also some of these solutions did not scale linearly which causes a problem in large log files. By exploiting the frequency of occurrence property
in log files, we proposed a new solution to abstract semi structured log files in linear time. We performed an analysis of the time complexity of the algorithm to show that our approach would scale linearly. We then calculated the accuracy of our solution. A pure heuristic based approach had an accuracy of 95%, while the pure empirical frequency based approach had an accuracy of 93%. But a hybrid approach where we applied these two techniques one after the other yielded an accuracy of 97%.

- **Log Transformations**: We proposed two novel approaches to building the operational profile of a system based on its logs. Both solutions find the occurrence probability of sequences of events - high occurrence probability events and low occurrence probability events. Most other solutions provide only the probability of the most frequently occurring events [19]. In the first approach we would transform the log file to a suffix array data structure, and in the second approach we transform the log file to a weighted directed cyclic graph. Both solutions were automated. Two use cases in VCL that we identified as the most frequent ones, corresponded to the ‘loading a module for the reservation’ and the ‘database update’ uses cases. Both of these techniques scale linearly and we have provided the complexity analysis for the same.

To summarize, the contributions that we have made in this dissertation are:

- Identified the various steps required to analyze log files to provide the necessary information to for software engineering decision making.

- Proposed an adaptable end-to-end framework, whose components accommodate these steps.

- Built a set of solutions (tools, techniques, and algorithms) for each component of the framework.

- Discussed the other solutions available in the literature for each component of the framework, and under what context they would be better solutions.

As part of future work, we intend to take this research in two directions. Firstly we would like to determine other possible analysis techniques that can be applied to the log files to extract information that could help the different stake holders in their decision making. Also we would like to study what kind of information is useful to these stake holders that can be collected in log files and how we can collect them. We would like to build new and improved solutions to each of the components of our framework as the research in other related data mining fields are advanced.
REFERENCES


70


APPENDIX
Appendix A

Virtual Computing Laboratory

VCL “is an open source implementation of a secure production level on-demand utility computing and services oriented technology for wide-area access to solutions based on virtualized and bare-metal resources, including computational, storage and software resources” [10], [38], [43]. The infrastructure at NCSU has more than 2000 computers (most of them are IBM BladeCenter blades) and delivers diverse computing services to more than 40,000 users. There are close to 900 different software service stacks called ‘images’. An image can range from a bare-machine hypervisor or operating system load, to virtual guest operating system loads, to any combination of the system and middleware and applications. Service can be individual and group based (e.g., for synchronized classroom or lab use), server-oriented, sub-clouds, and HPC. VCL also allows integration of external cloud services and hybrid cloud services. At NCSU VCL is operated as a private cloud, and all transactions and communications are logged. This includes networking, reservations and usage, security, resource provenance and performance logs, and other relevant information. Service reservations can be one-time or recurring, stateless or stateful, and they can be requested on-demand, or for some future time.

Fig. A.2 illustrates VCL architecture. This architecture maps fully onto Fig. A.1 general cloud components, i.e., VCL provides all required cloud elements. The users initially go through a web portal to authenticate. Once authorized, VCL presents them with the choice of all the pre-built images available for them to reserve. The images available to choose from depend on the user privileges and category. The user then chooses one of the images, a cluster of images, or another mode of operation (e.g., stateful or stateless operation, creation of new service images using base-line images, etc.). They also request the time frame for which they want to use the resources. Their request is recorded in the database. Then a VCL checks on the availability of the resources and images, and hands the request over to one of its distributed resource management nodes (MN). Selected MN then loads the images (virtual or bare-metal) onto identified real or virtual hardware, and allows the user personalized access. Typically
Linux based images are given direct access via ssh, or via the application service they are running, while Windows based desktops are accessed using RDP protocol. Windows servers are accessed using appropriate application interface. Time-out, IP-locking, VLAN-ing and one-time and NCSU corporate passwords are used to secure the resources and services. VCL also has a protected private back-channel to all of its physical and virtual resources to load them, manage them and secure them. VCL can be used to seamlessly access resource on other VCL clouds, and commercial cloud services such as Amazon EC2, IBM Blue Cloud, and soon Azure.

A.0.1 VCL Management Node Log Format

VCL logs information at many levels. Centrally, its database has global information about all transactions (and failures) since the day VCL went into production (2004). However, each management node keeps extensive additional logs that allow optimization, debugging, forensics,
management and recovery operations. Furthermore, each VCL image can be instrumented using open-source or proprietary resource monitoring agents.

In this paper we will focus on two sources - the relevant information from the global repository, and a sample of MN-based logs to illustrate the dynamics of the detailed processes and failures. Fig.A.3 illustrates the content of a record in the VCL daemon (or MN) log. It has a time stamp field, a unique identifier field, the caller information field, and information about the VCLd code that is emitting the message. The unique identifier is in turn comprised of three parts, namely the process ID, the (request ID, reservation ID) pair and the state of the VCL system. We can determine which reservation a particular log line belongs to by extracting this detail.

MN log files are collected locally, in the same machine on which the VCLd daemon is executing, and so there is not much delay in collecting the information. Nevertheless, the time stamp is not the time at which the message is logged, but rather the time at which the logging method was called, i.e., when the particular event happened in the system. The MN clocks are synchronized to the NCSU campus time servers. The process ID is the thread ID that is currently handling this particular reservation. The (request ID, Reservation ID) pair is generated from the database at the time of image request and time of image reservation. They are two separate IDs since not all requests will result in a reservation. This can happen due to a failure after a request has been made but the reservation did not complete. The state of the system indicates which specific action of the reservation process the system is currently in. Using the combination of these three fields we can uniquely identify a particular action of a specific reservation. We need to identify each action in a reservation uniquely because each action can either complete execution successfully, or result in a failure. It is these failures that we are interested in. For each failure that may eventually become visible to the end-user (e.g., a failed reservation), there may be multiple action failures in the MN log.

The next field is the caller information field. This records which line of which file in the code makes the call to the logging method. This is used to identify which event has just executed. The last field in Fig.A.3, namely, the string field contains the actual log message. This is the field we inspect to determine if the action is a failure or not. The string contains the word ‘CRITICAL’ or ‘WARNING’ when a critical or warning type failure occurs.
Each of these fields are separated by the ‘—’ (pipe) symbol. We read the log file line by line. Then we split each line into tokens based on the separator symbol. Once we have the tokens we examine the token for the string field to see if it has the case sensitive words ‘CRITICAL’ or ‘WARNING’. If it does then we grab hold of the tokens for the timestamp and the unique identifier fields for that event. We write the time stamp to either the file with critical failures or the file with warning failures. Then we use the unique identifier and iterate through the log file till we reach the next action of the reservation. We do this because some times a specific action might have multiple warning or critical messages. This is due to the cascading effect of the failures. These warning and critical messages are often separated by mere seconds. Collecting them as separate failures would be incorrect as they are all of the same failing action of a reservation. Note that we consider failures that happen in different actions of the same reservation to be separate failures because it signifies that the reservation failed in multiple steps.

Note also that in the ideal implementation of this research, the analysis and prediction of failures would be done in real time. This paper however is an exploratory study, and here we extract and analyze failures from the log files after the fact.

A.0.2 VCL Database Logs

This set of logs provides global information about all VCL transactions at a particular installation. In the case of NCSU VCL, this encompasses 3 data centers, and users from NCSU and about 50 other NC institutions. While the internal MN logs are not open to public. A lot of useful information that can be gleaned from central transaction repository is presented to the VCL managers either through the Dashboard (Fig. A.4) or to general public through the VCL Statistics interface tabs.

Dashboard provides, for example, information about the number of active reservations, online computers, and failed computers, top 5 images used, top recent computer failures, and top recent image failures. This allows quick and efficient insight into VCL operation, but it also facilitates mitigation and management activities.

It also points to a richness of faults and failure categories that exist in a cloud environment. Most of them are related to infrastructure issues, capacity issues, changes in operational profile (frequency of usage), and erroneous image to resource mappings. However, some are related to image building functions, and some (rare but present) to actual bugs in VCL codes.

Statistics display page allows a comprehensive insight into VCL operational profile over its life, into its reliability (and which images fail most often - usually capacity and image to hardware mapping related, but sometimes related to experiments with images themselves), and to failures that range from unavailability of resources (e.g., hardware unavailable, license
unavailable, image off-line, etc.), to potential perception failures that have to do with speed of service (e.g., how long does it take to load an image), to actual failures on part of users to accept a reservation, and failures to prepare a computer for the end-user. Since VCL has built in fail-over mechanisms, not all recorded failures end-up being seen by the end-user.

For example, one can learn that in between March 7, 2010 and March 7, 2011 there were close to 200,000 image reservation requests that consumed close to 490,000 computer-hours, that all but 7,247 of the reservations where for immediate use ("now") rather than being scheduled for later, and that end-user failed to receive requested resources (unavailable) in only about 0.55% of the cases, indicates an estimated reservation reliability of about 0.9945. In the same period of time there were no system-wide failures (i.e., at any time there always were some VCL resources available), however individual users may have experienced network or access related outages of a few minutes. A conservative estimate of the system availability is in excess of 0.999-s.