Creating Operational Profiles of Software Systems by Transforming their Log Files to Directed Cyclic Graphs

Meiyappan Nagappan
North Carolina State University
Raleigh, NC, USA
mnagapp@ncsu.edu

Brian Robinson
ABB Corporate Research
Raleigh, NC, USA
brian.p.robinson@us.abb.com

ABSTRACT

Most log files are of one format - a flat file with the events of execution recorded one after the other. Each line in the file contains at least a timestamp, a combination of one or more event identifiers, and the actual log message with information of which event was executed and what the values for the dynamic parameters of that event are. Since log files have this trace information, we can use it for many purposes, such as operational profiling and anomalous execution path detection. However the current flat file format of a log file is very unintuitive to detect the existence of a repeating pattern. In this paper we propose a transformation of the current serial order format of a log file to a directed cyclic graph (such as a non-finite state machine) format and how the operational profile of a system can be built from this representation of the log file. We built a tool (in C++), that transforms a log file with a set of log events in a serial order to an adjacency matrix for the resulting graphical representation. We can then easily apply existing graph theory based algorithms on the adjacency matrix to analyze the log file of the system. The directed cyclic graph and the analysis of it can be visualized by rendering the adjacency matrix with graph visualization tools, like Graphviz.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools, Tracing*

General Terms

Algorithms, Measurement, Reliability

Keywords

Log Files, Directed Cyclic Graphs, Operational Profiling

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 repeting patterns.

TEFSE'11, May 23, 2011, Waikiki, Honolulu, HI, USA

In order to solve this issue, we propose a transformation of the serially ordered log file to a Directed Cyclic Graph(DCG). 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 anomaly detection, all that we need to do is use an already existing software library of the graph theory algorithm for finding all the paths between two nodes of a graph, and apply it on our DCG representation of the log file.

1.1 Contributions

Our contributions in this research project are as follows:

- Developed the transformation of a log file with serial events to a DCG.
- 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 [1].
- Built tools to analyze the DCG 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 [2], a cloud management application at North Carolina State University.

2. OUR APPROACH

In this section, we present our transformation of a log file with a set of serial events to a DCG. In the serial order each event is important as a stand alone event. But in the DCG 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 DCG. 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 DCG. The edges are labeled with 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 DCG is labeled with 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 DCG 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 [7]. 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 $M * M$ 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 [1].

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.

2.1 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. Hence the time complexity of step 2 is $O(N)$. In step 5 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 6 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.

2.2 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
```

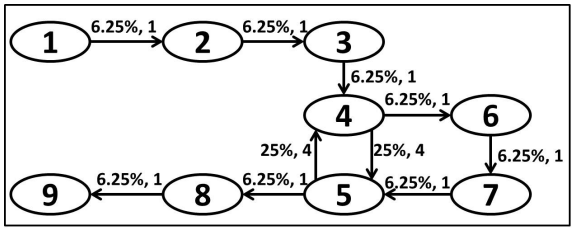

Figure 1: Directed Cyclic Graph of the Log File

Table 1: adjacency matrix $G[M][M]$

ID	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	4	1	0	0	0
5	0	0	0	4	0	0	0	1	0
6	0	0	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

```
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 *
7: Retrying:Move chunk * from * to *
8: Combining * chunks in * to form file *
9: Moved file * from * to *
```

The events in the log file are mapped to their corresponding event IDs:1, 2, 3, 4, 5, 4, 5, 4, 6, 7, 5, 4, 5, 4, 5, 8, 9.

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. 1. The graph generated from this transformation and rendered by the graphviz tool is shown in Fig. 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 apercentage value of 25% since they occur four times each.

2.2.1 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 <- 1 to M
  For j <- 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 highligted 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. [5], and Nagappan et.al [8] propose other log file analysis approaches that build the operational profile of the system. Hassan et.al.'s [5] 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.

3. RESULTS AND DISCUSSION

We performed a case study on a log file from Virtual Computing Lab [2], 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 465.70 MB in size having 2,595,258 events. This was log information collected over one week. The index file to this log file had 113 events (implies there were 113 unique events in the log file which had over 2.5 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 113 IDs. Then we build the adjacency matrix of the log file in 6.26 seconds. We then convert the graph to a DOT file and render it using Graphviz [1] 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 465 MB file.

We then applied the operational profiling algorithm to the adjacency matrix with a threshold value of 15%. We chose the value of 15%, as it focused the analysis on the largest two sequences in the log file. We traced these two sequence of events back to where they originate in the source code. They correspond to the VCL state intialization and VCL image request actons. These were expected to be the most frequent usage scenarios.

Given that there were 2.5 million events in total, this implies that these two pairs of events occurred almost 375,000 times 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 two pairs of events as the most frequent sequences because in the log file that we analyzed they were the most frequent sequences. However, if the most frequent sequence was a multi-event sequence, we still would have found it because our approach is not dependent on the length of the sequence of events.

We were able to reduce the size of the log file from a 465 MB flat file to a 104 KB DOT file or 20.3 MB image(gif format) file. 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.

4. RELATED WORK

There is extensive research on operational profiling [5][8], and debugging of systems [3][4] 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 [6], Jhala and Majumdar report the different ways to build state models of software to improve testing efforts. 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 [1].

5. CONCLUSIONS AND FUTURE WORK

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 Directed Cyclic Graph (DCG). 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 DCG. 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 (465 MB in size with more than 2.5 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.

As part of future work, we plan to apply this approach to log files from a production system at ABB Inc. We also want to detect anomalous execution paths by analyzing the DCG representation of the log file. We plan to adapt the graph theory algorithm that determines all paths between two nodes. Using this adapted algorithm we will be able to find the different execution paths between the starting and ending events of a particular use case. Our hypothesis is that the path that was executed fewer times will provide vital information to finding the root cause of the anomaly.

6. ACKNOWLEDGMENTS

This research was done as part of an internship at ABB Corporate Research. We would like to thank Dr.Mithun Acharya and Patrick Francis of ABB Corporate Research and Dr.Mladen Vouk of North Carolina State University for their insights on the project, and Aaron Peeler of the Virtual Computing Lab for providing the log files.

7. REFERENCES

[1] http://www.graphviz.org/ (last accessed 01/25/2011)

[2] http://vcl.ncsu.edu/ (last accessed 01/25/2011)

[3] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, k. Vaswani,"HOLMES: Effective Statistical Debugging via Efficient Path profiling," 30th Intl Conference on Software engineering, 2009, 34-44.

[4] Q. Fu, J. Lou, Y. Wang, J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," 9th Intl Conference on Data Mining, 2009, pp.149-158

[5] A.E. Hassan, D.J. Martin, P. Flora, P. Mansfield, and D. Dietz. 2008. An Industrial Case Study of Customizing Operational Profiles Using Log Compression. 30th Intl Conference on Software engineering, 2008, 713-723.

[6] R. Jhala and R. Majumdar, "Software Model Checking," ACM Comput. Surv. 41, 4, Article 21 (October 2009), 54 pages.

[7] M. Nagappan, and M.A. Vouk, "Abstracting Log Lines to Log Event Types for Mining Software System Logs," 7th Working Conference on Mining Software Repositories, 2010, pp. 114-117.

[8] M. Nagappan, K. Wu, and M.A. Vouk, "Efficiently Extracting Operational Profiles from Execution Logs Using Suffix Arrays". 20th International Symposium on Software Reliability Engineering, 2009, 41-50.