Identifying Faults in Large-Scale Distributed Systems by Filtering Noisy Error Logs

Xiang Rao, Huaimin Wang, Dianxi Shi, Zhenbang Chen
National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology
Changsha, P.R.China 410073
{*xrao, hmwang, dxshi, zbchen*}*@nudt.edu.cn*

Hua Cai, Qi Zhou, Tingtao Sun
Computing Platform
Alibaba Cloud Computing Corporation
Hangzhou, P.R.China
{*cai.ch, jackson.zhouq, tingtao.suntt*}*@aliyun-inc.com*

***Abstract*—Extracting fault features with the error logs of fault injection tests has been widely studied in the area of large scale distributed systems for decades. However, the process of extracting features is severely affected by a large amount of noisy logs. While the existing work tries to solve the problem by compressing logs in temporal and spatial views or removing the semantic redundancy between logs, they fail to consider the co-existence of other noisy faults that generate error logs instead of injected faults, for example, random hardware faults, unexpected bugs of softwares, system configuration faults or the error rank of a log severity. During a fault feature extraction process, those noisy faults generate error logs that are not related to a target fault, and will strongly mislead the resulted fault features. We call an error log that is not related to a target fault a noisy error log. To filter out noisy error logs, we present a similarity-based error log filtering method SBF, which consists of three integrated steps: (1) model error logs into time series and use haar wavelet transform to get the approximate time series; (2) divide the approximate time series into sub time series by valleys; (3) identify noisy error logs by comparing the similarity between the sub time series of target error logs and the template of noisy error logs. We apply our log filtering method in an enterprise cloud system and show its effectiveness. Compared with the existing work, we successfully filter out noisy error logs and increase the precision and the recall rate of fault feature extraction.**[1]

***Keywords*-error log; event filtering; fault injection; large scale distributed system;**

I. Introduction

In order to design an appropriate fault tolerance strategy for a large scale distributed system, the fault feature model of the system should be built up first [1]. A common way to model the features is to inject faults into the system and observe the subsequent anomaly symptoms from both the client QoS view and the system inner behavior view, e.g. error logs [2]. Event log has been widely used for characterizing software faults for decades [3]. The patterns of different log occurrence can help to extract the feature of a specific fault through pattern mining algorithms like Apriori association rule [4] and decision tree [5].

However, due to the boosting complexity and scale of distributed systems, there always exist noisy faults in a system. These noisy faults produce ambiguous noisy event logs that are unrelated to an injected fault, e.g. random memory errors, disk errors or network glitter, unknown bugs are touched or even configuration errors exist during a fault injection process. Except the noisy faults, the log severity level may be falsely set by developers either and cause to generate large amount of unrelated error logs. These problems are commonly happened in nowadays distributed systems. The existence of noisy logs hinders a fault feature extraction process in at least two ways: (1) some event logs will be falsely identified as a part of a fault feature; (2) the convergence of the precision and recall with target data set is slow and more learning data is needed to increase the precision of extracting a fault feature.

Filtering event logs is a challenging work and been widely studied and applied in large scale distributed systems. Existing work mainly focuses on improving the compression rate, such as filtering event logs from both spatial view and temporal view [6][7], or using causality analysis to remove the semantic redundancy [8]. However, under the existence of noisy log, unrelated logs are either falsely identified as a part of fault features or need to execute fault injection tests multiple times to get a relatively reasonable model. Coarsely removing all the ambiguous error log types that occur before injecting a fault may lose the precision of the fault features.

In this paper, we present an event log filtering method by modeling the event logs of a specific type into log time series and filtering the unrelated log via the similarity of the time series. Our method has two advantages compared with existing work: (1) event logs are filtered with a finer granularity and the suspicious event logs probably related to the injected fault are preserved; (2) less tests are required to extract a fault feature with comparable precision.

The rest of this paper is organized as follows: Section II introduces the preliminaries of our work. Section III presents the similarity based event log filtering method. Section IV shows the experimental results. The related work is reviewed in Section V and the conclusion is drawn in Section VI.

II. Preliminaries

The event logs in our target system record software behavior [9] rather than hardware status or failures [6][7][8]. The event logs are exemplified as follows:

[1]This work was carried out at Alibaba Cloud Computing Company

978-1-4577-0375-1/11/$26.00 ©2011 IEEE

[2010:11:24 11:01:13.193424] [*INFO*] [build/common/log_manager.cpp:744] *preparation LogHeader successes*

[2010:11:29 22:37:29.131597] [*WARNING*] [build/Schedule/master.cpp:420] *RoleMerge:Role Normal*

[2010:11:29 22:37:30.121176] [*ERROR*] [build/FileSystems/worker.cpp:823] *FailureProcessed:*$SSset = -1$

An event log mainly contains 4 attributes: (1) **Logging Time** is the time when the event happens with the accuracy of microsecond; (2) **Log Level** is the severity of the log; (3) **Log Path** is the exact line of the source code that generates the log; (4) **Description** is the detail information that the developers give to the log, including static text, the status variables and the stack trace. Log categorization is challenging and has been studied for decades for problem determination and failure prediction by using text extraction technology [10]. Although there are many other aspects that can be used to categorize event logs, we choose **Log Path** as the type of an event log. Because Log Path shows the exact line of the code producing the log and represents the logical context of an error.

Figure 1 shows a fault injection test case from a log time series view. The time window Wf ranges from injecting a fault till the recovery of the fault, and Wt is the time window before injection a fault. As can be seen from Figure 1, not every error log in Wf is related to the injected fault, and some of them have already been generated in Wt. When characterizing a fault, traditional pattern mining approaches such as $Apriori$ and $DecisionTree$ might falsely identify those logs as part of the fault features.

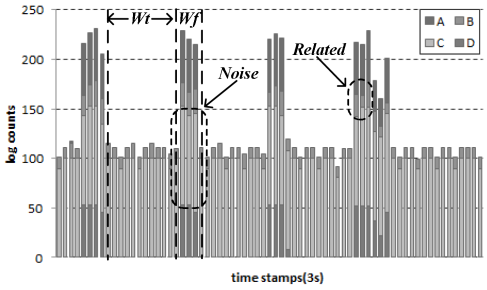


Figure 1. How Noisy error logs interfere the fault characterization process.

As shown in Figure 1, error logs of type B and C occur periodically in both Wf and Wt. When applying pattern mining techniques such as *Apriori* on all Wf, it could easily identify A, B, C, D as a frequent item. However, by manually examining the logs, we found that B is logged for a mis-configuration of the testing environment while C was caused by a common library and could be generated for a wide variety of faults. A coarse way to filter those error logs in Wf is to remove them if they are occurred in Wt. However, some candidate noisy error logs in Wt show an obvious pattern change of occurring counts after injecting a fault. For example, in Figure 1, the average counts of type B was increased from 10 to 25 when the fault was injected into the system, which means B might be related with the fault to some extent.

A. Definitions

Log categorization is challenging in log-based problem determination and failure prediction. A common way to tackle this problem is to use text extraction techniques [10]. We define an event log as a 2-dimension vector $\langle type, time \rangle$. Although the clock frequency of different hosts are not concurrency in a distributed system, we can limit the time shifting within seconds in our testing environment with the help of NTP (Network Time Protocol) service.

Definition 1 (Log time series of a log type α): Given a log type α, a natural number set N, a sampling interval s, a start time ts and an end time te, a time series of α in $[ts, te)$ is a n-dimension vector $P = \langle p_1, p_2, p_3, \cdots, p_n \rangle$, where $p_i (1 \leq i \leq n)$ and p_k is the count of the log type α that occur during $[ts + i \times s, ts + (i+1) \times s)$, and $ts + (i+1) \times s < te$. We call $P_{i,j} = \langle p_i, p_{i+1}, \cdots, p_j \rangle$ as a **sub time series** of P, where $i < j$ and $ts + (j+1) \times s < te$.

A common way to extract a time series' feature is to use the Discrete Wavelet Transform [11] techniques. With Discrete Wavelet Transform, we can reduce the noise and get an approximate time series that contains only the key characters of the original time series. This approximation process is done by wavelet decomposition and reconstruction. A wavelet *decomposing operation* decomposes a time series into *approximation coefficients* and *detail coefficients*. The key characters are presented in the *approximate coefficients* while the noise is stored in the *detail coefficients*. By reconstructing the time series from *approximate coefficients* and eliminating the *detail coefficients* (e.g. assign them to 0), we can get an **approximate time series**. We use **Trans**$(wavelet, w)$ to represent the approximation process, where $wavelet$ is the wavelet form used to decompose and reconstruct the time series and w is the original time series. $Trans$ returns the approximate time series aw.

In this paper, we use **Haar wavelet transform** for $wavelet$, which is a very basic and most widely used wavelet transform. Haar wavelet transform is done by a series of average and difference operations on the input time series. The 1-*level* decomposition and reconstruction process has a time complexity of $O(n)$, where n is the size of the time series. However, the choosing of Discrete Wavelet Transform Level is another open question. In this paper we choose 1-*level* wavelet transform by our experience, and its validity is justified by our experimental results.

Definition 2 (Similar time series): Given two time series W, W' and a threshold T, if $D(W, W') < T$, then W is

similar to W', where D is a similarity calculation function.

There are many methods for calculating similarity, such as Minkowski distance [12], Dynamic Time Warping (DTW) [13], *etc*. We choose DTW that uses a dynamic-programming methodology to find the best matching path between two time series to get the minimized Minkowski path. Compared with the original Minkowski distance, DTW is better in accuracy and the tolerance of time shifting. Giving two time series $W = \langle w_1, w_2, \cdots, w_n \rangle$ and $W' = \langle w_1', w_2', \cdots, w_m' \rangle$, the Dynamic Time Warping distance $D(W, W')$ is defined in Formula 1 [13] where $dist(w_i, {w_i}') = |w_i - {w_i}'|$, n and m is the size of W and W'.

$$\begin{aligned} &D(W, W') = dt(n, m) \\ &dt(n, m) = 0, dt(i, 0) = dt(0, j) = \infty \\ &dt(i, j) = dist(w_i, w_j') + \min \begin{cases} dt(i, j-1) \\ dt(i-1, j) \\ dt(i-1, j-1) \end{cases} \\ &(i = 1, ..., n; j = 1, ..., m) \end{aligned} \tag{1}$$

One drawback of DTW is its time complexity, i.e. $O(n \times m)$, where n and m is the size of W and W'. Though there are choices with a lower complexity, we prefer to get a better accuracy and select DTW as our similarity calculation function. We are trying to filter out the logs that may not relate to an injected fault under the following assumptions:

- A1 A stable workflow persistently invoking the common requests of the target service during the fault injection test.
- A2 All logs have been well categorized.
- A3 All injected faults are evidenced in logs by a signature despite there exist few failures without signatures.
- A4 The types of the logs that occur before a fault (in Wt) have the most suspicion to be unrelated with the injected fault.

A fault injection testing framework guarantees A1. A2 is guaranteed by using Log Path as the log categorization heuristics, it also means our filtering method can be easily extended to other log categorization method. For A3, although there may exist other failures that leave no signature on error logs, we believe such failures are rare. A4 is inspired by our observation and lays the basis of our filtering method, which means we will ignore the error log types that are not occurred in Wt for a specific fault injection test.

Definition 3 (noisy logs of type α): Given Wt as the time range $[ts, te)$ staring from the start of a test till injecting a fault, Wf as the time range $[te, te')$ starting from injecting a fault till the recovery of the system, a sampling interval s, a log time series P of type t in Wt, a time series P' in Wf, $P_{i,j}$ as the sub time series of P, $P'_{k,l}$ as the sub time series of P' and a similarity threshold T. If $D(P_{i,j}, P'_{k,l}) < T$, then all the logs of type t in $[te + k \times s, te + (l+1) \times s)$ are identified as noisy logs, where D is the similarity function.

In this paper, we model event logs into time series and try to detect *amphibious* event logs by comparing the occurrence pattern with the candidate noise template. By filtering the "matched" event logs and leaving the "amphibious" event logs, we can successfully filter out the noisy event logs and increase the fault diagnosis precision rate and recall rate.

III. Similarity-based Log Filtering

Given a time series P of a log type α in Wt, a time series P' in Wf and a similarity factor $th \in [0, 1]$, the detailed Similarity-based Log Filtering (denote as SBF) method is shown in Algorithm 1.

Algorithm 1: Similarity based log filtering

Input: P is a time series of a log type α in Wt, P' is a time series of α in Wf, th is a similarity factor.
Output: P'', which is the filtered time series of P'

1 reconstruct P and P' with **Trans** and return TP and TP' ;
2 divide TP and TP' into sub time series with $segmentation$ and return sub time series sets TPS and TPS' ;
3 **foreach** i *of* TPS **do**
4 /*calculate the threshold for each noisy template independently*/
5 $T \leftarrow 0$; // T is the similarity threshold
6 $initP \leftarrow \langle 0, \cdots, 0 \rangle$; // initialize a zero time series and $|intiP|$ equals to $|i|$
7 $T \leftarrow D(i, initP) \times th$;
8 **foreach** j *of* TPS' **do**
9 $distance \leftarrow D(i, j)$;
10 **if** $distance \leq T$ **then**
11 delete the logs of type α from P' in the time range of j;
12 **Break** ;

Algorithm 1 mainly contains the following 3 steps:

1) Line 1 gets the approximate time series TP and TP' of P and P' through $Trans$.

2) Line 2 divides TP and TP' into sub time series set TPS and TPS' by $segmentation$. We use the nearest two valley of a time series as a heuristic to divide a time series into sub time series set. That is, for a time series $P = \langle p_1, \cdots, p_n \rangle$, if $p_i < p_{i-1}$ and $p_i < p_{i+1}$ then we get a valley point p_i. The original time series can be divided into sub time series by the nearest two valley point's time stamp. We use $segmentation(P)$ to represent this process. The time complexity of $segmentation(P)$ is $O(n)$ where n is the size of time series P.

3) From line 3 to 13, it first calculates a similarity threshold of noisy template i by th, than for each element j in TPS', it calculates the similarity with the ith noisy template by using similarity measurement D. If the resulted distance is less than the threshold T, j is a matched sub time series, and all the logs of type t in the time range of j should be deleted.

To calculate the time complexity of Algorithm 1, we assume: (1) $|TP| = n_1$, $|TP'| = n_2$, $|TPS| = k_1$ and $|TPS'| = k_2$; (2) each time series of TPS contains $m1$'s sampling points on average, while m_2 for TPS'. We have $n_1 = k_1 \times m_1$ and $n_2 = k_2 \times m_2$. Both Step 1 and Step 2 have a time complexity of $O(n_1 \times n_2)$, and $D(i, j)$ has a time complexity of $O(m_1 \times m_2)$ [13], while the iteration has been executed for $k_1 \times k_2$ times, which makes the overall time complexity is $O(k_1 \times k_2 \times m_1 \times m_2)$ = $O(n_1 \times n_2)$.

An example of the SBF filtering effect is shown in Figure 2 with $th = 0.1$. Picture (e) is the final filtering result, Algorithm 1 successfully filters most of the noisy logs and leaves those suspicious logs.

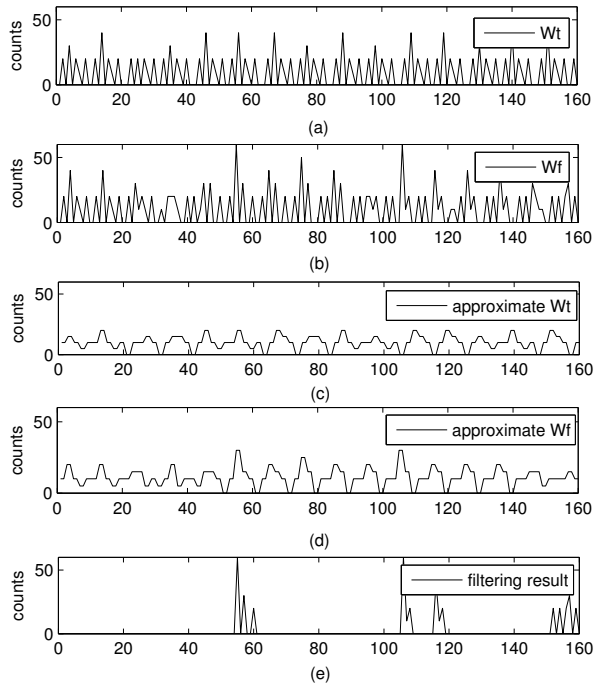


Figure 2. Example Effects of SBF. (a) and (b) are time series of a log type that extracted from Wt and Wf, while (c) and (d) are approximate time series of (a) and (b) after **Trans**('haar',w) process; (e) is the filtered result of (b) by using SBF

IV. EXPERIMENTS

The experiments are to evaluate the effectiveness of SBF log filtering method on improving fault detection precision and recall. The target system is a large scale distributed system built on the Master-Worker style. The experiments were done on a cluster composed of 100 nodes. There are 10 test cases in each test round, and totally 30 rounds of tests are executed. After each test round, the logs are gathered and imported into a centralized DBMS. Totally 2,800,973 logs are collected.

A. Test case overview

In our experiments, a test round contains 10 fault injection test cases that simulate the process crash scenario by killing the process. The basic statistic information of all test cases is listed in Table I. All test cases can be clustered into 3 categories: (1) crash a worker process, (2) crash a master process, (3) crash a communication agent. In each test case we only crash one process. For multiple worker processes, the test case will randomly choose one to crash. DBmr1 uses application requests while DBmr2 uses inner requests to judge the service's status. The average recovery time of each service crash is rather different, ranges from 16 seconds to 548 seconds. For each test case, a priori knowledge of the average recovery time is needed for setting the timeout of for each test case. For those timeout cases and short recovery time cases, we choose 100 seconds for Wt.

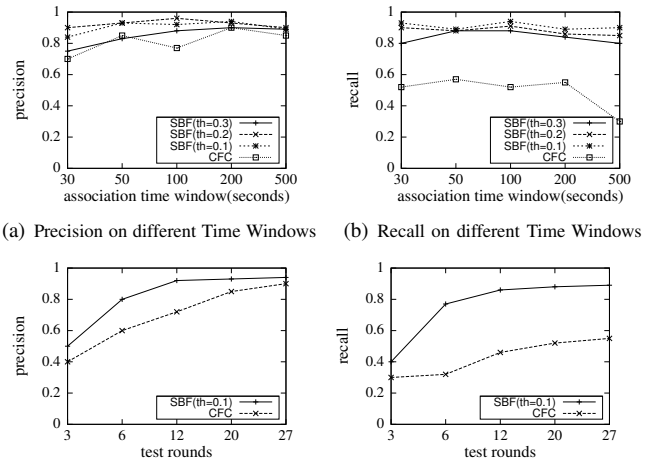
Table I
OVERVIEW OF TEST CASES

test case	average recovery time (seconds)	average Wt (seconds)
DBwr fail	156	150
DBmr1 fail	timeout	100
DBmr2 fail	16	100
FSwr fail	34	100
FSmr fail	548	300
SSwr fail	timeout	100
SSmr fail	80	100
NLwr fail	68	100
MSmr fail	80	100
CA fail	135	150

B. Experiments Results

We follow the basic processing framework to extract fault feature that in [8] (denote as CFC): (1) filter the raw logs from spatial and temporal views; (2) filter the semantically related combined events; (3) use the resulted combined events to construct the decision tree. The difference between our work and [8] is we add a step before (1), that is: (0) filter out the noisy logs by Algorithm 1.

A 10-fold cross validation process is applied for evaluating the precision and recall of the model. As was proposed in [8], we evaluate two metrics of fault detection: precision (i.e. the proportion of correct detections to all the detection results) and recall (i.e. the proportion of correct predictions to the number of injected faults).Due to different scenarios and system configurations compared with [8], we choose 30, 50, 100, 200 and 500 seconds as the candidate association



(a) Precision on different Time Windows (b) Recall on different Time Windows

(c) Precision on different size of data (d) Recall on different size of data

Figure 3. Experiment results. (a) and (b) compare the precision and recall of SBF and CFC under different settings. (c) and (d) compare the congestion of SBF and CFC in time window 200 seconds.

time window with regard to the average recovery time of different test cases.

As shown in Figure 3(a) and Figure 3(b), our presented method generally acquires better precision rate and recall compared with CFC when noise faults exist. For CFC, the noise faults strongly affect the recall rate. It is because the resulting association rules of CFC are affected by the noisy log, which produce the wrong judgment conditions in the decision tree and cause the failure of fault detection. Especially, when the association time window is large (e.g. 500 seconds), we get the worst recall rate of the experiment - 30%. However, the precision rate seems to be less affected by noisy logs in CFC. It is because the detected faults are actually related with some noisy logs from client libs, and the features of those faults are rather different.

SBF filters those unrelated noisy logs and provides more accurate association rules. To the effect of the time series threshold of SBF, there are no obvious differences when setting th to 10% or 20%. However, a little performance degradation is observed when setting th to 30%. That is because some logs are mistakenly filtered and some judgment conditions in the decision tree are lost. The best precision of SBF is 96% in 100 seconds when $th = 20\%$ and the best recall is 94% in 100 seconds when $th = 10\%$, compared with 90% in 200 seconds and 57% in 100 seconds of CFC.

Another interesting comparison is done on evaluating the relationship between fault detection results and the number of test rounds used to learn the model. We choose time window of 200 seconds as an example. As shown in Figure 3(c) and Figure 3(d), SBF congests much faster than CFC. CFC should use more learning data to acquire a relative good precision rate and recall rate because of the noisy logs. With SBF, the system administrator can use less data to fast the feature modeling of a fault, which saves human and computing resources.

V. Related work

Event logs have been widely used to characterize fault features. The scenarios for generating logs can be categorized into two classes: getting logs from production system [15][16] and using fault injection tools like G-SWFIT[2] to generate logs during the testing phase. The latter always injects fault into source code to simulate logical mistakes, and then evaluates the impact of the faults from both QoS view and fault detection view [17]. However, as observed in our target system, process crashing is a much common fault than logical error. We choose to use crashing process as our injected fault, which can offer much practical result for daily system management.

However, nearly all logging systems have redundant and inaccurate logs. Existing log analysis work tries to solve it by filtering the log from both temporal and spatial views with high log compression rate [6][8], that is to remove the same type of events reported from different location (e.g. different host) within a time window. The purpose of such filtering process is to reduce the related set of alerts to a single initial alert per failure. It makes the ratio of alerts to failures nearly one. Zheng, Z. *et al.* give a practical log pre-processing method by combining regular expression based event categorization and Apriori association analysis rules [8]. It effectively filters out semantic redundant logs without sacrificing the precision of failure identification and improves the failure prediction by up to 174% with compression rate more than 90%. However, existing work ignore the noisy log that occurs during a fault injection process, which is difficult to distinguish the cause-effect relation between event logs and injected faults. With the existence of noisy faults during system running, existing work may extract imprecise fault model because of keeping unrelated logs or removing the related logs.

Compared with the existing work, we model the logs of a specific type into time series and use Haar wavelet to extract the log occurrence pattern. By filtering out logs that fits the noise template, we successfully filter the noisy logs in a fine granularity and improve the validation of fault features.

VI. Conclusions

Extracting fault features by fault injection tests is usually affected by noisy logs. We present a log filtering method by modeling the logs of a specific type into log time series and filtering out the noisy log via the similarity between time series. The main contributions of our work are: (1) logs are filtered in a finer granularity and the suspicious logs are preserved to better characterize fault features; (2) less learning data is needed to compute a fault feature with a comparable precision. Comprehensive experiments have

been carried out, and the results show a better precision and recall rate compared with the existing work.

ACKNOWLEDGMENT

This research is supported by the National 973 Program of China under Grant No.2011CB302603. We also thank Chen Xinyu, Chu Rui, Ding Bo, Feng Dawei, Li Xiao, Mi Haibo, Yuan Ling, Yin Gang, Zhou Yangfan, Zhu Jun and Zhu Yanxu for their great suggestions for our work.

REFERENCES

[1] S. Yacoub and H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Transactions on Software Engineering*, pp. 529–547, 2002.

[2] J. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, pp. 849–867, 2006.

[3] T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 419–432, 1990.

[4] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *ACM SIGMOD Record*, vol. 22, no. 2. ACM, 1993, pp. 207–216.

[5] A. X. Zheng, J. Lloyd, and E. Brewer, "Failure diagnosis using decision trees," in *Proceedings of the First International Conference on Autonomic Computing*. IEEE, 2004, pp. 36–43.

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

[7] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2006, pp. 425–434.

[8] Z. Zheng, Z. Lan, B. Park, and A. Geist, "System log pre-processing to improve failure prediction," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2009, pp. 572–577.

[9] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2010, pp. 457–466.

[10] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[11] F. Chan, A. Fu, and C. Yu, "Haar wavelets for efficient similarity search of time-series: with and without time warping," *IEEE Transactions on knowledge and data engineering*, vol. 15, pp. 686–705, 2003.

[12] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*. Springer-Verlag, 1993, pp. 69–84.

[13] D. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *AAAI-94 workshop on knowledge discovery in databases*, 1994, pp. 229–248.

[14] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos, "Approximate embedding-based subsequence matching of time series," in *Proceedings of ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 365–378.

[15] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–351, 2010.

[16] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proceedings of International Conference on Dependable Systems and Networks*. IEEE, 2007, pp. 575–584.

[17] R. Moraes, J. Duraes, R. Barbosa, E. Martins, and H. Madeira, "Experimental risk assessment and comparison using software fault injection," 2007.

[18] M. Steinder and A. Sethi, "Probabilistic fault localization in communication systems using belief networks," *IEEE Transactions on Networking*, vol. 12, no. 5, pp. 809–822, 2004.

[19] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 13–24, 2007.

[20] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. ACM, 2009, pp. 243–254.