2016 IEEE 35th Symposium on Reliable Distributed Systems

# CRUDE: Combining Resource Usage Data and Error Logs for Accurate Error Detection in Large-Scale Distributed Systems

Nentawe Gurumdimma*[†], Arshad Jhumka*, Maria Liakata*, Edward Chuah[‡], James Browne[‡]

*Department of Computer Science, University of Warwick, Coventry, CV4 7AL UK
[†]Email: N.Y.Gurumdimma@warwick.ac.uk
[‡]University of Texas - Austin

***Abstract*—The use of console logs for error detection in large scale distributed systems has proven to be useful to system administrators. However, such logs are typically redundant and incomplete, making accurate detection very difficult. In an attempt to increase this accuracy, we complement these incomplete console logs with *resource usage* data, which captures the resource utilisation of every job in the system. We then develop a novel error detection methodology, the CRUDE approach, that makes use of both the resource usage data and console logs. We thus make the following specific technical contributions: we develop (i) a clustering algorithm to group nodes with similar behaviour, (ii) an anomaly detection algorithm to identify jobs with anomalous resource usage, (iii) an algorithm that links jobs with anomalous resource usage with erroneous nodes. We then evaluate our approach using console logs and resource usage data from the Ranger Supercomputer. Our results are positive: (i) our approach detects errors with a true positive rate of about 80%, and (ii) when compared with the well-known Nodeinfo error detection algorithm, our algorithm provides an average improvement of around 85% over Nodeinfo, with a best-case improvement of 250%.**

***Index Terms*—anomaly detection; resource usage data; faults; detection;large-scale HPC systems; unsupervised;event logs;**

## I. Introduction

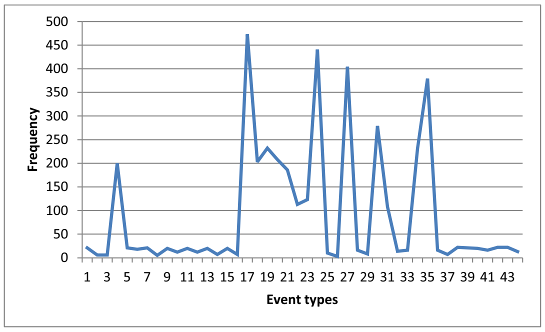
Event or console logs are system administrator's primary source of information about the state of a cluster or HPC system. Events generated by the system are logged sequentially in event logs. Most of the events generated are believed to be reports about the normal behaviour of the system. Events reported within a given time window form what we call an event sequence or pattern. For example, events that are reported within a one hour time period can be regarded as a pattern. In case of a system failure, faults that are likely to be symptomatic of such a failure, i.e., root causes, could be traced with the events preceding the failure. In such cases, the event logs contain an interleaving of normal event messages and error messages (symptomatic of faults that occurred in the system).

Detecting fault occurrences (i.e., errors) using event logs and other data has received good attention, with reasonably good results, e.g., [1]–[4]. Most of these approaches have focused on identifying individual error messages (that capture the nature of the faults) within the data. However, most of the times, individual events may not be sufficient to indicate an impending system failure. Other approaches for error detection within patterns in the recent past have used the concept of log entropy of the event log messages seen within the patterns [5] [6], [7]. Log entropy generally leverages the inherent changes in the frequency of events to capture the behaviour of the system. Oliner *et al.* [5] attempted to capture the sequence information, or what they called NodeInfo, of a nodehour, an area within which the log can be considered as faulty. A recent approach combines both entropy and the concept of mutual information of patterns to discriminate between faulty and non-faulty patterns [8].

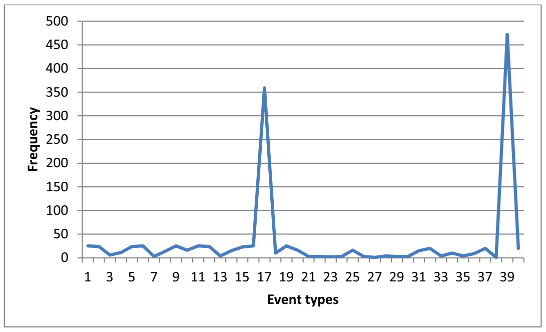
The challenge is that, even though the above mentioned attempts reported good detection results, they are however restricted mainly to failures which are remarkably characterised by frequent reporting of events, as seen in Figure 1(a). However, this is not the case for some types of failures experienced in HPC systems. For example, some failures caused by soft errors may not produce visible and abnormally frequent events that would signify a faulty behaviour. More so, some faults can induce a silent behaviour (i.e., events are sparse) and only become evident when the failure is about to occur (see Figure 1(b)). In such a case, this provides a small mean time to failure in the sense that the time between the error detection and the actual failure is small. In these cases, approaches based on entropy and mutual information, as mentioned above, will not totally help.

In general, event logs are known to be *incomplete* (the event logs provide insufficient information regarding the behaviour of HPC systems ) and *redundant* (the logs may contain several error events that relate to the same fault) and, as such, may not be readily used to accurately detect errors in the system. To circumvent the problem induced by the incompleteness issue of event logs, we complement these logs with *resource usage* data in this paper. In short, resource usage data captures the amount of resources consumed or produced by all the jobs in the system. For example, the resource usage data file may contain the amount of memory a job is using at a given time as well as the number of messages a job has output on the network. It has been generally argued that anomalous resource usages could be precursors to system failures [9]. Unfortunately, most supercomputer systems do not make such usage data available, making it difficult for researchers to verify this.

Leveraging both resource usage and error logs, we propose the *CRUDE* methodology to detect errors more accurately in the system. The CRUDE approach is a *novel* approach that uses both event logs and resource usage logs to extract deviations in behaviors in detecting errors. The intuition behind the use of resource usage data is that they will enable the detection of anomalous jobs which, when coupled with the error messages originating from the erroneous behaviour of nodes (as captured from the event logs), will provide a stronger evidence of the existence of an error in the system. The resource utilization would provide with better understanding of the behaviour of the system in the face of few



(a) bursty fault sequence



(b) Silent fault sequence

Fig. 1. Faulty sequence showing behaviour of fault events logged within an hour. The peaks indicates likely failure periods

precursor events in the event logs. The CRUDE methodology is *unsupervised*, meaning that our approach does not require that resource usage data or event logs to carry data labels. This is *extremely* valuable and relevant for two main reasons: (i) When resource usage data is used, given that little work has been done based on resource usage data, very little labelled data is available and (ii) the ability to provide labelled data requires extensive and detailed knowledge about the system, which may not always be available, but the labelling process is also time-consuming.

In this paper, we make the following *novel* contributions:

- We develop an algorithm that uses the event logs to detect erroneous behavior. The algorithm achieves this by *clustering* together nodes that exhibit similar behaviour, through the use of the mutual information and entropy concepts.
- Resource usage log data provides an understanding of the resource usage of the cluster system. Anomalous utilization may suggest a deviation in behaviour due to some errors in the system. To be able to capture the anomalous behaviour of jobs running within the system, we employ an unsupervised detection approach, namely the *Principal Component Analysis* (PCA), to do this. Anomalous jobs present within a pattern may signify the presence of faults which could ultimately lead to failure.
- Lastly, we utilised both the *mutual information* and *entropy* of a pattern from event logs and the outlier/anomaly level of sequences from its resource usage data to infer if a sequence is likely to lead to failure or not. We thus propose a detection algorithm based on these information from both event logs and resource usage data. We subsequently compare our method with an existing approach, namely *Nodeinfo*.

Our approach is very promising: it is able to detect faulty sequences with high accuracy and, when compared to the well-known Nodeinfo approach [5], it outperforms it greatly. For example, the CRUDE approach obtains a best-case improvement over Nodeinfo of $250\%$ and an average improvement of $85\%$.

## II. System and Fault Models

In this section, we first define the terms used and we then describe a general cluster system model. We subsequently explain the fault model and explain the structure of message or event logs and the resource usage data.

### A. Basic Definitions

- **Event:** A single line of text containing various fields (*time-stamp, nodeID, protocol, application, error message*) that reports a given specific activity of the cluster system. Such an event is also often called a *log message*.
- **Event logs:** A set or sequence of events capturing the various activities that occurred within a cluster system.
- **Failure event:** This is an event that is often associated with and/or indicative of a system failure.
- **Log sequence:** A log sequence (or simply sequence) consists of one or more consecutive events that are logged within a given time period. In this paper, we use the terms *sequence* and *pattern* interchangeably.

### B. System Model and Cluster Logs

Here we explain the system model and explain the structure of the logs, as well as the event types contained in the logs.

*1) Cluster System:* A cluster system contains a set of nodes, jobs or tasks, production time, job scheduler and sets of software components (e.g. parallel file system). The job scheduler allocates jobs to nodes to execute within a certain production time, and all the components involved write messages to a writing container. This is a common model for most of the cluster vendors like Cray, IBM etc.

*2) Event Logs/Message Logs:* Most Linux-based cluster systems log events using the POSIX standard [10]. Rationalized logs (*ratlogs*) [11] is an example. This standard allows freedom for the formatting of logs, which means that there are variations in different implementations. For example, the set of attributes used by IBM's Blue Gene/L to represent its components is different from that of the Ranger supercomputer. An example of Ranger ratlogs event can be seen below:

```
Apr 4 15:58:38 012324 mds5 kernel:
LustreError: 138-a: work-MDT0000:
A client on nid .*.*.5@o2ib was evicted
due to a lock blocking callback
to .*.*.5@o2ib timed out: rc -107
```

It has following attribute fields: the **Time-stamp** (*Apr 4 15:58:38*) containing the month, date, hour, minute and seconds at which the event was logged, the **Job-id** (012324) which specifies the particular job that reported the event, the **Node Identifier** or **Node Id** (*mds5*) identifies the node from which the event is logged, the **Protocol Identifier** (*kernel*) and **Application** (*LustreError*) provides information about the sources of events (or messages). A **message** (e.g., *A client on nid *.*.*.5@o2ib was evicted due to a lock blocking callback to *.*.*.5@o2ib timed out: rc -107*) contains alphanumeric words and English-only words sequence. The English-only words sequence (A client on nid was evicted due to a lock blocking callback to timed out) is believed to give an insight into the error that has occurred. They are referred to as *Constant*. The alpha-numeric tokens (**.*.*.5@o2ib ,rc-107*) also called *Variable*, are believed to signify the interacting components within the cluster system.

## C. Resource Usage Data

Resource usage data are collected by TACC_stats [12] (Texas Advanced Computing Center). It is a job-oriented and logically structured version of the conventional *Sysstat* system performance monitor. TACC_stats records hardware performance monitoring data, Lustre filesystem operation counts, and InfiniBand device usage. The resource usage data collector is executed on every node and is mostly executed both at the beginning and end of a job via the batch scheduler or periodically via cron. The collection of resource use data requires no cooperation from the job owner and requires minimal overhead.

Each stats file is self-describing and it contains a multi-line header, a schema descriptor and one or more record groups. Each stats file is identified by a header which contains the version of TACC_stats, the name of the host and its uptime in seconds. An example of a stats file header is shown, for clarity:

```
$tacc_stats 1.0.2
$hostname i101-101.ranger.tacc.utexas.edu
$uname Linux x86_64 2.6.18-194.32.1.el5
 _TACC #18 SMP
Mon Mar 14 22:24:19 CDT 2011
$uptime 4753669
```

A schema descriptor for Lustre network usage parameters is seen below:

```
!lnet tx_msgs,E rx_msgs,E rx_msgs_dropped,
E tx_bytes,E,U=B rx_bytes E,U=B ...
lnet - 90604803 95213763 1068
808972316287 4589346402748 ...
```

A schema descriptor has the character ! followed by the type, and followed by a space separated list of elements or counters. Each counter consists of a key name such as *tx_msgs* which is followed by a comma-separated list of options. These options include: (1) $E$ meaning that the counter is an *event counter*, (2) $C$ signifying that the value is a control register and not a counter, (3) $W =< BITS >$ means that the counter is $< BITS >$ wide (32-bits or 64-bits), and (4) $U =< STR >$ signifying that the value is in units specified by $< STR >$ (e.g.: U=B where B stands for Bytes.). From the schema descriptor above, *lnet - 90604802* gives records of the number of messages transmitted in the Lustre network. TACC_stats is open sourced and can be downloaded and installed on Linux-based clusters.

## D. Fault Model

Event logs are the primary sources of information for system administrators when the cluster system fails. When jobs are executing in the system, they produce mostly *mostly* normal messages (or events). However, there are events that are symptomatic of a problem within the system. In such a case, the job(s) may produce error messages that tend to capture the type of problem occurring in the system.

In this paper, we make the following assumptions: A fault occurs in the system that will affect one or more jobs. Once a job is affected, it will start to output error messages. For example, a network timeout will result in a "Network timeout" log message being recorded. Thus, the event log will contain an interleaving of normal messages and error messages. If the error is not handled by the jobs (or system), then a failure will eventually happen [13]. The failure is captured in the event log through a special message, which we call a *failure* event. For example, such a failure in IBM BlueGene/L is characterized by *FAILURE* severity level while, in the Ranger Supercomputer, these failures are characterized by a *compute node soft lockups*.

The rate at which error messages are generated by the jobs depend in part on the nature of the fault and also on the extent of communication among the jobs. Typically, the greater the communication among jobs, the higher the frequency of error messages being recorded.

## III. Problem Statement

Given the stated thrust of our paper is the development of a methodology for error detection in large-scale distributed systems, our objective can be stated as follows: Given an event sequence and resource usage data sequence, the problem is to determine whether the event sequence contains messages that indicate impending system failure.

## IV. Methodology

### A. Methodology Overview

We argued earlier that, given individual or single events, it is difficult to determine if a system failure would likely occur. We now briefly summarize our methodology. First, event logs are reported as stream of events occurring in time. However, to keep the log analysis tractable, it is beneficial to break a long sequence into smaller sequences of, ideally, similar size, which is measured in time units (obtained from the timestamps). The size of the small sequence is referred to as the *time window*. The choice of the time window is dependent on how informative events within such a time window are, i.e., the time window needs in general to be small, but large enough so as to contain enough informative events to characterise the time window. Given that event logs contain both normal and error messages [3] and are generally incomplete, we seek to complement these event logs with resource usage data to aid error detection. Here, our premise is that abnormal resource usage is indicative of a fault having occurred in the system that may lead to a system failure.

Next, we transform these data into a format that captures the system behaviour within the chosen time window and can be easily used by any error detection algorithm. We then create a feature matrix from the event log and another from the usage data. We subsequently extract the *mutual information* and *entropy* of the interacting nodes and event types respectively. This is done after nodes with similar behaviour are clustered together. We then determine the "outlierness" of a sequence by performing a principal component analysis (PCA) outlier detection on the resource utilisation data feature matrix. This step returns the anomaly score of a sequence which is then used in error detection. The detection process leverages the mutual information, entropy and anomaly score of the sequence to determine whether the sequence is likely to lead to system failure. This section focuses on detailing our methodology as seen in work flow of Figure 2.

## B. Data Transformation

**Notations and Terminologies:** Before explaining our methodology, we briefly mention the notations used for clarity. We will denote a specific entity using a small letter, while we will use the associated capital letter to denote the total number of that entity. For example, we will denote a specific node by $n$ (or $n_i$) and the total number of nodes in the system by $N$.

This section focuses on obtaining appropriate system data that can represent nodes and running jobs behaviour. These behaviours inherently describe the state of the system within a given time to us. The data (both event logs and resource usage data) need to be transformed into a format suitable for analysis. In order to detect anomalies in resource usage, we extract the attributes/elements which capture the state of the resources for every job on every node within a given time. Generally, we extract features from event logs first and then from resource usage data.

*1) Event logs Features:* For event logs, we denote by $e_i^l$ the number of events $e_l$ produced by node $n_i$ within a given time window. Given $E$ different possible events, we obtain a vector $[e_i^1 \dots e_i^E]$ which contains the number of occurrences of each event on node $n_i$. We call this vector a *event feature* of node $n_i$. This concept is analogous to the bag of words concept used in information retrieval, which has been proven to be effective in capturing relationships between terms/words/messages. We reuse the same idea because the event log messages are the primary source of data about the health of the system. The features that are produced depend of the frequency distribution of event logs. Hence, for a given time window $t_w$, each node will produce an event feature, resulting in a matrix, where each row represents an event feature of a given node and each column represents the number of occurrences of a given event on different nodes in the system. For a given time window $t_w$, we denote the resulting *event feature* matrix by $F_{t_w}$, as shown in Figure 3. Henceforth, where there is a mention of sequence, we are referring to the matrix representing the sequence.

*2) Resource Usage Features:* A system will typically have a number of counters that capture different resource aspects of the system during execution. These counters indicate the amount of resources they are associated with that are being produced or consumed. For example, a memory counter may capture the amount of memory that is being used by a given job at a given time. Denoting a given job by $j_k$ which executes on node $n_i$ during time window $t_w$, we denote the amount of a resource $r$ used by $j_k$ on $n_i$ during $t_w$ by $U_{k,i}^{r,w}$. We call the vector of resources used or produced by a given job $j_k$ on node $n_i$ during time window $t_w$ as a *resource feature* and we denote it by $U_{k,i}^w$ and is defined as $U_{k,i}^w = [U_{k,i}^{1,w}, \dots U_{k,i}^{R,w}]$. Whenever the exact location of a job is unimportant, we will denote the usage matrix by $U_k^w$, i.e., $U_k^w$ denotes the different resources used or produced by job $j_k$ in time window $t_w$.

Similar to event features, we construct a $J \times R$ *resource feature* matrix[1] for a given time window $t_w$, which we denote by $U^{t_w}$, where $J$ is the number of jobs and $R$ the number of resources. Each row of the matrix is a resource feature of a job and each column is the vector that captures the amount of the specific resource used or produced by all the jobs in the given time window. Specifically, the value associated with $U^{t_w}[k, r]$ represents the amount of resource $r$ that has been used or produced by job $j_k$ during time window $t_w$.

## C. Clustering and Feature Extraction

Given the event features created from the *event logs* as depicted by the matrix of Figure 3, we perform clustering to group nodes exhibiting similar behaviour. The underlying reason for this is that nodes executing similar jobs tend to log similar events. More importantly, this also means that similar errors could get to be reported by these nodes, enabling the capture of node level behaviours of the system through clustering. To this end, we employ the hierarchical clustering approach explained in [8]. Hierarchical clustering is appropriate due to the fact that two system runs, with exactly the same errors, may end up in different clusters due to the timing of the error occurrences. Hence, to reduce the impact of the temporal dimension of error occurrences, hierarchical clustering will cluster these two runs together. The mutual information and entropy of the individual features clustered are also computed.

*1) Clustering:* Makanju *et al.* [14] has shown that the state of systems can be discovered through information content clustering. As stated earlier, Oliner *et al.* [5] verified the hypothesis that similar nodes correctly executing similar jobs should produce similar logs. This implies that a fault in a node or job could cause a cascading effect or alter communicating jobs, which could result in producing similar error events. This also means that we can leverage the homogeneity of large scale systems to improve fault detection. To achieve this, we employ clustering to group nodes with similar behaviour to be able to extract the group behaviour.

Clustering groups similar data points together in such a way that those in different group are very dissimilar. In this case, we group nodes with similar behaviour in terms of the events they log within given time window. Hence, given a sequence of events within a given time window, the event features which are formed from the data transformation section is an $N \times E$ matrix, $F_{t_w}$, (see Figure 3) with $N$ being the number of nodes (rows) and $E$ the number of event types (columns). $e_i^l$ is the count of event types $e^l$ produced by node $n_i$. We employ a simple centroid based hierarchical clustering (see Algorithm 1) to perform this. We employ this clustering approach because we assume there are one or more outlier nodes behaviour within a sequence, and a centroid based clustering as this is not sensitive to outliers.

In any distance-based clustering, the distance metric is key to achieving a good result. For our purpose, we defined a distance metric that could capture the informativeness of each node within the sequence and/or the correlation between the events types in the sequence. Hence we employ the *Correlation Metric* as our clustering similarity distance metric.

[1]$U$ can be used to denote usage matrix when the $t_w$ is not used for purpose of clarity

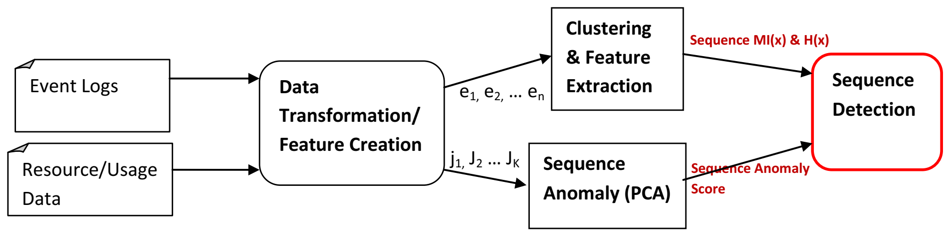


Fig. 2. Methodology work flow showing steps taken to achieve detection

$$F_{t_w} = \begin{pmatrix} e_1^1 & e_1^2 & . & . & . & e_1^E \\ e_2^1 & . & . & . & . & . \\ . \\ . \\ . \\ e_N^1 & e_N^2 & . & . & . & e_N^E \end{pmatrix}$$

Fig. 3. Data matrix $F_{t_w}$ of a sequence with $N$ nodes and $E$ event types, where $e_i^l$ denotes the number of occurrences of event $e_l$ by node $n_i$.

*Correlation Metric* (Corr), is based on the correlation within nodes of the sequence. Given two feature vectors of $F_{t_w}$ as $x_i, x_j$, then we compute the similarity as follows:

$\text{cov}(x_i, x_j) = \frac{1}{N}\sum_{k=1}^{N}(x_i^k - \overline{x_i})(x_j^k - \overline{x_j})$,

$std(x_i) = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(x_i^k - \overline{x_i})^2}$, where $\overline{x_i} = \frac{1}{N}\sum_{k=1}^{N} x_i^k$ is the mean of vector $x_i$;

$$sim(x_i, x_j) = 1 - \left|\frac{\text{cov}(x_i, x_j)}{std(x_i) * std(x_j)}\right|. \tag{1}$$

We treat $sim(.)$ as similarity or distance measure between two feature vectors.

**Algorithm 1** An algorithm for Clustering similar behaving nodes

```
1:  procedure CLUSTERING($\mathbb{C}, \lambda$)
2:      Inputs: $|\mathbb{C}|$ event features with $c_i \in \mathbb{C}, i = 1 \ldots n$; MinimumSimilarityThreshold $\lambda$.
3:      Initially assume each node vector as unique, with each as a cluster on its own;
4:      for $i = 1$ to $|\mathbb{C}| - 1$ do
5:          for $j = i + 1$ to $|\mathbb{C}|$ do
6:              $sim = sim(c_i, c_j)$        ▷ Find similarities of centroids of $c_i$ and $c_j$
7:              if ($sim \geq \lambda$) then
8:                  Merge $c_i$ and $c_j$;
9:              end if
10:         end for
11:         add the merged cluster to set of new clusters
12:     end for
13:     Repeat step 1 with new set of merged clusters until similarity less than $\lambda$ is achieved
14:     Return new $m$ sets of clusters of similar nodes
15: end procedure
```

*2) Sequence Feature Extraction:* The amount of information provided by the nodes within a given time window and also the informativeness of the event types could provide useful hints on unusual behaviour of such nodes within the time window. We believe that these behaviours can be succinctly represented by these two features of the sequence: *Mutual Information* ($I$) and *Entropy* (H) of the event types produced by each node within the given time window. To further support our approach, it has been argued that changes in entropy are good indicators of changes in the behaviour of distributed systems and networks [15]. This informed our decision to extract these event features. We assume that a higher uncertainty (entropy) with reduced mutual information (i..e, more unrelated information) could signify abnormal system behaviour or a failure sequence, with the converse being true for normal behaviour.

**Sequence Mutual Information (I)**
Mutual Information measures the relationship between two random variables especially when they are sampled together. Specifically, it measures how much information a random variable $e_i$ (event feature) carries about another variable $e_j$. Essentially, it answers the question about the amount of information event features $e_i$ and $e_j$ carry about each other. Hence, given clusters of event features $\mathbb{C} = \{c_1, ..., c_m\}$ and $c_i = \{e_1, ..., e_n\}$, containing set of similar event features, then,

$$I(c_i) = \sum_{j=1}^{N-1}\sum_{k=j+1}^{N} p(e_j, e_k)\log\frac{p(e_j, e_k)}{p(e_j)p(e_k)} \tag{2}$$

where $p(e_j, e_k)$ is the joint distribution of event features of cluster $c_i$, $p(e)$ is the marginal distribution of event feature $e$. Then, $I(\mathbb{C})$ is given by: $I(\mathbb{C}) = \frac{1}{|\mathbb{C}|}\sum_{c_i \in \mathbb{C}} I(c_i)$. Nodes within a sequence characterised by frequent events logging and probably similar events due to the same fault will have high mutual information.

**Sequence Entropy (H)**
Entropy, on the other hand, is a measure of uncertainty of a random variable. In other words, when the information content of event types of a sequence are highly unpredictable, then the sequence has high entropy. For each event cluster $c_i$,we define entropy as follows:

$$H(c_i) = -\sum_{e_j \in c_i} p(e_j)\log p(e_j) \tag{3}$$

where $p(e_j)$ is the distribution of event types of event features in $c_i$. The entropy of the sequence is then given by the average of the entropies of each cluster of the sequence. The changes observed in entropy are usually good indicators of changes in the behaviour in the cluster system. Such changes may point to imminent failure in cluster system.

## D. Jobs Anomaly Extraction

*1) Overview:* In a cluster system, a large amount of nodes are present with hundreds of jobs running. The resource usage data contains statistics about the usage of resources (e.g. CPU, I/O transfer rates, virtual memory utilization etc) of each job running on these nodes in the cluster system within given time. This information provides useful hints regarding some potential unusual behaviour of a given job in terms of its rate of resource utilization. Hence, the resource data is transformed into an analogous (job and counter) matrix as explained in Section IV-B2, and is used as the input data. The aim here is to obtain a set of anomalous jobs, which could significantly point to problem(s) in the cluster system, as observed by the anomalous jobs within a period. A column vector of the new matrix represents a job and a row represent a counter or attributes (e.g. *dirty_pages_hits, read_bytes, tx_msgs* etc) (see Table I for full list of counters).

There is a significant number of research work on approaches for detecting anomaly. However, since the data is not labelled with normal job behaviours, one option is to use an unsupervised approach to detect anomalous jobs. Principal Component Analysis (PCA) has been a widely and efficiently used feature extraction method in different fields [16] and also for identifying system errors [17]. We utilise PCA approach to find anomalous jobs in resource usage data.

*2) Principal Component Analysis (PCA):* In this section, we briefly explain the PCA technique and its application for anomaly detection purposes. PCA basically determines the principal direction of a given set of data points by first constructing the covariance matrix of the data and then finding the dominant eigenvectors (principal directions). These eigenvectors are also seen as the most informative of the original data space. Let $U = [j_1, j_2, ..., j_n]^T \in \Re^{n \times k}$, with each row $j_i$ representing a $k$-dimensional data instance of a job, and $n$ being the number of data instances or jobs, then PCA can be formulated as an optimization problem where it first calculates the covariance matrix $C$ of $U$ as:

$$C = \frac{1}{n} U U^T. \tag{4}$$

Then the *eigenvalues*, $\lambda_i$, ($i = 1...p$ and $p < k$) are calculated and sorted in decreasing order with the first being the highest eigenvalue. From this, a projection matrix $V = [a_1, a_2, ..., a_p]$ consisting of $p-$dominant eigenvectors is constructed, with each data point projected into a new subspace as:

$$X = V^T J. \tag{5}$$

The $p$-dominant eigenvectors produced are in decreasing order of dominance.

*3) Anomalous Jobs Detection in Usage logs Using PCA:* This section looks at the use of PCA to identify anomalous jobs running on a cluster system within a given time window. We employ an approach with similar property of principal directions, as one made in [16]. The basis for the approach is that, for every data point, removing or adding it contributes to the behaviour of the most dominant direction. This is because PCA relies on calculating the mean and the data covariance matrix in obtaining eigenvectors, and it is observed to be sensitive to the presence of an outlier. Hence, in this approach, the *outlierness* of a job can be determine by the variation in the dominant principal direction. Specifically, by adding an outlier, the direction of dominant principal component changes, but a normal data point will not change it.

**Sequence Anomaly**

The assumption here is that any anomalous job will cause a deviation from the leading principal direction. Therefore, a sequence anomaly (or "outlierness") is the average value of all the outliers or anomalous jobs present within such time window. For example, Figure 4 shows the distribution of resource usages by various jobs and the points in bold (above 0.0009 are likely anomalous jobs).

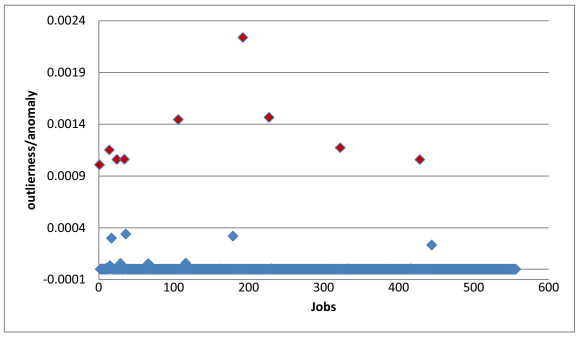


Fig. 4. Jobs outlierness of a sequence using PCA

So, given data points $U = [j_1, j_2, ..., j_n]^T$, the leading principal direction, $d$, of matrix $U$ is extracted. Then, for each data point $j_i$, we obtain the leading principal component, $d_i$, of $U$ without data point $j_i$. The outlierness, $a_i$, of $j_i$ is the dissimilarity of $d$ and $d_i$. The algorithm is as seen in Algorithm 2.

**Algorithm 2** Sequence Anomaly from Usage data

```
 1: procedure AnomalyScore($U, \gamma, j_i$)
 2:     $d$ = Leading Principal direction of $J$
 3:     $d_i$ = Leading Principal direction of $J$ without data point $j_i$.
 4:     $a = 0$                                   ▷ initialise anomaly score
 5:     $sim(d_i, d) = \frac{d_i \cdot d}{||d_i|| ||d||}$
 6:     $a = 1 - |sim(d_i, d)|$
 7:     if ($a \geq \gamma$) then
 8:         return $a_i$                          ▷ returns anomaly score
 9:     else
10:         return 0                              ▷ returns 0 for not anomalous
11:     end if
12: end procedure
```

Any data point with outlierness greater than given threshold $\gamma$ is regarded as *anomalous*.

## E. Detection of Failure Patterns

The algorithm presented in this section is aimed at detecting a sequence of events within a chosen time window that most likely indicates the presence of errors that will lead to a system failure. The algorithm leverage the features that capture the behaviour of nodes within a sequence (i.e., mutual information and entropy), where high values (i.e., above a threshold) of these features indicate the presence of errors which would likely lead to a failure. However, this may not be sufficient to conclude that a failure is imminent as event logs are generally incomplete. We complement the event logs with resource usage data, where we seek to determine if the identification of anomalous jobs can offer an improvement on the error detection confidence. Together with a high $I(s_i)$ and $H(s_i)$, an anomalous sequence $a_i$ of resource usage is *conjectured* to be a high indication of failure. Otherwise, we consider such a sequence to be normal. The detection algorithm will then depend on our definition of anomaly, as captures by the thresholds. We define thresholds for *Mutual Information* ($\mu$), *Entropy* ($\varphi$) and sequence *anomaly*,($\gamma$). The algorithm used for error detection is shown in Algorithm 3.

**Algorithm 3** Sequence Detection

```
 1: procedure DETECT($S, \gamma, \mu, \varphi$)
 2:     for each sequence $s_i$ in $S$ do
 3:         $I(s_i)$ =Mutual Information($s_i$)    ▷ uses the rows of the matrix of $s_i$.
 4:         $H(s_i) = Entropy(s_i)$ ▷ uses the columns of the matrix of $s_i$.
 5:         $a_i = AnomalyScore(S, \gamma, s_i)$ ▷ from resource Usage data of sequence
 6:         $f_i = I(s_i) - H(s_i)$
 7:         if ($I(s_i) >= \mu$ && $H(s_i) >= \varphi$ && $a_i > 0$) || ($a_i > 0$ && $H(s_i) >= \varphi$ && $f_i <= 0$) then
 8:             Return $True$                    ▷ faulty Sequence
 9:         else
10:             Return $False$          ▷ non-faulty or normal sequence
11:         end if
12:     end for
13: end procedure
```

The intuition behind the Algorithm 3 as seen from lines 6-10 is that if a number of nodes within a given sequence produce similar events with some degree of randomness and the the running jobs are anomalous, then such patterns a likely to end in failure. In summary, Algorithm 3 detects errors in a sequence that is likely to lead to a system failure by making use of $I$, $H$ and sequence anomaly score, $a_i$. Line 6 ensures that patterns with higher entropy than events mutual information are captured as failure inducing.

## V. Experiments and Results

The aim of this paper to develop a methodology to detect errors in an event sequence that is likely to lead to a system failure. We achieve this through the novel use of event logs and resource usage data. We evaluate our approach through experiments conducted on Rationalized logs (ratlogs) and resource usage data from the Ranger Supercomputer from the *Texas Advanced Computing Center (TACC)* at the University of Texas at Austin[2].

The resource usage data were collected using TACC_Stats [12] that takes snapshots at the beginning of job execution, in ten-minute interval and at the end of the job execution. Jobs generate their resource usage data, which are archived on the file system. The events are logged by each node through a centralized message logging system. The logs are combined and interleaved in time. We evaluate our approach on *four weeks* of resource usage data (32GB) and rationalized logs (1.2GB). These data were collected for the month of March 2012. It is worth noting that, within this time, the system experienced a high failure rate making the data sufficient for the analysis. The failures we considered in this research are the compute node soft lock ups, which are identified by experts and system administrators from TACC.

[2]*www.tacc.utexas.edu*

We extracted the 96 elements or counters from the resource usage log, as seen in Table I. The size of the time window ($t_w$) chosen was based on two factors: (i) lead time to failure - This is determined by conducting a *root cause analysis* of failures and research on Ranger Rationalized logs (ratlogs) has shown that the *minimum* lead time to failure of most occurring faults is about 120 $minutes$ [9], and (ii) The concept of nodehour [5] was shown to be fine enough to capture erroneous messages. To this end, we set $t_w = 60$ minutes. This time window also allows us to compare our work with Nodeinfo, a popular error detection approach. We conducted the experiment on 720 sequences of events logs, with also 720 corresponding resource usage data sequences, of which 182 are faulty sequences.

TABLE I
List of 96 Elements of Resource Usage Data

| *Type* | *Element* | *Qty* |
|---|---|---|
| Lustre /work | read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs,alloc node, setxattr, getxattr, listxattr, removexattr, inode_permission | 23 |
| Lustre /share | read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs, alloc_node, setxattr, getxattr, listxattr, removexattr, inode_permission | 23 |
| Lustre /scratch | read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs, alloc node, setxattr, getxattr, listxattr, removexattr, inode_permission | 23 |
| Lustre /network | tx_msgs, rx_msgs, rx_msgs_dropped, tx_bytes, rx_bytes, rx_bytes_dropped | 6 |
| Virtual memory | pgpgin, pgpgout, pswpin, pswpout, pgalloc_normal, pgfree, pgactivate, pgdeactivate, pgfault, pgmajfault_pgrefill_normal, pgsteal_normal, pgscan_kswapd_normal, pgscan_direct_normal, pginodesteal, slabs_scanned,kswapd_steal, kswapd_inodesteal, pageoutrun, allocstall_pgrotated | 21 |

### A. Evaluation Metrics

In measuring the performance of our detection algorithm, we employ the widely used *sensitivity, specificity* and *F-measure* metrics. *Sensitivity* (also called *true positive rate* or *recall*) measures the actual proportion of correctly detected failure sequences to the total number of failure sequences, as expressed in Equation 6. On the other hand, *specificity* (or true negative rate), measures the proportion of non-failure sequences which are detected as non-failure among all non-faulty sequences, as seen in Equation 7. *F-measure* here is synonymous with the usual *F-measure* in information retrieval, however, in this case, it is the harmonic mean of sensitivity and specificity (see Equation 8). Since *sensitivity* or *specificity* cannot be discussed in isolation, *F-measure* combines the two providing us with a balanced detection accuracy.

$$Sensitivity = \frac{TP}{TP + FN} \quad (6)$$

$$Specificity = \frac{TN}{FP + TN} \quad (7)$$

$$F - measure = 2 * \frac{Sensitivity * Specificity}{Sensitivity + Specificity} \quad (8)$$

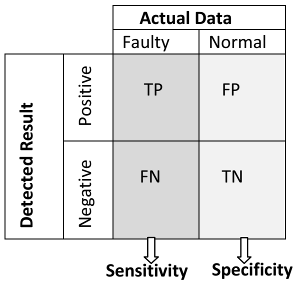


Fig. 5. Evaluation metrics

The parameters $TP$, $FP$, $TN$, and $FN$ denotes *true positives*, *false positives*, *true negatives* and *false negatives* respectively. Figure 5 demonstrates the relationship of these parameters. A perfect detection will have sensitivity and specificity value of 1, meaning the algorithm can detect all failure-inducing failures while avoiding false positives.

## B. Detection Performance

We aim to show the detection accuracy of our methodology and then compare our approach with Nodeinfo, a popular error detection approach proposed by Oliner *et al.* [5].

In the experiments, we evaluated our approach under various conditions. Since our approach is based on concepts such as anomaly score and entropy of a sequence, we show the effectiveness of our detection methodology under different values (see Figure 6). The aim is to find value combinations where detection accuracy is better achieved, i.e., a high true positive and true negative rate. We observed that a change in mutual information threshold ($\mu$) does not have as much influence on the detection result as do the entropy threshold ($\varphi$) and the anomaly threshold ($\gamma$) values. Figure 6 shows the result of detection with different values of $\gamma$ and $\varphi$. The best detection result is achieved for values of $\varphi = 0.4$ and $\gamma = 0.6$, achieving sensitivity (true positive rate) of about 80% and 78% respectively. This result also demonstrates how the best value of the features which affects detection the most are obtained. Note that the value of sensitivity increases with increase in $\varphi$ (see Figure 6(a)); however, specificity increases with corresponding increase in values of $\varphi$. Our approach is able to detect *80%* of errors that lead to failures. Further, with the high value of specificity, we conclude that *the false negative rate is also low*.

It is worth noting here that the detection threshold is not dependent on the system on which the approach is applied to. It basically is dependent on how anomalous the logs and the resource usage data of such are system are. Since it is difficult to discuss both sensitivity and specificity separately, we use *F-measure* as the more appropriate detection performance measure in that case.

Performing detection using event logs only achieved best sensitivity of 75%. This approach is detailed in [8]. However, combining both event logs and usage data, a sensitivity and specificity of above 83% is achieved as seen in Figure 6.

## C. Comparison CRUDE with Nodeinfo

NodeInfo is motivated by the assumption that similar computers executing similar jobs should produce similar log contents. In this respect, as long as log lines are tokenizable, the Nodeinfo technique can be applied. It equally leverages the "log entry" weighing scheme for calculating the entropy of a "nodehour" (nodes within an hour), which is analogous to documents in information theory. It first computes the amount of information each token conveys with regards to the node that reported it. Nodeinfo uses Shannon information entropy to calculate the information by each node. They further obtained and rank "nodehours" according to how high information terms (nodeinfo) they contain. Nodehours, or what we called sequence of 60 minute in size, with high nodeinfo, is regarded as faulty or containing error alerts.

We implemented this approach (Nodeinfo [5]) and evaluated its error detection performance on the event log data and we subsequently compare the performance of Nodeinfo with our approach (CRUDE). Figure 7 shows the detection *F-measure* of both methods. While our method performs consistently better with an increase in $\gamma$, Nodeinfo consistently decreases with increasing nodeinfo threshold. This means that, as we set the informativeness of a sequence to be high, Nodeinfo detected fewer faulty sequences. We make the following observations:

- The best improvement in error detection offered by our approach, compared to Nodeinfo, is achieved at an anomaly threshold of $0.2$, where the improvement is approximately $250\%$.
- Our method achieved on average (across all the thresholds used, using F-measure) an improvement of nearly $85\%$ over *Nodeinfo*.
- On the best anomaly threshold (0.6), our approach achieved an improvement of about $50\%$ over Nodeinfo, that is, our approach *accurately detected* an additional $50\%$ of faulty sequences over Nodeinfo.

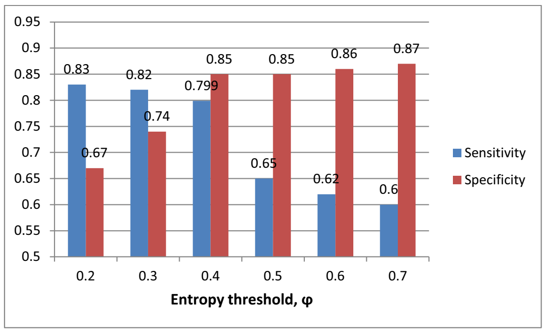
This shows that the use of resource usage as a complement to event logs, as proposed by our approach, can be effective in increasing the accuracy of error detection.

## D. Runtime Performance Analysis

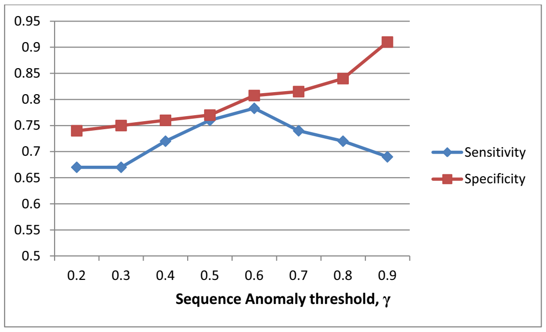
We implemented our methodology in Java and run on a system with CPU (Intel i5 @3.10GHz) to evaluate the runtime of the approach. The performance (runtime) of our methodology is not affected by detection thresholds, but only by the size of the data. This is mainly during the data transformation and obtaining the anomaly score. The PCA approach to anomaly detection slightly increased the runtime and this is applicable only if the data is large.

Figure 8 demonstrates that size of data[3] has impact on the runtime performance. As the size of the logs increases, the

[3]We add both resource usage data and error logs



(a) Detection with varying entropy threshold, with $\gamma = 0.6$



(b) Detection with varying anomaly threshold, with $\varphi = 0.4$

Fig. 6. Results showing accuracy of CRUDE under varying values of $\gamma$ and $\varphi$.

runtime also increase, however, the increase is not exponential. The increase is gradual, and for a data size of 300MB, the runtime is just about 4 minutes. This is not a challenge as logs may not be this large within a time window; even if it were so, the time taken to process it for detection is reasonably small. The detection process (Algorithm 3) is very fast with just about 2-3 seconds and it is not dependent on the data size at this point. The graph of Figure 8 shows the runtime of all the steps involved.

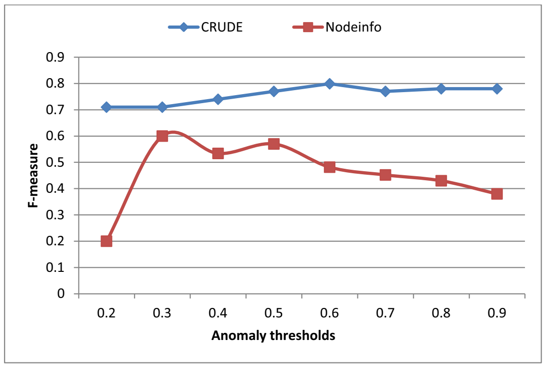


Fig. 7. Graph showing detection performance (F-measure) of *CRUDE* and *nodeinfo*

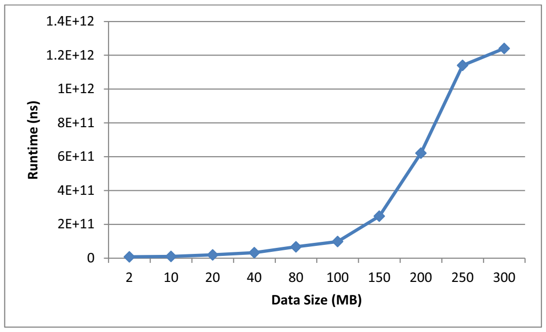


Fig. 8. Graph showing runtime performance of CRUDE

## VI. Related Work

Most approaches to detecting failure or faults concentrate on utilizing the log events of cluster systems to do that. These approaches predominantly use data mining and/or machine learning methods [1], [3], [18]–[20]. Others combine both these techniques with corrective measures like checkpointing. With the nature of the logs, massive, different and less structured depending on the cluster system, it is reasonable that data mining and machine learning approaches are mostly used by research for detecting failures in large-scale system.

Another approach explored void search for fault detection [4]. This approach demonstrated that a good detection measure using its sensitivity and specificity. This approach however, make use of different data,called environmental data of cluster systems. Ours is different as we make use of event logs combined with resource utilization/usage data. Chuah *et al.* [9] used event logs combine with usage logs to diagnose root-cause of some intermittent faults. Even though their method uses the same data with ours, we are solving a different problem in this paper, in that the authors of [9] perform a root-cause analysis of system failures while this paper focuses on the detection of faulty sequences in an unsupervised way. Similarly, other works such as [21] performed root-cause and correlation analysis to determine the impact of the various factors such as temperature on system failures.

In their work, Gainaru *et. al.* [18] [3] proposed a method for analysing systems' generated messages by extracting sequences of events that frequently occur together. This provides an understanding of the pattern of failures and non-failure events. They also use signal analysis technique to characterise the normal behaviour of system and how faults affects it; in this case, they will be able to detect deviations from the normal. We proceed differently, we capture systems behaviour and deviation through Entropy and mutual information of sequences. Utilizing the informativeness of these sequences, we detect deviations from normal and using resource usage data to determine if these sequences are anomalous based on jobs' resource usage. [22] and [1] used console logs combine with source code to create useful features that can be used by analysis algorithms for detecting faulty events. They used PCA to detect fault events in the logs. Source codes provide the extra effectiveness for detection as demonstrated by the authors, however, it is difficult to obtain program source codes as developers wouldn't make them public. For this reason, we do not utilized source codes, but we use the *event logs* and *resource usage data*; instead of detecting individual fault events, we detect faulty sequence of events preceding failures based on change in the entropy of the sequences and anomaly of the jobs within a sequence. Another approach that utilises both PCA and ICA to detect anomalous nodes is [17].

Oliner *et al.* [5], [23] and [6], [7], [14] employed an unsupervised approach to solving this problem. They leverage the notion of information content of the events produced by each node within an hour to identify possible presence of fault. They called it Nodeinfo. This approach calculates the informativeness of a node-hour by using the log.entropy weighing scheme for node - term weights. Nodeinfo produced good results as it has been produced as a tool and is currently in used. Other works on detection of failure patterns [8] relied only on the entropy and behaviour of nodes and similar sequences to characterise a failure sequence.

However, this approach is very dependent on the message logs, that is, a failure or faults not characterised by by many logs will not be detected. To solve this problem, we combine both the event logs and resource usage logs where faults characterised by less logs can still be detected through the anomalous running jobs.

## VII. Conclusion and Future Work

We proposed an approach for error detection in large-scale distributed systems. Our approach makes use of the novel combination of event logs and resource usage data to improve detection accuracy. Our methodology is based on the computation of (i) mutual information (ii) entropy and (iii) anomaly score to determine whether an event sequence is likely to lead to failure, i.e., is erroneous. We evaluated our methodology on the logs and resource usage data from the Ranger supercomputer and has shown to detect errors with a very high accuracy. We compared our approach with Nodeinfo, a well-known error detection methodology, and been shown to remarkably outperform Nodeinfo.

As a future work, we intend to adopt and implement it as an online detection approach. Secondly, Independent Component Analysis (ICA) can be used and compared with PCA. We hope to further investigate the performance this approach on different system data and different time windows.

## Acknowledgements

We thank the Texas Advanced Computing Center (TACC) for providing the Ranger logs. We also thank PTDF Nigeria and the University of Jos-Nigeria for partly funding this research.

## References

[1] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 117–132.

[2] X. Rao, H. Wang, D. Shi, Z. Chen, H. Cai, Q. Zhou, and T. Sun, "Identifying faults in large-scale distributed systems by filtering noisy error logs," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, June 2011, pp. 140–145.

[3] A. Gainaru, F. Cappello, and W. Kramer, "Taming of the shrew: Modeling the normal and faulty behaviour of large-scale hpc systems," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 1168–1179.

[4] E. Berrocal, L. Yu, S. Wallace, M. Papka, and Z. Lan, "Exploring void search for fault detection on extreme scale systems," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Sept 2014, pp. 1–9.

[5] A. J. Oliner, A. Aiken, and J. Stearley, "Alert detection in system logs," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on.* IEEE, 2008, pp. 959–964.

[6] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Fast entropy based alert detection in supercomputer logs," in *PFARM '10: Proceedings of the 2nd DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM).* IEEE, 2010.

[7] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "An evaluation of entropy based approaches to alert detection in high performance cluster logs," in *Proceedings of the 7th International Conference on Quantitative Evaluation of SysTems(QEST).* IEEE, 2010.

[8] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Brwone, "Towards detecting patterns in failure logs of large-scale distributed systems," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International.* IEEE, 2015.

[9] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, "Linking resource usage anomalies with system failures from cluster log data," in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, 2013, pp. 111–120.

[10] "1003.1 standard for information technology portable operating system interface (posix) rationale (informative)," *IEEE Std 1003.1-2001. Rationale (Informative)*, pp. i–310, 2001.

[11] J. L. Hammond, T. Minyard, and J. Browne, "End-to-end framework for fault management for open source clusters: Ranger," in *Proceedings of the 2010 TeraGrid Conference*, ser. TG '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:6.

[12] J. Hammond, "Tacc_stats: I/o performance monitoring for the intransigent," in *In Invited Keynote for the 3rd IASDS Workshop*, 2011.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[14] A. Makanju, A. Zincir-Heywood, and E. Milios, "System state discovery via information content clustering of system logs," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 301–306.

[15] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," *SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 217–228, Aug. 2005.

[16] Y.-J. Lee, Y.-R. Yeh, and Y.-C. F. Wang, "Anomaly detection via online oversampling principal component analysis," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 25, no. 7, pp. 1460–1470, July 2013.

[17] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 2, pp. 174 –187, feb. 2010.

[18] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 77:1–77:11.

[19] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines," in *Proceedings of the First USENIX Conference on Analysis of System Logs*, ser. WASL'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 5–5.

[20] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu, "Logmaster: Mining event correlations in logs of large-scale cluster systems," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, Oct 2012, pp. 71–80.

[21] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Mining console logs for large-scale system problem detection," in *Workshop on Tackling Computer Problems with Machine Learning Techniques (SysML), San Diego, CA*, 2008.

[23] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *International Conference on Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP*, june 2007, pp. 575 –584.