Efficiently Extracting Operational Profiles from Execution Logs using Suffix Arrays

Meiyappan Nagappan
North Carolina State University
Dept. of Computer Science
Raleigh, NC 27695-8206
mnagapp@ncsu.edu, (919)513-5082

Kesheng Wu
Lawrence Berkeley National Laboratory
Computational Research Division
Berkeley, CA 94720
KWu@lbl.gov, (510)486-6609

Mladen A. Vouk
North Carolina State University
Dept. of Computer Science
Raleigh, NC 27695-8206
vouk@ncsu.edu, (919)513-0348

Abstract

An important software reliability engineering tool is operational profiles. In this paper we propose a cost effective automated approach for creating second generation operational profiles using execution logs of a software product. 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 discuss a practical implementation of the algorithm in the context of the logs from a large cloud computing system.

Keywords: Software reliability, Operational Profile, Execution logs, Suffix Arrays, O(N).

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 [16]. 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 [18].

In studies of operational profiles, most existing efforts concentrates 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 [19, 21]. 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[4] to log their actions. Similarly HIPAA [2] 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 mil-

lion events collected over a 24 hour period [10]. *Hassan et al.* report getting few of the most frequently occurring scenarios within 2-3 hours with some human intervention [10]. *Vaarandi*, on the other hand, used a clustering algorithm to get the patterns from event logs [23].

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. 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 [14] 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 [5], [7].

1.1 Organization of the Paper

The paper is organized as follows: Section 2 presents some traditional techniques used to derive operational profiles. It also provides information on suffix arrays, the data structure that we use in our approach. In section 3 we briefly discuss some of the related work. In section 4 we explain our approach of finding frequency of patterns in log files using suffix arrays and the longest common prefix array. Section 5 discusses the complexity of our algorithm. In section 6 we present and discuss results from the experiments we conducted. We also discuss some of the limitations of our approach. Section 7 concludes the paper.

1.2 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 bio-informatics domain [14] and in analyzing whole program paths for software optimizations [20], 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 [10].
3. An automated solution that does not require users to tune any parameters.

2 Background

2.1 Deriving Operational Profiles

Operational profiles are traditionally created by a combination of system engineers, high-level designers, test engineers, product planners and marketing people [16]. 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 [10], [16]. 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 [24]. Code profiling and trace analysis are some of the available techniques.

The Eclipse Test and Performance Tools Platform (TPTP) [1], and Java Virtual Machine Profiler Interface (JVMPI) [3], 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 [11]. 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 [8]. 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.

Table 1. Example: SA of the text "abracadabra" and the LCP

i	SA(i)	Suffix	LCP(i)
0	11	a	0
1	8	abra	1
2	1	abracadabra	4
3	4	acadabra	1
4	6	adabra	1
5	9	bra	0
6	2	bracadabra	3
7	5	cadabra	0
8	7	dabra	0
9	10	ra	0
10	3	racadabra	2

These logs can be used to build operational profiles that are based on actual usage of the production system by the user.

2.2 Suffix Arrays

Manber and Myers invented the suffix array (SA) to identify all possible occurrences of a pattern in a text [14]. SA is a data structure that 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 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 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 [14]. 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 [13, 26], and can construct the LCP in O(N) time [12]. 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($P + logN + z$) time.

3 Related Work

The usefulness of analyzing log files has been long recognized. Tools like SEC [22], Splunk [6], and Swatch [9], 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. [10] and *Vaarandi* [23] 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. *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.

Operational profiles are often derived using clustering algorithms [15], [16], [17] which have non linear time complexities. *Vaarandi's* tool called Simple Log File Clustering Tool (SLCT) [23] 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.

4 Our Approach

Fig. 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.

4.1 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 [11]. Tools like SEC [22], an event correlation tool or Swatch [9], a log

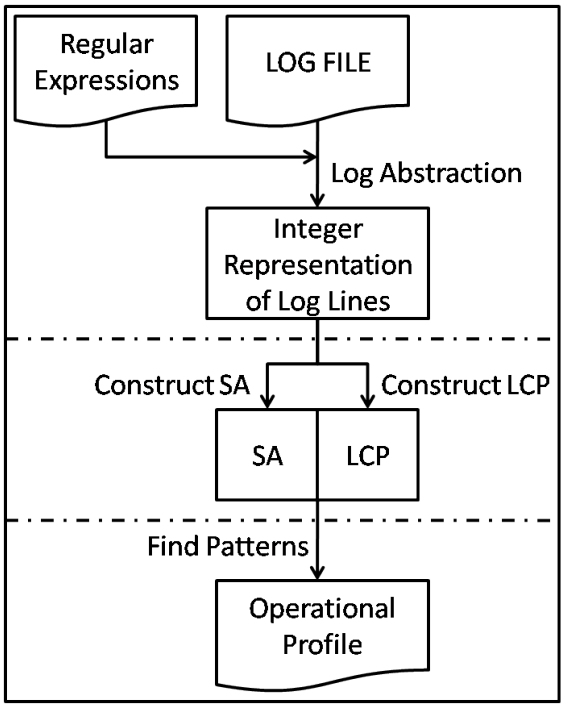


Figure 1. Main steps in our approach

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. In our approach we used a technique similar to the one used by *Xu et al.* [25]. Source code analysis is done to identify the statements that print to the log file.

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 `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) notify($ERRORS{'OK'}, 0, "$node ssh
port $port open");
(2) notify($ERRORS{'WARNING'}, 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:(.)+$
```

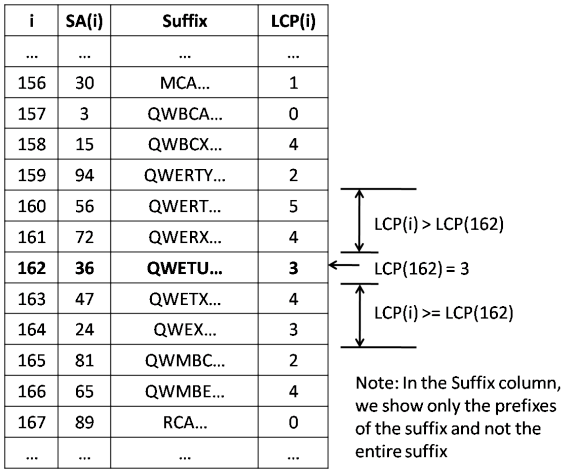
A particular line in the log file will match with at least one of these regular expressions. When more than one match occurs we choose the match with the regular expression of greater length. This matching is done using the boost-regex library available in C++ distributions. We replace each log line 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.

4.2 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* [26] and LCP construction algorithm by *Kasai et al.* [12] respectively. An example of SA and LCP for a sample string is illustrated in Table 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.

4.3 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. 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.



i	SA(i)	Suffix	LCP(i)
...	...	...	...
156	30	MCA...	1
157	3	QWBCA...	0
158	15	QWBCX...	4
159	94	QWERTY...	2
160	56	QWERT...	5
161	72	QWERX...	4
162	**36**	**QWETU...**	**3**
163	47	QWETX...	4
164	24	QWEX...	3
165	81	QWMBC...	2
166	65	QWMBE...	4
167	89	RCA...	0
...	...	...	...

Figure 2. Operational Profile from LCP and SA

Consider row 162, with value *(36, 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 (patterns) and the number of times each occurs in the log file (the text), and hence build the operational profile.

5 Algorithm Analysis

5.1 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. We read the log file in chunks of size $n(n << N)$ so that we can optimize on space required, when log files are 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. 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[26, 12]. 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.* [12] 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.

```
for each i less than N
  Initialize count to 0
  //To find the count of similar patterns
  //Scan lcp values after current index
  //Skip the iteration if lcp[i] < 0
  j = i
  while lcp[i] <= lcp[j + 1]
    Increment count
    if(lcp[i] == lcp[j+1])
      //negate lcp[j+1] to indicate
      //repetition
    endif
    j = j + 1
  end while
  //Scan lcp values before current index
  j = i
  while lcp[i] <= abs(lcp[j - 1])
    Increment count
    j = j - 1
  end while
end for
```

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).

5.2 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. 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 [14]. 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.

```
2009-03-22 21:45:28|10.192.7.5 ssh port 22 open
2009-03-22 21:45:28|VCL::Module::OS OS object successfully
created
2009-03-22 21:45:28|VCL::inuse object created and initialized
```

Figure 3. Examples of Log Entries from VCL

6 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) [5], [7], 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. 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.

6.1 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 2, shows the number of events in the sample, the time to find patters, the value of $\bar{k}$, and the total time needed to calculate and output operational profile

Table 2. VCL Log Data. Total number of events covered is 5,500,000.

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

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. 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 $\bar{k}$ 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 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

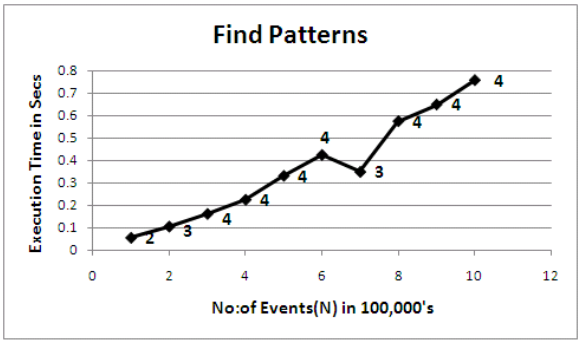


Figure 4. Execution Time vs. Number of events in VCL log. $\bar{k}$ is shown next to each data point.

Table 3. The Operational Profile Collected from VCL Logs

VCL Operation	% of occurance	Length of Sequence
Database Access	15.73	4
Load OS Modules	13.38	3

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 ma-

chine 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.

6.2 Discussion

Experimental results shown in Figure 3 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 the 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 interpretted. At that stage the human merely looks at less than a few hundred log lines. Also in the case study conducted by *Vaarandi*, 181 clusters were found [23], and in the case study conducted by *Hassan et al.* 7-10 patterns we found [10]. 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.

7 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 [10], or one staff month [16] 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 genera-

tion operational profiles more efficiently. Such profiles can then be used as optimization and anomaly detection diagnostic tool, and for construction of regression testing suites. Our tool will be released into the open source domain.

Acknowledgment

We would like to thank, Aaron Peeler for providing the VCL log files. We would also like to Aaron for his valuable inputs on the interpretation of the operational profile from the log files. We would also like to thank Dr.Stallmann for the discussions on the complexity analysis. This work was started as part of an internship at Lawrence Berkeley National Labs under the guidance of Dr.Shoshani. This research was funded in part by the DOE grants DE-FC02-ER25809, and DE-AC02-05CH11231.

References

[1] The Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/ (accessed 05/14/2009)

[2] HIPAA Regulations and Standards. http://www.hhs.gov/ocr/hipaa/ (accessed 05/14/2009)

[3] Java Virtual Machine Profiler Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html (accessed 05/14/2009)

[4] Sarbanes-Oxley Act of 2002. http://thecaq.aicpa.org/Resources/Sarbanes+Oxley/ (accessed 05/14/2009)

[5] Virtual Computing Lab(VCL). http://vcl.ncsu.edu/ (accessed 05/14/2009)

[6] Splunk http://www.splunk.com/ (accessed 05/14/2009)

[7] S. Averitt, M. Bugaev, A. Peeler, H. Schaffer, E. Sills, S. Stein, J. Thompson, and M. Vouk., "The virtual computing lab." In International Conference on Virtual Computing Initiative, Research Triangle Park, NC, May 2007. pp. 1-16.

[8] A. Hamou-Lhadj , T.C. Lethbridge., "A survey of trace exploration tools and techniques." In Proceedings of the 2004 Conference of the Centre For Advanced Studies on Collaborative Research, Markham, Ontario, Canada, October 04 - 07, 2004. pp. 42-55.

[9] S.E. Hansen, and E.T. Atkins., "Automated System Monitoring and Notification With Swatch." In Proceedings of the 7th USENIX Conference on System Administration. *System Administration Conference*. Berkeley, CA. November, 1993. pp. 145-152.

[10] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield and D. Dietz, "An Industrial Case Study of Customizing Operational Profiles Using Log Compression." In *ICSE '08* Leipzig, Germany, May 10-18, 2008, pp. 713-723.

[11] Z.M. Jiang, A.E. Hassan, P. Flora, G. Hamann., "Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper)." The Eighth International Conference on Quality Software, 12-13 Aug, 2008. pp.181-186.

[12] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park., "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications." In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, July 01 - 04, 2001. pp. 181-192.

[13] P. Ko and S. Aluru, "Space-efficient linear time construction of suffix arrays." Journal of Discrete Algorithms, vol. 3, no. 2-4, pp. 143-156, 2005.

[14] U. Manber, and G. Myers., "Suffix arrays: a new method for on-line string searches". In *Proceedings of the Symposium on Discrete Algorithms*, San Francisco, California, Jan 22-24,1990. pp.319-327.

[15] D.A. Menasc, V.A. Almeida, R. Fonseca, and M.A. Mendes., "A methodology for workload characterization of E-commerce sites." In Proceedings of the 1st ACM Conference on Electronic Commerce (Denver,United States, November 03 - 05, 1999. pp. 119-128.

[16] J. D. Musa., "Operational profiles in software-reliability engineering." *IEEE Software*, 10(2):1432, 1993.

[17] J. D. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin. "The operational profile." In Handbook of Software Reliability and System Reliability., 1996. pp. 167-216.

[18] M. Nagappan, M.A. Vouk, K. Wu, A. Sim, A. Shoshani. "Efficient Operational Profiling of Systems using Suffix Arrays on Execution Logs (Student Paper)." 19th International Symposium on Software Reliability Engineering, 11-14 Nov, 2008, Redmond, WA. pp. 313-314.

[19] A. Oliner and J. Stearley., "What Supercomputers Say: A Study of Five System Logs." In Proceedings of the 37th Annual IEEE/IFIP international Conference on Dependable Systems and Networks. Washington, DC. June, 2007. pp. 575-584.

[20] G. Pokam, F. Bodin., "An Offline Approach for Whole-Program Paths Analysis Using Suffix Arrays." LCPC 2004. pp. 363-378

[21] M. Steinle, K. Aberer, S. Girdzijauskas, and C. Lovis., "Mapping moving landscapes by mining mountains of logs: novel techniques for dependency model generation." In Proceedings of the 32nd international Conference on Very Large Data Bases. Seoul, Korea. September, 2006. pp. 1093-1102.

[22] R. Vaarandi, "SEC - a lightweight event correlation tool," IEEE Workshop on IP Operations and Management, 2002, pp. 111-115.

[23] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs." IEEE Workshop on IP Operations and Management, 1-3 Oct. 2003, pp. 119-126.

[24] E. J. Weyuker and A. Avritzer., "A metric for predicting the performance of an application under a growing workload." *IBM Systems Journal*, 41(1):4554, 2002.

[25] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan., "Mining Console Logs for Large-Scale System Problem Detection." SysML'08, Dec 2008. pp. 1-6.

[26] S. Zhang, G. Nong, "Fast and Space Efficient Linear Suffix Array Construction." Data Compression Conference (dcc 2008), 2008, pp. 553.