On Handling Redundancy for Failure Log Analysis of Cluster Systems

Nentawe Gurumdimma*
Arshad Jhumka† and
Maria Liakata‡

Department of Computer Science
University of Warwick
Coventry, CV4 7AL UK

Email: *N.Y.Gurumdimma@warwick.ac.uk
†arshad@dcs.warwick.ac.uk
‡M.Liakata@warwick.ac.uk

Edward Chuah§ and
James Browne¶

Department of Computer Science
University of Texas - Austin

Email: §chuah@acm.com
¶J.Browne@utexas.com

***Abstract*—System event logs contain information that capture the sequence of events occurring in the system. They are often the primary source of information from large-scale distributed systems, such as cluster systems, which enable system administrators to determine the causes and detect system failures. Due to the complex interactions between the system hardware and software components, the system event logs are typically huge in size, comprising streams of interleaved log messages. However, only a small fraction of those log messages are relevant for analysis. We thus develop a *novel, generic* log compression or filtering (i.e., redundancy removal) technique to address this problem. We apply the technique over three different log files obtained from two different production systems and validate the technique through the application of an unsupervised failure detection approach. Our results are positive: (i) our technique achieves good compression, (ii) log analysis yields better results for our filtering method than normal approach.**

***Keywords-Cluster Log Data; Unsupervised learning; Compression; Levenshtein distance; filtering*.**

I. INTRODUCTION

The size and complexity of computer systems required for computationally-heavy jobs such as scientific computations is increasing and failures are expected to be the norm rather than exceptions. The unscheduled downtime of such large production computer systems carries huge costs: (i) applications running on them have to be executed again, potentially requiring hours of re-execution, (ii) checkpointing has to be performed regularly and (iii) lots of effort is required to find and fix the causes of the downtime. These systems generate a large amount of data, typically in the form of system logs, and these data files represent the main avenue by which system administrators can gain insight into the behaviour of the systems.

Due to the size of such data files and the complexity of such systems, system administrators usually adopt a divide and conquer approach to analyse the data. Such log files are typically incomplete and redundant; that is, the files may not contain all the relevant events to characterize a failure while containing several interleaved events related to the same failure.

There are several possible ways to increase the dependability of these computer systems [1], with failure prediction [2] or failure diagnosis [3][4] being the most prominent ones. One of the basic tasks of *automatic analysis* of log files for the purposes mentioned above is *preprocessing*, which typically involves filtering the logs. However, such analysis techniques are invariably expensive due to the size and type of the logs being processed. Specifically, these log files are highly redundant and unstructured. To handle the lack of structure in log files, further information is added, often manually, to capture specific aspects of the data, i.e., the data is labelled using system information. To address the redundancy problem, the logs are *filtered*, or *preprocessed*, to aid the log analysis process. Specifically, to handle redundancy, *Filtering/compression techniques*, or redundancy handling techniques are used to remove events that are not useful for any analysis. A common problem with such compression techniques is that they may remove important information that are pertinent to the analysis phase, as captured by the targeted high compression or filtering rate [5]. The terms *compression* and *filtering* are used interchangeably.

This paper seeks to bridge this gap; we filter event logs based on their similarities. We propose a *novel and generic* clustering approach to log filtering where events that are not similar but causality related are also kept. This is to preserve events patterns that serve as precursor to failures. We focus on trying to achieve good failure analysis, hence we evaluate the impact of the filtering on failure pattern detection approach.

The rest of the paper is organised as follows: In Section II, we present the system logs and models we assumed in the paper. We present the methodology for achieving our objective in Section III, while failure pattern detection for performance evaluation is presented in Section IV. In Section V, we discuss the results when applying the approach

 ISBN: 978-1-61208-429-9

to log data from different supercomputer systems. Related works are presented in Section VI. We conclude the paper and provide direction for future work in Section VII.

II. Models and System Logs

A. Basic Definitions

We will refer to Figure 1 (sample logs from Ranger supercomputer) in this section for definition of terms.

- **Event:** A single line of text containing various fields (*time-stamp, nodeID, protocol, application, error message*) that reports the activity of a particular cluster system. Such an event is also often called a *log message*.
- **Event logs:** A sequence of events containing the activities that occur within a cluster system.
- **Similar Events:** These are events containing similar log messages based on the similarity measure used. From Figure 1, events 5 and 6 can be considered similar.
- **Identical Events:** These are events believed to be exactly the same and/or are produced by the same 'print' statement, e.g., events 7 and 8 in Figure 1.
- **Failure Event:** This is an event that is often associated with and/or is indicative of a system failure.
- **Sequence** A sequence consists of one or more consecutive events logged within a given time period. In this paper, *sequence* and *patterns* means the same and are used interchangeably.

Table I. summary of logs used from production systems

System	Log Size	Messages	Start Date	End Date
Syslogs	1.2 GB	$> 10^7$	2010-03–30	2010-08-30
Ratlogs	4.3 GB	$> 2 \times 10^7$	2011-08-01	2012-01-20
Blue Gene/L	730 MB	$4,747,963$	2005-06-03	2006-01-04

B. System Model and Cluster Logs

Here we explain the model of our system and explain the logs we work with 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 with certain production time, and all the components involved write logs to a writing container. This is a common model for most of the cluster vendors like Ranger, Cray, IBM etc. In this research, we use the log of two popular cluster systems, namely (i) Rationalised logs (ratlogs) from Ranger Supercomputer, (ii) syslog from Ranger supercomputer and (iii) IBM BlueGene/L. Table I shows a summary of the logs from the cluster systems we focused on in this research.

2) Cluster Event Logs: Different attributes are used by supercomputer vendors to represent its components. The IBM standard for Reliability, Availability, Serviceability (RAS) logs incorporates more attributes for specifying event types, severity of the events, job-id and the location of the event[6].

An example of Ranger's (syslog) event can be seen below: *Apr 4 15:58:38 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 five attribute fields namely: **Time-stamp** (*Apr 4 15:58:38*) containing the month, date, hour, minute and second at which the error event was logged. **Node Identifier** or **Node Id** (*mds5*) identifies the nodes from which the event is logged. **Protocol Identifier** (*kernel*) and **Application** (*LustreError*) provides information about the sources of logs. **Message** (*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. The English-only words (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*, signify the interacting components within the cluster system. The ratlogs have and additional field *(job-id)* which differentiates it from syslogs. Detailed example of the Ranger logs is seen in Figure 1 and IBM's Blue Gene/L (BGL) is seen in Table II.

C. Failure Model

The failures we focused on in this paper are those that cause the system to malfunction, i.e., execution of some jobs has stopped. 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*.

These failures usually occur as a result of faults occurring in the system, i.e., caused by the fault(s) of one or more subsystems/components in the system [7]. These faults result in a log entry or fault message in the log data file. For example, a network timeout will result in a "Network timeout" log message being recorded. Hence, in a typical sequence, there will be an interleaving of fault events and normal events.

III. Methodology

We first briefly explain the existing filtering approach, which we call *normal filtering*. We next explain our filtering approach which is based on simple iterative clustering. The clustering is based on the notion of Levenshtein's Distance (LD), defined on the events messages to capture their similarities.

Filtering based on defined heuristics is applied to purge out redundant events. The resulting event sequences are then transformed into term frequency matrices which serve as input to detection algorithm.

A. Normal Event Filtering

Filtering or compression as it may be called here, is meant to reduce the complexities that comes with analysing logs. It is generally agreed that filtering or pre-processing logs is an important process. The process helps eliminate redundant events from logs, thereby reducing the initial huge size. This

 ISBN: 978-1-61208-429-9

TABLE II. AN EXAMPLE OF EVENT FROM BLUE GENE/L RAS LOG

Rec ID	Event Type	Facility	Severity	Event Time	Location	Entry Data
17838	RAS	KERNEL	INFO	2005-06-03-15 .42.50.363779	R02-M1-N0-C:J12-U11	instruction cache parity error corrected

```
1:  Mar 29 10:00:44   i128-401  kernel:  [8965057.845375] LustreError: 11-0: an error occurred while communicating with *.*.*.36@o2ib. The
    ost_write operation failed with -122
2:  Mar 29 10:00:53  i128-401  kernel:   [8965077.319555] LustreError: 11-0: an error occurred while communicating with *.*.*.28@o2ib. The
    ost_write operation failed with -122
3:  Mar 29 11:27:16  i182-211  kernel:   [8981960.031578] a.out[867]: segfault at 0000000000000000 rip 0000003351c5b2a6 rsp 00007fffdcd318c0
    error 4
4:  Mar 29 11:27:16  i115-209  kernel:   [2073150.255467] a.out[22921]: segfault at 0000000000000000 rip 0000003ad725b2a6 rsp 00007fffbf1a6d40
    error 4
5:  Mar 30 10:02:24  i107-308  kernel:   [8966098.630066] BUG: Spurious soft lockup detected on CPU#8, pid:4242, uid:0, comm:ldlm_bl_22
6:  Mar 30 10:02:24  i107-308  kernel:   [8966098.642055] BUG: soft lockup detected on CPU#10, pid:21851, uid:0, comm:ldlm_bl_13
7:  Mar 30 10:09:25  i107-111  kernel:   [8966563.203631] Machine check events logged
8:  Mar 30 10:09:51  i124-402  kernel:   [8965663.148499] Machine check events logged
9:  Mar 30 10:10:22  master    kernel:   LustreError: 28400:0:(quota_ctl.c:288:client_quota_ctl()) ptlrpc_queue_wait failed, rc: -3
10: Apr  1 05:23:54  i181-409  kernel:  [9203054.301173] Machine check events logged
11: Apr  1 05:23:58  visbig      kernel:  EDAC k8 MC0: general bus error: participating processor(local node response), time-out(no timeout)
    memory transaction type(generic read), mem or  i/o(mem access), cache level generic)
```

Figure 1. Sample Log events for RANGER Supercomputer

however, must avoid removing useful events or event patterns that are important for failure pattern detection. In normal log filtering, events that repeats within certain time window are removed, only the first is kept. This simple log filtering is what we refer to as *normal filtering* in this work. Details can be seen in [6].

B. Preprocessing

Tokenization and Parsing

This phase involves parsing the logs to obtain the event types and event attributes, using simple rules. Tokens that carry no useful information for analysis are removed. For example, numeric-only tokens are removed but *attributes* (alpha-numeric tokens) and the *message types* (English-like only terms) are kept. Also, fields like protocol identifier and application are removed or omitted during the parsing and tokenizing phase.

Message part contains English words, numeric and alphanumeric tokens. The English tokens show a pattern providing information pertaining to the state of the system. The alpha-numeric tokens capture the interacting components or software functions involved. These interacting components, which do not occur frequently and show less or no pattern, are also important since we are interested in interacting nodes of the cluster system. The numeric only tokens are removed as they only add noise.

C. Filtering: Redundancy Handling

1) Logs Message types Extraction and Labelling through Clustering: Generally, data clustering techniques group similar data points together, based on some closeness measure. The output of such clustering algorithms is a set of clusters, where each of the clusters contain members (data points) that are similar (or close) to each other and very dissimilar to members of other clusters. In order to identify all the unique events in the logs, we first extract the message types and we introduce a clustering technique (see Algorithm 1) that partitions the logs based on events similarities given by an *edit* distance. Each cluster represents a unique event.

Edit Distance - (Levenshtein's Distance): The closeness of events is measured using Levenshtein's Distance (LD) [8]. It is a metric that measures differences between two strings. It is defined based on edit operations (insertion, deletion or substitutions) of the characters of the strings. Hence the Levenshtein's distance between two strings s_1 and s_2 is the number of operations required to transform s_2 into s_1 or vice versa. LD is an effective and widely used string comparison approach. We found it more useful as we easily can define it on tokens rather than characters. We equally found it to be more suitable here than cosine similarity as the later is a vector-based similarity measure.

Events Similarity: In our algorithm, we define LD as the number of operations required to transform one message type into another. Therefore, instead of defining the operations on characters of event message types, we define the operations on the tokens or terms t_i of the event types, $e_i = \{t_1, t_2, ..., t_n\}$. It should be noted that message types of event logs mostly do not have many terms or tokens, therefore the computational overhead is reduced.

Consider the log entries of Figure 1. Events 1 and 2 are both failed communication events by the same node; the communication is, however, with different nodes. Events 7 and 8 are both normal machine checked exceptions. The challenge is that these events greatly increase the feature space of distinct events, making it difficult to handle for any meaningful analysis. In solving this, we consider the following: (1) *Similar events need to be grouped together and considered the same* and (2) *identical events are also considered same and the redundant ones are removed*. We propose an algorithm (Algorithm 1 - see Figure 3) to first find the similarity between these events and then cluster those events that are similar. Then, events in the same cluster are indexed with the same identity (IDs).

From the sample logs of Figure 1, it is necessary that

 ISBN: 978-1-61208-429-9

	Event ID	Time-stamp	Node Identifier	Message
1	LEO	1269856844	i128-401	LustreError: error occurred while communicating with 129.114.97.36@o2ib. The ost_write operation failed with
2	LEO	1269856853	i128-401	LustreError: error occurred while communicating with 129.114.97.36@o2ib. The ost_write operation failed with
3	SEGF	1269862036	i182-211	segfault at rip rsp error
4	SEGF	1269862036	i115-209	segfault at rip rsp error
5	SSL	1269943344	i107-308	BUG: Spurious soft lockup detected on CPU, pid:4242, uid:0, comm:ldlm_bl_22
6	SSL	1269943344	i107-308	BUG: soft lockup detected on CPU, pid:21851, uid:0, comm:ldlm_bl_13
7	MCE	1269943765	i107-111	Machine check events logged
8	MCE	1269943791	i124-402	Machine check events logged
9	CQF	1269943822	master	client quota ctl ptlrpc queue wait failed,
10	MCE	1270099434	i181-409	Machine check events logged
11	GBE	1270099438	visbig	general bus error: participating processor local node response, time-out no timeout memory transaction type generic read, mem or io mem access cache level generic

Figure 2. Sample pre-processed logs

any similarity metric used must consider the order of the terms in the events for meaningful result. For example, the event messages *...error occurred while communicating with...* and *...Communication error occurred on...* may appear similar but semantically different. A similarity metric that does not take order of tokens/terms into consideration will cluster these events together, i.e., these events will be seen as similar, because they have similar terms. To address this challenge, we define an *edit distance* metric on terms without transposition, taking term order into consideration. Also, defining this metric based on terms or tokens reduces the computational cost incurred as opposed to when it is defined on string characters.

Finally, to capture the similarity of events, we define a similarity threshold, where the lesser the number of edits, the higher the similarity. Hence, we define the threshold such that, when the edit distance between a pair of messages is less than or equal to the threshold λ (hence highly similar), these events are regarded as similar and thus clustered together.

Event Similarity Threshold

It has also been observed that events that can be regarded similar do not have much difference in terms of the number of terms contained in the event messages. Using an *iterative approach* [9], we start with a small value of similarity threshold λ, then increase the value in small increments and monitor the output, until a satisfied similarity value is obtained. We observed that with a very small similarity threshold, only events that are exactly similar are clustered together. But, as the value of threshold is increased to values higher than 3, events that are often dissimilar were being classed as similar. Therefore, to have a more acceptable result, we chose a threshold of 2.

Clustering event logs and ID assignment The challenges addressed by this algorithm and its approach can be explained in two steps:

STEP I: The events are grouped based on the value of the edit distance or LD. In this step all events with equal terms or token length are clustered together. This is because they will have same value of LD.

Algorithm 1
Input: *Log events* $e_0 \ldots e_n$, *MinimumSimilarityThreshold* λ
Output: Log events with cluster IDs

initialise $s_0 = e_0$;

for all log events $e_i, i = 0 \ldots n$ **do**{ }
  Obtain events similarities using Levenshtein Distance *similarity*(s_0, e_i)=LD(s_0, e_i);
end for
for all log events **do**
  if *similarity* $<= \lambda$ **then**
    Assign log events to cluster
    Assign ID to log event representing its cluster
  end if
end for
Repeat step 2 for clusters with problem explained in **STEP II** above Until all log events are clustered
Return() {outputs log events with their cluster ID}

Figure 3. An algorithm that Clusters event logs base on similarity and assign event IDs (represent the clusters).

STEP II: Since LD gives the number of operations performed to transform one event to the other, different event types with same token length are clustered together from the step 1. For example, ... *"machine check event logged"* and ... *"Spurious soft lockup detected"* will belong to the same cluster. This step partitions clusters with such problems with smaller LD value. This step is performed recursively until the clusters contains only events of similar message type. More on this is shown is Algorithm 1 seen in Figure 3.

As example of the output of this step is shown in Figure 2. These logs still contain redundant events, for example, events 5 and 6 (please observe that events 5 and 6 are clustered together, though being slightly different, and are indexed using the same id).

2) Removing Redundant Events: There are several seemingly identical error events reported frequently in cluster logs. These events are then clustered together and have the same ID from the clustering step. The events are sometimes reported by the same cluster node and they occur within a small time difference (temporal aspect). Also, sometimes

some of these events are reported by different cluster nodes (spatial aspect) but still within the small time difference.

According to Iyer and Rosetti [10], occurrence of similar or identical events within a small time window might likely be caused by the same fault. Thus, these messages are related (and hence redundant) as they potentially point to the same root-cause. Therefore, removing these redundant messages may prove to be beneficial to the analysis stage. In another sense, removing the "redundant" events could be useful in understanding the behaviour of a particular fault in terms of the frequency of the event generated within the period. Therefore, in filtering of redundant log events we consider events in a sequence having the following properties:

- Similar events that are reported in sequence by the same node within a small time window are redundant. This is because nodes can log several similar messages that are triggered by the same fault.
- Similar events that are reported by different nodes in a sequence and within a time window. This could be triggered by the same fault resulting in similar misbehaviour by those affected cluster nodes.
- Identical events occurring in sequence and within a small time difference are redundant.

General approach to filtering will keep the first similar event of sequence and subsequent ones removed [5]. It is pertinent to note that it is possible that the same error messages logged by different nodes are caused by different faults and at close time interval. Some events are causally-related. In our approach, we keep such events. The process of identifying and grouping the error events exhibiting the above properties is done using a combination of both tupling and time grouping heuristics [9]. We define some heuristics that captures the properties outlined above.

With careful observation of the logs and experts' input, we realised that achieving high compression rate and yet preserving patterns are important and dependent on how informative and well-labelled a given log is. For example, Ranger's *Ratlogs* contains more information regarding the nodes and jobs involves which provides more information regarding an event. *Job-ids* in logs indicates particular job that detects the reported event. The job-ids when correlated with failure events, tells which job is the source of the failure. This implies that identical job-ids present on different events within a given event sequences would have high correlation as regards the faults and failure that is eventually experienced [11]. In order to achieve high *events compression accuracy* (ability to keep unique events) and *completeness* (remove redundant events), yet maintaining events which are possible precursor to failure (preserving failure pattern), we propose a filtering approach that removes redundant events or events that are related based on the causes, sources, similarity and time of their occurrence.

Specifically, given two events e_1 and e_2, with the time of occurrence Te_1 and Te_2 respectively, they are both causality related or emanates as result of same faults if:

- $nodeid(e_1) == nodeid(e_2)$ && $|Te_1 - Te_2| \leq t_w$ && $sim(e_1, e_2) \leq \lambda$
- $nodeid(e_1) == nodeid(e_2)$ && $jobid(e_1) == jobid(e_2)$ && $|Te_1 - Te_2| \leq t_w$ && $sim(e_1, e_2) \leq \lambda$,

where $sim(.)$ is the similarity given by LD, λ is similarity threshold.

IV. Case Study: Pattern Detection

A. Introduction

The aim of compression or filtering event logs of large-scale computer systems is to reduce the massive size by properly removing redundant events; and preserving the necessary events patterns to enhance any log analysis. Such analysis can be failure prediction, root cause analysis, failure detection etc. In this section, we introduce an unsupervised pattern detection approach in logs of distributed systems. This is an approach used to evaluate the accuracy and efficiency of our filtering approach. That is, if the approach preserve useful event patterns in logs that improves failure detection.

Filtered logs sequences are now extracted transformed into term-frequency matrix. This matrix comprises of row vectors representing distribution or counts of each event types within a given time and the column vectors representing the sequences or patterns.

We further utilise clustering approach [12] to group similar behaving patterns. Each of these patterns are then expected to be normal event sequences or comprising faulty events.

B. Failure Pattern Detection

According to Gainaru et al. [13], event log sequences can be categorised as noisy, periodic or silent in their behaviour. Noisy sequences occur with high frequency (busty or chatty) and the level of interaction of the nodes involved increases within short period. The characteristics of these patterns are captured through entropy [14] and mutual information. High entropy signifies that the cluster is likely *failure cluster*.

Hence, given a cluster with set of sequences or patterns $C = \{c_1, ..., c_m\}$ and each pattern c_i contains a set of similar events sequences,s, i.e., $c_i = \{s_1, ..., s_n\}$. Then detection is achieved as follows:

$$f(c) = \begin{cases} 1 & \text{if } \varphi(c) < 0 \\ else & \begin{cases} 1 & \text{if } \varphi(c) > \tau \ \& \ H(c) > 0 \\ 0 & \text{otherwise} \end{cases} \end{cases} \quad (1)$$

Where,

$$\varphi(c) = MI(c) - H(c) \quad (2)$$

and $MI(c)$ and $H(c)$ are the mutual information and the entropy of patterns c, τ is detection threshold, the value of $\varphi(c)$ for which we can decide if c contained failure sequences or not.

V. Experiments, Results and Discussion

A. Experimental Setting

We performed our experiments in order to evaluate the effectiveness of our filtering method is preserving useful events patterns that may potentially improve failure detection. Further, we aim to assess the efficiency of our unsupervised detection method on the various logs. The experiment was conducted on three different logs obtained from two cluster systems. The ratlogs and syslogs are obtained from the Ranger supercomputer sited at Texas Advanced Computing Center at the University of Texas at Austin, and the BGL logs from IBM Blue Gene/L supercomputer. These systems were chosen because of the availability of the logs and they are among the top 500 widely used supercomputers. Further, their event logging system is representative of a many other similar systems.

Following, a sequence, an input vector for the pattern detection algorithm is labelled as either a failure or non-failure. We implemented normal filtering approach as explained earlier in order to compare with our approach. Note that we could not implement the approach by Zheng et al. [5] to compare with ours because it is log-specific. It cannot be generalised with logs that are not labelled with severity levels, which is the case for most systems. Hence we compare with *normal filtering* method which is the most used.

To form the basis of our evaluation, we use information retrieval metrics *precision* (the relative number of correctly detected failure patterns to the total number of detections); *recall* (the relative number of correctly detected failure sequences to the total number of failure sequences) and *F-measure* (harmonic mean of precision and recall) to measure the performance of our approach. They are as expressed in Equations (3) - (5). We capture the parameters in the metrics as follows: *True positives (TP)*: Number of failure sequences/patterns correctly detected. *False positives (FP)*: Number of non-failure (good) sequences detected as failure. *False negatives (FN)*: Number of failure sequences identified as non-failure sequences.

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

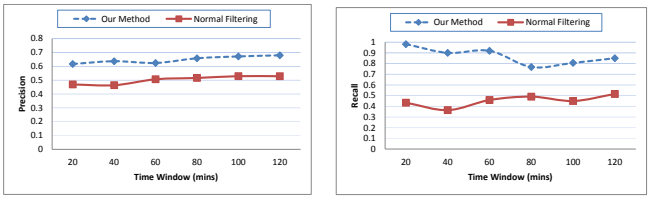
$$F - Measure = 2 * \frac{Precision \times Recall}{Precision + Recall} \tag{5}$$

B. Results

The results are captured in Figures 5, 6, 7 and 8. Each plot of the graphs contains two curves, one is representing the detection efficacy of *our method*, and the other representing that of *normal filtering*.

As mentioned in the introduction, we obtained a good log events compression from the original size. We obtained compression rate of 78%, 80% and 84% on syslogs, ratlogs and Blue Gene/L logs respectively with $LD = 3$. Normal filtering achieved an average compression of 88%. We show from Figure 4, the compression rate on syslogs data as the value of LD increases.

Figure 4. Compression rate on syslogs data against LD

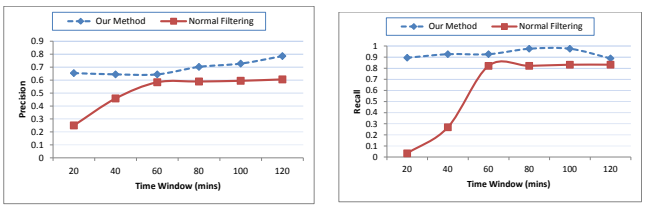


(a) Precision (b) Recall

Figure 5. Precision and recall showing effectiveness of failure detection on *syslogs* filtered with our method and normal filtering

syslogs: Results, as seen in Figure 5, shows that the *precision* and *recall* on logs filtered by *our method* is consistently higher than on those filtered by normal filtering through all the time windows captured. Furthermore, filtering using our method achieve highest precision and recall of 69% and 88%, respectively, *normal filtering* on the other hand is considerably lower with peak precision and recall of 53% and 52% respectively. Our filtering method achieved a relative improvements of about 16% and 26% over normal filtering, for precision and recall respectively.

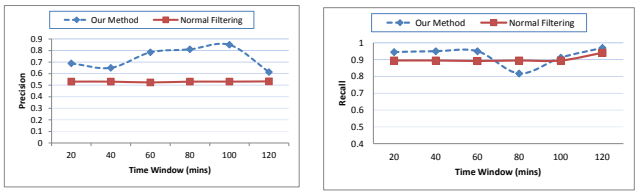
ratlogs: On ratlogs (see Figure 6), both *precision* and *recall* for our filtering method are consistently high across time windows, ranging between $67\% - 80\%$ for the former and between $79\% - 98\%$ for the latter. However, both precision and recall for normal filtering is inconsistently low, with maximum precision of 60% and recall of 82%.



(a) Precision (b) Recall

Figure 6. Results showing effectiveness of failure detection on *ratlogs* filtered with our method and normal filtering

 ISBN: 978-1-61208-429-9



(a) Precision (b) Recall

Figure 7. Showing effectiveness of failure detection on *BlueGene/L (BGL)* logs filtered with our method and normal filtering

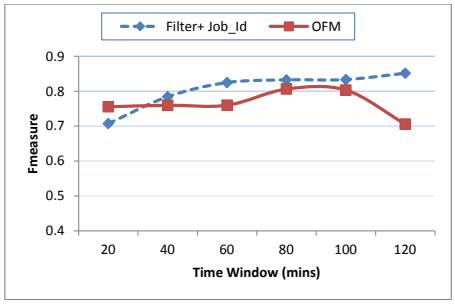


Figure 8. Detection performance on ratlogs using our filtering method without using additional structure (legend: *OFM*) and logs compressed with additional useful structure (legend: *Filter + Job_ID*).

Similarly, on **IBM BlueGene/L (BGL)**, the *precision* as seen in Figure 7 shows there is improvement in detection using our method over normal filtering. It achieved an average improvement of about 30% over normal filtering. *Recall* for both methods are high, however, our method performed better at smaller time windows.

What is the implication of these results? Our method achieved an improvement over *normal filtering* of about $10\% - 30\%$ across all logs used. One of the reasons for this is that, after careful manual investigation, we discovered that failures experienced as captured in these logs are often preceded by event patterns within a short time window. Further, our filtering method was able to preserve these precursor events to failures. This implies that our approach can aid system administrators take necessary failure preventive measures earlier.

We show the result of compression taking *job-ids* into consideration in Figure 8. This result is for ratlogs only, being the only logs with job-id field among the three logs used. The result shows that there is a remarkable improvement over not using job-ids for compression with an average detection improvement of about 15%. The increased detection in logs compressed with job-ids can be explained by the fact that events which are reported by same jobs and are semantically related, yet not similar are properly filtered.

VI. Related Work

Data mining and machine learning techniques are the mostly used in recent works that focused on analysing logs for failure analysis in cluster systems. These works can be found in [15][16][17][18] and [19], and they all developed algorithms that mine patterns of events in the logs. The works in [20] and [18] combines console logs with source code and employed PCA to obtain faulty patterns in the logs. The authors of [2] proposed a method for analysing system generated messages by extracting sequences of events that frequently occur together. In [21], the authors proposed a technique and developed a tool based on clustering called HELO, to extract event templates and describes the templates for system administrator's use. None of the above work considers removing any redundancy in the events logs. They considered every event useful for analysis.

Zheng et al. [5] proposed a method that pre-processes logs and removes redundant events without losing important ones, for failure prediction. In their approach, redundancy from both a temporal and spatial viewpoint is considered. They also filter events based on their causal relationship. Unlike this method, we assume that temporal events must occur in sequence to be removable and we believe that causally-related but semantically unrelated events are patterns or signatures to failure, therefore we keep them. Since the method of [5] cannot be implemented on logs without severity levels, we conjecture that the approach will yield either a high false positive or negative, should it be used on these types of logs. Hence we couldn't compare this method with ours.

Pecchia et al. [22] developed an approach based on heuristics combined with statistical techniques that provides likelihood of events produced by different nodes to removed unwanted events. Their approach is different from ours as they focused on analysing the effects of tupling on compression while we proposed a new filtering approach.

Other approaches that use clustering can be found in [23] and [16]. The latter mainly focus on extracting the message types that can be used for indexing, visualization or model building. One of the caveats of this approach is that it clusters events/message types that are believed to have been produced by the same print statement and their occurrences is non-overlapping. In contrast, our approach can cluster overlapping and non-overlapping events together.

VII. Conclusion and Future Work

We have presented a novel, generic compression algorithm that can be instantiated according to the structure of the log files. Our method did not only compressed logs, it first extract message types in logs. The clusters formed and indexed with ids represents message types. These message types are useful in log analysis e.g., visualization, indexing. Our compression method make use of event similarity (Levenshtein distance), and event structure to determine a redundant event. The efficiency of the compression technique is validated through a proposed pattern detection algorithm. The results from three different logs demonstrate that compression does not only reduce log size which leads to low computational cost of failure analysis, but also enhance better detection of failure patterns. As future work, we intend to use the result and perform failure prediction.

 ISBN: 978-1-61208-429-9

ACKNOWLEDGEMENTS

The data which was analyzed in this paper was available through the SUPReMM project funded by NSF grant ACI-1023604, and has utilized and enhanced the NSF-funded system Ranger (OCI-0622780). We thank the PTDF Nigeria for partly funding this research.

REFERENCES

[1] J.-C. Laprie, "Dependable computing: Concepts, challenges, directions," in *COMPSAC*, 2004, p. 242.

[2] A. Gainaru, F. Cappello, J. Fullop, S. Trausan-Matu, and W. Kramer, "Adaptive event prediction strategy with dynamic time window for large-scale hpc systems," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, ser. SLAML '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:8.

[3] E. Chuah, S. hao Kuo, P. Hiew, W.-C. Tjhi, G. Lee, J. Hammond, M. Michalewicz, T. Hung, and J. Browne, "Diagnosing the root-causes of failures from cluster log files," in *2010 International Conference High Performance Computing (HiPC)*, dec. 2010, pp. 1 –10.

[4] 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.

[5] Z. Zheng, Z. Lan, B. H. Park, and A. Geist, "System log pre-processing to improve failure prediction," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 572–577.

[6] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta, "Filtering failure logs for a bluegene/l prototype," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, June 2005, pp. 476–485.

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

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet physics doklady*, vol. 10, 1966, p. 707.

[9] J. Hansen and D. Siewiorek, "Models for time coalescence in event logs," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, jul 1992, pp. 221 –227.

[10] R. Iyer, L. Young, and P. Iyer, "Automatic recognition of intermittent failures: an experimental study of field data," in *Computers, IEEE Transactions on*, vol. 39, no. 4, apr 1990, pp. 525 –537.

[11] E. Chuah, G. Lee, W.-C. Tjhi, S.-H. Kuo, T. Hung, J. Hammond, T. Minyard, and J. C. Browne, "Establishing hypothesis for recurrent system failures from cluster log files," in *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 15–22.

[12] 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.

[13] 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.

[14] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *SIGCOMM Computer Communication Review*, vol. 35, no. 4. New York, NY, USA: ACM, Aug. 2005, pp. 217–228.

[15] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IP Operations Management, 2003. (IPOM 2003). 3rd IEEE Workshop on*, 2003, pp. 119–126.

[16] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," in *IEEE Trans. on Knowl. and Data Eng.*, vol. 24, no. 11. Piscataway, NJ, USA: IEEE Educational Activities Department, Nov. 2012, pp. 1921–1936.

[17] M. Aharon, G. Barash, I. Cohen, and E. Mordechai, "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ser. ECML PKDD '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 227–243.

[18] 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.

[19] 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.

[20] 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.

[21] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer, "Event log mining tool for large scale hpc systems," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–64.

[22] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. Iyer, "Improving log-based field failure data analysis of multi-node computing systems," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, June 2011, pp. 97–108.

[23] S. Jain, I. Singh, A. Chandra, Z.-L. Zhang, and G. Bronevetsky, "Extracting the textual and temporal structure of supercomputing logs," in *High Performance Computing (HiPC), 2009 International Conference on*, Dec 2009, pp. 254–263.

 ISBN: 978-1-61208-429-9