ELT: *E*fficient *L*og-based *T*roubleshooting System for Cloud Computing Infrastructures

Kamal Kc, Xiaohui Gu
Department of Computer Science
North Carolina State University
kkc@ncsu.edu, gu@csc.ncsu.edu

***Abstract*—We present an Efficient Log-based Troubleshooting(ELT) system for cloud computing infrastructures. ELT adopts a novel hybrid log mining approach that combines coarse-grained and fine-grained log features to achieve both high accuracy and low overhead. Moreover, ELT can automatically extract key log messages and perform invariant checking to greatly simplify the troubleshooting task for the system administrator. We have implemented a prototype of the ELT system and conducted an extensive experimental study using real management console logs of a production cloud system and a Hadoop cluster. Our experimental results show that ELT can achieve more efficient and powerful troubleshooting support than existing schemes. More importantly, ELT can find software bugs that cannot be detected by current cloud system management practice.**

I. Introduction

Cloud computing [1], [2] has become increasingly popular by obviating the need for users to own and maintain complex computing infrastructures. However, due to their inherent complexity and large scale, cloud systems are prone to various runtime problems caused by hardware failures or software bugs. Console logs are often the sole information source for system administrators to perform troubleshooting. However, those log files grow quickly during system runtime. For example, in the Virtual Computing Lab (VCL) [2] production system, a single 12 hour reservation log file contains over thirty thousand messages. It is a daunting task for system administrators to manually examine millions of log messages that can be generated in a week. The current practice is to search some keywords such as “critical”, “warning”, or “error” that are inserted in the source code by software developers. However, relying on labelled messages cannot capture those unexpected runtime problems that do not contain any critical or warning messages. Moreover, developers often mark some messages as critical or warning for debug purposes, which often produce many false alarms for the system administrator.

Previous work on log analysis (e.g., [8], [20], [13]) has examined coarse-grained or fine-grained log analysis separately, which makes them suffer from either high processing overhead or low detection accuracy. Moreover, even with automatic anomaly detection, the system administrator still faces the challenge of examining a large number of log messages to find anomaly causes. Unfortunately, little research has been done to automatically extract root cause relevant messages. In addition, existing solutions (e.g., [23]) mostly focus on detecting known problems, which require prior knowledge about anomalies.

In this paper, we present an *E*fficient *L*og-based *T*roubleshooting (ELT) system for large-scale cloud computing infrastructures. ELT adopts a novel *hybrid* log analysis approach: It first employs a hierarchical clustering algorithm over all examined instances using a simple coarse-grained log feature called message appearance vector (MAV); It then performs outlier detection within large clusters using a fine-grained log feature called message flow graph (MFG). The first step can quickly detect those anomalous instances that are very different from all normal instances while the second step can further separate those anomalous instances that resemble some normal instances. Our approach is based on two key observations: i) large-scale cloud computing systems are naturally redundant, which consist of a large number of similar normal instances (e.g., multiple reservations of the same VM image, data-parallel MapReduce tasks [6], [4]); and ii) there are anomalous instances that share certain similarity with normal instances, which makes them hard to detect using coarse-grained features only. ELT can detect both known and previously unseen anomalies using *unsupervised* learning methods. Moreover, ELT can automatically extract tens of key messages from thousands of log messages to significantly narrow down the examination scope for system administrators. ELT also supports invariant check to provide useful clues for anomaly causes.

We have implemented a prototype of the ELT system and conducted extensive experimental study using large-scale console logs collected on the VCL production system. As a proof of general applicability, we also tested ELT using a small-scale Hadoop [4] cluster in our lab. We compare the anomaly detection accuracy of different algorithms using the receiver operating characteristic (ROC) curves that are commonly used to show the tradeoff between detection rate and false positive rate. The experimental results show that ELT can achieve much higher detection rate and lower false positive rate than current production system practice and other existing schemes. ELT can extract 4-150 key messages

```
VCL log message:

2010-04-27 13:22:03|18457|952414:1027953|new|OS.pm:
     wait_for_response(479)|waiting for vclf1-5 to respond
      to SSH, maximum of 600 seconds

Hadoop log message:

2011-03-25 23:47:01,380 INFO org.apache.hadoop.mapred.
     TaskTracker: Task attempt_201103252330_0001_m_000033
     _0 is in commit-pending, task state:COMMIT_PENDING
```

Figure 1. Production system console log message examples.

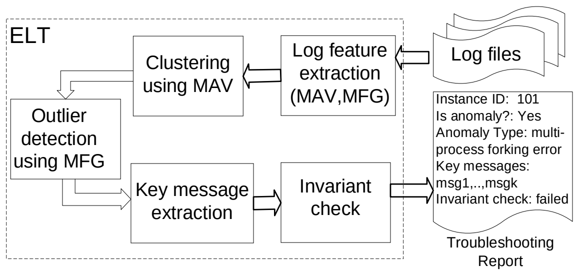


Figure 2. The ELT system architecture.

from large log files containing over 30K messages. Our invariant check scheme successfully found two software bugs in the VCL production system. ELT is light weight, which has more than one order of magnitude less overhead than fine-grained log analysis schemes. Performing invariant check over extracted messages reduces the check time by more than three orders of magnitude compared with checking against whole log files.

The rest of the paper is organized as follows. Section II presents the overview. Section III presents the design details of our approach. Section IV presents the implementation and experimental evaluation. Section V compares our work with related work. Finally, the paper concludes in Section VI.

II. Background and System Overview

Cloud systems often continuously produce console logs to record management operations during system runtime. In this work, we use the virtual computing lab (VCL) [2] and a Hadoop cluster [4] as our case study examples. VCL is a production cloud system that operates in a similar way as Amazon EC2 [1]. The system keeps a large number of virtual machine (VM) images recording different preconfigured reference systems (e.g., RedHat 4.3 with Hadoop, Windows 7 with Matlab). When the user submits a reservation request, one of the VCL management nodes creates the requested system by loading proper VM image files into the allocated hosts. Hadoop [4] is an open source implementation of Google's MapReduce system [6], which is one of the most representative data-intensive applications running in cloud.

Figure 1 shows two examples of console log messages produced by a VCL management node and a Hadoop tasktracker node. VCL log files record reservation operations such as host allocation, image loading, node reclaim, and others. Each log message consists of a set of fields separated by the pipe symbol ("|"). The message fields include *time stamp*, *process identifier*, *request identifier*, *reservation identifier*, *reservation state*, *caller information*, and *message content*. Hadoop log files record task operations such as creation, deletion, progress report, cleanup, and others. Each log message contains fields separated by blank spaces. The message fields are *time stamp*, *log level*, *output class name*, and *message content*.

In order to efficiently parse millions of log messages in a short period of time, we first extract a set of message templates to represent many different log messages. The message template consists of two parts: the text that does not change and the text that changes. We refer to the text that does not change as *message type* and the text that changes based on variable values as *message variables*. The log message template matching process only uses the message type part by replacing those message variables with the "*" symbol. In our experiments, we extracted 4164 message templates from the Apache VCL source version 2.1 and 167 message templates from the source files (mapred folder) of the Hadoop version 0.20.2. If the number of the message templates is large (e.g., VCL logs), we store all message templates in a hash table to achieve fast matching. Since the log message often consists of the code module name (e.g., `OS.pm` in the log message shown in Figure 1) where the message is generated, we hash log message templates using their code module names.

ELT explores a novel hybrid log mining approach to achieve efficient troubleshooting for cloud computing infrastructures, illustrated by Figure 2. ELT first groups log messages based on a certain identifier (e.g., reservation ID in VCL, task ID/task attempt ID in Hadoop) into different log files. Thus, each log file records all the execution operations for one troubleshooting instance (e.g., one reservation in VCL, one map/reduce task in Hadoop). ELT then parses each log file to extract a coarse-grained log feature called *message appearance vector* (MAV) and a fine-grained log feature called *message flow graph* (MFG). The MAV feature only captures which types of messages appear in the log file while the MFG feature captures the transition patterns between different types of messages.

Next, ELT performs hierarchical clustering over the log files of all troubleshooting instances using the MAV feature. The system will get two different types of clusters: i) small clusters consisting of anomalous instances only; and ii) large clusters consisting of both anomalous and normal instances. The system performs outlier detection within those large clusters using the fine-grained MFG feature to further separate the anomalous instances from normal ones.

After the anomalous instances are identified, ELT automatically extracts key messages that are most relevant to anomaly root causes. This is done by performing clustering over the *difference logs* of all anomalous instances. The *difference log* of an anomalous instance contains those messages that are not found in normal instances. Each of the resulting cluster represents an anomaly type and key messages for each anomaly type are the common message sequences among the anomalous instances of the same anomaly type. ELT provides further automatic troubleshooting support by checking system specific invariants over extracted key messages to provide useful clues for the anomaly causes.

Assumptions. Our current system implementation relies on the source code to extract message templates. This assumption is similar to previous console log analysis system [20]. Previous work also proposed techniques to extract message templates from log files without the need for source code [19]. Our work can easily leverage those techniques to eliminate the source code requirement if the source code is hard to obtain. Our clustering-based anomaly detection approach is based on the premises that the number of normal instances is larger than that of the anomalous ones, and each instance type (e.g., reservation of certain VM image) has abundant normal instances. During our experiments, we found that large-scale cloud computing environments generally follow our assumptions. Another assumption of our approach is that the anomalous instances contain either different log messages or different log message sequences from normal instances. We have found that most of the anomalous instances in cloud systems such as VCL and Hadoop follow this behavior.

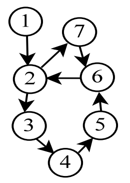
III. System Design

In this section, we present the design details of our system. We first describe the hybrid log feature extraction scheme followed by the hybrid log mining scheme. Next, we present the key message extraction and invariant check schemes.

A. Hybrid Log Feature Extraction

To achieve efficient log-based troubleshooting, the first step is to extract proper log features to characterize different instances. ELT first extracts the simple MAV log feature to describe the presence or absence of different message templates in each log file. Since the message templates present in the log file correlate with the system operations, MAV reflects which operations are conducted during the runtime of the examined instance. The size of MAV equals to the total number of message templates. The i^{th} element value in the vector is set to be 1 if the i^{th} message template is present in the log file. Otherwise, it is set to be 0. For example, if the total number of message templates is 4 and a log file contains message templates 1 and 3, the MAV of this log file is [1,0,1,0].

We chose MAV rather than previously used message frequency feature [20] because it is more robust. MAV is not



```
[1] initiating Linux post_load: * on *
[2] * is NOT responding to SSH, port 22 or  24
are not open
[3] waiting * seconds for * to boot
[4] waited * seconds for * to boot
[5] waiting for * to respond to SSH, maximum
of * seconds
[5] waiting for * to respond to SSH, maximum
of * seconds
[6] attempt *: waiting for * to respond to SSH
[2] * is NOT responding to SSH, port 22 or 24
are not open
[7] attempt *: code returned false, seconds
elapsed/remaining: */*, sleeping for seconds
[6] attempt *: waiting for * to respond to SSH
[2] * is NOT responding to SSH, port 22 or 24
are not open
```

Figure 3. Message flow graph (MFG) construction.

susceptible to repeating messages that often appear during normal operations. For example, in VCL, the management node continuously polls the reserved host, which results in some messages appearing frequently in the logs. Under such circumstances, the message frequency feature may cause false alarms by incorrectly categorizing reservations with different durations.

Examining MAV alone is insufficient since MAV cannot capture the execution sequence information. Thus, MAV can not distinguish two instances that consist of the same set of operations but executed in different orders. To address the problem, ELT extracts the fine-grained MFG log feature to capture the transition patterns among different message templates. The graph nodes in an MFG are message template identifiers and the edge from a node i and to a node j denotes that the message template i appears right before the message template j at least once. For example, Figure 3 shows an example of MFG for a VCL log snippet. The numbers in square brackets are the message template identifiers. In this example, the message sequence $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ is the first round of check for ssh service. During this check, the management node waits for some time to accommodate the boot delay. After the first round, the management node iteratively performs ssh service check in the loop $2 \rightarrow 7 \rightarrow 6 \rightarrow 2$. The loop terminates after the ssh service starts.

The console log message sequence reflects the execution sequence in the code path. Thus, MFG is useful for finding differences between instances that contain similar messages. Difference between the MFGs of two similar instances would indicate that there exist different execution sequences between them even though they have similar operations. However, the drawback of the MFG feature is its complexity. Analyzing the MFGs for all log files will inevitably incur big processing overhead. Thus, we need to design a hybrid log analysis algorithm that can leverage the advantages of both log features but still keep the log processing overhead low. We will describe the details of our log analysis algorithm in the next section.

B. Hybrid Log Analysis

We propose a new hybrid log analysis scheme that can achieve both high accuracy and low overhead. The hybrid

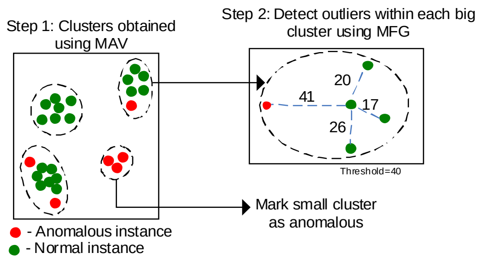

Step 1: Clusters obtained using MAV
Step 2: Detect outliers within each big cluster using MFG
20
41
17
26
Threshold=40
Mark small cluster as anomalous
● - Anomalous instance
● - Normal instance

Figure 4. Hybrid log analysis scheme.

log analysis algorithm consists of two steps, illustrated by Figure 4. First, we employ a hierarchical clustering algorithm using the MAV log feature to quickly obtain groups of instances with similar characteristics. Second, we perform a nearest neighbor based outlier detection within each group using the MFG log feature to accurately detect abnormal instances. We describe both steps in detail as follows.

1) Clustering using MAV: We employ a hierarchical clustering algorithm [10] to obtain groups of instances with similar characteristics. We choose hierarchical clustering over other common clustering techniques such as k-means and its variations because we do not want to make assumptions on the number of clusters that will form during production run. We use MAV in the clustering process for two reasons. First, MAV allows the clustering algorithm to form groups of instances with similar operations. Second, the cost of using MAV over all instances is much smaller than the cost of using fine-grained feature such as MFG.

We start from a single cluster containing all instances and continuously split a large cluster into two smaller ones. This process repeats until there isn't any cluster that can be split. Typically, hierarchical clustering uses dendogram height based stopping criterion which is not easy to determine algorithmnically. Due to this reason, our work uses *cluster diameter* as the stopping criterion. The cluster diameter denotes the largest distance between any two objects in a cluster. Any cluster with diameter greater than a specified threshold (e.g., 40) is selected for splitting. During each splitting step, we employ the k-medoid ($k = 2$) clustering algorithm to split a large cluster into two new smaller clusters.

We define the distance between two instances as the Manhattan distance between the MAVs of the two instances' log files. Let $D(L_i, L_j)$ denote the distance between two log files L_i and L_j. Let $[m_{i1}, ..., m_{ip}]$ and $[m_{j1}, ..., m_{jp}]$ denote the MAVs of the two log files. We formally define the distance as follows,

$$D(L_i, L_j) = \sum_{k=1}^{p} |m_{ik} - m_{jk}| \qquad (1)$$

The result of the clustering process is a collection of clusters of different sizes. Based on the premise that anomalous instances are relatively infrequent and contain different messages, most abnormal instances should form separate small clusters by themselves. However, in some cases of real systems, both normal and abnormal instances may contain similar messages. Thus, we cannot distinguish anomalous instances from normal ones based on the MAV feature only. To handle those cases, we perform outlier detection within all large clusters (e.g., size > 4) to further pinpoint those anomalous instances that mix together with normal ones in the same cluster. The size value (eg. 4) is an empirically decided value. We believe that large cluster size threshold can be methodically derived in an online version of ELT scheme. In online operation, our scheme is less sensitive to the small cluster size selection since the anomaly always starts from a small cluster and normal operations often have sufficient number of instances in a large-scale cloud computing environments.

2) Outlier Detection using MFG: The instances in the same cluster contain a similar set of message types but can have different message sequences. We use the MFG feature to distinguish those instances with different message sequences. We perform outlier detection using the MFG feature within each large cluster to further pinpoint anomalous instances. We use the graph edit distance to define the distance of two instances' MFGs. The graph edit distance between two graphs is the number of edge additions or deletions required to transform one graph to the other.

We use the nearest neighbor technique [18] to perform outlier detection. We compute pairwise graph edit distance between the MFGs of every two instances within the cluster. We define the nearest neighbor distance as the shortest distance for each instance. We label an instance as anomalous if its nearest neighbor distance is greater than the threshold $\bar{X} + 2\delta$, where $\bar{X}$ is the mean of all the nearest neighbor distances and δ is the standard deviation. Note that we only process the MFG feature within each cluster rather than over all the instances. As we will show later in our experiment results, the hybrid log analysis approach can significantly reduce the processing overhead by more than one order of magnitude compared to the scheme using the MFG feature only over all instances.

C. Key Message Extraction

Although automatic anomaly detection is very helpful, the system administrator still faces the dismal task of examining each anomalous instance containing thousands of messages for problem diagnosis. To further simplify the troubleshooting task, ELT provides key message extraction support, which can narrow down a small number of log messages that are most relevant to the anomaly cause. Our key message extraction scheme consists of three steps shown in Figure 5.

First, we obtain a difference log called *difflog* for each anomalous instance by comparing it with normal instances. We use the MFG feature during the difflog generation. If

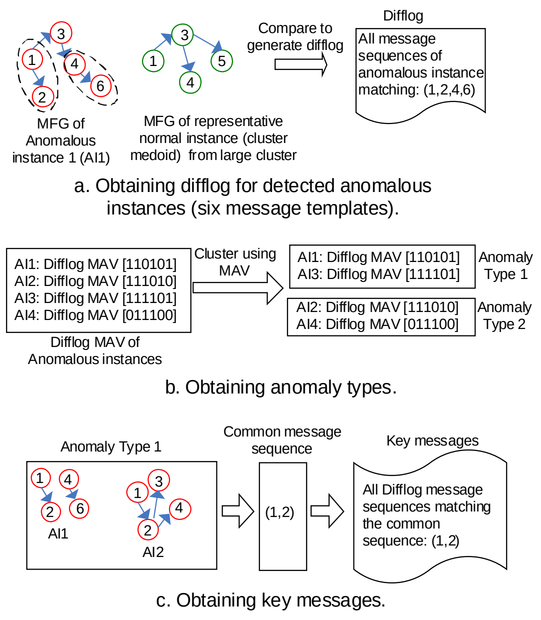

MFG of Anomalous instance 1 (AI1)
MFG of representative normal instance (cluster medoid) from large cluster
Compare to generate difflog
Difflog
All message sequences of anomalous instance matching: (1,2,4,6)
a. Obtaining difflog for detected anomalous instances (six message templates).
AI1: Difflog MAV [110101]
AI2: Difflog MAV [111010]
AI3: Difflog MAV [111101]
AI4: Difflog MAV [011100]
Difflog MAV of Anomalous instances
Cluster using MAV
AI1: Difflog MAV [110101]
AI3: Difflog MAV [111101]
Anomaly Type 1
AI2: Difflog MAV [111010]
AI4: Difflog MAV [011100]
Anomaly Type 2
b. Obtaining anomaly types.
Anomaly Type 1
AI1
AI2
Common message sequence
(1,2)
Key messages
All Difflog message sequences matching the common sequence: (1,2)
c. Obtaining key messages.

Figure 5. Key message extraction.

an edge in the anomalous instance's MFG is not present in any of those normal instances's MFGs to which it is compared, we add the sequence of the message templates denoted by the edge together with all message occurrences matching the sequence into the difflog of the anomalous instance. For example, in Figure 5(a), the unique MFG edges of the anomalous instance are $1 \rightarrow 2$ and $4 \rightarrow 6$. We add all occurrences of the message sequence $(1, 2, 4, 6)$ into its difflog. For the anomalous instance that belongs to a large cluster, we compare its MFG with the cluster medoid. For the anomalous instance that belongs to a small cluster, we compare its MFG with the medoids of all large clusters (i.e., the representatives of all normal instances).

Second, we extract the MAV features for all derived difflogs. We then perform hierarchical clustering over all difflogs using their MAVs to identify different anomaly types. The clustering algorithm used for difflogs is the same as the one used for processing original log files. As a result, we can identify different types of anomalies and the anomalous instances belonging to the same anomaly type will be grouped together in one cluster. For example, in Figure 5, we identify two anomaly types, each of which includes two instances.

Third, we extract common message template sequences among all the instances belonging to the same anomaly type by comparing their MFGs. For example, in Figure 5, if the edge $1 \rightarrow 2$ appears in both instances of the anomaly type 1, we add the message template sequence $(1, 2)$ into the common message template. Next, we match the message template sequence with the original log messages stored in the difflogs of all anomalous instances. If there are multiple matching message sequences in different instances, ELT

```
13:38:28|24817|..|new|..|collected public hostname=host1

13:38:31|27609|..|new|..|executing SSH command on host1: .. '/
usr/sbin/useradd -u 32494 .. '
13:38:31|27609|..|new|..|SSH command executed on host1,
returning (0, "none")

13:38:33|24817|..|new|..|SSH command executed on host1, '/usr/
sbin/useradd -u 32494 .. '
13:38:33|24817|..|new|..|detected user already has account
13:38:33|24817|..|new|..|executing SSH command on host1: '/
usr/sbin/userdel -u 32494'
```

Figure 6. Key messages extracted for the multiprocess forking error.

outputs all of them.

D. Invariant Checking Over Key Messages

By extracting a small number of key messages, ELT can speedup the troubleshooting task for the system administrator. To further automate the log-based troubleshooting process, ELT supports invariant checking to provide more clues for the anomaly causes. We observe that many real world systems need to preserve certain invariants. Violations of those invariants indicate incorrect operations. For example, in VCL, the system needs to preserve the following invariants: 1) a reservation request cannot be processed by multiple processes simultaneously; and 2) simultaneous commands (e.g., remote ssh) should return the same output. Note that we do not use invariant checking to perform anomaly detection but instead only use it to extract more properties about the detected anomalous instances. We assume that the invariants are specified beforehand and can be readily tested.

Figure 6 shows an example of the extracted key messages for an anomalous VCL reservation request with the *multiprocess forking error*. We can see that the log file contains two interleaving processes (process 24817 and process 27609) performing the new state operations simultaneously. We detected this bug by performing invariant check over all the processes in the extracted key messages. We first derived the execution intervals defined by the start and end timestamps for all processes. We then checked whether the execution intervals of different processes overlap with each other. In Figure 6, we can see that the process 24817 and the process 27609 have overlapped execution interval. This bug has been reported to the VCL bug database (bug entry VCL-149). Note that this bug cannot be detected by the current VCL management practice since the failed reservation request does not include any critical or warning message.

ELT allows system administrators to specify the system invariants it should check. Performing invariant checks over the extracted key messages is relatively inexpensive compared to the entire log. Although we can perform some invariant checks over original log files, it is more efficient to perform invariant check over the extracted messages. For example, in the multiprocess forking error example, a single reservation may contain a large number of processes for long running reservations. The cost of performing the invariant check over the entire log file is $O(N^2)$, where N is the number of processes included in the log file. In contrast, the extracted key messages only include a few processes, which can significantly shorten the checking time.

IV. Experimental Evaluation

In this section,we present our prototype implementation and experimental evaluation results. We applied ELT to three production logs of the VCL system and MapReduce execution logs of a Hadoop cluster.

A. Implementation and Experiment Setup

We have implemented a prototype of ELT in about 6K lines of C/C++ code. The prototype consists of five major modules: 1) log parser, 2) clustering, 3) outlier detection, 4) key message extraction, and 5) invariant check. When we apply ELT to different system logs, we only need to modify the log parser module. We used adjacency list to represent the MFG. For matrix operations, we used the armadillo library [3].

We now describe the log parser implementations for VCL and Hadoop systems. The VCL codebase is written in Perl. It uses the function "`notify()`" to produce the console log message. The third argument of the notify function call contains the log message. Thus, the log parser greps the VCL source code for notify() function calls and then parses the third argument of each function call to extract a message template. The message template may contain perl variables which typically start with a "$" or "@" symbol. We assign a unique identifier to each message template. In our experiments, we extracted 4164 message templates from the Apache VCL source version 2.1 [2].

We use a simple approach to extract message templates from the Hadoop source code. Hadoop is written in Java. The functions of LOG object, `LOG.info()`, `LOG.warn()`, `LOG.error()` and `LOG.fatal()` output the log messages. Our log parser greps those functions from the source code to form the templates by separating parts enclosed in double quotes (") as fixed texts and parts not enclosed as variables. We extracted 167 message templates from the source files in the mapred folder of the Hadoop version 0.20.2. For simplicity, our current log parser did not use the abstract syntax tree to disambiguate the templates that may have different stream combination values due to known variable values and object function calls such as toString(). We can extract more templates by applying the abstract syntax tree, which can further improve the accuracy of our anomaly detection algorithms. However, as we will show later, ELT can achieve very good detection accuracy for Hadoop console logs even with only 167 message templates.

Table I shows the detailed information of the log datasets used in our experiments. VCL-1, VCL-2 and VCL-3 are the production logs produced by six different VCL management nodes during a 60 day period from April 1st to May

System	Log messages	(anomalous instances, total instances)	Message templates	Log size
VCL-1	2 million	(39,623)	4164	240 MB
VCL-2	20 million	(172,3655)	4164	2.6 GB
VCL-3	17 million	(43,1114)	4164	2.0 GB
Hadoop-1	2 million	(257,18224)	167	300 MB
Hadoop-2	6.8 million	(975,68561)	167	1.1 GB

Table I
Datasets used during experiments.

31st, 2010. During this period, reservations were made for various softwares such as Matlab, Openoffice, educational software for class projects, Linux system, and others. VCL-1 contains only Linux reservation logs. VCL-2 contains Linux and Windows reservation logs produced by the first management node. VCL-3 contains Linux and Windows reservation logs produced by the second management node. Hadoop-1 and Hadoop-2 are the Hadoop logs produced by running MapReduce sort applications on a 20 node cluster during a period of 5 days. Hadoop-1 is the subset of Hadoop-2 dataset. All log analysis experiments were conducted on a Quad core 2.66 GHz Intel processor (8GB RAM) machine running Linux kernel 2.6.32.

The VCL and Hadoop system logs used in our experiment are single files that contain log messages for all troubleshooting instances. When we process the system logs, we group log messages based on the instances. Each of these instances is represented as a separate log file. So, the number of log files ELT uses is the same as the number of instances shown in Table I.

We evaluate our system in terms of both detection accuracy and processing overhead. Let N_{tp}, N_{fn}, N_{fp} and N_{tn} denote the total number of true positive, false negative, false positive and true negative observations, respectively. We calculate the detection rate A_D and false alarm rate A_F in a standard way as follows,

$$A_D = \frac{N_{tp}}{N_{tp} + N_{fn}}, A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{2}$$

We used the receiver operating characteristic (ROC) curve to show the tradeoff between A_D and A_F for different anomaly detection algorithms.

To identify true anomalies in the VCL logs, we spent several weeks on manually examining the VCL log files. We first use our clustering algorithms to separate different log files into different groups. We then manually examine each instance in one group based on the specific messages appeared in the instance. After manually identified those anomalies, we confirmed with VCL developers on our manual inspection results. For Hadoop log files, we injected faults on a subset of MapReduce tasks. We thus know which tasks are true anomalies.

For comparison, we also implemented a set of alternative console log analysis schemes: 1) *Critical keyword*: this scheme marks an instance as anomalous if its console log file contains the keyword "critical". Current VCL production system management adopts this approach and a critical message will immediately trigger an alert email sent to the system administrator; 2) *Warning keyword*: this scheme relies on warning messages to detect anomalous instances. Since developers tend to include many warning messages for debugging purposes, this scheme will inevitably mark many instances as anomalous; 3) *PCA*: this scheme applies principle component analysis using the message frequency vector (MFV) feature to detect anomalies, which has been used by previous console log analysis system [20]; 4) *Clustering + NN MFG*: this scheme uses the same hybrid log analysis algorithms as ELT but uses the fine-grained log feature MFG for both clustering and outlier detection; and 5) *PCA MFG*: this scheme uses the same PCA scheme as in [20] but uses the fine-grained log feature MFG instead of MFV.

B. VCL Results

Table II shows all anomalies present in the VCL production log files we processed. Among the listed anomalies, *multi-process forking error* and *multiple attempts to delete user error* do not contain any warning or critical message. The management node normally processes a reservation with one process at a time. But, due to a bug in the software, for some reservation instances, there were two redundant processes performing node allocation, causing the *multi-process forking error*. Both processes then attempted to create user account on the reserved host, setting different passwords on each attempt. Depending on which process notifies the user, the user might be provided with a wrong password and locked out of successfully reserved host. In the case of the *multiple attempts to delete user error*, a bug in the user deletion handling module makes the management node to issue a repeated delete command. The second delete command attempts to delete an already deleted user and the management node is not able to correctly process the return result. ELT can successfully detect both unexpected bugs in all examined anomalous instances.

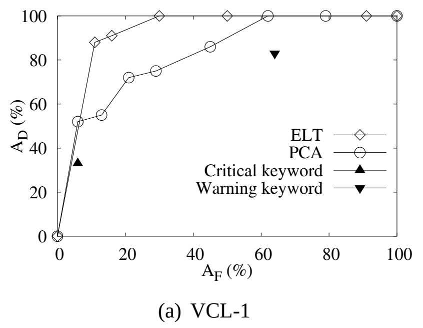
Figure 7 shows the anomaly detection accuracy comparison results among different log analysis schemes for the three VCL datasets. We evaluate the detection accuracy of different schemes using the Receiver Operating Characteristics (ROC) curves, which are commonly used to show the tradeoff between detection accuracy and false alarm rate. The x-axis and y-axis are false alarm rate (A_F) and detection rate (A_D) defined in Equation 2. We derive the ROC curve for ELT by tuning the diameter from 0 to 500. For VCL, we mark cluster of size smaller than 4 as anomalous cluster. This size value is empirically decided. We derive the ROC curve for PCA by changing the confidence interval of the Q-statistic value of the projections on non-significant principal components from 0 to 100%. Since the critical keyword and warning keyword schemes do not have tunable parameters, their ROC curves are not continuous.

Anomaly Types	**Description**
Image reloading failure	Occurs when the reloading operation fails twice to establish a new reservation.
Sanitization failure	Occurs when user account and activity are not cleared successfully during the reclaim operation.
Overlapping reservations	Occurs when more than one reservation is made on the single host.
*Multi-process forking error	Occurs when more than one process performs same state operation at the same time.
*Multiple attempts to delete user	Occurs when the management node attempts to delete the same user account more than once.
Reservation host not in contact	Occurs when the management node is not able to establish ssh connection with the reserved host after the host was reserved successfully.
Reservation failure due to cluster node failure dependency	Occurs when the reservation processing is not successful due to the failure of other dependent cluster nodes.
Predictive reloading failure	Occurs when predictive reloading cannot be performed at the end of a reservation.

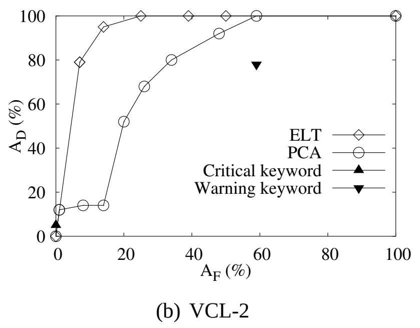
Table II
VCL ANOMALY TYPES."*" DENOTES UNEXPECTED ANOMALIES.

The results show that the critical keyword scheme has very low detection rates for VCL-1 and VCL-2 datasets, and about 70% detection rate for VCL-3. In contrast, the warning keyword scheme has very high false positive rates for VCL-1 and VCL-2 datasets since many warning messages are inserted by developers for debugging purposes. For VCL-3, the warning keyword scheme can achieve 100% detection rate with a slight lower false positive rate than ELT. The critical/warning keyword methods work better for VCL-3 than VCL-1 and VCL-2. The reason is that most anomalous instances in VCL-3 are expected problems that contain critical failure messages and most normal instances do not contain any warning messages. Compared to PCA, ELT can achieve up to 530% higher detection rate (i.e., 95% v.s. 15% true positive rate in Figure 7(b)) under similar false alarm rates (i.e., 15% in Figure 7(b)). The reason is that PCA cannot detect those anomalous instances that share significant similarity with some normal instances while ELT can detect them. We did not show the ROC curves for the PCA MFG and Clustering + NN MFG algorithms since both of them are too expensive to be considered for practical use. We will show their overhead results in Section IV-D.

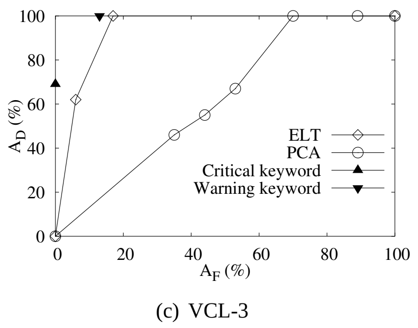
Table III shows the key message extraction and invariant checking results. We can see that ELT can extract 5 to 150 messages from log files containing 269 to 32957 messages on average. We use the full coverage metric to verify the correctness of the extracted messages. We say that the extracted messages have a full coverage if all the messages relevant to troubleshooting are included in the extracted messages. The invariant check results indicate whether a specific type of anomaly violates the system invariant. ELT also



(a) VCL-1



(b) VCL-2



(c) VCL-3

Figure 7. Anomaly detection accuracy comparison among different log analysis schemes.

Anomaly Types	**Num. of extracted key msgs/total msgs**	**Full Coverage**	**Invariant check**
Image reloading failure	18/540	Yes	Pass
Sanitization Failure	22/594	Yes	Pass
Overlapping reservations	50/2395	Yes	Pass
Multi-process forking error	150/4455	Yes	Fail
Multiple attempts to delete user	25/3036	Yes	Fail
Reservation host not in contact	60/2574	Yes	Pass
Reservation failure due to cluster node failure dependency	20/269	Yes	Pass
Predictive reloading failure	4/32957	Yes	Pass

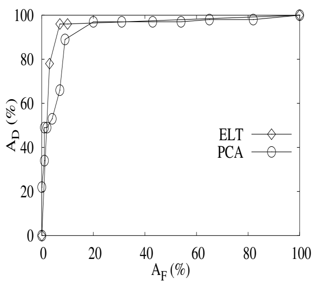
Table III
STATISTICS IN TROUBLESHOOTING REPORT.

```
21:39:28|..|..|deleted|..|executing SSH command on host2: '/
usr/sbin/userdel u1'
21:39:28|..|..|deleted|..|SSH command executed on host2,
returning (0, "none")
21:39:28|..|..|deleted|..|executing SSH command on host2: '/
usr/sbin/userdel u1'
21:39:29|..|..|deleted|..|SSH command executed on host2,
returning (0, "userdel: user u1 does not exi...")
21:39:29|..|..|deleted|..|attempted to delete usergroup for u1
```

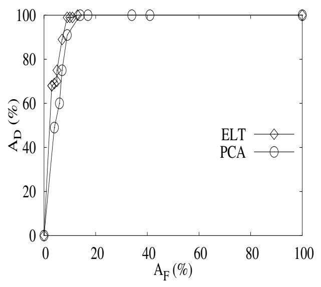
Figure 8. Key messages extracted for multiple attempts to delete user.

successfully reduced the overhead of performing invariant check compared to the original log. For VCL-1, VCL-2 and VCL-3 datasets, the average per-instance invariant checking time was reduced from 600ms to 0.3ms, 67ms to 0.2ms, and 150ms to 0.2ms, respectively.

Figure 6 shows the snippet of the key messages extracted for multi-process forking error, which has been described in Section III-D. Figure 8 shows the snippet of the key messages extracted for the *multiple attempts to delete user error*. The remote commands issued to delete the user generate improper return code. Thus, the management node attempts to delete the user again. We have confirmed with VCL developers that the extracted key messages are able to identify the problem and are useful for troubleshooting.



(a) Hadoop-1



(b) Hadoop-2

Figure 9. Hadoop ROC curves.

C. Hadoop Results

We consider three types of anomalies in Hadoop MapReduce jobs: i) *memory leak bug* where some map tasks contain a memory leak bug that eventually makes the host run out of memory; 2) *non-responding task* where some map tasks contains an infinite loop bug, which causes the jobtracker to mark them as non-responding and kill them; and 3) *disk out of space* where some tasks failed after processing a large amount of data because the system ran out of disk space. We used the sort application processing the RandomWriter data, both of which are provided by the Hadoop software distribution.

Figure 9 shows the detection results for ELT and PCA schemes. The results show that ELT achieves similar detection accuracy as PCA. Both ELT and PCA can achieve much better detection accuracy for the Hadoop datasets than for the VCL datasets. The reason is because most anomalous tasks have very different features from all normal instances. Both ELT and PCA can easily detect such anomalies. Hadoop also contains relatively fewer number of message templates. As a result, the log features are easier to distinguish than VCL.

In contrast to VCL reservation instances, Hadoop task instances have relatively fewer log messages. Anomalous tasks contain even fewer log messages than normal tasks since anomalous tasks often last for a very short period of time. Figure 10 shows the key messages extracted for the *disk out of space* anomaly. The second message shows that no space is left on the Hadoop Distributed File System

```
In TaskLauncher, current free slots : 1 and trying to launch
attempt_201103252330_0009_r_000018_4
Task: attempt_201103252330_0009_r_000018_4 - Killed :
java.io.IOException: No space left on device
attempt_201103252330_0009_r_000018_4 done; removing files.
LaunchTaskAction (registerTask):
attempt_201103252330_0009_r_000018_4 task's state:FAILED_UNCLEAN
```

Figure 10. Hadoop key messages extracted for disk out of space error.

Dataset	**ELT**	**PCA**	**PCA MFG**	**Clustering +NN MFG**
VCL-1	9±0.5 min, 600MB	4±0.5 min, 400MB	Impractical (>256TB)	8 hours, 600MB
VCL-2	135±2 min, 7GB	30±2 min, 3GB	Impractical (>256TB)	>25 hours, 7GB
VCL-3	20±1 min, 7GB	18±1 min, 3GB	Impractical (>256TB)	13 hours, 7GB
Hadoop-1	31±1 min, 2GB	10±1 min, 400MB	>8 hours, 1GB	>15 hours, 2GB
Hadoop-2	170±2 min, 7GB	25±1 min, 1.5GB	>10 hours, 2.5GB	>20 hours, 7GB

Table IV
PROCESSING OVERHEAD COMPARISON.

(HDFS), and the task is subsequently killed. On average, the anomalous instance of this anomaly type contains about 20 messages while normal instance usually contains up to 100 messages.

D. Overhead Results

Table IV shows the overhead comparison results. The results show that ELT has low processing overhead compared to other schemes (i.e., PCA MFG, Clustering+NN MFG) that use the fine-grained features. ELT achieves a low execution time of 9 minutes for VCL-1 and the highest of 170 minutes for Hadoop-2. The current prototype of ELT has larger memory consumption than PCA. The reason is mainly due to an unoptimized implementation for object allocations during the clustering process. We can significantly reduce the memory consumption of ELT by changing the object allocation implementation, which is part of our on-going work. PCA (using the message frequency vector) has the lowest runtime and memory consumption since it only uses coarse-grained log features. However, as shown by our previous results, PCA can only achieve very low detection accuracy for the VCL logs since the coarse-grained log feature cannot distinguish the anomalous instances that share certain similarity with some normal instances. PCA MFG cannot be run on a single machine for VCL datasets due to extremely high memory requirement (eg. >256TB). The high memory requirement is due to the space requirements of covariance matrix which is $O(p^2)$, where p is the size of feature vector (for the MFG of VCL, $p = 4164 \times 4164$). PCA MFG also has relatively longer processing time for Hadoop datasets. Clustering+NN MFG also has very high processing overhead.

V. RELATED WORK

Previous work has investigated unsupervised learning techniques for automatic log analysis. For example, Xu et. al [20] applied PCA based anomaly detection to Hadoop Distributed Filesystem (HDFS) console logs to identify the anomalous HDFS block instances. However, as we have shown in the experiments, PCA can only detect those anomalous instances that are very different from all normal instances. Moreover, PCA using the message frequency feature is very sensitive to repeating message patterns, which may cause high false positive rate of the PCA scheme. Lim et. al [13] applied frequency analysis technique to identify message patterns that mostly occur during known failure event. This approach is more suitable when the failure events are known a priori. In running production systems known as well as unknown failures can occur. Our scheme differs from previous work on being able to identify unknown failure as well.

Previous work has also explored other unsupervised learning techniques to detect performance anomalies, resource usage and prediction of fatal events. Fu et al. used the clustering technique to extract log keys and constructed finite state automaton to diagnose performance anomalies [8]. Kavulya et al. applied an instance based learning technique on logs of a production MapReduce cluster to characterize resource utilization patterns and find failure sources [11]. The SALSA project used Hadoop logs to derive the control flow and the data flow, and compared the probability distributions of state durations across hosts to identify anomalies such as disk hog and cpu hog [17]. Salfner et al. proposed a logfile structure consisting of hierarchical numbering of event types and sources such that it is amenable for automatic log analysis techniques like clustering [16]. Li et al. proposed an integrated framework to mine logs to infer temporal dependency between log events from the cumulative distribution function of the events' waiting times [12]. Palatin et al. employed a distributed outlier detection algorithm HilOut, a variant of the nearest neighbor approach, over processed log files stored in different nodes of a grid system to identify misconfigured machines [15]. Our work also explores unsupervised learning methods to achieve fully automatic log analysis. However, our scheme differs from previous work by adopting a hybrid analysis model to achieve both high accuracy and low overhead. Moreover, our scheme can not only detect anomalies but also simplify the troubleshooting task by extracting key messages and performing efficient invariant check.

Our work is also related to previous function call trace analysis work. Mirgorodskiy et al. used k-nearest neighbor approach on per function time profile to identify potential failure traces in distributed systems [14]. Jiang et al. used training data to construct automata to characterize normal traces of distributed system and then used it to detect abnormal traces [9]. Yuan et al. applied support vector machines classifiers to categorize system event traces and correlated them to known system problem to determine root causes [21]. Zheng et al. trained decision tree to user request traces with user visible failures and used the decision tree to identify the failure causes for runtime user request in Internet systems [22]. Dickinson et al. proposed to use clustering over program execution traces based on the function call relationships to identify failure executions [7]. Chilimbi et al. correlated program path profiles with program failures to identify the cause of program failures [5]. In contrast to execution trace analysis, our scheme is unobtrusive, which does not require modifying original program or installing extra system call tracing program.

VI. Conclusion

We have presented ELT, a practical and efficient console log based troubleshooting system for large-scale cloud computing infrastructures. ELT employs a novel hybrid log analysis approach that combines coarse-grained and fine-grained log analysis to achieve both high accuracy and low overhead. ELT can automatically extract a few key messages and perform invariant check to significantly simplify the anomaly diagnosis process. We have implemented a prototype of the ELT system and tested it using several real console logs collected on a production cloud computing system and a Hadoop cluster in our lab. Our experimental results show that ELT can achieve higher detection rate and lower false alarm rate than existing schemes for the VCL logs. For Hadoop logs where anomalous instances show distinct log features and the number of message templates is small, ELT achieves similar detection accuracy as previously proposed PCA scheme. ELT can extract correct key messages for all detected anomalous instances. More importantly, ELT found two software bugs that were missed by current cloud system management practice.

In the future, we plan to optimize the implementation of the ELT system (e.g., reduce the memory consumption of hierarchical clustering process) and develop online log analysis system based on ELT.

VII. Acknowledgment

We would like to thank VCL system administrators Aaron Peeler and Andy Kurth for providing us with the log data and their generous help on validation. We also thank Brent Miller from IBM for many useful discussions. We would also like to thank our shepherd Miroslaw Malek for his guidance. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, and IBM Faculty Award. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] Amazon elastic computing cloud. http://aws.amazon.com/ec2/.

[2] Apache vcl. https://vcl.ncsu.edu.

[3] Armadillo - c++ linear algebra library. http://arma.sourceforge.net/.

[4] Hadoop. http://hadoop.apache.org/.

[5] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proc of ICSE*, pages 34–44, Washington, DC, USA, 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[7] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. of ICSE*, pages 339–348, Washington, DC, USA, 2001.

[8] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of ICDM*, pages 149–158, Los Alamitos, CA, USA, 2009.

[9] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-resolution abnormal trace detection using varied-length n-grams and automata. In *Proc. of ICAC*, pages 111 –122, 2005.

[10] L. Kaufman and P. Rousseeuw. *Finding Groups in Data An Introduction to Cluster Analysis*. Wiley Interscience, New York, 1990.

[11] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proc. of CCGrid*, pages 94 –103, 17-20 2010.

[12] T. Li, F. Liang, S. Ma, and W. Peng. An integrated framework on mining logs files for computing system management. In *Proc. of KDD*, pages 776–781, New York, NY, USA, 2005.

[13] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Proc. of DSN*, pages 398 –403, 24-27 2008.

[14] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *Proc. of SC*, page 88, New York, NY, USA, 2006.

[15] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff. Mining for misconfigured machines in grid systems. In *Proc. of KDD*, pages 687–692, New York, NY, USA, 2006.

[16] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. In *Proc. of IPDPS*, 2004.

[17] J. Tan, X. Pan, S. Kavulya, R. G, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Proc. of WASL*, 2008.

[18] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[19] R. Vaarandi. Mining event logs with slct and loghound. In *Proc. of NOMS*, pages 1071–1074, 2008.

[20] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of SOSP*, pages 117–132, New York, NY, USA, 2009.

[21] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *Proc. of EuroSys*, 2006.

[22] A. X. Zheng, J. Lloyd, and E. Brewer. Failure diagnosis using decision trees. In *Proc. of ICAC*, pages 36–43, Washington, DC, USA, 2004.

[23] P. Zhou, B. Gill, W. Belluomini, and A. Wildani. Gaul: Gestalt analysis of unstructured logs for diagnosing recurring problems in large enterprise storage systems. In *Proc. of SRDS*, 2010.