ABSTRACT

DAS, ANWESHA. Predicting Location and Time of Anomalies in Large-Scale Computing Systems via Log Mining. (Under the direction of Rainer Frank Mueller).

Today's large-scale supercomputers encounter faults on a daily basis. Exascale systems are likely to experience even higher fault rates due to increased component count and density. Predicting which node will fail and how soon remains a problem for HPC resilience that needs to be solved to pave the way to exploiting proactive remedies before jobs fail. Triggering resilience-mitigating techniques is still difficult due to the absence of well defined failure indicators. Not only for increasing scalability up to exascale systems but even for contemporary supercomputer architectures does it require substantial efforts to distill anomalous events from noisy raw logs. System logs consist of unstructured text that obscures essential system health information contained within. In this context, efficient failure prediction via log mining can enable proactive recovery mechanisms to improve reliability.

This thesis makes the following contributions. Two novel solutions are proposed to pin-point node failures, which are unprecedented. First, a phrase extraction-style mechanism, called TBP (time-based phrases) demonstrates the feasibility to predict imminent node failures in Cray systems. Second, a deep learning-based framework, called Desh, predicts short lead times to failures via long short-term memory (LSTM) networks. These open up the door for enhancing prediction lead times for supercomputing systems in general, thereby facilitating efficient usage of both computing capacity and power. Next, an auto-generated inference scheme, called Aarohi, is developed to achieve speed up during online prediction. Aarohi's parsing is designed to be generic, adaptive, and scalable making it suitable for real-time inference. This compiler-based approach provides a fresh perspective for lead time optimization with a significant prediction speedup required for the deployment of proactive fault tolerant solutions in practice. Further, root cause analysis of node failures is explored using an integrated measurement driven approach to better understand how nodes fail. Our empirical observations about environmental influence and the application effect on failures along with feasible lead time enhancements can facilitate better failure handling in production systems.

Finally, we discuss our efforts to deploy the developed failure prediction solutions in a realistic setting to be able to circumvent the hurdles if any. The thesis concludes with a discussion of future research directions in the context of proactive fault tolerance and sustained resilience for large-scale computing systems.

© Copyright 2019 by Anwesha Das

All Rights Reserved

Predicting Location and Time of Anomalies in Large-Scale Computing Systems via Log Mining

by
Anwesha Das

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Xipeng Shen	Guoliang Jin
Michela Becchi	Rainer Frank Mueller Chair of Advisory Committee

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **ix**

Chapter 1 Introduction **1**
1.1 Supercomputing Systems 1
1.2 Challenges in Proactive Failure Management 3
1.3 Problem Statement 4
1.4 Hypothesis 5
1.5 Thesis Contributions 5
1.6 Dissertation Organization 7

Chapter 2 Predicting Which Node Will Fail When on Supercomputers? **8**
2.1 Introduction 8
2.2 Background 10
2.3 Predictor Design 14
2.4 TBP Framework 17
2.5 Experimental Evaluation 19
2.6 Related Work 30
2.7 Conclusion 32

Chapter 3 Deep Learning for System Health Prediction of Lead Times to Failure in HPC . **33**
3.1 Introduction 33
3.2 Background 36
3.3 Desh Overview 37
3.3.1 Phase 1: Training 38
3.3.2 Phase 2: Training 41
3.3.3 Phase 3: Testing 42
3.4 Evaluation 44
3.4.1 Prediction Accuracy 44
3.4.2 Lead Times 45
3.4.3 Unknown Phrase Analysis 47
3.4.4 Cost Analysis 51
3.4.5 Desh Comparison 51
3.4.6 Discussion 53
3.5 Related Work 54
3.6 Conclusion 56

Chapter 4 Making Real-time Node Failure Prediction Feasible **57**
4.1 Introduction 57
4.2 Background 59
4.3 Online Failure Prediction Design 61
4.4 Evaluation 68

4.5 Related Work 76
4.6 Conclusion 78

Chapter 5 Systemic Root Cause Analysis of Node Failures in Production HPC 79
5.1 Introduction 79
5.2 Background 80
5.2.1 Preliminaries 82
5.2.2 Methodology 83
5.3 Evaluation Results 84
5.3.1 External Influence on Node Failures 85
5.3.2 Application Triggered Failures 89
5.3.3 Node Internal Failure Analysis 91
5.3.4 Unknown Causes 93
5.3.5 Case Studies 94
5.3.6 Discussion 95
5.4 Related Work 98
5.5 Conclusion 99

Chapter 6 Summary and Future Work 100
6.1 Concluding Remarks 100
6.2 Future Work 102

BIBLIOGRAPHY 105

LIST OF TABLES

Table 2.1 System Details . . . 10
Table 2.2 Data Details . . . 11
Table 2.3 Node Shutdown Events . . . 13
Table 2.4 Examples of Node Failures . . . 14
Table 2.5 Topic Assignment . . . 17
Table 2.6 Phrase Extraction . . . 21
Table 2.7 Recurring Phrases . . . 22
Table 2.8 Evaluation Metrics . . . 22
Table 2.9 Major Failure Categories . . . 24
Table 2.10 Lead Time Improvement . . . 27
Table 2.11 Difficult Correlation Extraction . . . 28
Table 2.12 TBP Comparison . . . 29
Table 2.13 TBP Impact Assessment . . . 29

Table 3.1 Log Details . . . 37
Table 3.2 Phrase Vectors . . . 38
Table 3.3 Phrase Labeling . . . 39
Table 3.4 Example Failure Chain . . . 41
Table 3.5 LSTM Parameter Specifications . . . 42
Table 3.6 Test Data Statistics . . . 43
Table 3.7 Prediction Efficiency . . . 45
Table 3.8 Node Failure Classes . . . 45
Table 3.9 Unknown Tagged Phrases . . . 48
Table 3.10 Unknown Phrases with and without Node Failures . . . 49
Table 3.11 Desh Comparison . . . 51
Table 3.12 BlueGene/L Log . . . 52
Table 3.13 Desh vs. DeepLog . . . 53

Table 4.1 Log Variations . . . 60
Table 4.2 System Logs . . . 60
Table 4.3 Log Message Processing . . . 61
Table 4.4 Parser Grammar . . . 62
Table 4.5 Multiple Rule Matches . . . 66
Table 4.6 Efficiency Formulae . . . 69
Table 4.7 Speedup . . . 70
Table 4.8 Comparative Analysis of Aarohi . . . 73
Table 4.9 Aarohi Adaptability . . . 74

Table 5.1 HPC System Details . . . 80
Table 5.2 Log Data Details . . . 82
Table 5.3 Fault Breakdown . . . 85
Table 5.4 Root Causes of Failures . . . 92
Table 5.5 Sample Failure Cases . . . 95

Table 5.6 Findings and Recommendations 96
Table 5.7 Large-Scale System Evaluation 97
Table 5.8 Comparison 98

LIST OF FIGURES

Figure 1.1 Lead time to a failure 2

Figure 2.1 Overview of a Cray System 11
Figure 2.2 Correlation with Job Logs 15
Figure 2.3 Time Correlation and Data Integration 16
Figure 2.4 TBP Framework 17
Figure 2.5 TBP Prediction: Topic Modeling for Node Failure Prediction 19
Figure 2.6 Estimate of Node Failures 20
Figure 2.7 Phrase Likelihood 21
Figure 2.8 Mean and Std. Deviation 21
Figure 2.9 Sensitivity of Lead Times 23
Figure 2.10 Recall/Precision/FNR Rates 23
Figure 2.11 Failure Categories 24
Figure 2.12 Lead Times+False Positives 24
Figure 2.13 Phrase Reduction and Order 25
Figure 2.14 False Positive Rate 26
Figure 2.15 TBP Scalability 26

Figure 3.1 Desh with LSTM 35
Figure 3.2 Desh Overview 36
Figure 3.3 LSTM Phases 40
Figure 3.4 Prediction Rates 43
Figure 3.5 FP Rate and FN Rate 43
Figure 3.6 Lead Times+Failure Classes 46
Figure 3.7 Avg. Lead Times of Systems 46
Figure 3.8 Lead Times and FP Rate 47
Figure 3.9 Unknown Phrase Analysis 48
Figure 3.10 Cost Analysis 50

Figure 4.1 Two Phase Failure Prediction 58
Figure 4.2 Overall Design 59
Figure 4.3 Aarohi Design 59
Figure 4.4 Phrase Chains 63
Figure 4.5 ΔTimes 64
Figure 4.6 Offline Training to Online Testing 68
Figure 4.7 Phase 1 Efficiency 69
Figure 4.8 W/O Benign Phrases 69
Figure 4.9 With Benign Phrases 69
Figure 4.10 Diverse Platforms 70
Figure 4.11 O3 Optimization 70
Figure 4.12 Token Fraction 71
Figure 4.13 Lead Times to Failures 71
Figure 4.14 System Lead Times 72

Figure 4.15 System Prediction Times 72
Figure 4.16 Predictor Placement 76

Figure 5.1 Cray System[1] 83
Figure 5.2 Methodology 83
Figure 5.3 Failure Times 84
Figure 5.4 Dominant Cause 84
Figure 5.5 Failure Reasons 84
Figure 5.6 Node Faults 86
Figure 5.7 Heartbeat Faults 86
Figure 5.8 Blade/Cabinet Faults 86
Figure 5.9 Blade Counts 87
Figure 5.10 SEDC Warnings 87
Figure 5.11 Cabinet RPM Faults 87
Figure 5.12 Lead Times 88
Figure 5.13 Lead Times with FP 88
Figure 5.14 External Influence 88
Figure 5.15 Job Failures in S5 90
Figure 5.16 Job Failures in S2 90
Figure 5.17 Resource Overallocation 90
Figure 5.18 Root Causes 92
Figure 5.19 Blade Failures 92
Figure 5.20 Temporal Locality 92

Figure 6.1 Improved System Reliability 101
Figure 6.2 Real-Time Failure Prediction 103

CHAPTER

1

INTRODUCTION

The past decade has contributed resilience to HPC (High Performance Computing) by proposing fault tolerant solutions for applications [Vis16; Cha06], reactive recovery approaches such as checkpoint/restart [Ell12; Tiw14], gaining a better understanding of system logs [Wan17a; Gup15], and comprehending the requirements in terms of performance, resilience and power trade-offs [El-12a; Nie17]. While computing systems have matured in computing efficiency (operations/second), scale, and evolved architectures, fault manifestation has become complex. Faults are frequent and are expected to increase with shorter mean time between failures (MTBF) in the next generation systems [Eln08]. Currently, substantial compute capacity and power is wasted in recovering failed components [ES18]. Failure prediction with defined lead times is the need of the hour to combat such unprecedented faults and failing components.

1.1 Supercomputing Systems

As of November 2018, 28% of the top 100 supercomputers are Cray machines [Top]. In such systems, hardware, software, or application malfunctioning give rise to errors. Errors propagate as faults leading to failures. This work develops a better understanding of failure manifestations in the current systems and proposes efficient techniques to predict node failures before an unhealthy node stops responding. To do so, production logs are analyzed, befitting ML-solutions are leveraged, and appropriate methods are developed facilitating proactive resilience. In this dissertation, we use

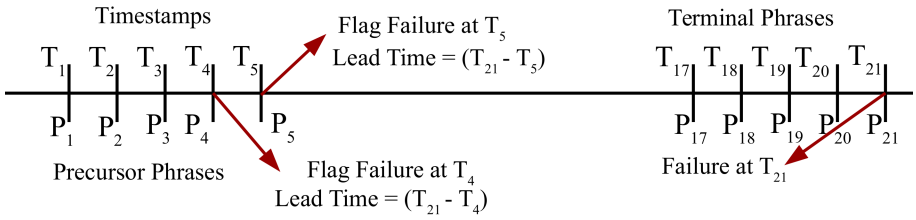

Figure 1.1 Lead time to a failure

the following terminologies:

- **Phrase**: A phrase refers to an event or a log message in any log file obtained from a computing system. Usually, log messages are timestamped, however, they need not necessarily be so.
- **Lead Time**: The messages or phrases are logged at some timestamp. Lead time is the time difference between the time at which a component became unresponsive and the time at which an impeding failure is flagged proactively at a precursor phrase. This precursor phrase may or may not be indicative of any error and precedes the terminal messages leading to a failure. Figure 1.1 illustrates the definition of lead time. Suppose a terminal message indicating a confirmed node failure appears in log message P21 at time T21. If our failure prediction framework flags this failure after checking phrase P5, the calculated lead time is ΔT = (T21 - T5). If it is flagged earlier after checking phrase P4 then the lead time increases to ΔT = (T21 - T4), the increase being (T5 - T4). The higher the lead time, the more time remains to take suitable recovery actions. However, flagging potential failures at much earlier precursor events may also lead to incorrect predictions (false negatives and false positives), thereby decreasing the accuracy. This makes a lead time sensitivity study essential.
- **Prediction Time**: Prediction time refers to the time taken to infer whether an incoming sequence of phrases will lead to a node failure or not. This is also referred to as the inference time during testing. Prediction time is decoupled from the offline training time.
- **Failure**: Failures can be in hardware, software or application unless explicitly mentioned. This dissertation focuses on compute node failures.

Supercomputers help conduct intensive scientific simulations and computing. However, their evolving complex system design and voluminous logs have made data mining-based failure diagnosis non-trivial. In that sense, most technical enhancements become a double-edged sword over a period of time. With applied machine learning-based fault tolerant solutions on the rise, log characterization to understand interconnect, GPU [Nie17], memory [BG16], hardware errors [Wan17a], and the affect of temperature on performance and reliability [El-12a] are being explored to determine failure prevention techniques. At this juncture, it becomes imperative that we take steps to propose

failure prediction solutions for improved resilience. This can avoid expensive checkpoint/restarts saving compute capacity and energy. This dissertation contributes methods for achieving lead times to node failures for predictive localization of anomalies for large-scale computing infrastructures.

1.2 Challenges in Proactive Failure Management

In spite of the fact that a substantial amount of research has been conducted in the context of anomaly detection [Lan10b; Du17; Sal10], log characterization [Gup15; Gup17; Nie17] and root cause diagnosis [Zhe12; Fu14], failure prediction still requires additional research efforts because of the following concerns:

1. The existing approaches do not evaluate lead times to failures, an indispensable requirement for proactive actions. Post-mortem diagnosis is not helpful even if accurate. This requires scalable unsupervised solutions, which can be made to work with noisy unstructured system logs.

2. Online failure prediction necessitates even more stringent timing requirements. Offline training and event correlation plus validation, even with improved recall and accuracy, do not ensure rapid anomaly prediction. The missing link is quick inference during testing so that the prediction time is minimal. This then allows a significant fraction of the remaining time before the component fails to be spent on taking recovery actions such as migration or rescheduling of jobs.

3. Root cause diagnosis considering diverse system conditions across time and space needs further investigation. Spatial and temporal analysis of different events in an HPC system reveal insights to failure patterns and their characterization. There are many internal as well as external conditions and diverse components to consider for resilience actions. Determining what faults and environmental conditions lead to failures in the operational context is challenging. The next step would be to build frameworks that can propose effective solutions to prevent failures based on the outcome of such characterizations.

The road blocks to answering the above questions have been that, the past researchers heavily relied on:

- Fatal severity levels to classify anomalies [Fu14]. In older HPC systems, logs were comparatively more structured than Cray systems. Fatal events served as indicators for feature extraction.

- Techniques of Principal/Independent Component Analysis (PCA/ICA) [Lan10a], Support Vector Machines (SVM), Decision trees, Markov chains, and Bayesian models have been applied for log mining-based anomaly detection. These methods are computationally expensive with increasing log size.

In recent times, system logs have become more complex and dense because of component scaling and diverse log sources. New data mining techniques have also been developed such as deep learning [Chi14; Du17]. With the upcoming exascale[1] era, scalable solutions are required for log analysis unlike prior approaches such as PCA or SVM, which become intractable with scale.

Another concern is the expected short lead time in large-scale computing systems, with MTBFs being reduced originally from hours to a few minutes. It is challenging to obtain short lead times before a failure happens since the major indicators are prior to the terminal failure messages. Analysis of a sequence of events over time and space irrespective of log-severity levels is required for accurate failure prediction. A fatal message could be fatal for a component failure during a specific time frame but benign in another context. Such fine-grained analysis needs a slightly different approach to failure prediction.

Besides the focus on time sensitivity, tracking the location of failure is also important so that recovery actions can be taken assuming system logs are vendor controlled and cannot be modified as per our convenience. Moreover, anomaly injection [Yu16], source code reference [Xu10], and binary instrumentation have been applied in several anomaly detection works in distributed systems but are inappropriate for HPC systems. Supercomputing systems produce lower-level Linux-style logs, which are not application centric. Instead, they consist of OS-level timestamped events. Low level hardware and software malfunctioning is a concern in addition to application bugs. Hence, HPC systems require extensive data correlation and integration to understand the intricacies of anomalies at the lower systems level.

It is time we investigate failure prediction in the light of the above concerns to improve the reliability of the current and next generation computing systems.

1.3 Problem Statement

This dissertation aims to answer the following questions:

"Can we predict that a node is about to fail before it stops responding? If so, how much ahead of time can this prediction be done and with what level of accuracy?"

It should be noted that we want to use the nodes exhaustively before any one node fails, i.e., we do not want to flag an impending failure too early when the node is functioning without any anomalies. In that sense, *just enough* lead time is essential in the context of finishing any feasible proactive action (e.g., live migration within 30 seconds). Otherwise, if a failure was flagged late, just

[1]The next milestone of the Exascale Computing Project [Ecp] is to accelerate the peak compute capacity to quintillion (10^{18}) double floating point operations/second (flops). Currently, the world's fastest supercomputer, Summit, is operating at 200 petaflops (10^{17}).

before the failure manifests (e.g., 10 seconds), proactive resilience may no longer be feasible, i.e., the node's state can no longer be migrated or saved by any means before the failure occurs. However, it may or may not be always feasible to procure *sufficient* lead times to failures, i.e., flag a failure in a timely manner. How much lead time can be achieved depends on how the nodes fail in different systems over different time frames. How similar or dissimilar the sequence of phrases are in the logs leading to a failure versus healthy logs influences the prediction accuracy. In this context, we wish to achieve sufficient lead time with an acceptable false positive rate. *Acceptable* indicates the fact that the majority of the failures can be predicted with minor prediction inaccuracies. We do not put a singular lower bound on the lead time, as different failures pertain to different patterns. Depending on an immanent failure, diverse lead times may be obtained with a limit on the false positive rate. This dissertation addresses this problem of lead time estimation for failing nodes in production systems.

1.4 Hypothesis

We have identified the prerequisites to enable practical failure management in computing systems (Section 1.2), i.e., scalable semi-supervised log mining solutions with defined lead times. The conjecture is that feature extraction from and dimensionality reduction on contemporary system logs is difficult. But, at the same time, sufficient numerical and textual information is contained in these logs to take timely precautionary actions. We may not be able to avoid inaccurate predictions at all times but lowering the false positives and false negatives will contribute to making systems more resilient and to saving compute resources. The challenge is to gain clarity in understanding how faults propagate in the system causing component failures and finding the middle ground where failures can be flagged with sufficient lead time, yet an acceptable false positive rate. We state the following hypothesis for developing failure prediction techniques:

"*The majority of node failures in HPC systems can be predicted via machine learning thereby identifying the precise node location and with sufficient lead time to engage in proactive actions for fault mitigation*".

1.5 Thesis Contributions

This dissertation contributes a characterization of achievable lead times to node failures in HPC systems by exploring contemporary machine learning/deep learning techniques. This can aid in assessing the potential for proactive failure management through a time sensitive study involving prediction times, lead times, false positive, and false negative rates. The dissertation makes the

following specific contributions:

1. Chapter 2 describes TBP (Time-Based Phrase), a topic modeling-based approach for producing lead times to node failures. This scheme demonstrates the requirement of continuous time probabilistic likelihood estimation to decipher emerging event phrases over diverse time intervals. The job logs and system logs are correlated before training. Trained node failure chains are formulated with the help of terminal messages and confirmed in consultation with the system administrators of the facility. TBP checks for impending failures in the test data by comparing with the trained failure chains. Depending on the phrases that form a failure chain, lead time is calculated from the terminal message to the specific phrase where a failure is flagged. TBP achieves as high as 86% recall and 16.66% false negative rate. With 2 minutes average lead time, the false positive rate does not exceed 23%. Additionally, this study shows that 15 to 20% of the nodes fail due to hardware errors, machine check exceptions (MCEs). and kernel panics, while bit errors and application-caused errors are minor contributors.

2. Chapter 3 proposes Deep Learning for System Health (Desh), a deep learning-based method to predict lead times to node failures. Desh uses long short-term memory (LSTM) to train events to form failure chains, re-train chains of events augmented with time differences to learn expected lead times, and eventually predict lead times during testing. Desh obtains at least 83.6% accuracy and 85.7% F1 score along with as high as 87.5% recall rates. It obtains on average 2 minutes lead times with no more than 25% false positives. Desh illustrates the importance of examining a timed sequence of events for anomaly prediction over a single log message, since a message may be fatal in one but benign in another context. Thus, log-severity levels are inaccurate indicators for failure prediction. Furthermore, this work finds that lead times of similar failure classes are comparable with insignificant variability while the lead times tend to vary over diverse classes of a specific system.

3. Chapter 4 contributes a fast node failure predictor called *Aarohi*. It is designed to be generic and scalable to effectively infer failures online through context free grammar-based rapid event analysis. Aarohi obtains more than 3 minutes effective lead times to failures with an average of 0.31 msecs prediction time for a chain length of 18. The overall improvement in inference speedup obtained w.r.t. the existing state-of-the-art is over a factor of 27.4x. This compiler-based approach depicts the merits of leveraging an efficient parser suitable for real-time streaming logs.

4. Chapter 5 explores holistic root cause diagnosis of node failures using a measurement-driven approach on contemporary system logs that can help vendors and system administrators support exascale resilience. Empirical evidence suggests that environmental influence is not strongly correlated with node failures in terms of the root cause. Though hardware and software

faults trigger failures, the underlying root cause often lies in the application malfunctioning. Lead time enhancements are feasible for nodes showing fail slow characteristics. This study excavates such helpful insights, which can facilitate better failure handling in production systems.

1.6 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents a node failure prediction framework using topic modeling, an NLP-based approach for supercomputing systems. Chapter 3 contributes a deep learning-based mechanism to predict lead times to potential node failures in HPC systems. Chapter 4 proposes a compiler-based failure predictor to infer imminent failures with enhanced speedup. Chapter 5 describes root cause analysis of node failures adopting a holistic approach considering external and internal event correlations. Chapter 6 discusses the deployment efforts of the developed failure prediction solutions and concludes with potential future work in the context of proactive fault tolerance in large-scale computing systems.

CHAPTER

2

PREDICTING WHICH NODE WILL FAIL WHEN ON SUPERCOMPUTERS?

2.1 Introduction

Significant efforts have been made to improve the resilience of HPC systems in recent times. Existing health check monitors and techniques such as root cause diagnosis and failure detection use diverse log sources to combat failures. However, they still fall short of strong means to handle node failures *proactively* in complex, large scale computing systems. First, supercomputing systems are constantly changing due to novel architectures, design, upgraded applications and logging mechanisms. Prior techniques of automated fault diagnosis do not suffice for the evolving changes [Bra15].

Second, existing HPC infrastructures with their increasing component count required for exascale ($\approx 10^6$ nodes) make accurate fault prediction hard. Aborted jobs due to node failures inflict significant energy costs [Mar15b]. More than 20% of the compute capacity is wasted in failures and recovery, as reported by DARPA [Eln08]. With increasing number of nodes, the mean time between failures (MTBF) reduces for a node, making fault identification and resolution even more difficult. Increasing complexity in emerging next generation systems obviates the need for adaptive fault aware solutions to address this critical reliability challenge.

To address this challenge, we present a fault-tolerant solution to pin-point potential node failures in HPC systems. Our study on Cray system data with an automated machine learning technique suggests that careful time series analysis of log phrases can be used to predict node failures. Recovery

techniques such as checkpoint/restart and redundancy/replication incur additional costs [Ell12]. Our methodology to identify *which nodes* are likely to fail (location information) prior to actual fail-stop behavior can reduce the overhead of failure recovery.

HPC resilience has been investigated extensively. Prior work has characterized system logs in the context of fault detection and prediction. While most work focused on anomaly detection in BlueGene systems [Fu14; Yu12; Ber14; Lan10a], time sensitive failure prediction in the context of Cray supercomputers with their lower-level Linux-style raw logs has not been researched exhaustively yet. As of November 2017, 29% of the top 100 and 40% of the top 10 supercomputers (e.g., Titan, Cori, Trinity) [Nov] were Cray machines. It is important to investigate Cray machines more closely to explore techniques that increase reliability. This paper discusses the inherent challenges of Cray systems and proposes a mechanism to predict node failures.

Motivation

From log data analysis to root cause diagnosis across various levels (hardware, system, application) researchers have studied failure manifestations in HPC systems and devised ways to improve recall rates [Gai13]. In spite of such a large body of work on resilience, further investigation is required for the following reasons:

1. Existing work performs prediction and diagnosis without sufficient emphasis on lead time requirements. Pin-pointing which nodes will fail well ahead in time to proactively counter performance disruptions still remains a challenge. Optimal learning window interval selection and determining appropriate lead times are important considerations for successful prediction of node failures.

2. Most prior studies [Yu12; Zhe12; FX07] use the same training data for future predictions over a long time frame. As hinted by Gainaru et al. [Gai13], correlations determined off-line are dynamically adapted, having limitations when using a short training set for a long future time window. This limitation makes prediction impractical on real production systems. Investigations of dynamic learning and scalable online prediction techniques are required to improve prediction efficiency.

3. There exist unpredictable failures [Gai13], and we need to understand HPC Linux/Cray systems in finer detail to determine where correlation extraction from system logs is hard.

4. Validation of predicted faults is commonly done through comparison with event logs (sometimes with identical training and testing data) or by consulting system administrators. Relying on manual human expertise or system administrator's knowledge is difficult at times. It is important to explore if alternate validation schemes could be formulated for good prediction accuracy.

We focus on the 1st and 3rd aspects in this paper. This work is a step forward to address the above mentioned hurdles in locating node failures in HPC systems.

Contributions

This paper shows a novel way to extract failure messages indicative of compute node failures for Cray systems. First, we provide an analysis of Cray system logs and job logs and show how failure prediction in such systems poses additional challenges compared to systems such as BlueGene.

Second, we discuss what node failure exactly means in the context of Cray systems, what traits govern normal shutdowns and abnormal reboots and how we can avoid trivial pitfalls in node failure detection. We provide frequency estimates of compute and service node failures highlighting their potential consequences on systems and user applications.

Finally, we propose a novel prediction scheme, TBP (time-based phrase), to extract relevant log messages indicative of node failures from noisy data. This scheme relies on phrase likelihood estimation considering continuous time-series data to elicit out useful messages. These events help forecast future failures with lead times ranging from 20 secs to 2 minutes.

2.2 Background

Let us provide a brief overview of the system logs studied highlighting the main components of such logs used for analysis. Table 2.1 summarizes the system and job logs collected from three contemporary systems, namely: SC1, SC2 and SC3. The timespan of SC1 and SC2 logs exceeds a year, SC3 logs cover less than a year. Size refers to the log data size and scale indicates the cluster size in terms of the number of compute and service nodes. Most of the systems belong to the Cray XC series that have been widely deployed and typically run more than 1,400,000 jobs/year.

Table 2.1 System Details

System	Duration	Size	Scale	Type
SC1	14 months	573GB	5600 nodes	Cray XC30
SC2	18 months	450GB	6400 nodes	Cray XE6
SC3	8 months	39GB	2100 nodes	Cray XC40

Cray System Architecture

Figure 2.1 shows a high-level overview of a Cray system. A job scheduler distributes user jobs on the allocated compute nodes. Production jobs are executed on the compute nodes and external clients access the cluster through the login nodes. The parallel file system (e.g. *Lustre*) and a network server (e.g., *Aries* implementation of the generic network interface (GNI)) communicate with the service

nodes and compute nodes. The **S**ystem **M**anagement **W**orkstation (SMW) administers and logs various cluster components and monitors resource usage. The **S**ervice **D**ata**b**ase Node (SDB) stores information of all the service nodes. The boot node manages the shared file system with service nodes. Login, boot and SDB are some of the service nodes of the system in addition to syslog, I/O and networking nodes. The **A**pplication **L**evel **P**lacement **S**cheduler (ALPS) processes such as `aprun`, `apbridge`, `apshed`, `apinit` etc. are responsible for user application submission and monitoring and run on both service and compute nodes.

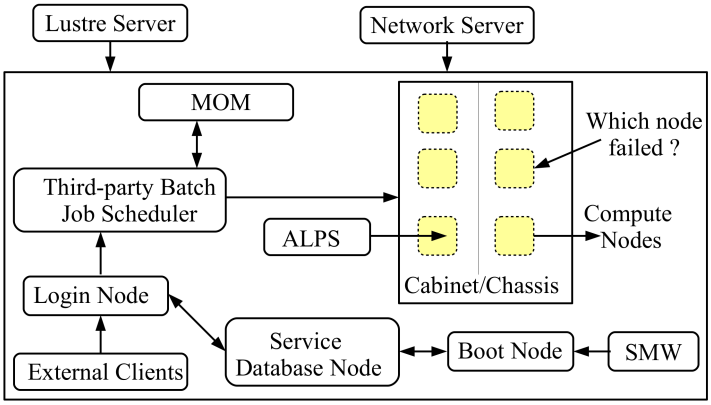

Figure 2.1 Overview of a Cray System

Table 2.2 indicates the major log sources. From these archived logs, we consulted p0-directories that contain comprehensive logs of the entire system with information pertaining to the internals of compute nodes including system and environment data. We use the acronym *id* to refer to node or job identifiers. The files (console/message) in these directories contain timestamped event logs including node ids (cX-cYcCsSnN) per line. We track the node ids per log line (event) while correlating and integrating data. Logs can contain extended node ids (nids), which can be converted to the cX-cYcCsSnN format to identify a node's location such as blade(S), chassis(C), cabinet(XY) and the node (N). Additional references to boot messages and job logs aid the prediction scheme since they provide the status of nodes and jobs over time.

Table 2.2 Data Details

Source	**Content**
p0 directory	Internals of compute nodes
Boot Manager	Boot node messages
Log System	`rsyslog` messages
Power/State Logs	Component power and state information
Event Messages	Event router records
SMW Messages	System Management Workstation messages
HSN Stats	High Speed Network Interconnect logs
Job Logs	Batch job/application scheduler messages

We found that noisy information pertaining to power management, SMW, the log system, network interconnect logs, events, and the state manager, are not very useful for node failures and, hence, have not been considered. These lower-level logs did not reveal significant textual indicators related to node failures. Manuals/system administrators help to understand logs, but even then post-mortem analysis for unsupervised information extraction remains non-trivial from such data.

Technical Challenges

The challenges of data diversity, system complexity, and overwhelming logging volume have been investigated in the context of failure detection [Lan10b; Zhe12]. Furthermore, Cray systems specifically have the following additional challenges:

1. BlueGene systems have Node Card, Service Card, Link Card and Clock Card components, each of which provide current, voltage, temperature data etc. In Crays, failure needs to be discovered by integrating a distributed set of events in space and time, coming from different system components, some of which are replicated. Feature identification even before dimensionality reduction is hard.

2. Binary or numeric values (normalized or mapped) of features as considered in prior work [Yu11; Yu12] do not suffice. Simple absence or presence of an event is not enough. Which event, when did it happen, is it related to the node under consideration and similar factors are also of importance here.

3. RAS logs in BlueGene systems contain fatal and warning flags indicating the severity levels with the log messages [Zhe11; Zhe09; Hac09] aiding researchers to segregate between failures and benign events. Critical and warning flags are present in Crays as well pertaining to certain components such as `netwatch`, `pcimon` etc. However, direct classification of log messages based on occasionally appearing flags is ineffective [Zhe09] for long-term time sensitive data since several non-critical messages could be a better indicator of failures over time. Besides, seemingly benign events in one context may lead to fatal events in another, which means BlueGene logs may result in shorter lead times. Hence, we consider phrases irrespective of flags/severity-levels.

4. There exists unsteadiness in timestamps between service nodes, job schedulers (Slurm/Torque) and compute nodes making time-based correlation non-trivial. Time-series analysis handles this and allows us to study lead time sensitivity.

Node Failure

The boot log reveals several clustered node failures caused by problems ranging from communication failures, network-interconnect and application-based errors, resource-contention, file system or hardware errors. Further study revealed certain patterns in the context of node failures. If nodes shut down in bulk (multiple blades) within a few seconds, the root cause tends to be maintenance. Such shutdowns often are massive (e.g., 98 or 126 nodes going down at once). But even in case of single compute node failures, the culprit could be either external or internal events (see Table 2.3).

Table 2.3 Node Shutdown Events

Internal Failures	**External Failures**	**Normal Shutdowns**
Application Bugs	Blade or Cabinet Controller Issues	Massive shutdown
Node System Bugs	File system or Network Server Issues	Maintenance Reboots
Node Hardware Issues	Router or other Hardware Issues	Periodic Node Reboots

- Internal Events are compute node specific events either caused by applications running on that node or hardware/software problems related to memory, kernel etc. pertaining to that node.
- External Events are events that occur outside a specific compute node such as Lustre server-related errors, a Link Control Block (LCB) failure, or a network interconnect failure causing multiple chassis to shutdown. Cabinet- or blade-controller problems can also manifest as massive node failures.

***Node Failure Definition*:** Not every node unavailability indicates an anomaly. Power outages, maintenance and deliberate shutdowns have been eliminated in our study. Cases related to the internal and external events (discussed above) are considered as node failures since they manifest as anomalies.

Periodic service reboots are common in Cray nodes. Nodes are rebooted several times before they come up as part of regular maintenance. Such spurious cases are not counted as node failures since these are not caused by any faults in the system. Counts of node failures do not consider unique nodes since a specific node can fail multiple times at different timestamps. Unresponsive nodes, stress testing, and changing power cooling conditions manifest in log messages and have been considered in our study; a failed heartbeat indicates failure with unknown root cause (network/OS/hardware failures resulting in lost connection) and are indistinguishable from anomalous node failures in terms of manifestation. We have timestamped logs, and based on the time and scale of shutdowns, we segregate failures. In many instances, service node failures impact compute node failures (see Sec. 2.5).

It should be noted that this work aims to *correctly predict node failures*. The goal is *not to identify the exact root cause* that provoked fault manifestation. In other words, the methodology is to identify

Table 2.4 Examples of Node Failures

bit flips caused failure	**hardware caused failure**	**app. caused failure**
4.25.30 pm LCB on and Ready	8.44.12 pm Hardware Overflow Error	2:44:49 am Matlab invoked oom killer
4.30.33 pm Micropacket CRC Error Messages	8.46.09 pm Lnet errors Recvd down event	2:54:14 am Out of memory: Kill process
4.35.29 pm Network chip failed due to too many soft errors	8.47.45 pm Lustre Errors Binary changed	2:58:14 am Killed process
4.36.42 pm Aries LCB operating badly, will be shutdown	8.48.06 pm Bad RX packet error	2:59:40 am Kernel panic not syncing:
4.37.31 pm Failed LCB components	8.52.37 pm Out of memory/Killed processes	3:00:00 am page_fault+0x1f/0x30
4.37.39 pm 2 nodes unavailable	8.55.13 pm Node unavailable	3:00:03 am Node unavailable
Failed within 12 min.	Failed within 11 min.	Failed within 16 min.

chains of time-based events, which eventually led to a failed node. The actual cause and the location of a root cause may not be indicated by our prediction methodology. E.g., in Table 2.4 column 1 (bit flips), after repeated soft errors a Link Control Block (LCB) went down. The cause of CRC error messages may range from hardware (link/NIC) problems to silent data corruption, but this cause is of no concern for our analysis.

***Illustrative Examples*:** Table 2.4 shows three cases of node failures. The first column shows a case of a failure caused by soft errors (bit flips) detected by the LCB. Due to too many errors the LCB went down after which 2 nodes of a blade went down within 12 minutes. The second column shows the case of a hardware error caused by the network interconnect. This triggered Lnet and Lustre errors followed by memory problems, and the node went down within 11 minutes. The third column shows an application (Matlab) failure caused by excessive memory allocation. This killed several tasks, followed by a kernel panic, failing this specific node. In these cases, if prediction happens a few minutes before the actual shutdown, some proactive measures can be taken.

2.3 Predictor Design

We model our method after unstructured phrase mining approaches applied in the context of Natural Language Processing (NLP) research. Our study shows that Topics over Time (TOT) [WM06] (an LDA-style [Ble03] unsupervised topic model) based learning can help pin-point faulty nodes. Identifying rare compute node failures by relating job ids, which are *re-run* several times on the faulty nodes, can help take subsequent proactive resilience actions.

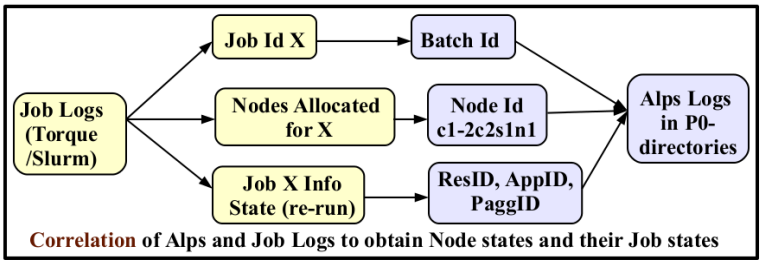

Figure 2.2 Correlation with Job Logs

Job Logs

Figure 2.2 demonstrates how jobs scheduled on allocated nodes are identified in the ALPS logs inside the p0-directories. The job server such as Torque provides information about job id, the allocated nodes for that job and the status. These can be referenced in the ALPS logs through a mapping conversion. The job ids map to batch ids. The node naming convention is derived from the boot logs, which provide the required id for system logs, and the job information is referenced using `res`, `app` and `pagg` like tags added by ALPS. The timestamp is considered for correlation by checking the amount of slack in logs from different sources (job, server/compute node). If it is within a given threshold (we observed 15 seconds to be appropriate), it is considered. This did not cause any correlation errors since the ids were matched correctly over time. In our experience, missing data in log archives and MOM (machine-oriented miniserver) node failures (see Figure 5.1) affecting job data complicates this correlation across the available logs. This arises due to stopped daemons/logging, with or without upgrades. Nonetheless, this presents a legitimate way to track the behavior of jobs running on nodes. However, upgrades and changes in job schedulers (ALPS-Torque, ALPS-Slurm) can create inconsistencies in the available data. But this does not hamper the prediction mechanism since the correlation has been confirmed by prior offline manual validation with system administrators, and only complete and consistent data is used for evaluation.

Data Integration

After successful correlation, a text document with timestamps, node ids and filtered log messages is formed to generate a viable input for the statistical probabilistic model (Figure 2.3). We do not use any particular environmental data, such as **S**ystem **E**nvironment **D**ata **C**ollection (SEDC) logs, since such information predominantly does not aid in phrase extraction. Temperature and voltage values may indicate an anomaly but our work intends to discover salient phrases providing symptoms of abnormalities.

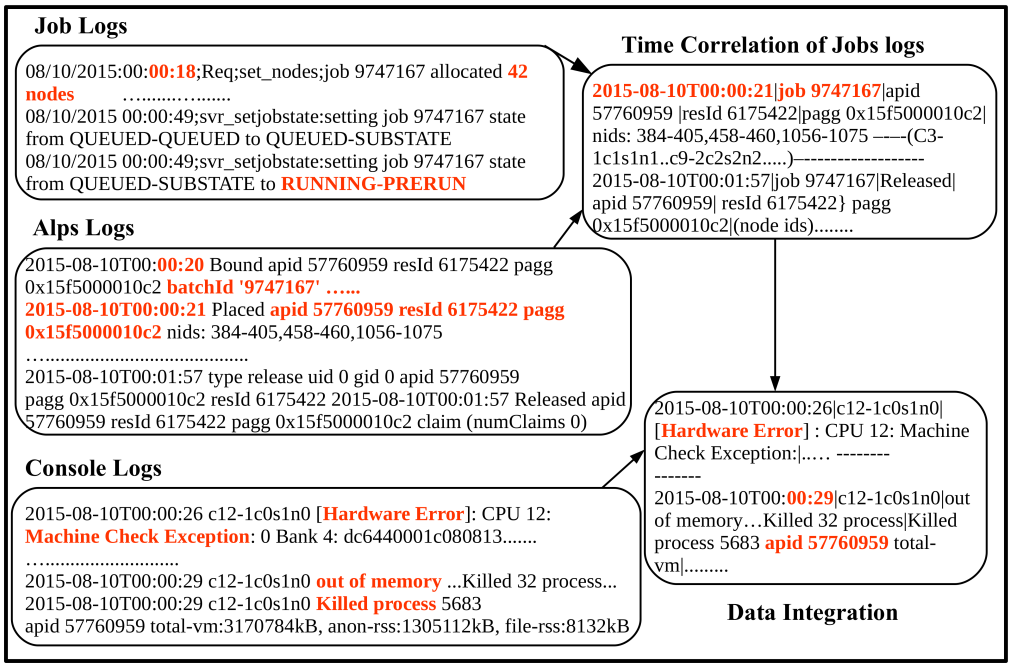

Figure 2.3 Time Correlation and Data Integration

Phrase Likelihood Estimation

One primary tendency observed in the available logs is the recurrence of failure messages over time and changing patterns that continuously evolve over time. A phrase is defined as an event log message corresponding to a specific node at a certain timestamp. This trend of time-based evolution prompted us to leverage a well known machine learning technique called Latent Dirichlit Allocation (LDA) [Ble03], a probabilistic model on discrete data. One important factor for log analysis targeting prediction is time. Hence, continuous time series-based evolution is required to extract patterns from the integrated document. To address this, we utilize the **T**opics **o**ver **T**ime (TOT) [WM06] algorithm to identify the top `N` topics (i.e., phrases or log messages) over a period of time and track how the topics change over time. TOT employs Gibbs Sampling and is useful for dynamic co-occurrence of patterns when an upsurge and downfall of phrases exists over time. Since TOT models time in conjunction with frequency of phrases, which is analogous to the case in temporal logs, TOT is a good choice for our study in contrast to other existing competitive approaches such as [Nak11; Zhe09; Hac09; Zhe10; Yu16].

On similar grounds, discrete-time **D**ynamic **T**opic **M**odeling (dDTM) [BL06] is inappropriate since there is a significant variation in occurrence of events (at the granularity of milliseconds) so that several chunks of logs cannot be clustered under a single time instance. Coarse-level time discretization fails to capture short-term time variations. The ability to identify a known delta time difference between two known messages is critical here. The main idea is that every phrase is

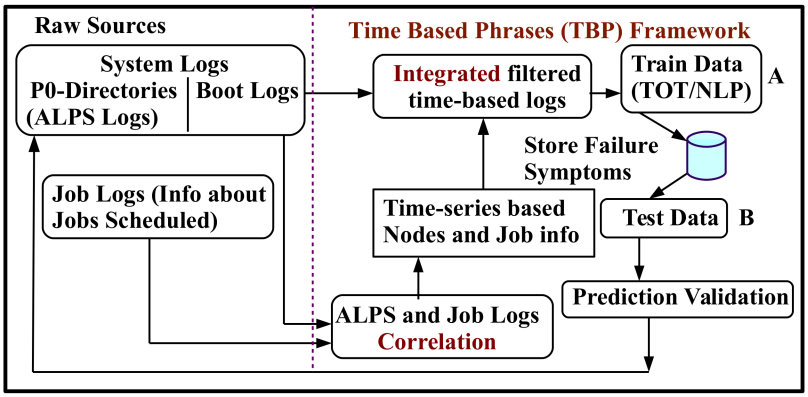

Figure 2.4 TBP Framework

assigned a topic. Multiple phrases can be assigned the same topic (see Table 2.5). We have finite number of topics for an integrated document. During the training phase, TOT learns top `N` topics referring to phrases. TBP forms sequences of phrases that correspond to failures in the past referring to the data. We use them to forecast future failures when those phrases reappear in the test data. Once we know the significant phrases, we denormalize their timestamps and refer to the nodes associated with them to identify nodes subject to future failures. We name our prediction mechanism "*time-based phrases*" (TBP).

Table 2.5 Topic Assignment

#	Event Phrase	Topic
1	Lnet: waiting for hardware..	Lnet
2	Lnet: Quiesce start..	Lnet
3	Debug NMI detected	NMI
4	DVS: uwrite2: returning error	DVSBug
5	Kernel panic/not syncing/Fatal Machine check	Panic
6	MCE threshold of fff..	MCE

2.4 TBP Framework

Figure 2.4 shows the work-flow diagram of our approach. After job and system data correlation and formation of an integrated document, we obtain the top ranked phrases over time and determine the nodes corresponding to them (Box A). Then, based on the time series, TBP obtains a chain of messages leading to the node failures. We use them on test data to compute recall rates and lead times (Box B). Let us clarify that we do not want to determine the distribution of node failures over time/space or failure characteristics on the susceptibility of a specific node to future failures [Gup15; ES13; Tiw14].

Time Correlation

Figure 2.3 illustrates the idea of time correlation. The job log indicates that at `00:18` a job with id `9747167` is allocated 42 nodes. This job is correlated with its corresponding ALPS message `apid 57760959`, `resId 6175422`, `pagg 0x15f5000010c2` logged at `00:20`, just 2 seconds later. Since 2 seconds are within the threshold (15 seconds), these become correlated. Simultaneously, we obtain the ids of nodes allocated to this job by converting from `nid` (extended node id) to the `cX-cYcCsSnN` format. These node ids and job ids can be time correlated to the rest of the logs (e.g., console logs) in a similar fashion as shown in Figure 2.3.

TBP Learning

TBP uses TOT to learn the failure chains from the training data. Topic assignment assigns a relevant topic to every phrase pertaining to that topic as shown in Table 2.5. We have used more than 100 topics in our training data. Multiple phrases can be assigned the same topic if the content of that phrase is not anomalous or if they have similar system event context (e.g., 1 & 2). A distinct phrase can also be assigned a topic if it indicates a unique event (e.g., 3 & 5). TOT chooses top `N` topics (i.e., phrases) as shown in Figure 2.5.

Let us explain how TOT picks the top `N` topics. The basic idea is that phrases chosen are localized in time. As the distribution of phrases changes over a continuous time frame, top phrases evolve since the phrase co-occurrence changes [WM06]. The 8 topics (firmware bug, ec_node_info, Lustre, DVS, LNet, hwerr, apic_timer_irqs and krsip) shown in the illustration Figure 2.5 (box 1) pertain to phrases containing that topic. TBP uses the top picked phrases over time to formulate the failure chains. Topic-based training helps to extract only the significant phrases relevant to failures (boxes 2+3).

We varied the value of `N` in different data sets to ensure that we are not missing relevant phrases. In our experiments, `N` ranges from 50 to 80 based on the amount of data considered. We manually inspected the output of TOT while choosing a subset of `N` considering time and space constraints. To clarify, `N` has been varied but it is impractical and inefficient to inspect too large of an `N` value. TBP has chosen a smaller subset (smaller `N`) at times to effectively collect indicative phrases.

Node Failure Prediction

How does TBP predict node failures based on top `N`? Figure 2.5 illustrates the key idea of prediction. T denotes timestamps, N stands for node ids, and P for phrase ids (for brevity we omitted job ids in the figure). The integrated document (Figure 2.3) is trained using TOT. We know the terminal node shutdown messages from sysadmins (e.g., `System halted, cb_node_unavailable`, see Table 2.10, last 10 phrases). TBP forms failure chains linking phrases among the top `N` with time-stamps and node-ids referring to the data. From the filtered phrases (box 3), TBP obtains the node

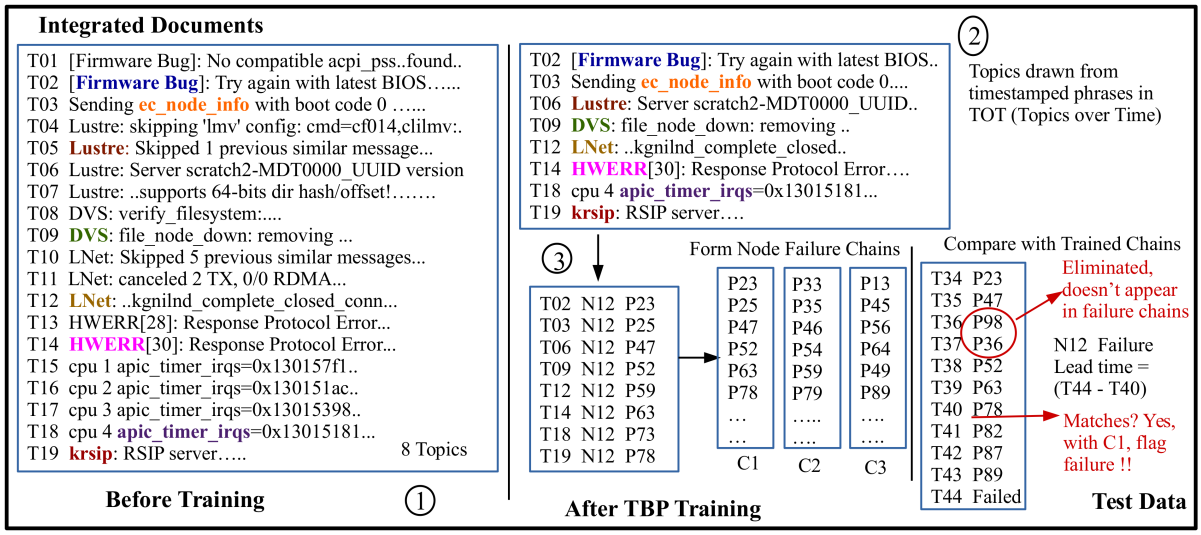

Figure 2.5 TBP Prediction: Topic Modeling for Node Failure Prediction

failure chains as shown by C1, C2 and C3. During testing, no top phrases are generated. TBP compares the incoming phrases with those in the failure chains. If chains with at least 50% similarity in log messages are formed, the corresponding node is likely to fail in the future. E.g., p98 and p36 are rejected since these were not seen in the training data, and the remaining phrases match with C1 till P78. In the test data, before N12 fails, we compare and predict it to be a potential failure. From N12 (i.e. cX-cYcCsSnN) we can easily derive the node's exact location in the blade(S), chassis(C) and the cabinet(XY) to potentially trigger proactive resilience actions before it fails at a future timestamp (T44). About 1 month's data is used for training. The test data is comprised of a moving time window of 3 days to 1 week for generating better lead times.

2.5 Experimental Evaluation

We have implemented a prototype of the TBP predictor using the factorie [McC09] library and python. We use TBP on 3 datasets from SC1, SC2 and SC3 supercomputers (Table 2.1) for evaluation. TBP achieves as high as 86% recall rates with acceptable lead times in predictive fault localization of nodes. Our time-series based phrase mining approach works for any HPC system with "text-based" logs. The evaluation focuses on Cray logs, but the approach is generic to modern HPC systems (see discussion at the end of this section).

Average Node Failure Estimate

Figure 2.6 shows an estimate of service and compute node failures over 4 months of the data in Table 2.1 for SC1, SC2 and SC3. SC2 had the highest overall number of failures but SC3 had a slightly higher frequency of failures (failures/month). The compute node failure count is higher than the

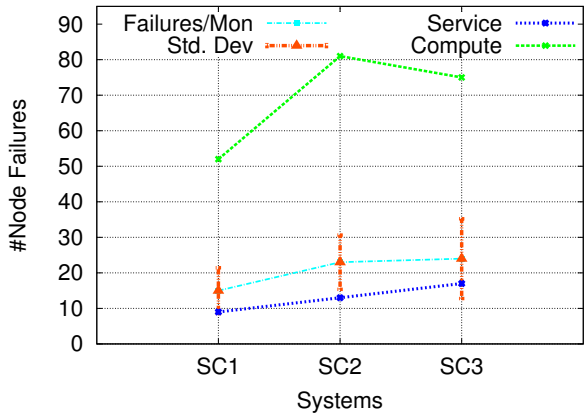

Figure 2.6 Estimate of Node Failures

number of service node failures. Our observation indicates that both service and compute node failures are randomly distributed over time. Hence, this could be misleading in terms of estimating failure rates over longer periods of time. E.g., over a period of 3 weeks, SC3 encountered 10 service node failures, but the subsequent time periods of 9 days and 2 weeks had 0 and 1 service node failures, respectively. There exists sufficient variability of failures over time across the three systems. This explains the standard deviation of the frequency ranging from ±6 to ±11. Nonetheless, this gives us an idea of the number of failures encountered in such systems.

One might argue, if node failure events are relatively rare compared to the overall scale of anomalies in the system, why do we need to predict them and take proactive actions? In this regard, we have summarized two main takeaways:

1. The number of compute node failures increases dramatically with an increase in service node failures. On multiple occasions, time periods with high service node failures have affected a large number of compute nodes around the same time due to external failures. We did not investigate further the exact root cause of each failed compute node, but in addition to maintenance related shutdowns, controlled service node failures can definitely prevent compute node failures, which benefits jobs of users running on them.

2. Rescheduling a job after a node failure delays the overall job execution time and utilizes additional resources. A single job, on average, is generally allocated to many compute nodes (up to tens of thousands for peta-/exascale capability jobs). If blades fail repeatedly, the effect is logged as independent job failures, which are rescheduled. Past work has observed that a significant fraction of applications fail due to system problems apart from user-related errors [Mar15b]. If user application disruptions are a concern and system-wide outages are to be reduced, node failure prediction is of paramount importance.

Table 2.6 Phrase Extraction

#	Phrases	Probability
1	Ensure file system is mounted on the server and then restart DVS	0.0214
2	LNET: waiting for hardware quiesce flag to clear	0.0145
3	nscd: nss_ldap: failed to bind to LDAP server	0.0167
4	LustreError: *:*:.....unable to mount	0.0298
5	startproc: nss_ldap: failed to bind/reconnecting to LDAP server	0.0302
6	Error: No data from cname	0.0178
7	Lustre: skipped * previous similar messages	0.0119
8	Lustre:*:*:vvp_io.c:*: vvp_io_fault_start	0.0176
9	reconnected to LDAP server	0.0247
10	Lnet: * Dropping PUT from *	0.0218

Phrase Distribution

Table 2.6 shows a sample snippet of 10 phrases over a time window of 2 weeks with their probability distributions depicting a compute node failure case caused by Lustre server-based errors. The table shows that Lustre, Lnet and filesystem-related messages are produced by TBP with higher probability than other system messages. These phrases are related to filesystem problems impacting the node. We do not care about individual probabilities as long as the top `n` phrases are of interest in the context of node failures.

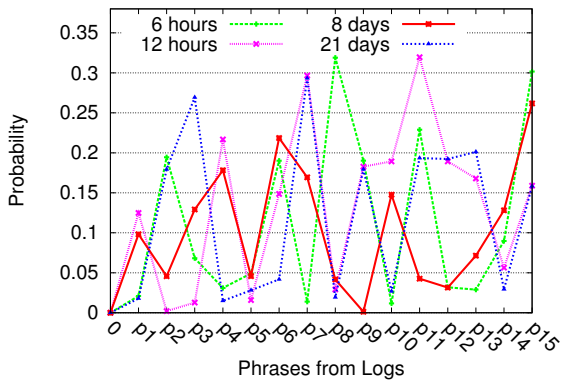

Figure 2.7 Phrase Likelihood

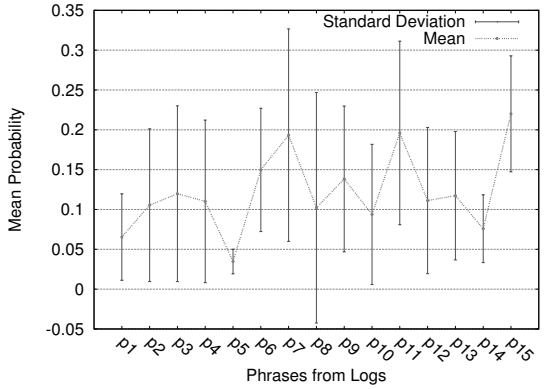

Figure 2.8 Mean and Std. Deviation

Table 2.7 Recurring Phrases

No.	Phrases
P1	crms_wait_for_linux_boot: nodelist: *
P2	Lnet: Quiesce start: hardware quiesce
P3	Wait4Boot: JUMP:KernelStart *
P4	krsip:RSIP server * not responding
P5	startproc: nss_ldap: failed to bind
P6	checking on pid *
P7	LustreError: *:*:.....can't find the device name
P8	GNII_SMSG_SEND + *
P9	No .....bios_settings file found
P10	Lnet: Added LNI *
P11	DVS: file_node_down: removing *
P12	Lustre: skipped * previous similar messages
P13	Lnet: skipped
P14	<node_health:*> RESID* xtnhc FAILURES
P15	Bad RX packet error

Table 2.8 Evaluation Metrics

Metric	Formula & Implication
Recall	TP/(TP+FN) # Node failures, TBP *correctly* predicted
Precision	TP/(TP+FP) # Total node failures, TBP predicted
FP Rate	FP/(FP+TN) # False Positive Rate
True Positive (TP)	# Actual node failures, TBP successfully predicted
True Negative (TN)	# Nodes actually didn't fail, TBP did not predict as failure
False Positive (FP)	# Nodes actually didn't fail, TBP predicted as failure
False Negative (FN)	# Actual node failures, TBP failed to predict

Observation 1: *Significant phrase variation exists over short time intervals. HPC logs indicate event changes at a high frequency, which calls for continuous-time statistical models that can handle this variability prior to identifying phrase relevance for node resilience.*

Figure 2.7 and Table 2.7 illustrate the variation in probability of occurrence for the same 15 phrases over four continuous time intervals (6 hours, 12 hours, 8 days and 21 days) for SC3 data. The 4 disjoint time frames have been selected from over 3 months of data and illustrate the lack of a uniform distribution. Table 2.7 depicts frequently occurring system log messages pertaining to Lustre, Lnet etc. Figure 2.7 shows the fluctuations of phrase probabilities over different time intervals. We provide a quantitative analysis of such variations to signify the lack of discernible features. Pattern extraction with such non-uniform distribution of unstructured log messages but without clearly flagged errors is hard.

Figure 2.8 quantifies this variance through mean (curve) and standard deviation (indicated by the error bars). The standard deviation for most phrases is high (e.g., for `p7`, `p8` and `p11`), except for `p5`. As an example, message `p7` is emitted by Lustre for an unmounted device with a higher probability distribution in the 2nd (12 hours) and 4th (21 days) time intervals, but the device was mounted and the message occurred less in the 1st (6 hours) and 3rd (8 days) time intervals. Similarly, `p11` indicates a Data Virtualization Service (DVS) server failure, which is removed from the available mountpoint list. The failover event occurred with a different magnitude over the intervals for multiple nodes. Message `p5` related to the LDAP server had less deviation since the connection to the LDAP server

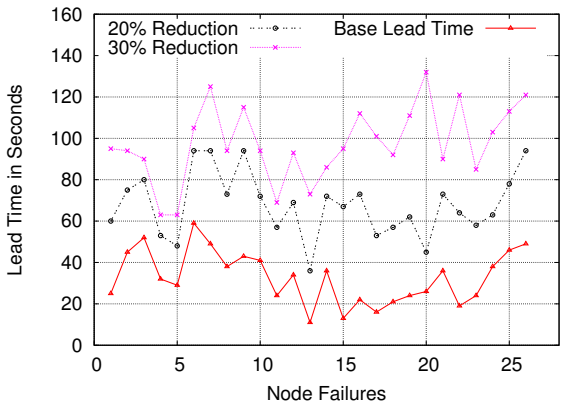

Figure 2.9 Sensitivity of Lead Times

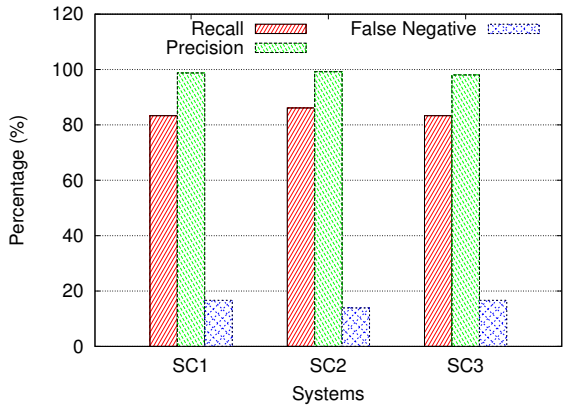

Figure 2.10 Recall/Precision/FNR Rates

was successful after a few attempts in those time intervals. Discrete-time statistical methods [Hac09] are ineffective under such variability.

Prediction and Lead Time Analysis

Phrases detected during the learning phase of around 4 weeks help in node failure prediction on new test data. TBP checks the test data for phrase similarity relative to the training data (Figure 2.5). If similar, we obtain the node ids corresponding to those phrases and compute recall rates. N-fold cross validation is less effective for time-series data. Our train and test data split respects temporal event ordering, (lower-order time-series training, higher-order testing). TBP uses the standard evaluation metrics of Recall and Precision to estimate prediction efficacy. Table 2.8 enumerates their formulae and the implications in the context of node failure prediction. The recall rate is defined as the fraction of node failures that are correctly predicted by TBP, and precision rate as the total fraction of node failures predicted by TBP (need not be correct). The FP rate is the false alarm rate, the ratio of actual failures missed by TBP. Validation is performed by manually checking the logs with the timestamps of actual failures. In the test phase, we consider 3 days' to 1 week's data, and compare the phrases with the obtained trained data. We re-train (4 weeks data) and move the test data time interval with shifts of 1 week to predict impeding failures and procure lead times. The base lead times in Figure 2.9 considering the trained failure chains are in the range of 20 to 60 seconds without any phrase reduction. We have optimized TBP's lead time sensitivity further through phrase pruning.

Observation 2: *TBP achieves high recall and precision with a modest number of false negatives and as high as 1 minute base lead time.*

Figure 2.10 shows the recall, precision and false negative rates. We observe a precision rate of

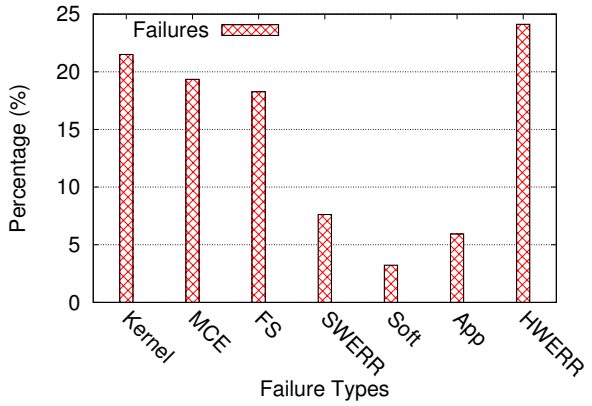

Figure 2.11 Failure Categories

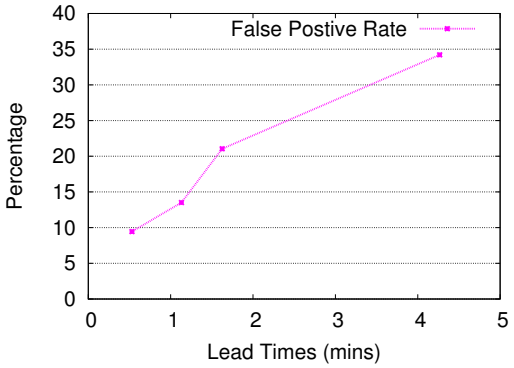

Figure 2.12 Lead Times+False Positives

Table 2.9 Major Failure Categories

#	Failure Type	Class
1	Trap invalid opcode/Segfaults, File System Bugs	Software
2	cb_hw_error: failed_component, Firmware Bugs, NMI Faults	Hardware
3	[hwerr]: Machine Check Exception (MCEs), Memory faults	Hardware
4	RCU CPU Stalls/Hangs, Kernel Panic/Fatal Exception, Stack Trace	Software
5	Bit/Packet Errors (dla_overflow error. Msg protocol error)	Soft Errors
6	Machine Check Events (Node heartbeat faults)	Hardware
7	Job Server/Task related errors	Application

up to 99%, which indicates a low rate of false positives. This indicates that the trained failure chains were indeed indicative of node failures. The recall rates are 86% or lower. TBP aims not to miss actual node failures irrespective of the causes and correlations between them. Even if correlated failures are removed, precision and recall exceed 80%. Across all the 3 systems the false negative rate is as high as 16.66%. This is partly because of the new phrases seen while testing and partly due to the corner cases where phrase extraction is difficult. SC2 has a relatively low false negative rate since failure events learned during training were mostly seen during testing with similar failure types (e.g., networking problems). Figure 2.10 rates correspond to the base lead time without any phrase pruning.

Observation 3: *Hardware errors, MCEs and kernel panics often cause node failures. Minor causes are failed components in the network interconnect, bit errors, filesystem caused errors and application based errors.*

Figure 2.11 shows the proportion of different types of failures observed in the data, namely Kernel panic, MCE (Machine Check Exceptions), FS (filesystem errors), SWERR (Software errors), Soft

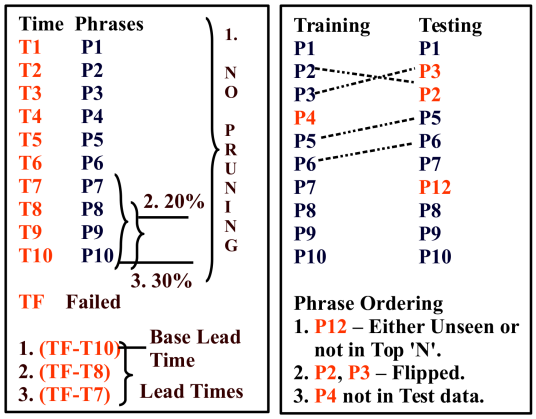

Figure 2.13 Phrase Reduction and Order

(bit/packet/protocol related soft errors), App (application errors), and HWERR (hardware node heartbeat faults), respectively. Table 2.9 lists the major classes of anomalies manifesting in node disruption in such systems. While 15 to 20% of the nodes fail due to H/W errors, MCEs and kernel panics, bit errors and application caused errors are minor contributors. Our observations conform to failures reported by Gupta et al. [Gup17].

We predict node failures on average a minute in advance; this is an improvement over scenarios where nodes fail within 20-30 seconds of the occurrence of the first reported event that can be linked to a later failure by system administrators. However, certain failures such as NMI faults cause instant failures without a chance to communicate anymore. It is impossible to take proactive actions in those cases. Moreover, many indicative messages occur just prior to the failure.

Lead Time Improvements

Starting from the last phrase considered from prior learning, we prune backwards by omitting additional phrases to increase lead times and assess the impact on false positives. Figure 2.13 (left) depicts backward pruning as an example. Suppose 10 phrases are considered named P1, P2...P10 with increasing timestamps T1, T2,..T10. A node failure occurred at TF. When 20% phrases are pruned, the last 2 phrases with ordered timestamps are removed from consideration, i.e., those phrases are not checked in the test data to qualify as a failure chain. The percentage of reduction is increased to gain longer lead times. The earlier correct failures are flagged, the higher the lead time will be. Lead times improve from (TF-T10) to (TF-T7) with 30% phrase reduction as shown in Figure 2.13 (left). In reality, the number of phrases considered are higher than 10 (30-50). Figure 2.13 (right) depicts cases of phrase mismatches. Observed phrases in the test data may not be in the same order as the trained phrase set under consideration. E.g., P2 & P3 or unseen phrases (e.g., P12) may be present or a phrase seen in the trained chain earlier is missing (e.g., P4) in the test data

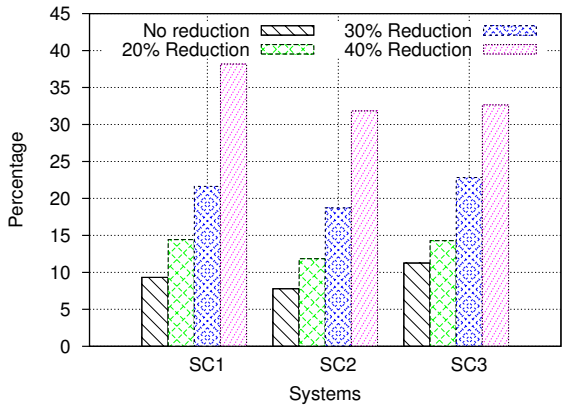

Figure 2.14 False Positive Rate

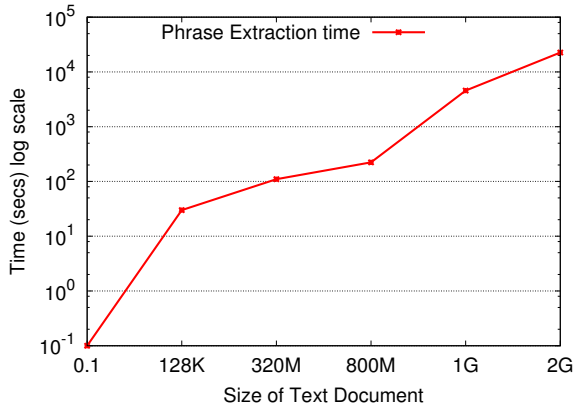

Figure 2.15 TBP Scalability

interval. In such cases, TBP ensures a similarity of 50% or higher and otherwise discards phrases as unmatched. Figure 2.9 illustrates the increased lead times with 20% and 30% phrase reduction for each of the 26 node failures across different machines. (The rest of the failures have similar lead times). With phrase pruning lead times increase. Corresponding to a specific average lead time, we calculate the false positive rate from the data set.

Observation 4: *In general, lead times are as high as 2 minutes, with most of them higher than 1 minute after a 20% reduction, not exceeding a 23% false positive rate.*

With a 30% reduction, a few lead times exceed 2 minutes. After phrase reduction, few sequences of phrases were incorrectly identified as failures. Figure 2.12 illustrates the rise in average false positive rate as average lead times prior to node failures increase over the three systems. The average lead times of 0.5, 1.1, 1.6, 4.2 minutes are calculated for the cases of no phrase reduction, 20%, 30% and 40% phrase reduction, respectively. Figure 2.14 shows an increase in the false positive rate in the test data as the amount of phrases pruned is increased from 20% to 40%. Since the false positive rate is more than 30% with 40% tail reduction, even though we could procure lead times as high as 4 minutes in certain failures, we restricted our experiments to 30% tail reduction. Most indicative logs appear just prior to the failure, and backward pruning at times increases the false positives by 5%, as seen in Figure 2.14. Presence of false positives may cause unnecessary migration, but this is much cheaper than costlier checkpoint/restarts without predicted node failures. Table 2.10 shows a partial example where 3 min. 11 sec. lead time could be procured before the node failed at 21:23:26. This is because none of the previous messages were indicative of a failure w.r.t. the learned failure chains. If we aggressively prune more than 30% phrases, TBP can procure as high as 10 minutes lead time in this case.

Table 2.10 Lead Time Improvement

Timestamp	Event Phrase	Lead Time
21:13:01.60	slurm_load_partitions: Unable to contact	10min 25secs
21:14:30.83	Lustre:(ptlrpc _expire_one_request...	8min 56secs
21:17:10.37	Unloading nic compatibility device	≈6min
21:19:10.37	bpmcd_exit: No local access to power statistics..	4min 16secs
21:20:15.48	Invoking ..slurm stop ..stopping slurmd:	3 min 11secs
21:21:19.01	slurmd is stopped..	2 min 7secs
21:21:17.01	Unmounting /dsl/.shared/..	
21:22:18.06	Shutting down...	1 min 17secs
21:23:20.10	rpcbind: cannot save any registration..	6 seconds
21:23:21.63	Shutting down DÂ◌Bus daemon..	
21:23:21.63	Removing... SLURM.Failed..	
21:23:21.63	umount /dsl/var/run 1 ...	
21:23:21.63	Unloading XPMEM driver..	
21:23:23.15	Stopping DVS service:	
21:23:23.15	Could not unmount /dsl...	
21:23:26.22	System halted	Node Down
21:23:35.47	cb_node_unavailable	
21:23:35.49	RCADSVCS : shutdown	

Feasible Proactive Measures in 2 minutes

We have found that lead times ranging from 20 seconds to 2 minutes can be obtained with an average lead-time of more than 1 minute. Live job migration, process cloning, lazy checkpointing [Tiw14], and quarantine technique are some relevant actions that can be taken based on node failure information. Wang et al. [Wan08] report that proactive live migration of jobs can range between 0.29 to 24.4 seconds. Rezaei et al. [RM15] demonstrate node cloning duration within 200 seconds to aid redundant execution during failures. Gupta et al. [Gup15] quantify that 5% to 9% of future failures can be avoided if blades/cabinets are quarantined (no jobs scheduled on the nodes) for some node-hours after a failure is observed on them. Such proactive recovery based prior work shows that 2 minutes often suffice. If the unhealthy nodes are known ahead of time, future jobs can be scheduled on the healthy nodes. Spare nodes are optionalnot required, jobs can be delegated on the existing healthy nodes. Future job scheduling can be delayed in case all nodes are busy and unhealthy nodes need repair. Root cause diagnosis coupled with additional log correlation could further increase the lead time confidence. However, a low false positive rate is also a necessity. Further details of proactive actions are beyond the scope of this paper. TBP locates failed nodes with exact location information (Section 2.4) and can predict failures with acceptable lead times, which can definitely aid recovery actions on specific cabinets/blades/nodes in practice.

Case Studies

We assess a case of node failure to highlight TBP's strength and show rare instances where correlation extraction is hard.

2.5.0.1 A True Positive Case

TBP captured an excerpt of messages, two of which are *out of memory* and *killed process* errors. These indicate a node failure correlated to memory errors. This node failure was caused by a CPU group running out of memory. On further investigation we found that the Slurm node had caused the memory crisis. Subsequently, all the processes sharing that cgroup were killed. This caused a machine hang followed by a few Lustre errors. The compute node went down and was rebooted later. In this case, the co-located phrases in the test data were less frequent than the relevant phrases indicative of the failure, thus TBP identified them. Other compute node failures in the cabinet followed due to this Slurm caused hang. This is a true positive case.

2.5.0.2 Difficult Correlation Extraction

TBP missed certain node failures, which do not necessarily conform to the past seen failures. In those cases, phrase extraction to indicate future failures is hard. They possess similar probabilities for errors that are not fatal. TBP detected some hardware errors (e.g., `correctable aer_bad_tlp`) but it failed to predict failures related to messages listed in Table 2.11. The job-related errors are less frequent but certain errors have probabilities similar to benign messages. We verified one case where the `correctable MCEs` was ranked low so that any $n \leq 150$ excluded it, i.e., TBP discarded it. More research is needed to address these cases.

Table 2.11 Difficult Correlation Extraction

#	Error	Description
1	Console	Processor has catastrophic error
2	Console	DVS: lnet_mapuvm: page count mismatch
3	Job	Node id has a different configuration
4	Console	logged... correctable MCEs

TBP Performance

TBP takes more time to train as the size of the document increases. This is normally the case for any data mining solution. One reason for this is that our logs were not preprocessed, which has the potential to reduce the size of data slightly. Another aspect is the unsupervised learning model of TBP, which operates on unlabeled data. However, text processing (looking for topics in the phrases) takes longer than analysis of just numerical RAS logs of environment data or other features.

Observation 5: *TBP is scalable, taking* $\approx$ *50 msecs to flag a node failure, and below 2 mins to process 320 MB data.*

Table 2.12 TBP Comparison

Solutions	Method	Lead Time	Recall	System	Location
Hora [Pit17]	Bayesian Networks	10 mins	18%	Dist. RSS Feed Reader	Component specific
Zheng+ [Zhe10]	Genetic Algorithms	10 mins	60%	Blue Gene/P	Rack-level
Li+ [Yu12]	SVM, KMeans	10 mins	N/A	Blue Gene/P, Glory	Component with sensor
hPrefects [FX07]	Stochastic Model, Clustering	N/A	N/A	256 node HPC cluster	H/W, S/W components
[Nak11]	Decision Tree	N/A	80%	HPC (LANL)	Node-level
[Hac09]	Neural-gas [Mar93]	N/A	N/A	Blue Gene	Node-level
TBP	Topic Modeling	2 mins	86%	Cray	Node-level

Table 2.13 TBP Impact Assessment

Features	HPC Systems			Distributed Systems		
	TBP	[FX07]	[Zhe10]	[Yu16]	[Xu09]	[Zha16b]
No Source Code	✓	✓	✓	✓	×	✓
Lead Time	✓	×	✓	×	×	×
Location	✓	×	✓	×	×	×
Prediction	✓	✓	✓	×	×	×
Scalability	✓	×	×	✓	×	✓
Unsupervised	✓	✓	×	✓	✓	✓

Figure 2.15 shows that the phrase extraction time (y-axis, on a logarithmic scale) grows with increasing data sizes. This is due to the complexity of TBP but does not present a problem in practice as TBP would eventually be deployed to process log events within a time-limited window below 320 MB representing 6 or more hours, i.e., its cost would be below 2 minutes. TBP takes ≈ 30 to 50 msecs to detect a node failure during testing. In real-time, even if data gets large too quickly, parallel message filtering approaches can aid rapid failure prediction, which is subject to future work. We highlight TBP's novelty distinguishing itself from prior work in three unique ways: it predicts node failures, not system failures; explores semantic information using NLP unlike numerical M/L-based studies; and adheres to time-series-based analysis to showcase lead-time sensitivity.

TBP Comparison

We perform both quantitative and qualitative analysis of TBP with some of the current state-of-the-art techniques pertaining to failure prediction in HPC systems. *To the best of our knowledge, this work is the first of its kind for failure prediction on Crays.* Table 2.12 shows some of the proposed failure prediction techniques. TBP achieves as high as 4 minutes lead time with higher false positive and lower recall rate (≈80%). Hence, we chose to restrict ourselves to 30% phrase reduction. Hora [Pit17]

and Zheng et al. [Zhe10] procure 10 minutes lead time with much lower recall rates. Li et al. [Yu12] and hPrefects [FX07] achieve prediction accuracy greater than 90% and 70% respectively. Nakka et al. [Nak11] and Hacker et al. [Hac09] predict node failures without lead time analysis using less scalable methods.

Observation 6: *TBP is effective in pin-pointing potential node failures, with acceptable lead times, without fault injection or source-code reference w.r.t the state-of-the art approaches.*

We further assess the overall impact of TBP in Table 2.13 by highlighting its traits amongst *log mining solutions* proposed in both HPC systems and distributed systems. We observe that failure diagnosis in distributed systems may refer to the source code [Xu09], and even though some solutions are scalable and unsupervised, they lack location and timing information of anomalies for practical usage. Also, logging practices in the distributed system and software engineering often modify or even add log messages in the source code, and their performance diagnosis is very application- and problem focused. In contrast, we do not augment logging messages as such modifications can only be performed by the vendor at the runtime/operating system layers of HPC systems. Even without such augmentation, we show that predicting failure locations is feasible.

Discussion

How generic is TBP? TBP has been evaluated on Cray systems, but we believe TBP to be applicable for most contemporary HPC systems. Before developing TBP, we went through BlueGene logs of two different systems. These logs, as researched in the past, are comparatively more structured, have easier to detect failure indicators such that the existing approaches suffice. In contrast, TBP handles more complicated logs in an unsupervised manner. With appropriate integration and pre-processing, TBP can certainly be adapted to non-Cray systems with simpler logs or generic Linux logs (as Cray logs are mostly a superset of those). The recall rates and accuracy will depend on the quality of data and the failure diversity.

2.6 Related Work

Several facets of resilience have been studied in the recent past. We present them in four categories, namely a) log mining tools and failure characterization, b) anomaly detection and prediction, c) recovery and root cause diagnosis, and d) log mining in distributed systems. We subsequently mention how TBP enhances or compliments them.

Log mining tools and failure characterization

[Zhe09; Gai11; Gai12b; Mar15a; Ham16a; Di17] propose useful log mining tools such as HELO, ELSA, LogDiver, LogMine, and LogAider. These aid in studying event pattern-based correlations, spatial/temporal event analysis, and application-level resilience [Mar15b] on complex HPC logs. [Ghi16] characterizes node failures through temporal and spatial correlations without any anomaly detection scheme. TBP, unlike any event-correlation framework, exhibits efficient failure prediction.

[SO08; Yu12; Fu14] propose data preprocessing and filtering mechanisms to extract salient features and show symmetry in system and job logs, to improve fault prediction. [ES13; Sri15] study environmental effects on HPC nodes and, analyze memory errors showing their unpredictability and instability in face of scale change. Based on a study of the Titan supercomputer (Cray XK7), [Gup15] finds that the application mean time to interruption (MTTI) can be improved exploiting spatial locality along with temporal correlation by quarantining locations. [Bau16] proposes a dynamic checkpointing scheme adapting to regime changes based on fault events. [Mar14] shows that hardware contributes to only 23% of system downtime, software being the main culprit (74% contribution) of system-wide outages on Cray systems. [Gup17] gives insights to non-uniform spatial distribution of failures and the fact that temporal recurrence is significantly different for diverse failure types but similar across systems. These are field-data based failure characterization studies and do not perform timely prediction like TBP. [Bra15] studies the Trinity platform (Cray XC40) and highlight how new features in hardware, software and HPC subsystems manifest differently in performance along with complexities of log data sources making correlation-extraction difficult. Hence, even if prior work has shown solutions to similar goals in different systems, they may be invalid on new systems. This reinforces our motivation.

Anomaly detection and prediction

[Sal10] proposes a taxonomy of prevalent failure prediction approaches for system reliability. [Gai12a; Aup12; Gai13] discuss the impact of fault prediction strategies and propose a hybrid approach with signal analysis and data mining techniques to predict faults and mathematically model the optimal value of check-pointing. [Lan10b; Lan10a; YL16] propose techniques to find abnormal nodes employing dynamic training to find changing patterns that improve prediction accuracy. [Jia15; Ber14; FX07] propose hardware fault-monitoring and online/offline failure prediction frameworks using ideas of principal component analysis (PCA), co-variance modeling and void search (indicating scarce faults). [Bos16; Hac09; Nak11] develop an algorithmic model using a gossip protocol, apply the neural-gas method and use decision tree classifiers for cluster analysis to detect failed nodes with high probability. These either deal with more structured logs or lack timing analysis, combined with less efficient and less scalable solutions. In contrast, TBP employs unsupervised topic-modeling for node failures. DeepLog[Du17] uses stacked LSTM for anomaly detection without any lead time

analysis, and flags any log key not in top g as an anomaly. In contrast, TBP's anomaly definition is in terms of a failure chain. Costa et al. [Cos14] design a proactive memory management system analyzing memory logs, successfully preventing 63% of the memory-driven failures. Hora [Pit17] studies fault injection and models architecture dependency for component failure prediction unlike TBP. Moreover, TBP's recall rates (83%) are higher than Hora's (73%). Klinkenberg et al. [Kli17] study node soft lockups using supervised classifiers, leveraging the node lock/unlock information from the batch history, unlike TBP. They obtain average lead times of 17 and 22 minutes. However, the effect on the false positive rate is not discussed. We cover more general node failures.

Recovery and Diagnosis

[RM15; Tiw14; Ell12] propose efficient methods of checkpointing using ideas of redundant execution and temporal locality patterns of failures incurring low overhead in recovery. [Wan08; Nag07] leverage techniques of live process migration and proactive task migration for predictable failures. [Zhe12; Chu13] perform root cause diagnosis exploiting log correlation to failures. However, these either focus on improving known recovery techniques like migration and checkpoint/restart or have limitations in root cause diagnosis (e.g., assuming that non-fatal events may not manifest as faults in future). These are complementary to our work.

Log mining in distributed systems

[Xu09; Cho17; Nag12; Yu16; Zha16b] mine logs in cloud-based distributed systems (e.g., Hadoop, OpenStack) using application console logs adhering to data center concerns. They either refer to the application source code for log parsing not applicable for our work or lack lead time analysis, which is crucial for TBP. Moreover, these studies target smaller, less complex systems, and unlike TBP, do not address the problem of predicting node failures.

2.7 Conclusion

This paper proposes a novel, time-based phrase (TBP) model for node failure prediction leveraging topic modeling for text mining. Our scheme achieves no less than 83% recall rates, 98% precision and as much as 2 minutes of lead time. The paper shows valuable insights to Cray data revealing that service node and compute node failures affect user applications considerably. Node failure classification can segregate abnormal failures to address the significant variations observed in phrases over short time intervals. Continuous time series-based temporal phrase mining approaches are an effective way to handle unstructured Cray logs and obtain lead times suitable for proactive actions.

CHAPTER

3

DEEP LEARNING FOR SYSTEM HEALTH PREDICTION OF LEAD TIMES TO FAILURE IN HPC

3.1 Introduction

Significant research efforts [Lan10b; Gai13] have been invested on anomaly detection and failure prediction for enhanced system reliability. With the transitioning from the petascale to the exascale computing era, failures in large-scale HPC systems are anticipated to increase and the MTBF (mean time between failures) will decrease, wasting considerable compute capacity [Cat12]. This obviates research efforts to investigate the trade-offs between power, performance and resilience with alternate solutions. Several failure characterizations [Gup15; Gup17], and machine learning (ML) solutions [Lan10b; Gai12a] for anomaly detection exist for large-scale computing systems. Past work identified faults of gradually failing components with hours of lead time [Zhe12], but most faults occur within a much shorter window. The state-of-the-art lacks in two key aspects. First, faults need to be predicted even when lead times are short (in the order of minutes), together with their exact fault location. In other words, pin-pointing the component (e.g., which node) of impending failures and to do so just in time so that proactive recovery actions can be taken (such as job migration [RM15] or quarantining unhealthy nodes [Gup15]) are equally important. Second, the

large component count of extreme-scale HPC presents a challenge to data mining techniques such as support vector machines (SVM) [Ful08] or principal component analysis (PCA) [Lan10b] due to their limited scalability since prediction has to be performed in real time, and results have to be available prior to the actual failure. Hence, novel scalable and optimized data mining solutions are required. Moreover, the natural language of unstructured logs produced by the computing systems gives rise to two problems. First, since the data lacks any structure and labels, conventional ML techniques suffer from limitations in processing it, e.g. forming feature vectors or classifiers is non-trivial. Second, it is infeasible to infer intricate patterns from high dimensional data quickly, unless the data is processed and fed with appropriate input representation. Deep learning has made tremendous progress in these aspects recently, particularly in natural language understanding [JZ16]. This motivates the need to explore scalable unsupervised deep learning techniques in the context of node failure prediction. Researchers agree that failure prediction is useful even if imperfect and with limited precision [Cat12]. Suppose 50% of the node failures are correctly predicted and the remaining ones are incorrectly predicted (false positives), we can then prevent half of the expensive checkpoint/restarts that require global coordination with much cheaper process migrations.

HPC systems suffer from various kinds of failures at the hardware, software, and application levels. While some failures are critical and obvious to detect, such as kernel panics, most anomalies are not easy to track. Which component will fail and how it will impact the system is not known ahead of time. The observed symptoms of anomalies in the system may or may not reflect the exact root cause. For example, a kernel panic may be caused by a Lustre file system bug or a hardware machine check exception. However, the unwanted consequences, such as node failures, job abortions, etc., can be mitigated if the anomalous patterns are detected well ahead of time by incorporating fast data mining techniques.

This paper proposes Desh[1], Deep Learning for System Health, to predict node failures using a recurrent neural network technique called LSTM (long short-term memory). Desh obtains an average lead time of more than 3 minutes, which suffices for commonly known recovery mechanisms (see Section 3.4.6). While deep learning has been investigated extensively in the areas of vision [Don15] and speech recognition [Hua17], its efficacy in the context of fault prediction and localization for large-scale systems needs more investigation. While DeepLog [Du17] detects anomalies using LSTM without any lead time analysis, Desh predicts node failures. We discussed several other differences to prior work in Section 3.4.5.

Challenges: Recent failure prediction approaches either do not consider lead time to failure, use fault injection [Pit17; Yu16] and synthetic data for evaluation, modify systems with custom log augmentation [Zha16b] relying on the source code format, or do not consider the semantic information of the log entries [Oli08]. In contrast, we use real logs from four supercomputing systems without modifications or augmentation as these logs are vendor controlled, and as are the software

[1] *Desh* means *Country or Native Land* in Hindi

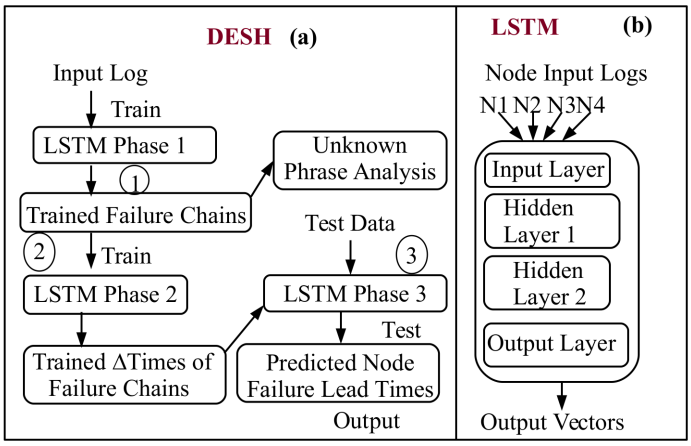

Figure 3.1 Desh with LSTM

layers emitting them, i.e., they cannot be modified by us. The efficacy of the approach is determined by investigating which parts of the log are pertinent for failure prediction, by discarding benign events, and by leveraging expert-labeled ground truth. The main challenge in efficient prediction lies in processing the timestamps between two events across nodes correctly to predict accurate lead times. The data has to retain the location information of the anomaly, keeping track of the time differences between correlated events that are generally not adjacent in the logs. This requires a methodology that considers the anomalous phrases leading to node failures and estimates the time of failure based on prior learning.

Contributions: Desh uses LSTM to estimate lead times for impending node failures. We perform phrase analysis of unlabeled log entries, which may or may not belong to the failure chain. Desh uses a novel three-phase deep learning approach to first train to recognize chains of log events leading to a failure, second re-train chain recognition of events augmented with expected lead times to failure, and third predict lead times during testing/inference deployment to predict which specific node fails in how many minutes. The use of time differences between log entries in a chain requires this second re-training step as failure chains are unknown prior to the first training phase, i.e., it is unknown which events of a log are safe or erroneous until the initial training has formed such chains. Only after chains are known can time differences of earlier events in the chain to a terminal log event indicating a node failure be calculated. Based on time differences and along with the phrases, Desh infers failure chains and ultimately reports specific node ids (identifiers) leading to node failures. Based on trained failures, Desh predicts failures with acceptable lead times before a node stops to respond. Thus, Desh not only helps in flagging failures to take recovery actions, it also gives insights as to what phrases indicate node failures based on this statistical analysis.

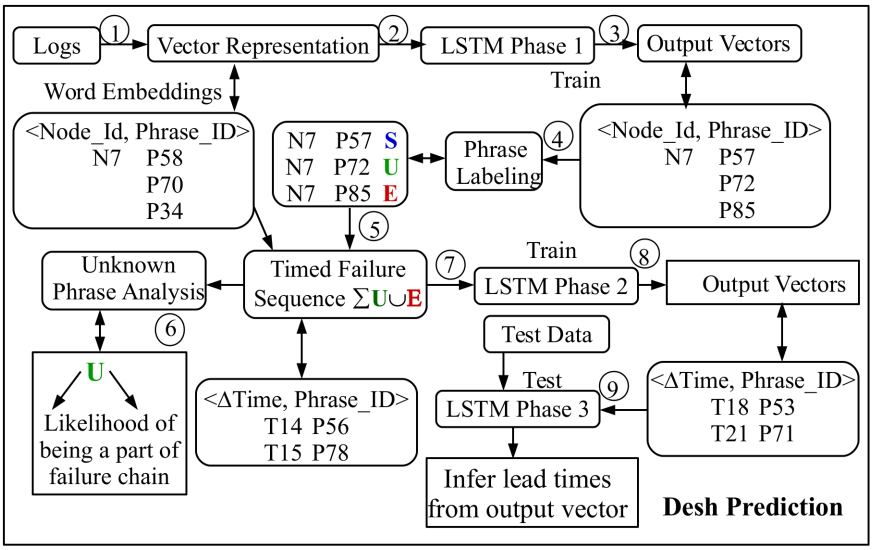

Figure 3.2 Desh Overview

3.2 Background

Traditional language modeling uses frequency counts of variable length sequences (n-gram model) in a vocabulary of words considering every word an individual unit. In other words, the probability of a word given a history is based on maximum likelihood estimation (MLE); two histories are similar if the last (n-1) words are the same. N-gram models [Bro92] do not correlate semantically close words since words are indivisible. However, recurrent neural networks (RNNs) have the power to predict future data based on sequences of past data considering the semantic closeness since they exploit a distributed representation of words, i.e., word vectors of real values. Hence, in RNNs semantically similar words can be close together in the vector space. LSTMs (long short-term memory) are RNNs augmented with memory and logic gates, known to retain a long-term memory of short-term data chains that represent events correlated in time and space (see [HS97] for details). LSTMs contain hidden layers, which strengthen their memory persistence. Supercomputing logs have unstructured textual data with short-term failures from seconds (e.g., kernel crash in 20 seconds) up to minutes (e.g., link control block failure in 5 minutes). Moreover, time-stamped logs have diverse events logged in the granularity of microseconds, and patterns evolve over varying intervals of time that have to be remembered over a long time (days to weeks).

Prior Text Mining Techniques: Past solutions based on probabilistic models, PCA/ICA (principal/independent component analysis) [Lan10b], and Markov chain and decision trees (e.g., random forests) worked for systems with comparatively more structured logs, where structure aids in feature extraction and offline anomaly detection. They are less efficient when it comes to unstructured text data mining with time constraints. Prevalent approaches such as Support Vector Machines

(SVMs) [Ful08] and sequence mining [Fu14] either require complex feature extraction or are unable to capture long-term dependencies making systems intractable with scale. Very recently Coates et al. [Coa13] demonstrated that large-scale training can be done through deep learning on HPC infrastructures with acceptable classification performance and scalable efficiency. LSTM works well for time sensitive data. It can unlearn and relearn over time making it a preferable choice for Desh over other RNNs such as logistic regression and multilayer perceptron (MLP).

Node Failures: This paper uses the term "node failures" frequently. We define node failures as abnormal node shutdowns caused by some system anomaly triggered by software or hardware. These failures differ from intentional maintenance-related massive node shutdowns or periodic service reboots. As any other prediction solution, Desh does not investigate the exact root cause of the anomaly leading to a failure, it instead emphasizes phrase mining to identify impending node failures ahead of time. We have investigated normal service patterns of node shutdowns. Large-scale node reboots clearly indicate service-oriented shutdowns. Desh considers anomalous node failures, that are identifiable by a terminal log message, which is verified in consultation with the system administrators.

3.3 Desh Overview

Table 3.1 provides an overview of the system logs studied, which are collected from four contemporary HPC systems, namely M1, M2, M3 and M4. All of these systems are production level clusters typically running more than 1,200,000 jobs/year and tens of thousands of compute node hours. Duration refers to the time duration of the datasets used for evaluation. Size refers to the log data size and scale indicates the cluster size in terms of the number of nodes. 40% of the top 10 supercomputers belong to the Cray series [Top]. Our system logs have been procured from contemporary Cray machines for prediction evaluation.

Table 3.1 Log Details

System	Duration	Size	Scale	Type
M1	10 months	373GB	5600 nodes	Cray XC30
M2	12 months	150GB	6400 nodes	Cray XE6
M3	8 months	39GB	2100 nodes	Cray XC40
M4	10 months	22GB	1872 nodes	Cray XC40/XC30

These logs are analyzed by our framework called Desh, with its three-phase solution design (Figure 3.1a). (1) In the first phase, a sequence of phrases leading to node failures is extracted, which trains LSTM phase 1 to learn/recognize such chains based on training data. (2) In the second phase, the formulated chains from phase 1 is fed to LSTM phase 2 to make it aware of the cumulative time differences (Δ times) of just those phrases that belong to a failure chain relative to the terminal

Table 3.2 Phrase Vectors

#	Timestamp (T)	Node Id (N)	Phrase (P)	
			Static	**Dynamic**
1	16:25:48.301744	c1-0c1s1n0	kernel ∗ LNet: hardware quiesce ∗	p0-20141216t162520, All threads awake
2	16:39:59.507009	c4-0c0s0n2	Running ∗ using values from ∗	sysctl, /etc/sysctl.conf
3	00:01:16.704832	c2-0c0s15n2	hwerr ∗ Correctable aer_replay_timer_timeout error∗	[28451]:0x6624, Info1=0x500: Info2=0x18:..
4	10:47:39.417963 (T1)	c0-0c0s0n2 (N1)	hwerr *:ssid rsp a_status_msg_protocol_err error∗ (P1)	:Info1=0x4c00054064: Info2=0x0:Info3=0x2

phrase in the respective chain. (3) In the inference phase, learned chains from the second phase allow us to estimate the lead times of future failures with an indication of the future location from test data that is disjoint from the training data (Figure 3.1a). Desh features an RNN with input/output layers along with multiple hidden layers that form a stacked LSTM to facilitate training and prediction on system logs (Figure 3.1b). Figure 3.2 elaborately depicts the overall solution design for lead time prediction described in detail below.

3.3.1 Phase 1: Training

The first phase entails learning failure chains from raw data. The raw logs of Cray/Linux systems contain phrases with anomalies interspersed with considerable amounts of noise and benign events, much in contrast to IBM logs [Blu]. The phrases with timestamps pertaining to specific nodes are separated. For example, a log message has a timestamp T1, an event phrase P1 and a node N1 (see row 4 in Table 3.2). Similarly, several such timestamped phrases will correspond to specific nodes.

Tracking the node ids helps to retain the specific failure location in the cabinet/blade/chassis. We train datasets node wise in this phase (see Figure 3.3a). In other words, logs from each node are concatenated and fed to the same LSTM. This has two advantages. First, there is no overhead of storing the node id and processing it in the vector, which saves memory and computation costs. Second, to learn the patterns of failure chains observed, node identity is of no consequence, what phrases appear in a sequence leading to various node failures is required. We do not predict time in this phase, but sort the log messages by their timestamps for the same reason. The order of the phrases matters here, not their time difference. Time is taken care of in phase 2. As seen in Figure 3.3, phase 1 uses sequence of phrase ids as the vector.

Each event phrase is then segregated into static and dynamic contents to identify the constant message subphrase separating it from the variable component (e.g., error identifier, IP address) as shown in Table 3.2. Thus P1, breaks down to its static component and variable component (see

Table 3.3 Phrase Labeling

#	Safe	Unknown	Error
1	Mounting NID specific	LNet: No gnilnd traffic received from	WARNING: Node * is down
2	cpu ∗ apic_timer_irqs	∗ invoked oom killer	Debug NMI detected
3	Setting flag	LNet:∗ gnilnd:kgnilnd_reaper_dgram_check	cb_node_unavailable
4	Wait4Boot	PCIe Bus Error: severity=Corrected	Kernel panic not syncing
5	Sending ec_node_info with boot code	ERROR: Type:2; Severity:80;	Stack/Call Trace

Table 3.2). The dynamic component is discarded. Once the constant messages are extracted they are encoded to a uniquely identifiable number. These phrases are hyphenated multi-word entities. Now, if we have a sequence of encoded phrases, namely, {45, 67, 89, 40} for node N1, LSTM cannot comprehend their semantic or syntactic relevance in this discrete form. Vector space models with distributed representation help establish semantic correlation. These encoded phrases are then vectorized using word embeddings. Embeddings are defined contexts that check what appears before and after a target event phrase in a sequence of events. We use the traditional skip-gram model [Mik13] for word embeddings of TensorFlow [Aba16] to vectorize the data. For all the encoded phrases in the data, we have a set of phrase embeddings referring to their context such as Lustre, Lnet, Hwerror etc. Using the embeddings and what appears before and after a target, the vector space is formed. Window sizes of 8 and 3 are used, respectively, to consider the number of phrases left and right of a specific target phrase. Desh trains via the stochastic gradient descent optimizer (sgd) with categorical cross-entropy since log analysis is a multi-class problem. Desh predicts the target phrase using the nearby phrases (e.g., for sequence {45, 67, 89, 40, 89, 102}, it provides the probability that 40 is observed when {45, 67, 89} appears before it and {89, 102} appears after it). This conforms to the vector representation (2) in Figure 2. Now, these vectors are fed into the stacked LSTM using two hidden layers (3) to perform 3-step prediction (to predict the next 3 phrases). A larger history size and a higher number of hidden layers increase accuracy, but also the computation time. Experimentation proved 3-step prediction with 2 hidden layers, to have ≈85% accuracy taking ≈0.65 milliseconds in time. More than 1 hidden layer strengthens LSTM's efficacy to remember past phrases to make predictions. This unsupervised LSTM phase 1 training emits the trained sequences of phrase vectors.

Phrase Labeling: From these vectors, we decode the phrases to filter (4) them into three categories: safe, error and unknown as shown in Table 3.3. *Safe* represents the benign phrases, which are definitely not related to any system anomaly (e.g., *Wait4Boot*). *Error* refers to those phrases, which are definitely indicative of some anomaly (e.g., *Stack Trace*). The *Unknown* tag is given to those phrases that may or may not be indicative of some anomaly (e.g., *PCIe Bus Error: severity=Corrected*).

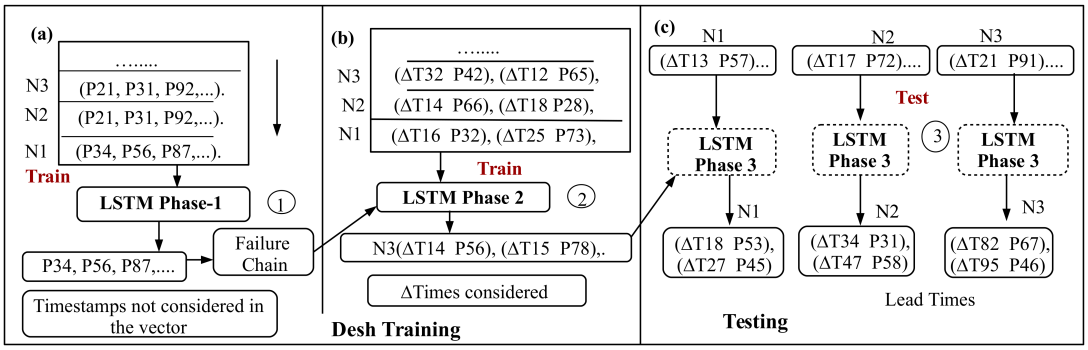

Figure 3.3 LSTM Phases

It should be clarified that tagging a phrase as Error does not imply that it will always be a part of node failures, it may or may not be a part of node failure. However, these phrases are either terminal messages (e.g., Stop NMI detected) or major hardware, software malfunctioning (e.g., page faults) seen in Linux logs. We do not consider the log severity levels even if present. This phrase grouping is based on consultation with the system administrators. After categorization, we have phrases with Safe (S), Error (E) or Unknown (U) labels. Safe (S) phrases are eliminated now, since our primary interest is in the error and unknown phrases.

Phrase labeling is deliberately not done before vectorization since training is more robust with noise. Moreover, indicators for erroneous or benign events are not an a priori for unsupervised learning. Desh incorporates labeling to optimize computation costs in phase 2 to determine the likelihood that unknown phrases lead to node failures. A sequence of events leading to a node failure is formed using Unknown (U) and Error (E) tagged phrases after referring to the raw data, since terminal messages indicating a node going down are known (e.g., Shutdown events, cb_node_-unavailable). Desh forms trained failure chains (5) learning contextually relevant phrases temporally close as shown in Table 3.4. At this juncture, we evaluate statistically how certain unknown phrases form a failure chain, while others never appear in any chain. This likelihood estimation unveils insights to log messages eventually leading to node failures. We discuss the analysis of unknown phrases in Section 3.4.3.

LSTM Phases: Figure 3.3 depicts a unified view of the three phases. Desh phase 1 and phase 2 are training phases that process the vectors concatenated, i.e., one node after the other (Figures 3.3a and 3.3b). Desh learns failure sequences from different nodes sequentially. The difference is the contents of the input vector for each log message pertaining to a node. While phase 1 contains only phrase ids, phase 2 contains the time differences between phrases along with phrase ids (Section 3.3.2). In phase 3, Desh detects failures based on the previous training applied to the new test data (different from the training set). As seen in Figure 3.3, the vectors fed to LSTM phase 3 are from a specific node (not concatenated). The vectors contain time differences (ΔT) of phrases along with phrase ids similar to phase 2 (Section 3.3.3). The idea behind this is to first learn from all nodes

Table 3.4 Example Failure Chain

#	Timestamp	Phrase	Label	Phrase Vector
P1	03:59:58.466 (T1)	CPU *: Machine Check Exception:	U	ΔT1=07.822, P1
P2	03:59:59.543 (T2)	[Hardware Error]: Run the above through 'mcelog –ascii	U	ΔT2=06.745, P2
P3	04:00:00.477 (T3)	[Hardware Error]: RIP !INEXACT! 10:	U	ΔT3=05.811, P3
P4	04:00:01.706 (T4)	Kernel panic - not syncing: Fatal Machine check	E	ΔT4=04.582, P4
P5	04:00:01.731 (T5)	Call Trace:	E	ΔT5=04.557, P5
P6	04:00:06.288 (T6)	cb_node_unavailable	E	ΔT6=00:000, P6

and use that assimilated learning to detect failures per individual node during testing and inference.

3.3.2 Phase 2: Training

The objective of the second phase is to predict lead times based on the learned failure chains. In phase 1, the presence of noise makes ΔT calculation infeasible, since we need to consider anomalous phrases leading to node failures without interspersed Safe phrases. In this phase, we segregate the phrases forming the failure chains from the rest (not part of a failure chain), and compute the time differences between phrases in the failure chain to enable lead time prediction. Desh then trains multiple failure chains to learn the diverse ΔTs with phrase ids to eventually predict what times are expected in the future. This enables the capability of lead time prediction of Desh.

We know the timestamps and the phrases (either U or E) pertaining to a detected node failure from phase 1. Table 3.4 demonstrates how the time differences between the phrases in the failure chain are converted to an input vector for LSTM phase 2, for a specific node failure. This failure was caused by hardware processor corruption. Here, a CPU experienced a hardware machine check exception, followed by kernel panic with a call trace, after which the node failed. Table 3.4 enlists a few phrases for brevity.

ΔTime Calculation: Our target is to predict the lead time to a node failure. Anomalous messages precede terminal messages in a failure sequence. The manifested failure is indicated by the higher order time-series. Hence, we sort the data in descending order of timestamps and calculate ΔTs, which is the cumulative time difference between the current phrase and the last phrase (highest order) in the sequence. The highest timestamped phrase in the sequence is assigned ΔT=0 since there is no phrase left in the sequence to calculate the time difference. For example, in Table 3.4, ΔT6 is assigned 0. Next, we compute the ΔTs subtracting timestamps of every phrase from T6 (e.g., T6-T5, T6-T4, T6-T3 etc.) in the failure chain, respectively, and obtain the time differences in seconds (7.822, 6.745, 5.811, 4.582, 4.557, 0). The input to LSTM phase 2 is a 2-state vector with the ΔT and the phrase id, as shown in the Phrase Vector column of Table 3.4. As seen in Figure 3.3, the vectors are concatenated across node logs, but with added Δ times unlike phase 1. Training on these time differences helps LSTM learn how late the terminal phrase is expected to appear in the sequence

Table 3.5 LSTM Parameter Specifications

#	Input Vector	Output Vector	#HL	Steps	#HS	Loss Function, Optimizer
Phase-1	(P1, P2, ...PN)	(P11, P15, ..PN)	2	3	8	SGD, Categorical Crossentropy
Phase-2	(ΔT1, P1), (ΔT2, P2), ..	(ΔT11, P11), (ΔT22, P22), ..	2	1	5	MSE, Rmsprop
Phase-3	(ΔT4, P4), (ΔT5, P5), ..	(ΔT15, P15), (ΔT16, P16), ..	2	1	5	MSE, Rmsprop

based on the previously seen phrases. In phase 2 the LSTM is fed with these vectors with a history size of 5 to perform 1-step prediction for every sequence with 2 hidden layers.

This training performed offline on multivariate time-series expects the predicted value to be close to the actual value seen in the training data, thus the objective function employed is the mean squared error (MSE) loss minimization combined with the RMSprop optimizer.

3.3.3 Phase 3: Testing

In the third phase, LSTM is used on the test data to validate trained failure chains from Phase 2. The test data is processed to form encoded vectors with time differences and phrases similar to the discussion of Table 3.4 in Section 3.3.2. It should be noted that here we need to track the node id to know which node is expected to fail with how much time is left before any failure. Please note in Figure 3.3c, the vectors are not concatenated across nodes as in phase 1 and 2. Instead, the log of each node is passed to an identical trained LSTM. Sequences of vectors containing ΔTs and phrase ids pertaining to nodes are formed from the test data. We form batches corresponding to distinct nodes with their sequence of phrase vectors. Suppose we have 100 distinct nodes in the test data, then we have 100 batches of size M, i.e., M 2-state vectors in each batch. This represents node ids in a way that saves computation cost and conforms to the input vector format. LSTM uses this test data and the target data obtained from the previously trained failure chains for evaluation. LSTM predicts the next sample from the trained data (failure chain), compares with the vector in the test data and computes the MSE. Notice that this is not a binary classification problem. While validating phrases in a sequence the prediction if a node failure will happen is determined by trying to find a close match to the actual target failure chain.

We use a threshold of 0.5 for inferring node failures. In other words, when LSTM obtains MSE≤0.5, we consider those outcomes to check for failure. Based on experimentation, more than 0.5 MSE in the test data emitted chains that are quite dissimilar from those in the trained failure chains. Suppose the prediction is {P1, P3, P7} and the failure chain is {P1, P3, P8}. We may consider it a failure and cross validate with the ground truth data. We can then extract the ΔTs from the vectors and compute the predicted lead times. Let us say we have {(6.2, P4), (2.57 P6), (1.4, P9)}, with time

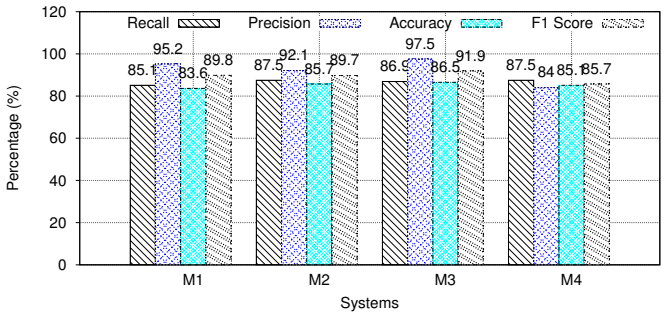

Figure 3.4 Prediction Rates

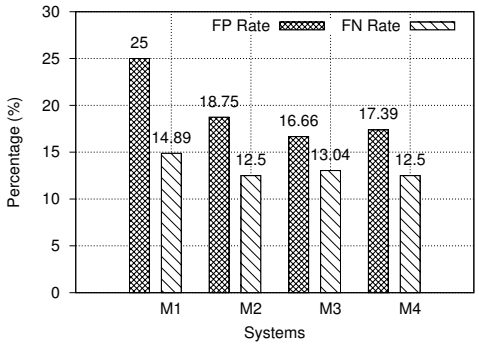

Figure 3.5 FP Rate and FN Rate

in minutes, then a lead time of 4.8 minutes is predicted before the node fails. Overall, Desh obtains node failures with as high as 87.5% recall and an avg. of 19.43% false positives. Table 3.5 depicts the major parameters of LSTM training for Desh in phases 1, 2 and 3. The input/output vectors show the details of the vector entries; HL indicates the number of hidden layers used; Steps refers to the number of steps of prediction, i.e., how many samples to predict based on the history provided; and HS refers to the History Size, the window size of samples given during training based on which prediction happens. Lastly, the loss function and optimizer required for LSTM is indicated per phase.

Table 3.6 Test Data Statistics

System	FC-related
M1	84.9%
M2	87.17%
M3	88.3%
M4	72.46%

Breakdown of Test Data: Table 3.6 depicts the fraction of FC-related phrases present in the overall test data used for evaluation for the four systems. It should be noted that the test data is split across different time-frames during testing. The table indicates the percentage of phrases that are part of failure chains, the remaining are those that are not part of any failure chains. This percentage alone does not indicate the expected recall or precision, since FC-related phrases part of a sequence dissimilar from the failure chains (e.g., not having the subsequent phrases of an FC or not ending with a terminal message) are not actual failures, they have partial matches. Several nodes encounter similar messages during the day, however, not all fail. These estimates give an indication of the overlap of common phrases for events leading to a failure versus those not leading to a failure.

3.4 Evaluation

We have built a prototype implementation of Desh using Keras [Cho15] with a TensorFlow [Aba16] backend. Our evaluation uses actual system logs of several supercomputing machines to demonstrate the efficacy of Desh. Desh obtains as high as 3 minutes lead time, with no less than 83.63% accuracy. We split the dataset for all the systems for training and testing. 30% of the data is used for training and the remaining is used for testing. The experiments are performed on the Intel platform. Our experimental evaluation quantifies the prediction efficacy of Desh in terms of the standard performance metrics, performs lead time sensitivity and determines the implications of unknown phrase analysis in the context of system reliability.

3.4.1 Prediction Accuracy

Table 3.7 tabulates the metrics used for statistical analysis, namely, recall, precision, accuracy, F1 score, false positive rate and the false negative rate. Their corresponding formulas are indicated in Column 2, where TP = True Positives, FN = False Negatives, TN = True Negatives and FP = False Positives, respectively. Correctly predicted failures are true positives, incorrectly predicted failures are false positives, failures missed by Desh are false negatives, and the sequence of phrases not predicted by Desh as failures, which are actually not failures, are true negatives. Additionally, we have computed the F1 score, which evaluates Desh's failure prediction accuracy considering the weighted average of recall and precision.

Figure 3.4 illustrates that, both accuracy and F1 score are relatively high for all the systems. While the recall rates do not vary much (85.10% through 87.5%), M4 has comparatively low precision.

Observation 1: *Desh has ≥84% precision, ≥83.6% accuracy and ≥85.7% F1 score along with as high as 87.5% recall rates across all the four systems.*

This implies that the information gained from the learned failure chains aided Desh in making accurate predictions for the overall test data. In other words, new patterns or unknown failures are rare, hence, our model was a good match.

The failure chains were reliable to sustain the model's predictive power over the events encountered during testing. Another reason for Desh's performance is the history window size is 5 to 8 in Desh. More history improves accuracy consuming more time. Reducing the history size to 3 brings down the accuracy by 10% to 14%.

The false positive and false negative rates for the systems are shown in Figure 3.5. While the false positive rates range from 16.66% to 25%, the false negative rates do not exceed 15%. They vary between 12.5% to 14.89% indicating that Desh is effective in not missing actual failures. M1 has a

Table 3.7 Prediction Efficiency

Metrics	**Formula**
Recall (%)	TP/(TP+FN)
Precision (%)	TP/(TP+FP)
Accuracy (%)	(TP+TN)/(TP+FP+FN+TN)
F1 Score (%)	2*(Recall*Precision)/(Recall+Precision)
FP Rate (%)	FP/(FP+TN)
FN Rate (%)	FN/(TP+FN), (1-Recall)

higher false positive rate and higher true negatives. For M1, incorrectly flagged failures were higher. Since the overall accuracy and F1 score are high, a 25% false positive rate is acceptable and Desh is capable of predicting most of the node failures.

Table 3.8 Node Failure Classes

#	**Class**	**Failures**	**Avg. Lead Times(secs)**
1	Job	Job scheduler (Slurm)-based errors, Task/Application related bugs	81.52
2	MCE	H/W Machine Check Exceptions, Page/Memory Faults, Processor Corruptions	160.29
3	FileSystem (FS)	Lustre/DVS Bugs, Packet/Protocol Errors	119.32
4	Traps	Segmentation Faults, Trap invalid opcode	115.74
5	Hardware (H/W)	NMI Faults, critical hardware errors, Node heartbeat errors	124.29
6	Panic	Stack Trace, Kernel Panic	58.87

3.4.2 Lead Times

Our research goal is to obtain sufficient lead times while retaining the prediction accuracy. We have analyzed lead times across three dimensions, seeking to answer the following questions:

1. How does the diversity of node failures affect the lead times?
2. How high is the average lead time for each of the systems?
3. How sensitive are the lead times w.r.t. the false positive rates?

To this end, we classify node failures considering their predominant context of failures. We have investigated various chains leading to failed nodes and determined the prominent phrases causing

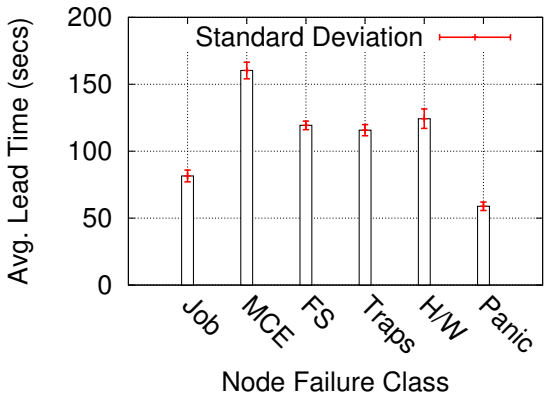

Figure 3.6 Lead Times+Failure Classes

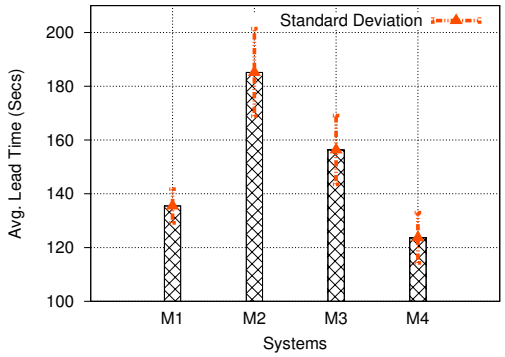

Figure 3.7 Avg. Lead Times of Systems

anomalous node shutdowns. Table 3.8 enumerates those classes evaluating the sequence of events. `Job` class refers to the node failures caused by the slurm job scheduler due to slurm controller connectivity problems or resource outage caused by application aborts etc. `MCEs` refer to hardware machine check exceptions frequently encountered by compute nodes causing processor corruptions, memory faults and other hardware interrupts. `FileSystem` bugs are mostly Lustre (parallel file system) errors, mount problems of DVS (data virtualization server) etc. Several bit errors, packet and protocol errors are also considered in this class, which are common in many failure chains. While `Traps` are typically software interrupts, exceptions, segmentation faults and invalid opcode errors, the `Hardware` errors are node heartbeat fault messages, NMI faults, interconnect failures etc. Lastly, a kernel `panic` followed by a stack trace can be triggered by both hardware faults and software exceptions. Kernel panics commonly cause nodes to fail. Figure 3.6 helps to answer the first question. The standard deviation as seen is low. It is intuitive that kernel panics do not have high lead times since the anomalies happen just prior to the failure, without enough lead time for proactive job migration. Their avg. lead time is ≈1 minute. `MCEs` and `FileSystem` failures have comparatively higher lead times, job scheduler-based failures are rare with ≈82 seconds lead time.

Observation 2: *Based on the class of failures, procured lead times differ. LSTM training is efficient when trained data contains major failure classes with correlated log messages preceding an actual terminal message where node stops responding.*

Figure 3.7 shows the average lead times of each of the 4 systems with their standard deviations. M2 has higher lead times than the rest since M2 features more node failures caused by `Hardware` and `Filesystem` classes and fewer `kernel panics`. All the systems have more than 2 minutes average lead time. The standard deviations of lead times of a specific failure class is lower than it is in a specific system. On investigation we found that, a system has higher variations of failures with

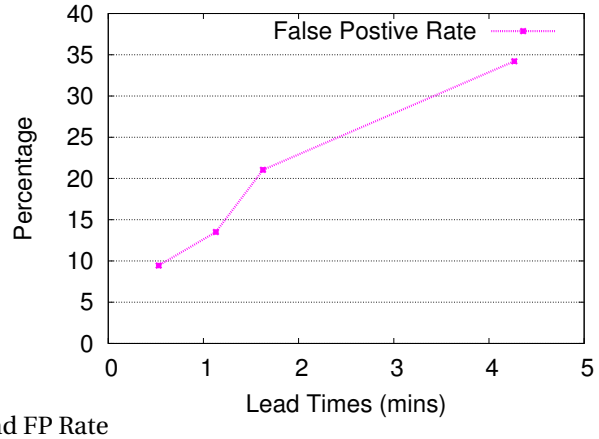

Figure 3.8 Lead Times and FP Rate

diverse failure class, hence the obtained lead times vary. For all the node failures of the same failure class, the deviation in lead times are not that high as seen in Figure 3.6. Figure 5.13 shows the lead time sensitivity. Let us recall the cumulative ΔT calculation Desh performs in phase 2 training, as described in Section 3.3.2. Suppose, we have a sequence of events in the test data, {(4, P1), (3.1, P2), (2.5, P3), (0, P4)} with ΔTs (in minutes) and phrases in each sample vector. Now, if Desh predicts a failure while testing P4, we obtain 0 lead time (P4 is a terminal message), if a failure is flagged after checking P3 we get 2.5 minutes lead time. In other words, the earlier we flag the longer the lead time. The caveat is that there are several other sequence of events similar to a target failure chain not leading to a failed node. In those cases, if failure is flagged after checking P2 or P1, we obtain 4 minutes lead time at the expense of an increasing false positive rate. This makes the sensitivity study important. We aim at longer lead times, yet need to limit the false positive rate. Figure 5.13 indicates that a false positive rate in the range of 18% to 30% results in a lead time of 105 seconds to 196 seconds. As the lead time increases to more than 4 minutes, false positives rise to 39%, finally reaching 44% with $\geq$6 minutes lead time.

Observation 3: *Desh obtains more than 2 minutes lead time with acceptable false positive rate (16.66% to 25%) and false negative rate (12.5% to 14.89%) across all the 4 systems.*

Observation 4: *The standard deviation of lead times to node failures of a specific failure class is lower than the standard deviation across all node failures in a system indicating that different failure classes have unique and reproducible lead times to failure.*

3.4.3 Unknown Phrase Analysis

Table 3.9 depicts a subset of commonly encountered phrases in Cray logs that are not individually benign or erroneous by itself but can manifest as failures based on other system events. Their

Table 3.9 Unknown Tagged Phrases

#	**Phrase**	(%)
P1	LustreError *	56
P2	Out of Memory/Killed Process	15
P3	Lnet: Critical H/W error	36
P4	Slurm load partitions error: Unable to contact slurm controller	42
P5	hwerr[*]: Correctable AER_BAD_TLP Error *	12
P6	Sent shutdown to llmrd at process *	17
P7	AER: Multiple corrected error recvd *	21
P8	Trap invalid code * Error *	8
P9	modprobe: Fatal: Module * not found *	27
P10	<node_health> * Warning: program * returned with exit code *	29
P11	DVS: Verify Filesystem	60
P12	BUG: unable to handle kernel NULL pointer dereference	25

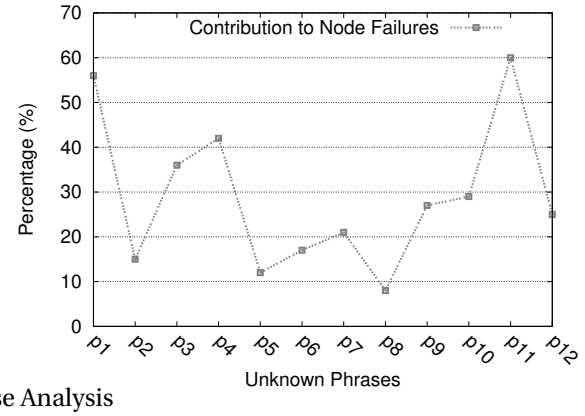

Figure 3.9 Unknown Phrase Analysis

percentage of appearance in node failure chains is indicated in Column 3. Some phrases are seen in many abnormal node shutdowns such as LustreError (P1) and DVS: Verify Filesystem (P11). It is interesting to note that an appearance of anomalous messages such as software trap (P8) or network critical hardware error (P3) do not necessarily result in *failing nodes*. We have observed these phrases in sequences of events pertaining to nodes without unusual node shutdowns during those time frames. We distinguish between anomaly-based node failure versus intended node shutdowns such as maintenance service or periodic reboots. Those have simpler patterns in manifestation. Figure 3.9 demonstrates that while hardware correctable errors (P5) and out of memory/killed process errors contribute less to failures, Lustre filesystem bugs are quite common. This is because several hardware faults trigger software faults that in turn trigger other software errors, even if the root cause may be a hardware error (conforms to the observation by Gainaru et al. [Gai14]). This observation is subject to the events during the time frame considered. Even if the statistics vary based on the target systems and datasets used, this estimate drives home an important point, namely that the presence of software traps or critical hardware errors does not always lead to node

Table 3.10 Unknown Phrases with and without Node Failures

#	Failure 1	Failure 2	Not Failure 1	Not Failure 2
1	H/W Error: MCE Logged	LustreError *	nscd: nss_ldap reconnected	LustreError: Skipped
2	Corrected Memory Errors on Page *	DVS: Verify Filesystem: *	<node_health> program * returned with exit code *	nscd: nss_ldap reconnected
3	<node_health> program * returned with exit code *	DVS: * no servers functioning properly	Trap Invalid Code	Hw Error: MCE Logged
4	mce_notify_irq: *	Startproc: nss_ldap: failed..	Killed process *	Corrected DIMM Memory Errors
5	Lnet: critical hardware error: *	Stop NMI Detected	Out of memory *	MCE_notify_IRQ
6	[Gsockets] debug [0]: critical h/w error	Slurm load partitions error: Unable to contact slurm controller	Lustre: * binary skipped *	Lnet: H/W Quiesce
7	Stop NMI Detected	Slurmd Stopped	hwerr[*]: RSP A_status_msg_protocol_error*	Corrected Memory Errors on Page *
8	<node_health> warning: * node is down	System: halted	<node_health> * failures: The following tests * failed	Lustre: * connected to *

failures, if their cause can be corrected eventually. Our aim is not to perform root cause diagnosis here. In contrast, erroneous phases such as kernel panics/stack trace, NMI faults, CPU stalls and several hardware machine check exceptions (MCEs) do result in failed nodes.

Table 3.10 depicts 4 sample sequences of events. The first two are node failures caused by anomalies, the last two are not node failures. In fact, node shutdown messages were absent during those time frames. We pick snippets of important phrases to highlight cases where such unknown phrases may or may not lead to an anomaly. The first node failure was caused by too many hardware machine check exceptions causing the CPUs to get corrupted. The non-failure case 1 (3rd column) shows a sequence of messages that did not eventually cause any node failure, although the node encountered traps, followed by jobs getting killed and hardware protocol errors. Note that the 10th phrase (program * returned with exit code) from Table 3.9 is present in both a failure case (1st column) and a non-failure (3rd column). The 2nd node failure was caused by filesystem bugs and a slurm controller connectivity error. The non-failure case 2 (4th column) shows Lustre errors in the beginning, yet the node endured several MCEs and DIMM memory errors without failing immediately. In fact, hardware MCEs and corrected memory errors were logged in both failure (column 1) and non-failure (column 4) cases, as are Lustre errors (column 2+4). Overall, it is clear that phrase mining-based anomaly detection is non-trivial and appearance of anomalous phrases neither indicates root causes nor certainty of eventual node shutdowns. We have provided an estimate of how much the unknown phrases contribute to a node failure. Terminal messages such as *Stop NMI detected* usually appear whenever a node goes down (either normal or anomalous),

debug NMI detected is more often seen with anomalies. What are the implications of these insights? Most vendors and HPC administrators expect accurate failure indicators to save resources, what messages lead to failures may not be anyone's concern. However, such characterization implies the following observations:

Observation 5: *A log message with a given phrase may be benign in one context while it is part of a failure chain in another one, or it may lead to a fault that is later corrected so that the fault is masked resulting in no failure.*

Major hardware bugs and software panics are known to cause failure [Gup17] in HPC systems. But in-spite of the appearance of similar messages (traps, critical hardware errors), nodes do not always fail. Events external to a specific node (e.g., interconnect bugs, temperature conditions) can cause such bugs and there exists conditions under which these faults do not cause failures. This is different from the known spatial correlation [Gup15] that node failure correlation is higher within the same cabinet than a blade. It will be interesting to determine what faults under what condition do not cause failures.

Observation 6: *In general, tags such as warning or critical with a log message should not be uniquely associated with a log event as the context of correlated events in time and space in a failure chain is indicative of anomalies, not a single event by itself.*

In the past, researchers have heavily relied on *fatal* severity level to formulate detection schemes. With contemporary system logs, understanding the sequence of events enhances machine learning solutions.

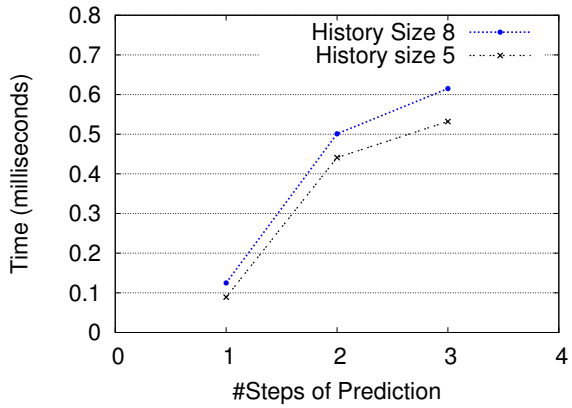

Figure 3.10 Cost Analysis

Table 3.11 Desh Comparison

Solutions	Method	Lead Time	Recall	Precision	Anomaly Injection	System	Location
Hora [Pit17]	Bayesian Networks	10 mins	83.3%	41.9%	✓	Dist. RSS Feed Reader	component specific
Gainaru et al. [Gai14]	Signal Analysis	N/A	60%	85%	×	Blue Waters	N/A
Islam et al. [IM17]	Deep Learning	N/A	85%	89%	×	Google Cluster	Job-level
UBL [Dea12]	Self-Organizing Map (SOM)	50 secs	N/A	N/A	✓	RUBiS, Hadoop, System S	N/A
CloudSeer [Yu16]	Automatons, FSMs	N/A	90%	83.08%	✓	OpenStack	N/A
Desh	Deep Learning	3 mins	86%	92.2%	×	Cray	node-level

3.4.4 Cost Analysis

Desh operates in three phases as discussed earlier (Section 3.3). The training phases 1 and 2 are performed offline and have no consequence to the prediction performance. However, we report the time taken to perform predictions based on the history size. As shown in Figure 3.10, with a larger history window, the time taken is slightly longer. As expected, 3-step predictions take longer than 1-step prediction. The times taken by 2-step predictions are comparable for both the history sizes of 8 and 5. The size of the vector entries and computing platform used to run LSTM also determine the cost. Optimizing detection speedup is not Desh's goal. Nevertheless, the prediction time can be insightful to provide further optimizations in log mining solutions.

3.4.5 Desh Comparison

The closest work related to Desh is DeepLog [Du17]. DeepLog uses stacked LSTM on HDFS, Openstack and BlueGene logs to detect anomalies. However, there are fundamental conceptual differences that makes Desh's application of LSTM unique. We discuss them briefly here:

1. DeepLog injects anomalies for cloud systems (e.g., during VM creation). Desh considers failures extracted from real field data, with no synthetic injection performed. Their anomaly detection with BlueGene/L RAS logs is similar to our Cray system log analysis. However, their log labeling and inference of anomaly differs substantially. Their normal and abnormal labeling relies on normal execution patterns to detect deviations for abnormal messages, which is not our work. Moreover, these logs are more structured than Cray logs, which are unstructured because of diverse log sources. For example, in Table 3.12, the first two phrases are considered abnormal and the last two normal based on the a priori information about

Table 3.12 BlueGene/L Log

#	Log Message	Label
1	kernel Info total of 2 ddr error(s) detected and corrected	Abnormal
2	kernel Info CE sym 9, at *, mask *	Abnormal
3	App fatal ciod: Error creating node map	Normal
4	kernel fatal MailboxMonitor::serviceMailboxes	Normal

BlueGene/L system event logging. Desh does not consider an individual message as abnormal unless it contributes to a node failure chain. We also explicitly demonstrate that severity labels are insufficient indicators of system malfunctioning (Section 3.4.3). Although DeepLog's workflow construction is analogous to our failure chain formation, they consider performance anomaly on a per-log entry level, i.e., if the actual value does not appear in the top g predicted keys, it is considered an anomaly. Our failure definition is more refined, it is based on a sequence of event vectors observed. An individual phrase by itself does not determine an anomaly, since the node failures are flagged based on the trained failure chains.

2. We design LSTM to predict lead times and track the node ids to pin-point failure location. In other words, Desh can warn, `In 2.5 minutes, node X located in Y is expected to fail`. The node id (e.g., cA-cBcCsSnN) contains the exact location information (cabinet: AB, chassis: C, blade: S, number: #N). This indication can prevent further scheduling of jobs on node X, existing applications can be migrated from that node to another healthy node. Such proactive actions can mitigate service disruptions and future job failures. DeepLog is not designed to predict lead times, it can detect performance anomalies in the system to aid diagnosis. Desh aims to strike a good balance between lead times and false positives. Increasing lead times hurts the false positive rate. Instead, acceptable lead times with low false positive rates are desirable.

3. Apart from the differences in anomaly definition (per log entry level vs. sequence level) and research goals (performance anomaly detection vs. node failure prediction), HDFS and Openstack logs are at a higher software layer (atop native filesystems and JIT engines) than low-level Linux-style Cray logs. Hence, performance anomalies recurring in such systems are different from the hardware critical errors appearing in HPC systems. Moreover, Desh exploits word embedding for vectorization along with cumulative Δ time calculation, differing from DeepLog's solution paradigm.

We have further enumerated the major capability differences between DeepLog and Desh in Table 3.13. DeepLog has performed online model updates to improve false positive rates and does not aim at tracking the component location. We have performed lead time characterization w.r.t. false positive rates.

Table 3.13 Desh vs. DeepLog

Features	Desh	DeepLog
No Source-Code	✓	✓
Lead Time	✓	×
Component location	✓	×
Sequence-level Anomaly	✓	×
Injected Failures	×	✓
Node Failures	✓	×
Cloud+HPC	×	✓
False Positive Rate	✓	×

Apart from DeepLog, there exist several state-of-the-art failure prediction solutions for system resilience. We have compared Desh with a few of those solutions in Table 4.8. CloudSeer performs automaton-based workflow monitoring on OpenStack, and UBL [Dea12] predicts performance anomalies exploiting self-organizing maps, both using anomaly injection. UBL (Unsupervised Behavior Learning) calculates lead time as the difference between the points in time of actual SLO (service level objective) violation and of anomaly detection for Hadoop, RUBiS and System S using application logs. In contrast, Desh predicts lead times of node failures using LSTM for low-level system logs. Hora uses fault injection to perform prediction in the Netflix RSS feed reader, which is architecturally different from HPC systems. Islam et al. [IM17] uses LSTM like Desh, obtains high precision and recall but targets the job failure problem rather than node failure. Gainaru et al. [Gai14] integrates their older research tool, ELSA [Gai12a], to perform online anomaly detection on Blue Waters. They obtain comparatively lower recall (60%) rates. Overall, Desh demonstrates a novel way of predicting node failures with low false positives and high accuracy.

3.4.6 Discussion

With increase in scale of production computing systems, the mean time between failures (MTBF) is expected to decrease. Failures are expected to occur in shorter intervals of time [Tiw14]. Failure lead time prediction can decrease the failure rate in large-scale systems. Desh incorporates stacked LSTM efficiently to have high prediction accuracy (85.71%), acceptable false positive rates (19%) and 3 minutes lead time warnings. Unknown phrase analysis further reveals the prospects of understanding system characteristics when a set of events leads to a failure versus conditions when the same set of events does not lead to a failure. In other words, the failure manifestation condition has several correlated system parameters indicating non-deterministic symptoms. Hence, further investigations of phrase mining-based failure prediction look promising.

How generic is Desh? Desh has been evaluated on Cray Logs. How generic is it for other system logs? We have thoroughly investigated prior HPC logs such as BlueGene that are comparatively easier for feature classification. Several solutions exist in this context [Lan10a; Zhe11]. Hence, prior

researched techniques suffice. Several studies [Wan17b; Jia17; IM17; Zha16a] have employed LSTM for system anomaly detection in cloud computing systems and service-oriented architectures, making Desh's solution paradigm highly generic. How efficient and computationally inexpensive is the question. That depends on the problem goal and system characteristics. Cray systems contain dense unstructured logging from multiple log sources, which makes log mining non-trivial. Desh can certainly be adapted to other large-scale production systems with a different logging paradigm with some customizations. Our approach in itself is unsupervised and remains unperturbed by the chasms of diverse computing infrastructures.

How much lead time is sufficient? What actions are feasible? Desh reiterates the requirement for sufficient lead times. So how much lead time is good enough? Several proactive recovery mechanisms have been investigated such as job migration, process cloning, lazy checkpointing and similar techniques that are successful in mitigating job failures. Traditional reactive checkpoint/restarts are expensive. Process-level job migrations [Wan08] take 13 to 24 seconds, skip/lazy checkpointing [Tiw14], or quarantining nodes [Gup15] by preventing future job scheduling on them are all feasible proactive actions that can be taken in practice, if Desh can indicate impending node failures with lead times longer than these mitigation actions require. Dino [RM15] proposes node cloning service in 90 seconds. Three minutes lead time suffices for the discussed recovery options. With a reduced false positive rate, even lower lead times ($\approx$1.5 minutes) can be helpful for proactive resilience actions. Further discussion about failure recovery is beyond the scope of this paper.

3.5 Related Work

Data mining solutions for enhanced system reliability are being investigated both in cloud computing and HPC systems. Understanding performance and resilience trade-offs is important for failure prediction in large-scale systems. We categorically highlight features that are distinct in Desh.

LSTM Applications: Recent works such as [Wan17b; Jia17; IM17; Zha16a] have leveraged LSTM for failure prediction. Wang et al. [Wan17b] uses LSTM to improve the quality of service in service-oriented architectures. Researchers [Jia17; Zha16a] have utilized binary classification for predicting failures without any lead time discussion. Islam et al. [IM17] perform failure prediction on cloud computing systems focusing on resource usage and jobs/tasks termination and have obtained 87% accuracy. Their high-level goals are similar to ours. However, Desh formulates failure chains and focuses on predicting lead times from real data. The obtained lead times are promising for taking proactive recovery actions in practice. DeepLog [Du17] is the closest work to Desh. Apart from the target logs such as Openstack, which are very different from Cray logs, DeepLog's definition and detection of anomaly differs from Desh. They predict a single phrase as an anomaly, while Desh evaluates a chain of events to flag an anomaly.

Failure Prediction: Nie et al. [Nie17] study GPU errors using neural networks. Nodeinfo [Oli08]

proposes an unsupervised alert detection system using ideas of information entropy and binary scoring. Bouguerra et al. [Bou13a] find that the failure distribution correlates with the false negative distribution and that the temporal correlations of failure needs to be understood. Xie et al. [Xie17] assess the Lustre file system of the Titan Supercomputer to propose a statistical regression approach, which predicts output performance in petascale file systems. Chilimbi et al. [Chi14] propose Adam, a scalable deep learning training system. Bautista-Gomez et al. [BG16] discuss the spatial and temporal analysis of DRAM memory errors in HPC systems. However, their objectives differ as they do not aim at node failures in computing systems. Hora [Pit17] formulates a Bayesian model for component failure prediction using component dependencies unlike Desh, which is unaware of such system dependencies and solely relies on the data for inference. Gainaru et al. [Gai14] discuss an online failure prediction model using feedback simulation with their toolkit ELSA [Gai12a]. In contrast, Desh uses deep learning to predict lead times to failures.

Log Analysis: Event block detection by Baseman et al. [Bas16b] focuses on tokenizing unstructured text. Pecchia et al. [Pec11] find that grouping events based on predetermined time thresholds performs badly. Additional consideration of likelihood of entries improves field data analysis. Di Martino [DM13] uses the MTW (multiple time windows) heuristic to group supercomputing error logs. Prior log analysis techniques have studied various event correlation methods [Gai12a], time coalition techniques [DM12] and log parsing methods [He16]. Our work uses word embeddings to vectorize the data. While processing the log input is similar to other unstructured text parsing, Desh exploits the relevant context (word embeddings, discussed in Section 3.3.1) for node failure prediction. Tiwari et al. [Tiw14] discuss lazy checkpointing to reduce overhead and describe temporal characteristics of failures in multiple HPC systems. Failure characterization [Gup17; Gup15] has been studied to understand the requirements of exascale computing, which provides important statistics for system characterization to help understand the challenges before embarking on log mining-based resilience studies.

Anomaly Detection in Distributed Systems: Log mining-based performance diagnosis on cloud computing systems [Xu10; Zha16b; Yu16] has garnered considerable attention recently. The solutions do have some similarities with the HPC resilience frameworks (e.g., log parsing, applied ML techniques). One concern with these solutions making them incompatible with HPC systems are fault injections [Dea12; Yu16; Du17; Pit17] and source code referenced log statement parsing or augmentation [Xu10]. Moreover, logs from distributed systems such as HDFS and OpenStack are at a much higher level in the software stack than HPC architectures. Hence, failure prediction gets complex with diverse log sources compared to per-log level anomaly identification. Data center log analytics based on workflow monitoring [Yu16; Cho17] and system tracing [Che02] differ in their log analysis objective (e.g., lack of lead time sensitivity study, retaining component location information), distinguishing them from Desh.

In summary, failure prediction is considerably researched. Our novel contribution is an efficient

way of predicting lead times in contemporary HPC systems, which may pave way for further analysis of deep learning-based failure prediction.

3.6 Conclusion

Desh provides a powerful technique to process HPC logs using LSTM for efficient failure prediction. Desh uses a novel three-phase deep learning approach to first train to recognize chains of log events leading to a failure, second re-train chain recognition of events augmented with expected lead times to failure, and third predict lead times during testing/inference deployment to predict which specific node fails in how many minutes. Desh obtains more than 2 minutes average lead times with an F1 score as high as 89.88%. Our lead time sensitivity study and its correlation with diverse failure classes can aid system designers comprehend what is required for faster prediction in the upcoming exascale era. We use actual field data from supercomputing sites without any failure injection for anomalies. Our insights can have implications on real-time approaches required for quick anomaly detection online, on the significance of novel phrase mining paradigms befitting for contemporary large-scale computing systems and on new statistical paradigms to leverage unknown log phrases in system anomaly detection.

CHAPTER

4

MAKING REAL-TIME NODE FAILURE PREDICTION FEASIBLE

4.1 Introduction

The research community has solved pertinent problems related to failure prediction for enhanced system reliability. Even for the contemporary HPC clusters such as Cray systems (e.g., Titan, Trinity at National Labs) unstructured log mining-based failure characterization [Bra15; Alv16; Gup17] has been investigated. Be it system failures, such as memory/DRAM errors [Hwa12; Sri15; BG16], hardware failures [Wan17a], software bugs [Gup15], GPU errors [Tiw15; Nie17] or application failures [Mar15b; Ash18], the past decade has contributed to automated system resilience via machine learning and gaining a better understanding of system logs. Such field data analysis has revealed interesting statistical properties of log events including failure distributions [Bir14], their spatial and temporal correlations [Gup15], the effect of temperature and power consumption on reliability and performance [El-12a; Nie17] to identify how faults manifest that lead to failures, their repair times, and root causes. Consequently, ML- and deep learning (DL)-based anomaly detection and failure prediction solutions have been formulated [FX07; Lan10b; Gua11; Ber14; Fu14; Pit17; Du17; Tun17] incorporating techniques such as clustering, SVM (support vector machines), PCA/ICA (principal/independent component analysis), Bayesian models, decision trees, signature extraction, and neural networks.

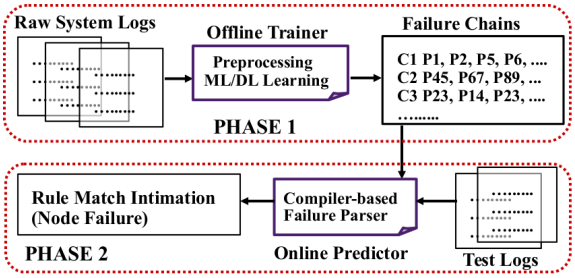

Figure 4.1 Two Phase Failure Prediction

However, the efforts invested in unveiling the wealth of information from ML-based studies will pay off only when we take steps to build realistic frameworks for online failure prediction such that the overhead of costly checkpoint/restarts and wastage of compute capacity due to recalculations and waiting is reduced. In the upcoming exascale era with a quintillion (10^{18}) floating point operations per second [Ecp], scaling the proposed solutions to work efficiently is critical. Vendors are expected to handle higher failure rates with decreasing mean time between failures (MTBF) in the order of minutes [Cap09; Gar14]. Moreover, higher component density (e.g., 10^5 nodes with GPUs and 1 TB RAM each), which require a shorter optimal checkpoint interval [Tiw14], and constantly evolving system architecture [Bra15] impose additional challenges for timely resilience in practice [Sni14].

To this end, we propose Aarohi[1], an efficient *node failure predictor* that promises to prevent impeding failures of computing systems online and in real-time. Aarohi predicts failures by analyzing logs on average in 0.31 msecs for a phrase chain of length 18, i.e., a speedup of over 27.4X w.r.t. the existing state-of-the-art [Yu16; Du17]. Our solution is applicable to any ML-based pre-trained chain of events leading to a node failure. Chain construction with precursory events of propagating anomalies for failure detection has been demonstrated for HPC systems [Sni14]. Even with an improved recall and precision [Fu14; Du17; Pit17; IM17] obtained by ML-based solutions, the inference time (not mentioned in some works [Gua11; GF13; Fu14; IM17]) may not be short enough to enable real-time failure prediction. This paper focuses on transitioning from an offline trainer irrespective of its algorithmic complexity to a scalable, adaptive online predictor. This predictor is automatically generated from the specifications obtained during training and, due to this automation, even may be dynamically updated if new training data becomes available. Figure 4.1 shows the two phases of failure prediction. First, the offline training phase of system logs achieves high recall and accuracy in formulating failure chains. Second, the online prediction phase strives to enhance the inference speedup while achieving sufficient lead times to failure. This paper is not about Phase 1, Aarohi's novelty is in automatically generating an inference tool for Phase 2 based on failure patterns identified in Phase 1.

[1] *Aarohi* means *ascending* in the *Sanskrit* language. It personifies the gradual event-wise progression towards successful failure prediction.

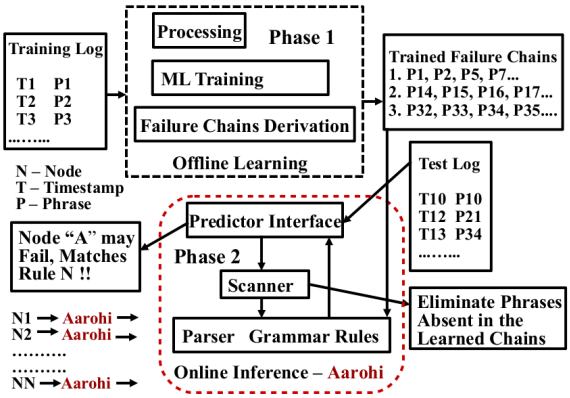

Figure 4.2 Overall Design

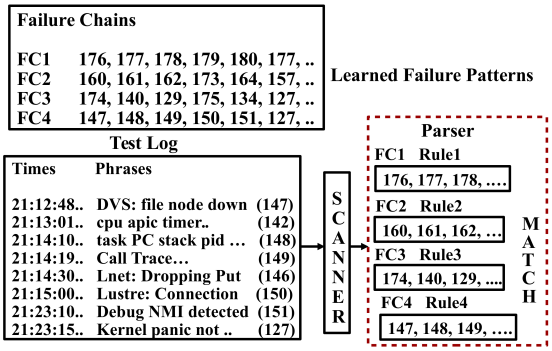

Figure 4.3 Aarohi Design

4.2 Background

Performance optimized training has produced high recall rates and accuracy [Du17]. However, once failure indicators have been trained, inferring impending failures from the new test data is not fast enough to aid real-time prediction. Even though improved learning-based solutions exist, practical deployment of such schemes for successful online prediction requires fast mechanisms to reduce the failure inference time for proactive fault tolerance. As an example, DeepLog [Du17] and Cloudseer [Yu16] incur 1.06 and 2.36 msecs, respectively, to check a single log message for online detection using techniques of LSTM and task automatons. For a failure chain of length 10, they might require as much as ≈10.6 and ≈23.6 msecs, respectively. Yet log messages can be as low as μsecs apart in time. Can we predict anomalies any faster? To what extent? We address this challenge by automatically generating an online predictor from training-derived event patterns in an adaptive, generic and fast manner.

Challenges: While *many* failure prediction solutions have been proposed, most of them cannot be used online [Lan10b] to take timely proactive recovery actions (e.g., job migration, process cloning [RM15]). Some of the hurdles for real-time failure prediction in large-scale computing systems are as follows:

1. ML-based schemes are effective offline trainers, but their analysis speed is unsuitable for real-time failure mitigation. Even with accurate learning and acceptable lead times to failures, slow inference with insufficient speed may fail to finish proactive actions before the fault freezes a component.
2. The pace of analyzing incoming event logs by the predictor should be compatible to the inter-arrival times of the adjacent logs in the system (msecs versus μsecs).
3. An online predictor should be reusable with evolving event patterns and diverse system types. It should accommodate software and logging paradigm variations with minimal overhead, without receding efficacy over time. This is non trivial since upgrades and new systems introduce new features [Bra15] and unseen log messages, thus new failure patterns emerge. Apart from re-training,

a robust predictor needs to be adaptive and portable so that the core prediction scheme remains functional and the approach becomes sustainable across systems.

Table 4.1 illustrates that there exist differences in logs obtained from Cray XC versus XE systems. In fact, even within the Cray XC series (XC40/XC30), different HPC sites can produce different logs based on their specific hardware and software. This elucidates that log upgrades are common, and even similar systems belonging to the same family (IBM, Cray) possess variations in their logs since they contain vendor-specific templates [Bra15] and feature diverse rates of logging.

Table 4.1 Log Variations

Features	**Cray XC40**	**Cray XE**	**Cray XC30**
Processor	Haswell, KNL	AMD Opteron	Haswell, IvyBridge
Burst Buffer, Job Scheduler	Yes, Slurm	No, Torque	No, Slurm
Interconnect	Aries (DragonFly)	Gemini (Torus)	Aries (DragonFly)
System Log Data	Controller (bcsysd), Boot-logs, SEDC differ from XE	Controller (syslog-ng), Boot-logs, SEDC differ from XC	Controller (bcsysd), Boot-logs, SEDC differ from XE

Contributions: Aarohi automatically generates a fully unsupervised parser from a DL-based training. It aims to provide significantly faster failure prediction via novel event stream parsing of phrase chains. This paper makes the following contributions:

1. We propose an efficient predictor, which can proactively flag failures in online streamed test data using grammar-based rules. The predictor works with trained failure chains, which are confirmed patterns of node failures (derived in consultation with experts and system administrators).
2. We describe the process for translating a set of failure chains identified via ML for a specific system to a set of grammar rules. This is a generic approach that can be adapted to any system (with specific failure definitions) as it automates the process of rule generation.
3. We illustrate how our predictor adapts to different systems and incurs low overhead for log variations across diverse system types. This demonstrates that the overall mechanism is robust for cross-system portability.

Table 4.2 System Logs

System	**Time Span**	**Size**	**Scale**	**Type**
HPC1	5 months	150GB	5576 nodes	Cray XC30
HPC2	6 months	98GB	6400 nodes	Cray XE6
HPC3	8 months	27GB	1630 nodes	Cray XC40
HPC4	6 months	15GB	1872 nodes	Cray XC40/30

Log Details: The system logs used for the study have been obtained from 4 HPC systems. Table 4.2 enumerates the system details including the duration of log collection, size and system scale. These are production HPC clusters serving millions of compute node hours. Offline training uses these logs to identify node failure patterns for later online prediction. Once patterns of failure chains are

Table 4.3 Log Message Processing

Timestamp	Phrase	ΔT (secs)	Token
04:58:57.640 (T1)	[Firmware Bug]: powernow k8: * (P1) E	0	<T1 174>
04:59:06.317 (T2)	DVS: verify filesystem: * (P2) U	8.323 (T2-T1)	<T2 140>
05:00:26.823 (T3)	DVS: file node down: * (P3) U	16.506 (T3-T2)	<T3 129>
05:00:51.669 (T4)	Lustre: * cannot find peer * (P4) U	24.846 (T4-T3)	<T4 175>
05:01:14.297 (T5)	Lnet: critical hardware error: * (P5) E	36.372 (T5-T4)	<T5 134>
05:03:24.403 (T6)	cb node unavailable: (P6) E	130.106 (T6-T5)	<T6 127>

learned from the training data, we discuss how Aarohi infers failures from the test data efficiently for proactive counter measures to be completed before a node seizes to respond.

Node Failures: Aarohi predicts node failures. Various system faults can lead to node failures. Root cause diagnosis and the intricacies of DL-based training are not the main focus of this paper. We build on prior work that identifies failure chains [Yu16; Das18a]. The novelty lies in the second phase, the automatic generation of the prediction engine. Node failures are anomalous node outages caused by hardware, software or application malfunctioning. We exclude intentional node shutdowns that are maintenance related or periodic shutdowns and reboots triggered by the operator. We further confirmed normal symptoms and abnormal ones, i.e., *failed messages* in the logs, by consulting with the system administrators.

4.3 Online Failure Prediction Design

Figure 4.2 depicts the overall design of the node failure prediction scheme, Phase 1 for offline learning followed by Phase 2 for online prediction. We briefly outline the offline training phase (obtained from [Das18a], see there for details) used before detailing the online prediction scheme, which is our contribution.

DL-based Training (Phase 1): The following steps summarize the use of an LSTM-based approach to learn from raw system logs such that it can infer node failure chains:
1. The raw logs are trained using LSTM to learn the message patterns based on the past history of the training data.
2. In consultation with the system administrators, the messages that are definitely not benign (e.g., erroneous or unknown) along with *failed messages* (e.g., Node X is down and will not be checked, cb_node_unavailable) corresponding to anomalous node shutdowns are segregated a priori to identify node failures. Based on this phrase labeling, sequences of events that lead to node failures are formed from Step 1. The rest of the phrases are omitted from consideration while forming the failure chains. Thus, we have node failure chains (FCs) composed of anomaly relevant phrases.

Table 4.3 shows 6 phrases leading to a node failure (FC3 of Fig. 4.3) of which P1, P5, and P6 are erroneous while the rest are unknown. All messages are pertaining to a specific node (e.g., c0-0c2s0n2), but the node identifiers are removed here for brevity. For a specific node, Table 4.3 depicts

the calculated ΔTs in the 3rd column computed from the adjacent phrases. We use the discussed LSTM-based training methodology [Das18a] for Phase 1. Notice that any learning technique will work as long as the predictor can be fed with a sequence of coherent phrases leading to failures, i.e., our approach is not dependent on LSTM, rather it works generally for failure chains. For a specific system, how a failure is defined, a chain is formed, and what technique is used for deriving the chain may vary and does not affect the predictor. Of course, higher accuracy of the failure chains implies better prediction efficacy.

Predictor Design (Phase 2): The online prediction method, Aarohi, consists of the following major steps:

1. The incoming stream of log events with their timestamps are scanned through regular expressions (RE) via the auto-generated rules of the scanner. These phrases are tokenized and sent to the parser. The events that do not appear in any of the trained failure chains (FCs) are skipped as those are of no interest to the predictor.

2. The parser consists of rules expressed in a context free grammar (CFG), which are directly and automatically generated from the learned failure patterns (Phase 1). Incoming events are checked against these rules to predict future failures. The rules are formulated based on the message sequence and the time difference between two adjacent events. The latter captures the contextual relevance of events over time. This token-based rule check is fast and is the source of our speedup during the inference phase.

Table 4.4 Parser Grammar

Notations	**Meanings**
G = (N, T, P, S)	LALR(1), 1 Lookahead, Start Symbol S
N	Non-Terminal Symbols
T	Terminal Symbols
P	R=N∪T, P⊆R^+ (Production Rules)
FC1 FC5	(176 177 178 179 180 137) (172 177 178 193 137)
P_FC	S: (176 177 178 179 180 137) | (172 177 178 193 137)
P_LALR	S→(176 C 137) | (172 C 137) C→(B 179 180) | (B 193), B→(177 178)

For each node in the cluster, we dedicate a predictor instance that processes messages of that node only (see Fig. 4.2). The Token column in Table 4.3 illustrates the handling of tokens corresponding to the timestamps and phrases. Figure 4.3 demonstrates how multiple rules of parsers are checked for diverse FCs over a stream of log events. The CFG rules are automatically derived from the learned FCs (e.g., FC1 - FC4). The test data is tokenized, and phrases not appearing in the set of learned failure chains are removed (i.e., 142 & 146). Then, a specific rule is selected based on the starting phrase (token match). For the example in Table 4.3, the sequence matches FC3. This

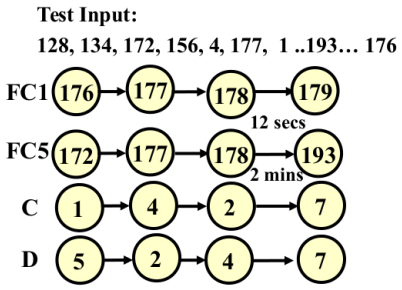

Figure 4.4 Phrase Chains

continues until one of the two conditions are met: a) no more phrases are left in the test data, or b) a match has been found. If a match is found before the test data ends, the parser resets, beginning with the first event phrase seen after the last phrase of the matched failure chain.

From Failure Chains to Rules: Table 4.4 defines a CFG G with a set of nonterminal symbols (N), terminal symbols (T), start symbol (S), and production rules (P). The start symbol is a non-terminal and leads to productions with leading terminals of FCs. The context free productions (P) are a subset of rules (R) that are formed as a union of nonterminal and terminal symbols. We have 1 lookahead, i.e., every phrase in the sequence is checked one at a time to decide on the parser action and select a production. Figure 4.4 highlights certain features of the observed sequence of phrases (FCs):
1. FCs usually have short subchain matches (e.g., 177 & 178 in FC1 and FC5), and they may end with a common failed message (e.g., 7 in C and D).
2. The ΔTs (time difference) between adjacent phrases are usually $<$ 2 mins. Figure 4.5 depicts the cumulative phrase arrivals for nodes A and B on a log scale with inter-arrival times in msecs. For A, 92.05% (278) phrase arrivals have $\leq$ 2 mins ΔTs (order of μ & msecs) in sample data of 302 phrases, and $\approx$13 of them have ΔTs $\geq$ 17 mins. For B, 98.6% arrivals happen within $\leq$ 1.1 mins in sample data of 71 phrases. For a single day, while B's logs spanned across $\approx$3.5 hours, A's logs stretched $\approx$8.75 hours (hence, the higher count). Similar trends are observed at other time frames for other nodes as well. As seen, sometimes there exist steep rises in cumulative arrivals for certain ΔTs (e.g., A: 51 & 19, B: 9 & 11 arrivals for ΔTs of 25 & 26 msecs, respectively). These are phrases pertaining to DVS (Data Virtualization Service), Lustre or LNet. The routing latency from a remote service (DVS) can cause intermittent delays in phrase arrivals within a burst of messages from the same log source. The filesystem/interconnect related delays can be caused by various environmental factors. A defined timeout can help identify unexpected delays during parsing (e.g., 4 mins when 93% of the phrase inter-arrival times are $\leq$ 4 mins) based on such observed ΔTs (checked by semantic actions).
3. Common subchains exist but the starting phrase is usually different for FCs. Few common phrases/swaps (e.g., 2 & 4 in C and D) may occur in sequences of phrases over time.

The chain of terminal symbols leading to an accept state is the distinguishing feature of any system-defined failures. To clarify, we used *failed message* earlier to refer to typical node shutdown

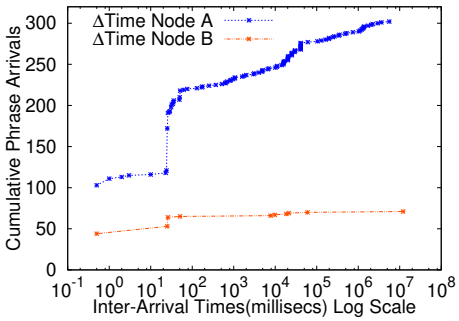

Figure 4.5 ΔTimes

messages (e.g., P6 in Table 4.3). The chain of terminal symbols of our grammar, G, refers to any relevant message of an FC (e.g., 172, 177, ...in FC5), not just the last phrase/event of an FC (e.g. 177, 157, 127 of FC1 - FC4 in Fig. 4.3). From these chains (e.g., FC1 and FC5 in Fig. 4.4), the nonterminals are derived automatically by combining common sequences of terminals (e.g., B→(177 178) of P_LALR in Table 4.4). Since prefixes and common phrases exist in failure chains, we formalize our parser as an LALR(1) [Aho86] grammar. The derivation of grammar rules is shown in Table 4.4 (P_FC and P_LALR) for chains FC1 and FC5.

Tokenization: During failure inference, log messages are processed a single line (event) at a time and parsed adhering to the templates of the FCs from Phase 1. As an example, consider the following two phrases from the test logs:

P1: *DVS: verify filesystem: file system magic value 0x6969 retrieved from server c4-2c0s0n2 for /global/scratch does not match expected value 0x47504653: excluding server. Ensure file system is mounted on the server and then restart DVS*

P2: *pceport 0000:00:03.0: [12] Replay Timer Timeout*

The first input phrase is parsed until it reaches `DVS: verify filesystem:`, which matches a template of a log message pertaining to some FC (e.g., P2 in Table 4.3). The remaining content has variable components, such as `0x6969`, node id `c4-2c0s0n2`, or directory path `/global/scratch`, none of which are further considered. Based on this template match, the corresponding token (say 140) is used for matching parser rules. For the second phrase, `pcieport...Timeout`, the parser checks up to the end of lexical rules, finds no match with any FC-related template, and discards it. Raw log tokenization and rule check-based inference are closely integrated in Aarohi, unlike prior online log parsers such as Spell or Drain [Zhu19]. Similarly, the remaining templates of the FCs are considered by parsing the tokens emitted by the lexer when receiving log input and then used to flag predicted node failures.

Consider Figure 4.4 for a test input 128, 134...4,..., 176. The scanner discards phrases 128, 134

and similar unrelated messages during tokenization since they do not match any of the FC-related phrase templates. Next, phrase 172 matches the starting phrase of FC5. Hence, that rule, 172, 177, 178..., is checked. The parser continues until a mismatch is encountered when it finds 4, another event that is tokenized as it occurs in chains C and D, yet it expected 177 as the next token (phrase) as per FC5. The parser skips such mismatches and continues parsing until a ΔT-based threshold violation occurs, or it reaches the end of test log. This is important as the test data can contain messages that have never been seen before in any FC during training. We transform common prefixes into a singular nonterminal production rule up to the end of the common substring. Based on left-to-right token matches, a matching rule is parsed until a rule is fully matched, i.e., at token 193 in the example. A regular parser would exit after reaching this accept state. Our parsing harness, however, proceeds with the next token, 176, and invokes another parsing instant, in this case to check if rule FC1 matches.

Interleaved Rule Matches: In theory, an incoming set of phrases can match any of the failure patterns recognized in the past. This necessitates the need to simultaneously check for multiple FCs because log events of a single node can match a rule partially when, before reaching an accept, tokens pertaining to another rule may be encountered. Aarohi's set of rules each match a unique failure chain with the ability to start/stop an FC based on which event (token) is encountered next. While evaluating a specific rule, say FC5, during testing, the following cases may occur:
1. The first phrase of another rule (say 1 of C) matches the incoming phrase, but the parser continues to check FC5. If the test data does not match FC5 completely (only partially) but could have matched C, then Aarohi misses this match with C, which would result in a false negative. This is the case for any partial rule match token-wise interleaved with another full rule match.
2. Tokens may be intertwined across rules, say by alternating tokens from FC5 and C, such that both rules could be matched. Here, only the first rule with a token match results in an accept while the other rule is never parsed, also resulting in a false negative. However, the first match already indicates a failure and thus subsumes a subsequent failure during the same time frame, i.e., the false positive of C is irrelevant for our application scenario for lagging node failures. Notice that there are no cases where this parsing methodology results in false positives.

In practice, we found that case 1 does not occur in the inspected test logs (see Table 4.5) but case 2 is seen, e.g., interleaved tokens from FC5 & C are observed. Although, case 1 is theoretically possible, it was not observed because:
1. Healthy node events tend to present a mismatch for learned FCs pertaining to failures. Unhealthy nodes experience a complete match with FCs with only rare cases of interleaving. Recall that the earlier a rule matches, the larger will be the remaining time for proactive measures irrespective of future rule matches if a node is going down.
2. Occasional interleavings exist (Table 4.5), but once a specific rule has been partially matched, it tends to be safe to skip subsequent rules as our inspection shows that the first rule tends to match,

and later phrases do not tend to lead to cases with true positives (full failure chain match).

Table 4.5 Multiple Rule Matches

System	Duration	Missed Rules	Interleaved	#Nodes
HPC1	4 mons	No	Yes	23
HPC2	3 mons	No	Yes	19
HPC3	3 mons	No	Yes	15
HPC4	4 mons	No	Yes	20

Table 4.5 provides empirical evidence of the absence of cases where multiple rules match completely in close temporal proximity (with interleaving) in our data. In the systems studied, either unhealthy node logs do match the FCs or, once an FC match starts, other interleaved FCs do not result in a false negative. However, there can be cases in the test data, where a new pattern (never seen in the training data) is encountered, which would be missed. But this is an inherent problem of any training-based scheme. On most nodes, partially matched FCs and occasional interleavings of different FCs are seen, but most frequently do not result a complete match (true negative). More commonly, different nodes may fail simultaneously in time when matching the same FCs, or a node may fail successively over *different time frames*. This substantiates that our scheme well suffices, and that we are not missing imminent failures. Since partial matches do not enhance Aarohi's resilience capability, we chose a simple, yet effective rapid inference model implemented by this parsing methodology.

Algorithm 1 enumerates the steps in automatically translating a generic set of FCs to parser rules. This translation is performed offline, i.e., its time complexity is not the critical path. Assume we have a set of FCs from the output of Phase 1 training for an HPC cluster. The distinct phrase templates (e.g., Phrase column in Tab. 4.3) encompassing all the FCs are enumerated uniquely. Each phrase ID is then tokenized by assigning (#5) a global token (e.g., {101 102 ...} → {P1 P2 ...}) forming a token list. Unique rules are formed with the corresponding tokens (#6) based on the sequence of phrases in the FCs, such as:

FC1: {123 135 ...} → R1: {P23 P35 ...}, FC2: {141 152 ...} → R1: {P41 P52 ...} ...

With a similar single chain rule set (#8), recursive rules (#15, #16) can be derived by substituting subchains (#14), if any, between multiple rules. These common subchains form the non-terminals of the LALR(1) grammar (see P_LALR in Table 4.4).

Algorithm 2 encapsulates Aarohi's operation during test data inference. Once the parser is invoked, each phrase in the test input is tokenized (#6) to check if it matches any of the tokens of the Token List generated from Algorithm 1. On a match, the token with its arrival time is sent to the parser (#7), else it is irrelevant and discarded (#8). For parsing, the Rule List obtained from Algorithm 1 is used with suitable semantic actions (R^+, #9, #10) such as:

R1: {P23 P35 P45 ...}, R2: {P41 P57 P62 ...} ...

Algorithm 1: From Failure Chains To Parser Rules

input : FC List
output: Rule List

```
1  T ← ∅, S ← ∅ // T ← Token List, S ← Rule List
2  foreach (FC ∈ FC List) do // Failure Chain
3      R ← ∅
4      foreach (Phrase ∈ FC) do // Form Token List
5          if (Phrase ∉ T) then T ← T ∪ Phrase
6          R ← R ∪ Phrase // Unique Chain Rule
7      end
8      S ← S ∪ R // Rule List from Unique Rules
9  end
10 // Derive LALR(1) Rules from Rule List S
11 foreach (V ∈ S) do
12     foreach (U ∈ S) do
13         if (V ≠ U) then // Substitute Subchain
14             foreach (C = Subchain(U, V)) do
15                 V' ← head(V) ∪ C ∪ tail(V)
16                 U' ← head(U) ∪ C ∪ tail(U)
17                 S ← (S \ {U, V} ∪ {U', V'}) // Update S
18             end
19         end
20     end
21 end
```

For the first match, an incoming token matches with one of the rule's first token (say rule R1), and that ($R1^{th}$) rule is checked. On an error (i.e., incoming token differs from expected token), if the difference between the current time and the last matched token's arrival time (ΔT) exceeds a predefined threshold, the parser aborts (#13), else it continues (#12), because, in practice, inordinate delays between incoming phrases of known failure chains do not belong to the same failure pattern. Similar semantic actions continue until a reset is triggered or no more phrases are left in the test log. Skipping tokens (#12) is essential for rule checking to discard the non-relevant phrases in between FC-related phrases. Multiple rule matches may occur back-to-back in an input stream. For any remaining unprocessed phrases in the test data, the online prediction continues from the phrase appearing after the last phrase of rule R1 in the Test data (after a match, #10) or the last processed token before reset (#13). This heuristic's complexity is linear in input size (i.e., log file) and, together with grammar-based parsing, aids in Aarohi's inference speedup w.r.t. the existing detection schemes.

Figure 4.6 summarizes the workflow required for Aarohi to facilitate transitioning from offline training to online prediction for any HPC system. Training Phase 1 produces FCs, which, when run with Algorithm 1, produce parser rules. Algorithm 2 with equivalent grammar rules, appropriate

Algorithm 2: Aarohi Prediction

input : Test Data, Token List T & Rule List S from Algo. 1
output : Matched FC Rule

```
// Online Inference
while (Test Data≠ NULL) do // Incoming Phrase
    Parse(Test Data) // Call the parser
end
// Parser Rules
Token ← Phrase // Tokenize
if (Token ∈ T) then return Token + Arrival Time // Relevant Token
else Skip Token (not relevant)
R+ ← S // Rule List, e.g., R1
P1 error P2 error P3... ← Sequence Matched Rule 9
if (∃ error) then // Mismatch while parsing
    if (ΔT≤Timeout) then Skip Token, Continue
    else Reset after Current Token #P // Restart
end
if (Test Data≠ NULL) then // On a Reset
    Parse(Test Data) // Start after Token #P
end
```

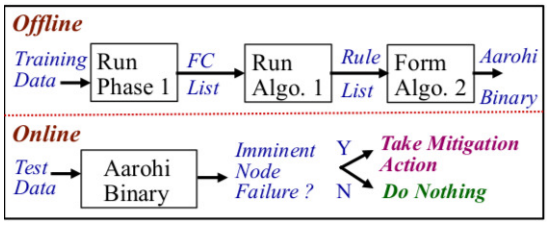

Figure 4.6 Offline Training to Online Testing

error handling, and other semantic actions (R^+) produces the Aarohi binary. Aarohi is run with new test data for online prediction to flag immanent node failures on a rule match, and only on such a match a suitable mitigation (proactive/reactive) action needs to be triggered.

4.4 Evaluation

Aarohi is implemented using the flex/bison parser in C++. Our FCs contain sparse subchain matches for which non-recursive chain rules suffice. Aarohi's token handling ensures continuation of parsing by skipping unexpected phrases in the test stream with appropriate semantics. Performance of parsing is reported for an Intel quad core processor running at 2.83 GHz. *The test log data used for prediction is different from the training data used for learning the FCs for any of the systems.*

We report prediction times, i.e., the time taken to check if a variable length sequence of phrases (not a single log message) matches any of the FCs. The inference times are obtained with compiler optimization level O3 enabled and trace output for debugging disabled. From the timestamped

Table 4.6 Efficiency Formulae

Formula	Implication
Recall(%)=TP/(TP+FN)	Fraction of node failures *correctly* identified
Precision(%)=TP/(TP+FP)	Fraction of node failures predicted
Accuracy(%)= (TP+TN)/(TP+FP+FN+TN)	Fraction of correct predictions in the entire set
FNR (%)=FN/(TP+FN)	Rate of missed failures
True Positive (TP)	Correctly predicted failures
True Negative (TN)	Correctly rejected as not failures
False Positive (FP)	Incorrectly predicted failures
False Negative (FN)	Incorrectly rejected as not failures

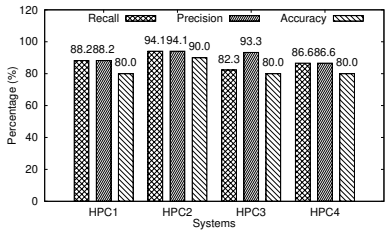

Figure 4.7 Phase 1 Efficiency

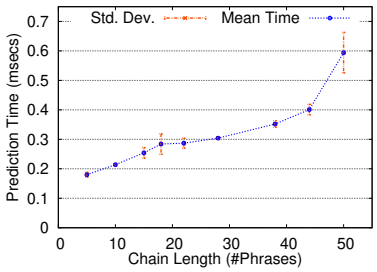

Figure 4.8 W/O Benign Phrases

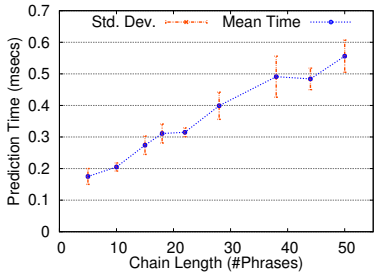

Figure 4.9 With Benign Phrases

node *failed message* in the test data to the event phrase at which the predictor flags match, we compute the expected lead times to imminent node failures. Aarohi's evaluation metric is predictor speedup over high accuracy in the context of real-time failure prediction.

Table 4.6 lists the standard efficiency metrics such as Recall, Precision, Accuracy and False Negative Rate (FNR) with their formulas. The terms TP, FP, TN and FN are defined in the context of node failures. Figure 4.7 shows recall, precision and accuracy obtained in Phase 1, implying its overall efficiency in predicting node failures. For the considered node failures in each of the 4 systems, the false negative rate ranges from 5 to 17.6%. This does not affect the prediction times in Phase 2, but it is indicative of the efficacy of FC-based rules used for inference. The precision exceeds 86% in all cases.

Observation 1: *Recall, precision, and accuracy exceed 86%, 88%, and 80%, respectively, across all 4 systems with a moderate false negative rate below 18%.*

Prediction Time: Figure 4.8 shows the prediction times of 9 phrase chains. The test data contains phrases that exist in some FC. In this case, the parser skips a token on a mismatch unless any of the termination conditions are met. Aarohi takes 0.18 msecs to 0.6 msecs for chain lengths ranging from 5 to 50 with a std. deviation $\leq \pm 0.068$ msecs. Figure 4.9 depicts the prediction times with log messages that include benign phrases that are not part of any FCs (as they appear in the test logs). These get discarded by the scanner without tokenization. In such a case, Aarohi obtains inference times ranging from 0.17 to 0.56 msecs with a std. deviation $\leq \pm 0.065$ msecs. This is a realistic case

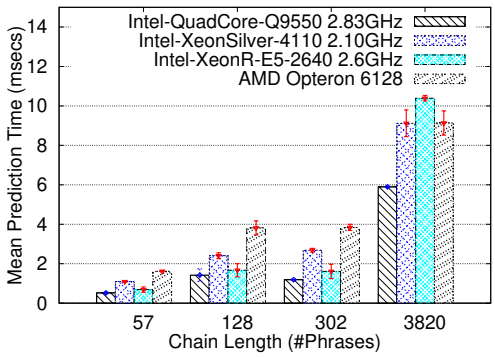

Figure 4.10 Diverse Platforms

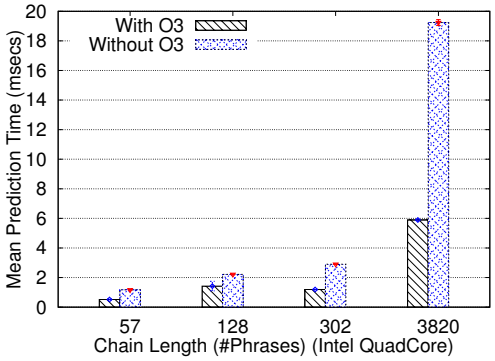

Figure 4.11 O3 Optimization

containing many benign phrases and some FC-related phrases. These times are comparatively lower than the former because, in the previous case, all phrases are tokenized but later skipped by the parser during rule checking.

To understand the variation of inference times over different CPU architectures, for increasing chain lengths, we ran experiments on Intel Quad Core, Xeon, Xeon Silver and AMD Opteron platforms. Figure 4.10 shows that Opteron takes more time than the Intel platforms, however, with increased number of phrases the difference in prediction times between Xeon, Xeon Silver and Opteron is less than 2 msecs. Overall the std. deviation do not exceed ±0.67 msecs. Please note that an increase in chain length or number of log messages in a sequence do not necessarily indicate higher prediction times. The individual phrase size varies (e.g., P1 & P2 under *Tokenization*) along with the proportion of FC-related phrases. This is why Aarohi checks a 302-length chain in less time than one of 128-length (see Tab. 4.7). The former contained long phrases in most lines unlike the latter, which had short phrases including call traces. Figure 4.11 shows the difference in prediction times with and without O3 optimization. For a 302-length chain, 58.96% improvement is observed (2.9 msecs to 1.19 msecs). For a stream of 7443 messages, with and without O3 flag takes 45 msecs and 77 msecs, respectively. Their corresponding sizes vary between 4K to 712K.

Table 4.7 Speedup

Approach	**Prediction Times (msecs)**				
	Chain Lengths				
	1	10	50	128	302
Aarohi	0.05	0.205	0.556	1.427	1.1904
Desh	0.12	1.856	8.761	19.356	32.681
DeepLog	1.06	10.6	53	135.68	320.12
CloudSeer	1.81	18.1	90.5	231.68	546.62

Observation 2: *Aarohi consistently obtains less than 11 msecs inference times across diverse CPU*

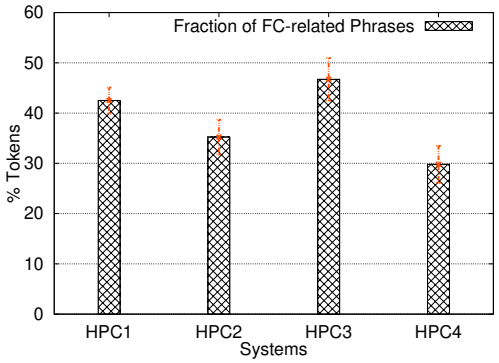

Figure 4.12 Token Fraction

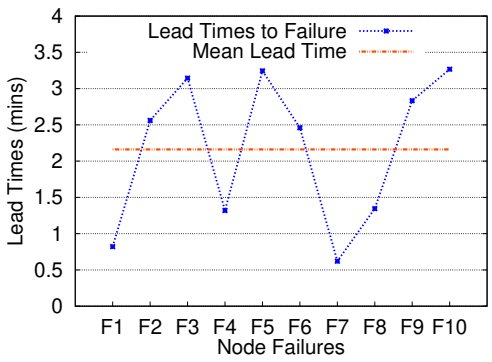

Figure 4.13 Lead Times to Failures

platforms. Such a compiler optimized approach enables rapid inference for diverse chain lengths with varying message sizes. Speedup is not linear w.r.t. the chain length or message size, hence, per log entry times are not appropriate indicators of time complexity.

Recent anomaly detection solutions [Yu16; Du17; Das18a] report the testing times of a single log entry. Cloudseer [Yu16], Desh [Das18a] and DeepLog [Du17] are suitable candidates for comparison since they employ contemporary techniques, such as automatons and LSTMs for log sequence analysis. Other ML-techniques are expected to consume more time. Aarohi's metric is a chain of messages, as opposed to a single anomalous message. Table 4.7 highlights that Aarohi is considerably faster than Desh, DeepLog and Cloudseer for increasing chain lengths. Lack of similar system logs and source code make an exact comparison difficult. However, by actually implementing Desh (as in the paper), we obtained 2x-27.4x improvements. Also, the reported times in DeepLog and CloudSeer are for single log entry checks pertaining to anomalous messages. If events are optional and are skipped, additional time will be incurred. Moreover, it is not clear if raw log tokenization time has been accounted in prior work. Differences in speedup become longer and discernible with increasing chain lengths. For a 302-length chain Aarohi is 27.4x faster than Desh (32.68 msecs to 1.19 msecs). Python-based frameworks with ML-libraries are slow runtime interpreters compared to C++, facilitating Aarohi's speedup and making it suitable for online prediction.

Observation 3: *For chain lengths 1 to 302, Aarohi is over 27.4x faster than the current state-of-the-art approaches underlining Aarohi's potential for real-time failure prediction. The speedups increase with chain lengths.*

Figure 4.12 shows the fraction of phrases in the test data that correspond to some FC across all the systems. As seen, most phrases are dissimilar to FC-related phrase templates. This is because healthy node logs dominate and their phrases are not part of any FCs. The percentages of phrases

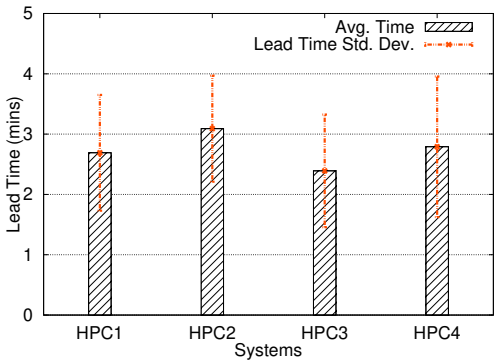

Figure 4.14 System Lead Times

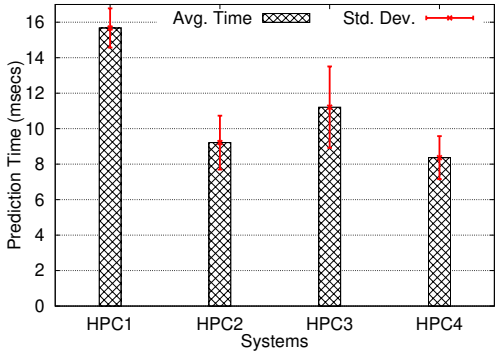

Figure 4.15 System Prediction Times

pertaining to any FC-related token range from 29.81 to 42.51%. This determines the portion of messages discarded during lexical scanning.

Observation 4: *The fraction of FC-related phrases eventually tokenized are below 47% in the test data indicating that only a fraction of log events need to be tokenized and then parsed during inference from online streaming logs.*

Lead Time Sensitivity: Figure 4.13 depicts the lead times obtained for 10 node failures before the terminal message appeared in the test data. The last phrase matched in the FC corresponds to a timestamp, which is subtracted from that of the eventual node failure message to compute the lead times. Similar lead times are obtained for other node failures. These node failures correspond to chains of length 5 to 50. With prediction times below 0.65 msecs, we can obtain effective lead times (with prediction times deducted) higher than 3 minutes. This means that for failure F5 in Fig. 4.13, suitable mitigation action can be taken in 3.24 mins (3.245-0.0006). The average lead time is more than 2 minutes for node failures across all the 4 systems. The chain lengths do not affect the lead time. However, the time differences between the log messages affect the lead time. Shorter ΔTs in the sequence matched in a FC lead to shorter lead time since the upcoming terminal message is immanent. That does not imply that longer ΔTs indicate higher lead times. ΔTs are a refinement to prevent false positives during inference rather than to increase the lead times to failures.

Observation 5: *With Aarohi, effective lead times to node failures are higher than 3 minutes with an average of 2 minutes. Prediction times are <11 msecs for 3820-length. Flagging the failure at an earlier precursor to a terminal message can further increase lead time, at the cost of possible false positives. Early prediction is required to leave sufficient time for failure mitigation approaches (e.g., process migration) before a node stops responding.*

Cross-System Comparison: Figures 4.14 and 4.15 depict the average lead times and prediction

Table 4.8 Comparative Analysis of Aarohi

Research Solutions	Approach	Unsupervised	Lead Time (mins)	Test Time	Online	Target	Objective
Zheng et al. [Zhe10]	Genetic Algorithm	No	2 to 10	N/A	✓	BG/P	Failure Prediction
Hora [Pit17]	ARIMA (Autoregression)	No	10	98 predictions/2 mins	✓	Netflix	Mem Leak, System Overhead, Node Crash
Perfscope [Dea14]	Signature Clustering	N/A	N/A	9.5 mins (Hadoop Bug)	✓	HDFS, Cassandra	Performance Bugs
Fu et al. [Fu14]	Frequent Episode Mining	No	N/A	N/A	×	Hadoop, LANL, BG/L	Root Cause Diagnosis
Nosayba et al. [ES17]	Random Forests (RF)	No	N/A	N/A	×	LANL, Google	Job Failures
Berrocal et al. [Ber14]	Void Search/PCA	No	N/A	4 secs/node	×	BG/Q	Fault Prediction
DeepLog [Du17]	LSTM	No	N/A	1.06 msecs/log entry	✓	OpenStack, BG/L	Anomaly Detection
CloudSeer [Yu16]	Automatons	N/A	N/A	2.36 msecs/log entry	✓	OpenStack	Anomaly Detection
Desh [Das18a]	LSTM	No	3	0.12 msecs/log entry	×	Cray-HPC	Node Failures
Klinkenberg et al. [Kli17]	Classifiers (RF, MLP etc.)	No	17 & 22	N/A	×	HPC Cluster	Node Failures
Aarohi	Compiler-based	Yes	3	0.31 msecs (length-18)	✓	Cray-HPC	Node Failures

times of node failures across all the systems. While the lead times exceed 2 mins, the average prediction times are less than 16 msecs for over 100-length chains. The standard deviation of prediction times is $\leq \pm 2.3$ msecs, which is higher than in Figure 4.9. This variation is caused by the diversity of node-specific test sequences on different systems w.r.t. their corresponding FCs. The average lead time is $\approx$2.74 mins for all the systems with a moderate standard deviation of $\leq \pm 1.16$ minutes.

Observation 6: *Aarohi obtains more than 2.3 minutes average lead times to node failures with an avg. prediction time of no more than 16 msecs across systems. The lead times are sufficient for proactive recovery actions such as job migration and quarantining unhealthy nodes. The standard deviation of prediction times are higher across systems than across different failure patterns of similar chain lengths. This is due to significant variations in the test sequences relative to the FCs across systems.*

Our work make novel contributions by demonstrating that automated translation of FCs as grammar rules can be used to effectively infer failures and thus enable proactive fault tolerance in practice. CFGs have been leveraged in the past in various contexts [Che04; BG13; Yu16] and online log parsers have been proposed for efficient log analysis [He17; Zhu19]. Aarohi goes beyond these works by closing the manual translation gap between failure chain identification by machine learning and fast parser-based inferencing. Aarohi automatically generates lexing and parsing specifications for a

language of FCs suitable for online failure prediction as part of inferencing, which would also allow Aarohi to be deployed in unsupervised dynamic re-training and re-generation of a new parser for enhanced failure chains as they are being observed. Overall, Aarohi obtains prediction times low enough to provide effective lead times to failures, which is of primary concern.

Comparative Analysis: Apart from DeepLog, Desh and Cloudseer, whose testing times are compared with Aarohi, several other researchers have studied failure prediction. Table 4.8 illustrates how they differ. Most solutions do not perform lead time analysis [Fu14; Ber14]. While some solutions are applicable for both DS and HPC logs [Fu14; ES17], most of the solutions either focus on HPC [Kli17; Zhe10] or data center platforms [Dea14; Yu16]. Approaches such as genetic algorithm, supervised clustering, and sequence mining are not effective for online failure prediction in contemporary systems. Klinkenberg et al. [Kli17] obtain higher lead times for known node soft lockups through supervised classification, unlike generic inference of Aarohi. Although past work stresses on building online solutions [Zhe10; Pit17], lack of lead times and fast-paced prediction mechanism pose deficiencies in practice. Aarohi's contribution can be effective for both cloud and HPC systems for proactive fault management.

Adaptability: In the production HPC clusters, the following cases are prevalent (see Table 4.1 discussion):

1. Systems belonging to different generations have syntactically different but semantically equivalent logs (e.g., Cray XE vs. XC40).
2. Different systems belonging to the same generation but with dissimilar hardware or software have syntactic log variations (e.g., Cray XC30 vs. XC40).
3. Software is updated on a given system after a period of time, which changes the log's syntax (e.g., several Cray systems upgraded to Slurm from Torque as their job scheduler or incorporated burst buffers, an intermediate storage layer).

Table 4.9 Aarohi Adaptability

#	HPC		DS	
	HPC 5 (Cray XK*)	**HPC 6 (IBM BG/P)**	**Cassandra [Cas]**	**Hadoop [Had]**
P1	GPU* PMU communication error	MMCS detected error: power module	Unable to lock JVM memory	No node available for block
P2	L0 heartbeat fault	Network link errors detected	Cassandra server running in degraded mode	Could not obtain block*
P3	Voltage Fault Node	DDR correctable single symbol error(s)	Not starting RPC server as requested	DFS Read: java IOException*
P4	Machine Check Exception (MCE)	Kernel panic: soft-lockup: hung tasks	No host ID found	No live nodes contain current block
P5	Kernel Panic, Call Trace	Kill job * timed out	Exception in thread Thread*)	DFSClient: Failed to connect
P6	GPU* memory page fault	Node System has halted	Exiting due to error while processing commit log	NameNode: shutdown msg:

In face of such software upgrades or log variations, phrase re-mappings and rule updates can suffice, without changing the overall workflow of Aarohi. Table 4.9 enumerates 6 phrases from 4 diverse clusters[2] of which 2 are HPC systems, namely a Cray XK* and a BlueGene/P, and 2 are distributed systems (DS), namely Cassandra and Hadoop. The DS logs correspond to 2 application bugs [Cas; Had]. In practice, distributed system logs do not have node identifiers in every log message as the logs are application layer centric. Assuming required preprocessing and correlation is performed specific to any system and failures, we discuss Aarohi's generic adaptability.

Phase 1 training is necessary for every system. This step is a prerequisite before our predictor adapts to new FCs. As seen, certain BG/P phrases have similar meanings as Cray logs (e.g., P6 in BG/P). In such cases, phrase mappings can be updated in the scanner (e.g., XC: 7→cb node unavailable, changes to BG/P: 7→node system halted) without any change in grammar rules. While some phrases remain the same (e.g., P4 & P5 in XK & XC), few undergo minor changes (e.g., heartbeat failures for Cray XK and XC). The scanner thus requires minor additions and updates. However, for Cassandra and Hadoop, the FCs change due to major log variations. It is not enough to update the mappings as the context differs. In such cases, the scanner produces new tokens and the grammar rules have to be reformulated with the new phrase identifiers (e.g., P1 to P6 in Table 4.9).

Discussion: Let us discuss a number of important considerations for real-time failure prediction.

(1) Predictor Placement: One pertinent question arising in a large-scale cluster is: Where can an online predictor be located for efficient failure handling? Figure 4.16 depicts the high-level overview of HPC vs. data center facilities. Cray systems utilize an HSS manager (hardware supervisory system) to administer the chassis and blade controller that manage the compute and service nodes. Moreover, nodes link to the System Management Workstation (SMW) via managers to collect system logs from the cabinets. HSS is an aggregate workspace over the entire interconnect where logs are accessible. Predictors such as Aarohi can be placed on the HSS network for failure prediction. This helps in two ways: First, daemons running on compute nodes can affect the jobs running on the cluster. Moving away from the compute nodes can eliminate any possible impact on the job resource consumption. Moreover, if resources are scarce or load imbalance arises due to heavy workloads, the overall computation in the cluster may be affected. Logs from different blades are available at the HSS, which could facilitate the online prediction scheme.

In data centers, the aggregation layer is a centralized monitoring and processing layer, which performs batch and event streaming of data in real-time (e.g., Google Cloud Platform [Gcp]). The data centers commonly assume a multi-tier model unlike the server-cluster model of HPC systems, making the design of a global predictor non-trivial. For a data center with 1000s of compute nodes, aggregating logs from all the hosts to a centralized location could throttle the network bandwidth.

[2]Cassandra and Hadoop logs were generated in the lab after bug reproduction, HPC 5 and HPC 6 logs were obtained from researchers who used them in the past.

The bandwidth can be a bottleneck based on the capacity of the switches and the network topology used. Every physical host runs diverse virtual machines, and each compute node can monitor its own health. This can facilitate application-centric anomaly detection, but remains less effective for node failure prediction in the cluster. A centralized predictor may not be effective in such a case. Further deployment experiences, subject to future work, can unveil insights about addressing such practical concerns.

(2) Proactive Recovery Actions: We mentioned that enhancing inference speedup can aid proactive recovery actions such as migration and cloning [RM15]. While Wang et al. [Wan08] show live migration times < 24 secs, Ouyang et al. [Ouy11] demonstrate that process migrations can be completed in 3.1 secs (10x speedup over conventional approaches). Gupta et al. [Gup15] report that quarantining unhealthy nodes can reduce future failures by 5.07% to 7.21%. Apart from live migrations, adaptive lazy checkpointing [Tiw14; ESS14b] schemes can aid mitigating upcoming failures. Shutting down or preventing future job assignments onto flagged unhealthy nodes can prevent future job disruptions, even if the ongoing jobs have to be aborted. In <16 msecs prediction time, and >2 mins *effective* lead time, such proactive recovery solutions become feasible in most cases.

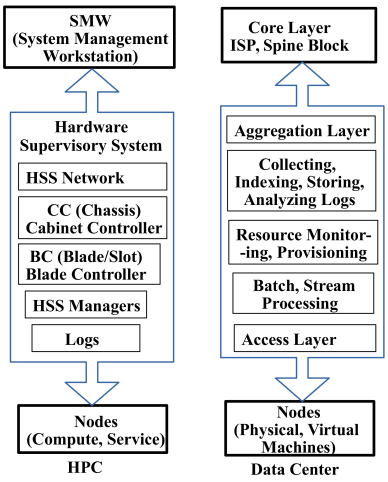

Figure 4.16 Predictor Placement

4.5 Related Work

Scientists have investigated anomalies in large-scale computing infrastructures across diverse research directions. We briefly discuss and contrast them to Aarohi.

Log Parsers: In the context of real-time inference, log parsing or preprocessing contributes to the overall inference time. [Ahm17; He17] have proposed efficient parsing algorithms to reduce the processing time. Efficient parsers require high accuracy along with speedup. [Ahm17] proposes non-domain specific detection model taking ≈11 msecs for an input. Logmine [Ham16b], Drain [He17]

and Spell [DL16] are similar parsers developed for online streaming logs. [Zhu19] uses several recent log parsing tools for online log tokenization and, [Aus18] parse logs without mentioning the timing constraints. Unlike these Aarohi tokenizes and performs rule checks right after, for failure inference.

Log Mining-based Failure Detection: While [FX07; Fu09; Lan10b; Ber14; Fu14; Pit17; Tun17] have concerted efforts in predicting failures in HPC systems, [Gua11; GF13; IM17; Du17] have similar objectives in distributed systems (e.g., Google, OpenStack). [Sal10] describes a detailed taxonomy of all the statistical approaches used in the context of system anomaly detection. ML-techniques used in the past (e.g., PCA, SVMs, Decision trees, Bayesian models etc.) do not scale well and are not efficient for contemporary system logs. [Gai13] performs online failure prediction through correlation extraction without inference time analysis while we do infer time. Klinkenberg et al. [Kli17] propose lead times as high as 22 mins with the use of supervised classifiers, which is less efficient for real-time stream processing. [ES17] predicts job failures indicating machine outages as one of the root causes for jobs to fail for Google and LANL clusters. This asserts Aarohi's importance in timely prediction of node failures.

Online Anomaly Prediction: Several recent studies have focused on real-time anomaly identification. While Hora [Pit17] explicitly uses system architectural information for predicting component failures, PreFix [Zha18b] predicts switch failures in data centers. [Jha13] discusses a high-level fault tolerance approach using existing detection schemes for cloud systems and [SH15] defines independent/dependent variables on synthetic transactions to detect SLA (Service Level Agreement) violations. [Wat12] performs classification and uses severity levels for failure detection which is a simpler failure model (no failure chain analysis) unlike Aarohi. LogLens [Deb18] deploys online sequence anomaly detection scheme using Logmine [Ham16b] for cloud systems without inference time analysis. [Lou10; Bas16a] use source code intrusive program invariants, random forests and density estimation after fault injection for anomaly detection. [Cha12] uses fault trees and ARMA (Autoregressive Moving Average) methods for failure prediction in a simulated cluster. We neither inject faults nor rely on source code as they are usually not available for HPC systems, and our failure model is chain-based. Predominantly, most of these works lack lead time analysis and prediction speedup, which is our primary concern.

Failure Mitigation Schemes: [Guo13] shows that although recovery is important for proactive fault tolerance, it may occasionally cause problems so that, at times, it is better not to recover for cloud systems. Similar investigations have been done in HPC systems [ESS14b] to evaluate whether or not to checkpoint. [Bou13b; ESS14a; Tiw14] propose mechanisms to optimize checkpoint overheads. [Ouy11; Wan08; Ell12; Sho16; Ash18] propose application recovery strategies through combinations of redundancy, checkpointing and process migrations. Such recovery schemes can be leveraged after Aarohi successfully pin-points the future node failure location in a timely manner.

Distributed Machine Learning: Model compression techniques [Che17] are being developed to accelerate deep learning. Distributed training [Str15] on GPUs leverage the benefits of parallelism

improving the training speed. However, these methods are not aimed at fast inference. Apart from the specialized hardware and software support required for these, such techniques are more suitable for performance optimized training (phase-1). In practice, real-time testing for failure prediction requires assessment of a single incoming log message processed one at a time as opposed to a batch of messages, limiting deep learning acceleration. In such a situation, parallelization may not provide benefits. Large datasets are the main motivation for model and data parallelism or hardware-based ML-optimizations. Moreover, such methods incur high computational costs. It is worthwhile to develop a simpler, flexible and faster compiler-based scheme for clusters such as Aarohi.

Compiler-based Analysis: Regular expressions and CFGs have been used in different contexts in various ways [Che04; BG13; C.15]. [Yu16] proposes an automaton-based anomaly detection scheme for Openstack. PerfScope [Dea14] uses FSM (Finite State Machine) for online performance bug inference after its offline reproduction via signature extraction. However, compiler-based static analysis and rule-based bug checkers used in the past have not addressed generic machine translation of failure chains like Aarohi. Moreover, their premise (system call tracing, profiling, application bugs, binary instrumentation, source code reference) is considerably different from failure prediction in HPC systems.

Current techniques of log mining-based failure prediction helps only to a certain extent for proactive fault management. Prediction schemes such as Aarohi can further enhance the predictor's competence and, thus, facilitate failure mitigation schemes and recovery actions in practice.

4.6 Conclusion

This paper proposes an online node failure predictor called Aarohi for HPC systems.

Aarohi flags failures in a timely manner, i.e., on average in 0.31 msecs for an event chain of length 18. The observed inference speedup is over 27.4x w.r.t. the current state-of-the-art prediction approaches. Our predictor obtains as high as 3 minutes effective lead times to failures considering prediction time, which is sufficient for prevalent failure mitigation approaches. Aarohi can be adapted with minimal overhead across diverse system types.

The deployment of proposed failure prediction schemes is important for HPC resilience. Aarohi demonstrates the feasibility of an auto-generated prediction scheme based on parsing logs resulting in a speedup over prior methods. Additionally, it gives insights to log variations across systems and highlights the requirement of adaptability for sustainable prediction schemes. Our work can identify caveats and provide a yardstick to system practitioners for possible follow-up work on effective online anomaly prediction applicable to contemporary and next generation HPC systems.

CHAPTER

5

SYSTEMIC ROOT CAUSE ANALYSIS OF NODE FAILURES IN PRODUCTION HPC

5.1 Introduction

Powerful supercomputers require high availability to run scientific applications, enabling researchers from diverse technical domains to address grand challenges in computational simulations. As engineers are designing energy efficient exascale nodes [Rig17], current computing platforms require robust failure handlers to keep up with system scale, density, software complexity and computational speed. This necessitates the need for proactive solutions that can flag ahead of time an impending component failure. Before failure mitigation, having a better understanding of how failures materialize is essential. Recent research on log mining-based failure characterization [Gup17; Gup15], prediction [Kli17; Das18b] and recovery [Bau16] have revealed helpful insights to address failures in supercomputers. The time difference between the failure manifestation and the timestamp of the anomalous log message (precursor) when a impending failure is flagged is defined as lead time. Proactive fault tolerant solutions [Das18b; Kli17] when supported by root cause diagnosis can improve lead times and help in responding to both manifested or imminent failures effectively. Prior studies on root causes of node failures in data centers [VN10; Gun16] and HPC [Zhe12; Mar14] either provide a high-level categorization with limited correlations or lack holistic root cause inference for proactive resilience that can ameliorate the failure impact with an effective fix. Advanced system log

analytics unveil better understanding of node failures. This can further strengthen computational capabilities.

Root cause analysis of any complex system requires more attention towards the events encountered by its components. While researchers have focused their attention on specific components [Kum18; Tiw15] and interfaces depending on their target problem, answering how compute nodes fail needs a more integrated approach towards correlation-based log mining. Our methodology adopts a system-wide view to track root causes of node failures prevalent in computing systems. Our work is novel in that it considers system environment conditions along with inter-component dependencies to increase lead times to failures enhancing failure prediction schemes.

Table 5.1 HPC System Details

System	**Duration**	**Log Size**	**#Nodes**	**Type**	**Interconnect**	**Job Scheduler**	**FileSystem OS**	**Processors**	**GPUs, Burst Buffer**
S1	10 mons	373GB	5600	Cray XC30	Aries Dragonfly	Slurm	Lustre, SuSE	IvyBridge	×
S2	12 mons	150GB	6400	Cray XE6	Gemini Torus	Torque	Lustre	Haswell	×
S3	8 mons	39GB	2100	Cray XC40	Aries Dragonfly	Slurm	Lustre, SuSE	Haswell	Burst Buffer
S4	10 mons	22GB	1872	Cray XC40/XC30	Aries Dragonfly	Torque	Lustre, CLE	Haswell, IvyBridge	Burst Buffer
S5	1 mon	3.1GB	520	Institutional	Infiniband	Slurm	Lustre, RedHat	Haswell	GPUs

5.2 Background

For better fault-tolerance in contemporary HPC clusters, the current state-of-the-art lacks in the following aspects:

1. The various layers (software [Gup17], hardware [Tiw15], application [Mar14]) of these large-scale systems are mostly studied independently without exploiting their correlations during root cause analysis [Fu14].

2. Diverse components of the system affect each other (e.g., interconnect [Jha17b], GPU [Tiw15], DRAM [BG16]). Focusing on a specific component in isolation provides a local view, yet lacks a global perspective. As an example, analyzing interconnect errors [Kum18] at the intersection of network and applications alone ignore the implications of event and console logs generated elsewhere. In the context of root cause, having an understanding of how much these components correlate in failure manifestation can prevent recurring faults and re-investigations, thereby reducing unnecessary overhead.

3. Once a failure manifests, corrective actions need to be enhanced. Failures during interconnect failover and recovery procedures [Jha17b] have been shown to increase network congestion. A deeper understanding of how failures happen (beyond spatial and temporal characteristics) can aid in choosing the appropriate action for long-term system health.
This work investigates root causes of node failures considering software, hardware and application malfunctioning across diverse components with recommendations for effective system health checkers for applicability in practice.

Challenges: The following impediments and insights make data mining-based root cause analysis challenging but important for large-scale computing systems:
1. Events occurring transiently may not get manifested as logs. Production logs occasionally contain missing (specific time duration) or partial information (absence of certain environmental logs) due to logging discrepancy or issues with sharing policies. Deciphering transient faults that lead to cascading node failures is non-trivial.
2. Components can exhibit gray failures or fail-slow characteristics, which is different from fail-stop behavior [Gun18]. The latter is easier to diagnose, but the analysis of the former is equally important to infer the causes behind these failures.
3. Additional inputs may be required from operators to understand the implications of low-level logs in order to analyze the root causes of failures accurately.

After the detailed failure analysis study on Blue Waters by Martino et al. [Mar14] for contemporary HPC systems, this paper is the first to investigate how node failures happen with insights to their reasons. Unlike the prior work on failure characterization [Mar14] on a single Cray XE6 system (Blue Waters; Gemini interconnect, Lustre file system with AMD processors, and NVIDIA GPUs on Cray XK7 nodes) considering manual failure reports, our diagnosis is purely log-based from 5 different production sites with no human written reports. Similar to several prior statistical analyses [Gun16; Gun18], our study did encounter unknown reasons and cases that could not be analyzed either because of insufficient data or system logs did not reveal any directions to root causes.

Contributions: This paper answers the following questions to enhance fault resilience in HPC systems:
1. Are there spatial or temporal correlations amongst failed nodes w.r.t. similar root causes?
2. How much do the environmental factors (e.g., heartbeat faults, temperature violations) directly influence node failures? If there exist early external indicators, can the lead times be enhanced considering the same?
3. What faults do not lead to failures? Can these faults instigate failures under specific conditions?
4. How do jobs allocated on compute nodes contribute to failures? While jobs fail because of node failures, jobs can trigger nodes to fail as well. Is there any presence of temporal locality for the failed compute nodes running the same specific application?

Insights to these questions can help system practitioners gain better awareness about failures.

Past work [Mar14; Gun18; Gun16] has performed high-level characterization of potential root causes without analyzing the existence of external influence over correlated failures. These investigations had a limited view of isolated node failures. We dig deeper to inspect the above questions to infer the global view. To summarize, our work makes the following contributions:

• We analyze node failures of 5 different HPC systems, 4 of which are well used production clusters, incorporating inter-node correlations to understand the failure cause. We provide estimates of commonly occurring faults not leading to anomalous shutdowns.

• We quantify lead time enhancements if feasible leveraging the external indicators for the failed nodes.

• We analyze job characteristics on failed nodes to drill down to the root cause, apart from spatio-temporal correlations. Based on the insights we obtain from such temporal system-wide measurements we discuss their implications for enhanced system health.

Our solution is not for deployment, however, understanding such empirical observations can aid the community to develop more sustainable recovery approaches.

5.2.1 Preliminaries

Table 5.1 entails the characteristics of peta-scale HPC clusters whose logs have been analyzed. As evident, 4 out of 5 are Cray systems of substantial scale, all used heavily for various scientific applications. Since this work is about root cause, we have briefly mentioned the system configuration details to provide a clear description about the log sources. The time line of the logs spanned across 3 years (2014 to 2016). S5 is a small scale institutional cluster with a local file system using an Infiniband interconnect, unlike the rest. We did not have external environmental logs for this system, we discuss our findings on 4 weeks data, only to quantify application-based failures compared to the other machines. Apart from S2, which has Gemini interconnect, all the Cray systems use an Aries interconnect, with a Lustre filesystem. While S1, S3 and S5 use Slurm as their job scheduler, the rest use Torque.

Table 5.2 Log Data Details

Node Logs	**Content Description**
Internal	console/messages/consumer (p0-directories)
External	controller/event (erd)/SEDC/slurm/torque

Table 5.2 depicts the major portions of the logs consulted for this study. The compute node internal logs reside in the p0-directories in Cray systems. The console, consumer and messages logs are used to obtain the node-specific events. Log messages pertaining to blade, cabinet and environmental data are analyzed using controller and event router daemon (erd) logs. These contain SEDC (System Environmental Data Collections) warnings and additional hardware fault alerts to aid failure analysis. The job scheduler logs from torque or slurm are analyzed to investigate job-based

failures.

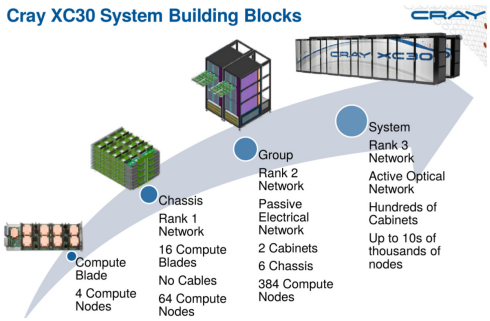

Figure 5.1 Cray System[1]

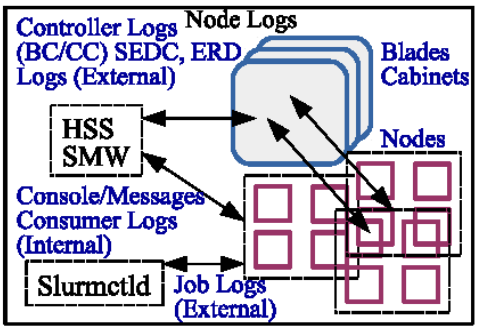

Figure 5.2 Methodology

5.2.2 Methodology

Figure 5.1 highlights the integrated components of a Cray supercomputer from a finer to coarser granularity. We consider system-wide environmental logs and blade/cabinet characteristics along with the node-specific internal events during the *unhealthy* time frame. We move from node to blade to cabinet to understand fault conditions and derive early indicators of impending failures. The controller logs coupled with event router messages provide deviations (higher/lower than the normal range) in sensor measurements (e.g., fan speed, temperature) to warn about health problems. Encompassing such features with job-based events aid in root cause analysis. Generality and automation is not the goal of this work. As case studies will reveal, it is impractical to formulate a generic algorithm for diverse failure reasons. However, statistical analysis with correlations over time and space to deduce the potential root causes using the aforementioned log sources is the objective. We have consulted the Cray documentation and relevant findings published in the CUG (Cray User Group) for our work. Figure 5.2 highlights our investigation procedure. We perform failure analysis in the following manner:

1. We track confirmed failure indications in the node-specific logs. These encompass the *console*, *messages* and *consumer* logs of the compute node internals. This initial step involved cluster administrator's knowledge about anomalous failure symptoms validating the ground truth.

2. Considering the time-frame of node failures derived in (1), we investigate the state of the nodes residing in the same blade as that of the failed nodes, mining the *controller* logs containing blade-specific information. This helps us to understand the presence of spatial correlation. We then correlate the SEDC warnings of the *event* logs to elicit any external influence evident over the blades with unhealthy nodes.

3. Additionally, we analyze the jobs allocated on the failed nodes from the *scheduler* logs to under-

[1]Source: `http://www.pefarrell.org/wp-content/uploads/2015/05/hpc.pdf`

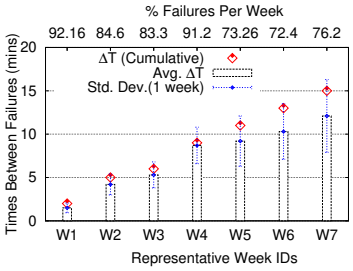

Figure 5.3 Failure Times

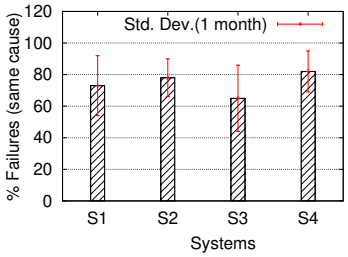

Figure 5.4 Dominant Cause

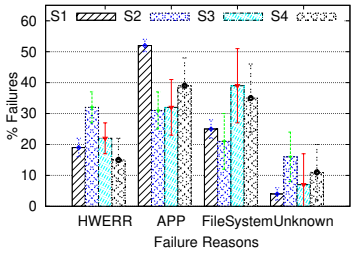

Figure 5.5 Failure Reasons

stand their affect on the compute nodes. This helps us to narrow down the root cause of failures in terms of deciphering whether application-triggered node failures appear in conjunction with early external indicators such as hardware errors and sensor reading variations or not.

Figure 5.2 shows the SMW (System Management Workstation) together with HSS (Hardware Supervisory System), which manage the system logging. Slurmctld is the application controller daemon that collects job logs.

We present systemic observations followed by the inferred failure reasons obtained through this methodology. Additional consideration of performance logs [Age14] or resource usage based application diagnosis [Tun18] is beyond the scope of this work. Since most production sites either do not maintain manual reports (that vendors, e.g., Cray specialists, may have) or do not share them because of restricted distribution policies or IP protection, validating our statistical inference becomes practically unattainable.

5.3 Evaluation Results

System-wide outages (SWOs) making the entire system unavailable are present in our logs and tend to be mostly service related, intended node shutdowns or file system caused failures. They contribute to less than 3% of the overall anomalous failures. Our study addresses single and multiple node failures, unlike SWOs, caused by diverse anomalies in the system. We evaluated over 1200 node failures for our analysis. In most Cray systems, 4 nodes reside in a single blade (slot). While not all failures[2]can be predicted with considerably increased lead times, some of them can be flagged ahead of time if external environment conditions are diagnosed. We provide carefully chosen time-intervals of representative samples to make observations, i.e., changing the duration does not alter the overall inference.

Inter-node Failure Times: Before digging into the root cause, we checked how far apart failures are per day and how many nodes share the dominant failure reason. We consider multiple node or blade failures over 7 weeks and calculate the cumulative node failures over different inter-arrival times. Figure 5.3 shows that most node failures happen within 2 to 15 minutes of each other. 92.16 &

[2]In this paper *failures* typically refer to *node* failures, unless otherwise mentioned

76.15% of the failures happen within 2 & 15 minutes, respectively, over W1 & W7. Similar observations on different days on all systems indicate that time between node failures ranges from a few seconds to more than 2 hours. Though there are days without failures, on other days nodes fail just minutes apart. Unlike SWOs appearing more than 6 hours apart in Blue Waters [Mar14], single or multiple node failures have shorter inter-node failure times.

Next, we identify single or multiple *dominant* failure reasons per day and compute the fraction of failed nodes on the same day corresponding to the dominant failure reason (e.g., H/W MCEs, kernel oops, Lustre Bugs). Figure 5.4 indicates that in 30 days, 65 to 82% of the nodes share the same failure cause with a standard deviation ranging between 12 & 21. Certain days have high job-triggered failures when most nodes run the same application. On other days, failures relate to diverse causes over short temporal distances. It is interesting to note that, if the dominant fault gets fixed, over 50% of the node failures can be recovered per day. Figure 5.5 shows the major breakdown of the high-level failure reasons. While S1 and S4 endured higher application-triggered failures, S3 had several file system caused failures, with overall variations between ±3 and ±12. Failures with uncertain root cause range between 4 & 16%, respectively.

Observation 1: *Time between failures has reduced (hours to minutes) in recent years compared to prior work [Mar14]. On average, more than 65% of the failures per day are caused by the same component malfunctioning, indicating the importance of analyzing short lead times and root causes of failures. Similar unknown cases whose root cause cannot be inferred exist in all studied Cray systems.*

Table 5.3 Fault Breakdown

Health Faults	**SEDC Warnings**	**Kernel Oops**
Node Heartbeat (NHF)	Temperature	Kernel Bug
Node Voltage (NVF)	Voltage	Lustre Bug
BC Heartbeat (BCHF)	Air Velocity	App-Exit
Module Health	ECB Fault	Hung Task
RPM Fault	Cabinet Sensor Check	oom-killer

5.3.1 External Influence on Node Failures

In order to understand the environmental influence on nodes we consider the blade and cabinet-specific health faults logged during the unhealthy time frames of the corresponding node failures. Blades encounter health faults and SEDC warnings (e.g., Electronic Circuit Breaker (ECB) faults) triggered by the blade controller (BC) software related to power monitoring. The cabinet controller (CC) software logs similar cabinet health status and sensor reading deviations such as temperature, voltage, and air velocity. A breakdown of the observed controller faults and warnings is shown in columns 1 & 2 of Table 5.3. We correlate the blade ID and cabinet ID of the corresponding failed

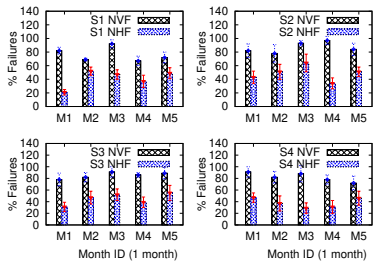

Figure 5.6 Node Faults

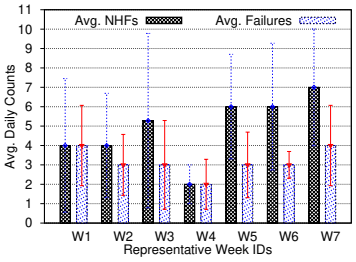

Figure 5.7 Heartbeat Faults

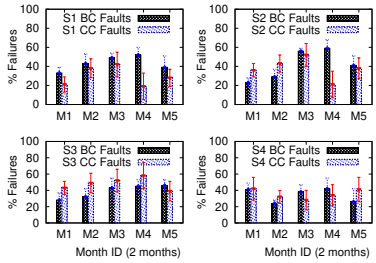

Figure 5.8 Blade/Cabinet Faults

node ID and inspect the logs around the failure time for systems S1 to S4.

Figure 5.6 highlights that 67 to 97% of the observed node voltage faults (NVF) correspond to failed nodes over 5 different months. These occur rarely, but when they do, they often relate to failures. On the contrary, 21 to 52% node heartbeat faults (NHF) actually manifest as failed nodes. This is because if a node fails a health test or skips a heartbeat, it is suspected to be dead, but empirically we observe that only about 45% NHFs actually fail. Figure 5.7 shows a finer breakdown of the NHFs, over 7 weeks. Most NHFs were failures in W1 & W4, while in the others more than 50% NHFs eventually caused a node to fail. Many NHFs turn out to be failures caused by hardware machine check exceptions (MCE), while NHFs that do not fail are nodes with their power turned off or those with skipped heartbeats. Unlike prior work [Ste12] (2% of the NHFs fail), we observe higher correspondence of NHFs with failures. For job-caused failures, NHFs may not be present in the logs, because the node passed the health checks at the communication level, but later job-caused malfunctioning launches the node health checker (NHC), which, when in *suspect mode*, may turn the node to *admindown* [Ste12] based on failed tests. Early NHF indicators for blades can warn about likely malfunctioning affecting some nodes of the blade.

Figure 5.8 indicates what fraction of the failed nodes belongs to faulty blades or cabinets that encountered some health faults or warning during the critical time frame. Over periods of 2 months, 23 to 59% of the failures belong to faulty blades, and 21 to 52% of the failures belong to cabinets that elicited some warnings or faults. While this alone does not give any clear indication about correlations between blade/cabinet health and specific node failures, hardware bugs and file system errors are observed in the node internal logs during these times. For days with no failures, such health faults are seldom found. For specific BC heartbeat faults, we observed only a fraction of the nodes in that blade to fail, but not all.

Observation 2: *On average, more than 21% NHFs and more than 67% NVFs result in failures. For blade-related faults a fraction of nodes in the blade are observed to have failed. NVFS and NHFs can be leveraged as early indicators of system malfunctioning for failure prediction schemes.*

What faults do not cause failures? We have identified characteristics, which clearly indicate

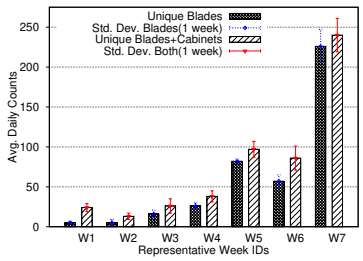

Figure 5.9 Blade Counts

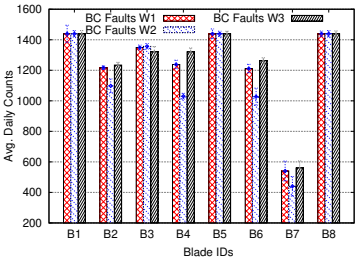

Figure 5.10 SEDC Warnings

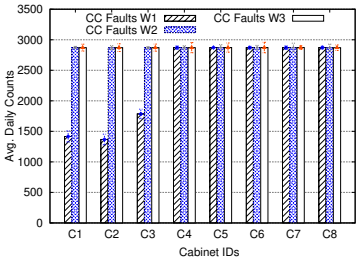

Figure 5.11 Cabinet RPM Faults

that certain repeatedly occurring environmental warnings and errors do not cause failures and are mostly benign. A small fraction of the blades and cabinets in the event logs (ERD) encounter *ec_sedc_warnings* with BC and CC sensor reading deviations. These predominantly contain warnings for temperature, voltage or velocity falling below the minimum allowed system threshold. Figure 5.9 shows the total number of unique blades and cabinets, which encountered various SEDC *warnings* over a week for S1. The unique blade count varies between 5 and 226, while the cumulative count of unique blades and cabinets ranges from 24 to 220. The blade count for encountered health *faults* is mostly higher than the *warnings*. Figure 5.10 depicts the frequency of multiple warning types occurring throughout the day for S2. While blades B1, B5 and B8 encountered more than 1400 mean recurring warnings, B7 stopped seeing them after a certain hour of the day, for which its std. deviation is as high as ±65. In 3 weeks, 8 blades underwent voltage/temperature violations, but the specific failed nodes did not belong to any of these blades.

Analyzing the environmental influences at the appropriate granularity (months vs. hours vs. days) brings more clarity for short term failures, as certain warnings (e.g., *sedc_warnings*) occur repeatedly while others (e.g., NVF) happen infrequently. Averaging frequently occurring faults at coarse-level granularity may hide the possible characterizations.

Figure 5.11 indicates that cabinet-level faults are logged more frequently than those of blades, with more than 1400 mean daily counts. Only 32.14% (e.g. 9 out of 28) failures belonged to these faulty cabinets over a week. The evident cabinet RPM faults do not affect the collocated nodes of the blades in respective cabinets. The root cause of these faults do not get fixed right away (within hours). It is likely that nearby fans provide sufficient cooling so that no failures occur, i.e., by ramping up their RPMs in response to higher temperatures. Such RPM ramp-ups do not enter failure logs, i.e., we do not have sufficient data to analyze our hypothesis with time correlation. But evidence collected with a sample node where fans were disconnected corroborate this as RPMs of resulting fans ramped up in response to the control-feedback subsystem governing node cooling. Cabinet faults do not have correlations with a certain failure type since the corresponding blade of the faulty node on several occasions does not encounter such signals. Furthermore, many healthy blades with no failures experience similar temperature or voltage violations. Temperature variations can trigger hardware errors or processor corruptions affecting sensor readings of an entire blade

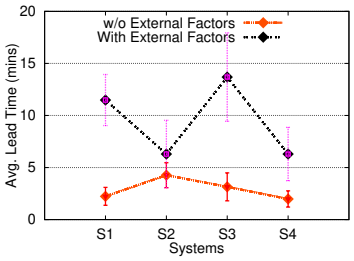

Figure 5.12 Lead Times

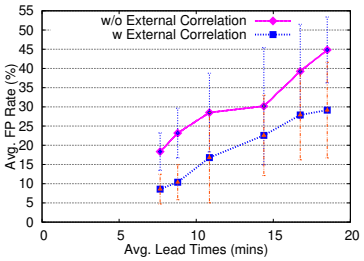

Figure 5.13 Lead Times with FP

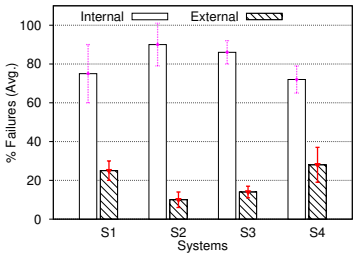

Figure 5.14 External Influence

causing the air velocity to be automatically reduced by the firmware in the cabinet. However, similar incessant warnings and faults on healthy time frames and blades do not help in pointing to the potential root cause. Hence, we did not investigate further. Our observation conforms to prior work on temperature variability [El-12a], i.e., there is no clear evidence that hotter nodes contribute to higher node outages or downtime based on studies with LANL HPC clusters. Our work did not find notable traits in threshold violations to help quantify node reliability.

Observation 3: *Blade and cabinet-level external indications are not primary causes of node failures. A small percentage of failed nodes belongs to faulty blades and cabinets. While SEDC warnings and health faults persist on days with observed failures, there is no clear evidence of sensor reading violations causing nodes to fail.*

Lead Time Enhancements: Most studies [Kli17] have referred to lead time considering internal console logs containing a sequence of fault indicative messages (e.g., fatal) leading to failure. We inspect whether additional environmental messages appearing before the indicative internal logs aid in improving the potential lead time.

While a major fraction of external faults and warnings are not the primary root causes, certain failures possess early indicators such as *ec_hw_errors* in the event logs indicating hardware malfunctioning. Typically, such errors appear in conjunction with multiple indicative root causes such as processor corruptions, node heartbeat faults, firmware bugs, kernel panics etc in the internal logs. While hardware errors appear during healthy times as well, additional internal failure patterns affirm their correlations with node failures. Blade-level faults and SEDC warnings along with the evidence of failed nodes with buggy console messages enable lead time enhancements. These environmental alerts imply fail-slow characteristics unlike fail-stop, similar to fail-slow hardware evident in the production data centers [Gun18] such as DRAMs. For certain failures, hardware errors sustain for a long time unlike their insignificant frequency during other times. Indicators in the external logs timestamped earlier are correlated to node, blade, and cabinet-ids over the failure times in the internal logs. The time difference between the relevant external and internal timestamps are computed to obtain viable lead time increments. Figure 5.12 shows that, considering the external

faults and warnings, the mean lead times can be increased by about 5 times, compared to just the internal node logs by themselves, in systems S1 to S4. For these failures, the early indicators were absent during normal operation. To assess the effect of additional external correlations on the false positive rate, the existence of similar correlations in the healthy node logs around similar failure times are analyzed. Figure 5.13 highlights that the false positive rate is lower (e.g., from 30.18 down to 22.57%) with external correlations considered than otherwise. This is because healthy node logs that appear *similar* to multiple correlations across diverse log sources (for unhealthy node) occur less frequently, reducing the number of false positives. It is well known that if the application is the primary root cause, no early indicators are usually available in the external logs. To confirm that lead times cannot be enhanced further if failure is solely caused by the application, several external faults and warnings are analyzed. The environmental faults do not correlate, which makes lead time enhancements infeasible in such cases. Figure 6.1 illustrates the fraction of node failures whose lead times could be enhanced considering early indicators over 4 different weeks. For many failures, absence of external warnings prevented lead time enhancements; however, improvements were feasible for 10 to 25% of the failures. 72 to 90% of the failures could not have improved lead times. This fraction depends on the contribution of application-caused failures. Across all the systems the std. deviation ranges between ±3 & ±15.

Observation 4: *Certain failures caused by hardware errors or file system bugs possess early indicators in the external logs. Lead times can be enhanced by about a factor of 5 times in such cases with a lower false positive rate w.r.t. the failures without external correlations. However, such enhancements are not possible for application-triggered node failures, since early failure indicators are absent in this case.*

5.3.2 Application Triggered Failures

The following patterns are observed primarily for failures triggered (directly or indirectly) by applications:

• The internal node logs often contain major hardware or software errors apart from file system or interconnect errors and node shutdown messages. These occasionally appear in conjunction with job-specific errors due to NHC tests.

• No external environmental indicators exist for these failures, making lead time enhancements infeasible.

• Strong temporal correlations exist. Multiple blades or nodes failing at similar times of the day share the same job ID. For such cases, failed nodes need not be quarantined as these nodes recover once new jobs run on them since the problem is with the application.

• Once *oom-killer* (out of memory) is invoked and processes are killed, associated modules in the

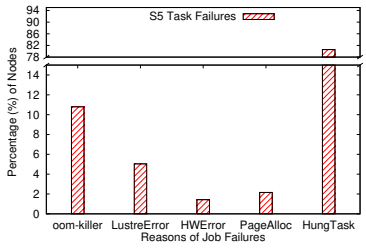

Figure 5.15 Job Failures in S5

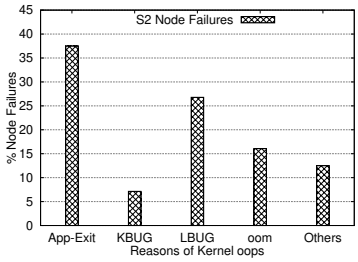

Figure 5.16 Job Failures in S2

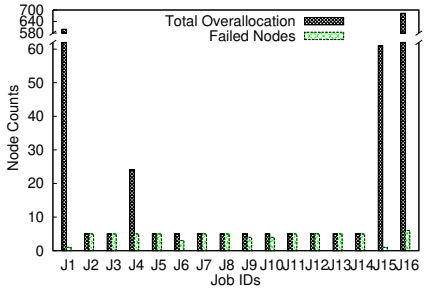

Figure 5.17 Resource Overallocation

stack traces (e.g., xpmem, dvsipc, lustre) often indicate filesystem inconsistencies. Processes also get killed by the epilogue of the job scheduler that removes any user job from a node before it is reallocated.

Table 5.3 indicates the root causes of kernel oops observed across the systems in column 3. Hardware, software and application faults tend to trigger callback traces. Across all 5 systems, memory crunch, abnormal app-exits, job-triggered kernel bugs and Lustre bugs are prevalent reasons for node failures. Even without application malfunctioning, MCEs, CPU corruptions and file system bugs result in kernel panics. Jobs do get aborted because of failed nodes. But nodes also fail because of the running application's computational requirements. This depends on what the allocated nodes experience during task execution. Our measurements show that spatially distant nodes have temporal locality of failures because of the common jobs running on them. E.g., in Figure 5.3 week W1 experienced 92.16% of the failures within 2 minutes. More than 42% of those failures belonged to different blades distant from each other on a specific day; however, all of them executed under the same job ID during the time of failure.

We analyzed the call traces of S5's logs. Figure 5.15 shows that 10.59% of the nodes were running low on memory, which triggered the *oom-killer* that killed processes and added "kernel oops" messages to the log. 5.04% of the nodes had Lustre errors without any call trace causing jobs to fail. While 1.43% of the nodes encounter hardware errors, such as GPU or disk errors, 2.16% had software errors, such as page allocation faults and segmentation faults. Several jobs got canceled in the interactive session, around 11% of the jobs failed to complete because they got affected by the state of the allocated nodes. 80.57% of the nodes encountered *hung task timeout* errors followed by a call trace indicating slow system I/O, unable to flush the data to free memory within the stipulated time (2 minutes). Hung task-based kernel oops are commonly seen in the institutional cluster, but they do not cause nodes to fail and are not observed on the Cray systems.

For S2, Figure 5.16 shows that 37.5% of the failures occur due to anomalous app-exits (application), failing NHC tests turning the node down. 7.14% of the failures were caused due to critical kernel bugs (e.g., invalid opcode) and 26.78% because of file system bugs (e.g., race in threads spawned in the code), all prompted by the compute jobs. 16.07% of the failures happened due to

memory resource exhaustion without additional software bugs. The other 12.5% of kernel oops were due to CPU stalls and other driver/firmware bugs, with an overall std. deviation ranging from ±2 to ±6. On the surface, these bugs (KBUG & Others in Fig. 5.16) seem to emanate from the OS, but careful analysis reveals the potential of application-triggered file system bugs, which indirectly propagate as kernel bugs or CPU stalls. Similar characteristics were observed on the other systems. As seen in Figure 5.5, for all systems at least 30% of the node failures happen due to applications. Runtime errors triggering kernel oops cannot be handled ahead of time since tangible software or hardware errors appear only after the affected kernel module is invoked. Fine-grained application performance diagnosis studies may consider such runtime anomalies to reduce systemic errors.

Figure 5.17 illustrates how memory overallocation causes failures in production HPC systems. On a specific day, 53 failures occur with a total of 16 jobs scheduled on them. Of the allocated nodes, a subset of them suffer resource *overallocation* errors. As evident, for jobs J5 & J8, all overallocated nodes fail, while only a few of them fail for jobs J4 & J15. J1 and J16 had 1 & 6 failures for 600 & 683 overallocated nodes, respectively. When a small fraction of the allocated nodes fail, the jobs fail to complete, consequently, job re-allocations are performed for recomputations.

Observation 5: *Unlike institutional clusters, file system bugs are more frequent in Cray systems indicating the application's influence on Lustre contention and resiliency. While oom-killers are often invoked in HPC systems, 37% of the app-exits occur because of incapable nodes implying the need for better resource aware scheduling. When job requirements exceed a compute node's resource capacity, quarantining nodes may not be effective. Instead, those applications can be monitored and their corresponding users can be informed.*

5.3.3 Node Internal Failure Analysis

Failure causes evident from internal node logs have been studied in the past [Mar14; Gup17]. We observed similar failure causes such as MCEs, processor corruptions, file system bugs, we performed additional correlations with external heath faults and application impacts to scrutinize external influences. We summarize our findings here:

1. Many driver/firmware bugs appear after application exits and result in kernel oops. These may or may not relate to hardware faults, *ec_hw_errors* are observed in the external logs. Such hardware bugs may get incited only when specific modules of the code is executed (e.g. row hammer [Xia16] attacks on DRAM).

2. Many segmentation faults originate from applications. Programs invoking page allocation requests often cause nodes to fail due to memory limits.

3. Software traps (e.g., invalid opcode) generally do not fail nodes, unless exception handling disturbs the file system, which may eventually fail the node.

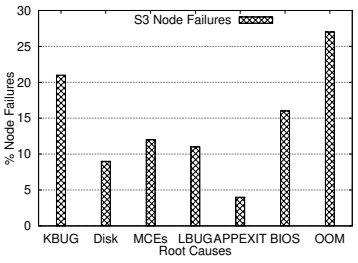

Figure 5.18 Root Causes

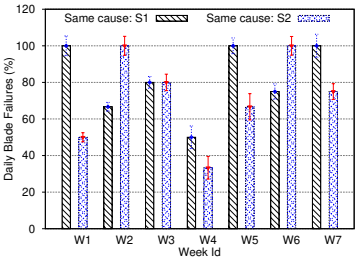

Figure 5.19 Blade Failures

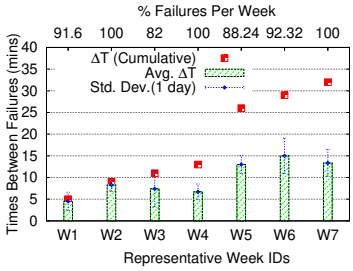

Figure 5.20 Temporal Locality

4. Disk errors & job caused inode errors can render a file system inaccessible, causing nodes to be slow or non-responsive. Finer analysis of such bugs indicates the root cause to be at the application layer, even though the failures manifest inside the OS kernel.

Even though job-specific malfunctioning is not indicated in the node failure logs upfront, the root cause often lies in the application. For Lustre or kernel bugs, several kernel oops, firmware bugs, and fork or memory allocation errors, the original fault propagates from the job running on the compute nodes. The high-level breakdown of the failure causes has been discussed in the literature [Gup17; Mar14] and is consistent with Table 5.4 column 1. Additionally, our work affirms most of such root causes to be application-triggered through fine-grained correlations. While some of the root causes can help create fault-aware solutions (e.g., CPU corruptions, MCEs), several high impact tangible indications happen only during application runtime (e.g., invalid opcode, segmentation fault, resource exhaustion). This implies the need for better performance diagnosis for application resilience mechanisms in HPC systems.

Table 5.4 Root Causes of Failures

Reasons	**Stack Trace Modules**
Application	sleep_on_page
Kernel Bug	ldml_bl
MCEs, Processor Corruptions	dvs_ipc_mesg
FileSytem Bug	mce_log
Segmentation/Page Fault	rwsem_down_failed

Table 5.4 indicates the overall root causes of failures and the predominant kernel modules reported by stack backtraces. Since kernel oops are frequently observed, we examined the preliminary stack traces indicating the modules linked to the trace such as *dvs_ipc_mesg, mce_log* etc. While few modules clearly refer to file system problems, *ldml_bl* and *sleep_on_page* are job-triggered. Hang bugs or concurrency bugs due to application code cause kernel hangs. We did not investigate the entire trace, but there are indications of application-caused (which in turn may affect the fileystem) versus file system-caused failures.

Figure 5.18 shows the distribution of root causes over a period of 4 months for S3. Hardware faults such as BIOS errors, MCEs, and disk errors contribute to 37% of the failures, software faults (Kernel/Lustre bugs) contribute to 32%, and applications contribute to 31%. 27% of the failures were

caused by memory resource exhaustion. There exist other time frames with very frequent *app-exits* when nodes failed the NHC (node health checker) tests.

In terms of failure locality, we observed two primary trends in the context of root cause:
1. Days with multiple node failures, spatially distant in physical location, usually fail with different root cause, unless they are causally correlated in terms of sharing the same application executable.
2. Presence of multiple blade failures is often caused by the same application executing on those nodes around the failure time. Blade failures display temporal locality in some systems with similar failure reasons with short (≈5 minutes) inter-node failure times.

Figure 5.19 depicts the avg. fraction of blade failures with same failure reasons for S1 and S2 over 7 weeks. Most of the days in both systems, at least 50% of the blades fail. Most days experience single or multiple node failures on diverse blades. Considering blades with all failed nodes, often the manifested symptoms appear to be similar. On some days, all blade failures were due to the same hardware faults and application-triggered software faults. For both S1 and S2, errors are less than ±7.2 hinting at the consistent spatial locality of nodes in terms of the same failure cause. When blades fail at the granularity of microseconds, they share the same system malfunction as the root cause. Figure 5.20 indicates that the time between node failures over 7 weeks, considering job-triggered node failures, does not exceed 32 minutes. As evident, W1 encounters on avg. 91% of the failures within 5 minutes. W6 & W7 experienced more than 90% of the failures within 29 to 32 minutes. Inter-node failure variations occur in the order of minutes, temporal locality (same application) caused failures are well evident. These are much shorter than the MTBFs (mean time between failures) observed in LANL systems in prior work [Tiw14] (> 5 hrs).

Observation 6: *Although primary root causes are similar to observations in the existing studies on Cray systems, finer analysis implies that the origin of most bugs lies in the application. Jobs often trigger kernel panics, firmware bugs, application exits, and exhaust resources. Performance diagnosis for application resilience in conjunction with failure prediction schemes can improve system health.*

Observation 7: *More than 50% of the blades can fail in a single day pertaining to the same root cause. Nodes sharing an application often fail during similar times. These nodes may belong to different blades (spatially distant), but exhibit temporal locality w.r.t. the failure root cause.*

5.3.4 Unknown Causes

During our analysis, we could not infer potential root causes for three failure patterns. First, the appearance of *ERROR: Type:2; Severity:80; Class:3; Subclass:D; Operation: 2*, which may indicate BIOS problems. These are commonly seen in the systems for benign healthy cases as well. Several nodes encountered anomalous shutdowns with these errors without any other helpful patterns. Application

or hardware do not show faults that influence these failures. It is not clear what conditions could trigger failures with this hardware error.

Second, *L0_sysd_mce* related to memory errors appear before nodes fail. However, insufficient information prohibited us to understand what causes these to appear. During failure times, other nodes in the blade do not fail, and there is no correlation over time. The name implies causes related to the blade controller (L0/BC), but the observed symptoms did not unveil anything helpful.

Finally, there are failures with no logs or simply shutdown messages with no prior indicative failure symptoms. The affect of cosmic radiations on hardware (e.g., memory, disk, DRAM, GPUs) and silent data corruptions (SDCs) have been shown in the research literature [Tiw15; BG16; Gun18]. We suspect such solar flares could cause nodes to fail, which are undetectable in the logs. Over the unhealthy time frames, we could not examine such uncertain failures based on the available weather reports, hence this remains speculative. What is more probable is operator error, i.e., manual shutdowns of good nodes by accident (e.g., wrong button, IPMI command).

Observation 8: *Several failures either do not have logs or have insufficient information, where we cannot deduce any potential root cause. Such unknown cases may or may not relate to known operator errors or rare cosmic radiations. These cases require more investigation subject to operator-level/vendor support.*

5.3.5 Case Studies

Let us briefly analyze 5 representative cases of root cause inference that provide insights to the variety of problems arising in production HPC clusters. The abridged analysis of these failures is summarized in Table 5.5.

Case 1: A single node failure occurs with *L0_sysd_mce* errors in the console logs without additional hardware, software, or application problems. The other nodes in the same blade do not fail and experience correctable hardware MCEs (which are mostly benign) and hardware SSID errors. No additional external influence or job errors are reported around the failure time. These indications did not lead to potential root cause and do not suffice to understand why the node was shutdown.

Case 2: 3 nodes fail hours apart belonging to different blades with similar node internal failure patterns with no temporal correlations. All 3 nodes encounter hardware errors followed by MCEs and kernel oops. Interconnect errors and temperature threshold violations are present in the SEDC logs, but not around the failure time. No application problems are reported. Processor corruptions and critical MCEs turn out to be the root cause in this case.

Case 3: 6 nodes fail around similar times (seconds to minutes apart) with similar node internal failure patterns. We confirmed that the nodes were running the same job around the failure time. No environmental faults or warnings were reported. *oom-killer* was invoked because of memory

Table 5.5 Sample Failure Cases

#	#Failures	Internal Indicators	External Indicators	Root Cause Inference
1	1 failure	L0_sysd_MCE followed by NHC warnings, other nodes of the same blade encountered correctable H/W errors and SSID errors	No environmental indications or job malfunctioning reported around the failure time	Potential root cause could not be deduced
2	3 failures	Neither spatially nor temporally close (4 am, 12.38 & 3.21 pm), however, similar patterns (H/W error → MCEs → kernel oops)	Aries link error and temperature threshold violations distant from the failure time, no job malfunctioning indications	CPU corruptions and MCEs affecting the filesystem causing failure
3	6 failures	oom-killer invoked → kernel oops (app-based call trace) at similar times, similar patterns on all nodes	No external indications around the failure time, same application running on all the nodes	Application-caused memory exhaustion, nodes fail NHC tests leading to failure
4	1 failure	LustreErrors → Unable to handle kernel paging request, other nodes of the blade did not fail	Benign link errors/temp. threshold violations distant from the failure time, scheduled job aborted	Application-triggered filesystem bug causing failure
5	1 failure	H/W MCEs → critical errors, other nodes of the blade encountered benign events	Early indicators of ec_hw_errors & link errors prior to the failure time, no job errors evident	Fail-slow symptoms of memory eventually failing the node (degraded hardware can be triggered by software)

exhaustion. Several processes were killed followed by kernel oops. The modules linked to the call trace were indicative of the running application. Consequently, NHC warnings were observed. This is an application resource exhaustion caused failure with no additional hardware problems.

Case 4: A failure occurs with a kernel bug unable to fulfill a paging request preceded by Lustre errors. External indicators include interconnect link errors and temperature threshold violations, which were distant from the failure time. The job running on this compute node did not terminate gracefully. In this case, the application triggered a filesystem bug eventually failing the node.

Case 5: A node fails with critical hardware MCEs with no kernel oops or major additional software bugs. In the external logs, *ec_hw_errors* and link errors were reported several minutes before the failure time. For the other blades such sustained environmental errors around that time were absent. No application misbehavior was evident. This case is a hardware caused failure with fail-slow characteristics. Lead time enhancements are feasible in this case.

5.3.6 Discussion

From these results we infer that a generic approach with a formal algorithm is impractical as a means to identify failure causes. While a fraction of them can be incorporated into a diagnosis framework, automation is not the goal of this measurement-driven work. Although its generality and applicability in practice may be a question, our statistical inference over 4 production clusters can aid in better understanding of the missing ingredients for sustainable system health. While some results affirm past findings, this work throws light on the feasibility of lead time enhancements without degrading the false positive rate based on the observed fail-slow characteristics, which can be useful. Further, the fact that major external health faults are not the primary culprits of

Table 5.6 Findings and Recommendations

#	Major Findings	Suggested Recommendations
1	Node health faults & short-term multiple blade failures often indicate unhealthy system state. Several daily failures relate to similar root causes	Non-critical health faults (e.g., NHF) and temporal locality of failures can be considered before launching checkpoint/restarts making reactive approaches more aware of the potential root cause
2	Major Blade & Cabinet level health indicators are not strongly correlated with the primary root cause	Frequent appearance of SEDC warning and threshold violations can be ignored unless major indicators are observed in the node internal logs
3	Fail-slow hardware symptoms exist for certain software triggered hardware failures aiding in lead time improvements	Node failure prediction schemes can incorporate external correlations for possible lead time enhancements for proactive fault tolerance
4	Node quarantining can be ineffective when the root cause is triggered by application misbehavior	Instead of sequestering nodes, users can be intimated about their malfunctioning job (by cluster operators) or buggy jobs can be blocked (by NHC)
5	Across all the systems, a considerable number of node failures involve Kernel oops with long stack traces. These can be triggered by the hardware, software, and application based on the fault propagation chain	It may be worthwhile to conduct a machine learning guided detailed study of call traces from large-scale systems to narrow down the buggy code or function emanating from the application or filesystem to segregate job-triggered (which in turn affect the filesystem or produce driver bugs) versus job-caused failures with better confidence
6	Spatial-temporal correlations of node failures exist w.r.t. the application-caused failures	System administrators can incorporate additional health tests in NHCs to account for the nodes failing incessantly due to abnormal application exit, to track the buggy APID besides *rebooting* or turning the node to *admindown*
7	A significant number of failures are primarily triggered by the applications, which in turn may affect the file system or hardware	Application resilience schemes (performance diagnosis) can be used in conjunction with system failure prediction tools to infer future system health

node failures is not obvious (e.g., temperature variability influence node reliability in the data centers [El-12b; Wan17a]). We empirically show what faults do not cause failures. Besides, the stack trace logs indicating the associated modules (hints to root cause) and job-triggered failures (that can be indirectly caused) imply the importance of application diagnosis with failure prediction (i.e., the need to be more inclusive of system events to assess health). The implications based on our work can facilitate the improvement of existing failure prediction schemes [Kli17] or health diagnostic tools (e.g., NHC) with a more integrated approach towards system inspection. This has the potential to benefit the system administrators, vendors, users and the community as a whole. Based on the major observations, Table 5.6 summarizes our findings with recommendations for enhanced resilience.

Implications and Suggestions: We provide evidence about multiple failures with causal or temporal correlations w.r.t. the root cause. We specifically observe the fact that environmental indications are less correlated to the root causes of node failures on Cray platforms and to the existence of fail-slow symptoms. While operators can be less concerned about external health warnings in the absence of internal faults, it is better to equip the failure prediction & checkpoint/restart (C/R) schemes as well as node health checkers to be aware of early indicators of health faults and malfunctioning applications to reduce recomputation cost. Instead of quarantining a node, buggy

Table 5.7 Large-Scale System Evaluation

Studies	Focus	Findings
[Bau16; Gup17; Jha17a]	Overall System	Regime detection, Wasted time analysis, Integrated monitoring infrastructure design, Reliability consistency over time, Failure categorization
[Wan17a; VN10; Gun18]	Server Failures, Hardware Faults	Spatial/Temporal distribution, Component correlation, Root causes: Hard disk, Raid controller & Memory faults, H/W failure rates, Fail-slow symptoms, Environmental causes
[El-12a; BG16; Man16; Gho18]	Disk, DRAM	Multi-bit errors corrupt non-adjacent memory bits, DRAM power/current consumption modeling, Temp. & energy influences
[Tiw15; Jha17b; Gar18; Kum18]	Interconnect, GPUs	Interconnect error breakdown, Load imbalance effect on lane degrades, Reasons of SWOs, Failover and lane recovery impact, Radiation expts. & GPU temp. sensitivity
[Mar15b; ES17; Amv18]	Application, User Jobs	Job failure probability increases with system scale, Failure impact on production workload, Jobs consuming higher CPU utilization is more fallible, Task failures cause job failures
[Mar14]	Node Failures	High-level root cause breakdown of SWOs & Single/Multiple node failures for a single system
Our work	Node Failures	Fine-grained external influence analysis, Faults not leading to failures, Feasible lead time enhancements for 4 systems

applications can be monitored with intimation to the user for a better solution, since application malfunctioning creates node reboots incurring additional computation costs. A full stack trace can provide finer details about the file system and application errors. Automated diagnosis tools for inference of anomalies from call traces [Arn07] can further help in anomaly detection and pinpointing the root cause. Finally, it seems equally important to invest time on improving application resilience mechanisms through performance diagnosis. Root causes often originate from the application affecting other system components causing failed nodes.

Comparative Analysis: Field data analysis for failure characterization and root cause analysis have benefited the community to comprehend upcoming resilience requirements. Table 5.7 encapsulates the major failure analysis studies performed for large-scale systems (data centers and HPC). Analogous pattern recognition, analyzing spatial/temporal locality and correlations with applications have been applied to different systems, such as Google, Hadoop, BlueGene, Cray and other HPC Clusters, with varied research objectives encompassing different layers and components. Findings have indeed help to understand what causes a higher impact on failures and needs more consideration. Past work has provided invaluable insights to system failures, our work further refines this wealth by adopting a more integrated approach towards root cause analysis. While certain findings conform to the past, newer insights narrow down those influences that are not the primary culprit of failures, providing empirical evidence of purely application-triggered failures and feasible lead time enhancements.

Table 5.8 compares our work with two other similar recent failure analysis studies. [Gun18]

Table 5.8 Comparison

#	Features	Our Work	[Mar14]	[Gun18]
1	Root Cause Analysis	✓	✓	✓
2	Node Failures	✓	✓	×
3	Stack Trace Analysis	✓	×	×
4	External/Job Correlations	✓	✓	×
5	Fail-slow Symptoms	✓	×	✓
6	Lead Time	✓	×	×
7	Cloud	×	×	✓
8	HPC	✓	✓	✓

studied detailed hardware faults for 12 production clusters characterizing transient and fail-stop behavior. However, they do not focus on node failures, which require more correlations and lead time analysis. Moreover, they provide anecdotal evidences without any empirical analysis like ours. [Mar14] performs statistical analysis of failures on the Blue Waters peta-scale system. Unlike our work, they study SWOs but do not perform correlations over multiple times and diverse log sources for detailed root cause inference with lead time enhancements. Moreover, our study is based on 5 contemporary Cray systems with different configurations providing evidence of fine-grained external influence and implications of stack trace analysis.

5.4 Related Work

Existing work on system log-based failure analysis can be grouped into three categories: a) failure characterization in HPC and clouds, b) root cause diagnosis frameworks, and c) failure prediction schemes. We briefly discuss them to understand how our work differs from past system evaluations.

[Jha17b] discusses cases where interconnect failures (lane/link) and overheating cause job failures, system-wide outages (SWOs), and failures during recovery. While [Jha17b] reports early indicators of interconnect faults and SWOs due to overlapping interconnect and filesystem fault recovery events, [Kum18] characterizes interconnect errors showing evidence of load imbalance causing lane degradation. Shiraz [Gar18] improves system throughput by intelligently scheduling applications. [Tiw15] studies GPU faults specifically, and [Gup17] conducts general system fault measurements. These focus on specific components or layers and affirm the need for a holistic study to understand the causes of failures from a system-wide perspective. [Gup15; Ghi16; Bau16] study spatial/temporal correlations of failures, derive their logical correlations, and propose dynamic checkpointing schemes after detecting system degradation.

Several studies address reliability concerns in cloud-based data centers. Deepview [Zha18a] performs timely virtual hard disk (VHD) failure diagnosis through global system analysis for Azure. [Che18] discusses transient and lasting failures in production data centers through cooling profiles based on thermal variations. VAMPIRE [Gho18] proposes a better DRAM power model, and [Wan17a;

Xu18; VN10] study data center hardware failures, predict disk errors, and characterize server failures. [Hua18; Gun18] emphasize the investigations for fail-slow behavior (gray failures) of hardware. [El-12a] studies the effects of temperature on node reliability. While some of their observations are similar to ours (e.g., disk-based failures), we have shown evidence of transient failures in HPC systems as well.

Multiple root cause diagnosis techniques [Zhe12; Att12; Fu14; Mar14; Kas15; Wan15] either point out high-level failure layer (hardware, software, application) or perform causal dependency analysis without holistic considerations or are application-centric. In contrast, we include controller logs to understand external influences and quantify increases in lead times.

[Jha17a] possess similar research objectives as ours, but unlike our statistical quantitative failure analysis, it proposes a high-level architecture with case studies for holistic application monitoring. Failure analysis on Blue Waters [Mar14] is the closest to our work. Our study provides different empirical observations and new insights from five production clusters unlike [Mar14]. Recently, machine learning (ML)-guided failure prediction schemes [Das18b; Kli17] have been developed to proactively respond to failures. Our work compliments such solutions in providing sustainable recovery schemes.

5.5 Conclusion

Accurate holistic root cause diagnosis is an indispensable step toward failure mitigation in practice. Proactive fault tolerant solutions with estimated lead times may serve as a short-term cure. Since the root causes are not fixed, the same failures may recur unless we clearly understand how node failures happen. Better awareness has the potential to enhance recovery approaches. We present inferred root causes of node failures observed in well used production HPC systems based on temporal correlations of internal and external logs. We recognize application characteristics that cause nodes to fail and provide an estimate of feasible lead time enhancements. Our work identifies that environmental influences are not strongly correlated to single or multiple node failures. Choosing a suitable mitigation action (proactive/reactive) consciously with an understanding of the root cause when a failure is imminent can have long-term benefits in progressive computation instead of restarting from checkpoints, which requires recomputation.

CHAPTER

6

SUMMARY AND FUTURE WORK

Lead time is a crucial metric for failure prediction in any computing system. Unless the impending failures are identified sufficiently ahead of time, no effective recovery action can be taken. Established approaches to combat failures, such as checkpoint/restarts, job migrations, and redundant computing, incur considerable overhead that reduce compute capacity and increase power consumption [ESS14b]. As systems are scaling with evolving hardware and software, timely failure prediction is necessary to act on unhealthy components before they seize to respond. Shorter MTBFs and higher frequency of failures obviate the need to investigate proactive failure management techniques. With the advancement of computing systems, fault tolerance strategies need to adapt for compatibility with the changing operational context. This thesis takes a step in that direction for the current petascale and next-generation exascale computing systems.

This chapter summarizes the thesis contributions and discusses some possible future directions.

6.1 Concluding Remarks

Figure 6.1 illustrates the connections between the four pieces of work contributed to improve system reliability. While TBP and Desh devise methods to determine the lead time estimates with minimal supervision, Aarohi strives to achieve low inference times to allow effective response during the available lead time. Root cause analysis further guides in deciding what suitable actions to take

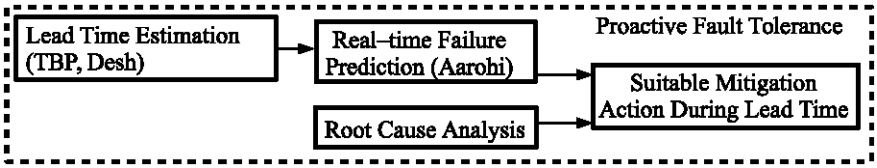

Figure 6.1 Improved System Reliability

(i.e., how to respond) during the available lead times. This dissertation contributes to the following solutions for proactive failure management in supercomputing systems:

1. First, we design a topic modeling-based framework, called TBP (Time-Based Phrase), to predict node failures. During training, TBP forms node failure chains from the top ranked phrases extracted by TOT (Topics Over Time). During testing, TBP matches the incoming phrases with trained failure chains to examine potential failures. TBP achieves no less than 83% recall, 98% precision and as much as 2 minutes of lead time. Furthermore, this study highlights the existence of short-term phrase variations in system logs making continuous time statistical analysis important. Hardware errors, machine check exceptions (MCEs), and kernel panics are found to cause frequent node failures, besides interconnect and filesystem caused failures.

2. Second, we propose a deep learning-based solution, called Deep Learning for System Health (Desh), to predict lead times to node failures. Desh operates in three phases: First, training is used to recognize sequences of log events pertaining to nodes. Second, re-training is applied to the chain recognition of events augmented with expected lead times to failure. Third, lead times are predicted during inference and a specific node subject to failure is identified. Desh obtains more than 2 minutes average lead times with an F1 score as high as 89.88%. This scheme also demonstrates the importance of considering a sequence of events irrespective of log-severity labels for examining failures accurately through unknown phrase analysis.

3. Third, we contribute a compiler-based solution, called Aarohi, to infer imminent failures during testing with enhanced speedup. Aarohi demonstrates generic machine translation of failure chains with intertwined tokenization and parsing. Prediction times are consistently under 12 msecs for test sequences as long as 3820 phrases over diverse platforms, leading to effective lead times over 2 to 3 minutes. Per log entry prediction time is not a suitable estimator of the overall prediction efficacy, since the message size, count and the fraction of relevant tokens parsed influence the time to predict failures.

4. Fourth, we conduct root cause analysis of node failures correlating external and internal node logs. The existence of temporal correlations across spatially distant node failures caused by the same application malfunctioning is evident. While environmental health faults and warnings

are not strongly correlated to the root cause, early external indicators exist to enable lead time enhancements. Similar empirical insights can facilitate taking better mitigation actions for sustained system health.

Both TBP and Desh procure 2 to 3 minutes of lead times to node failures. With approximately 2 minutes lead time, TBP obtains a false positive rate of less than 23% versus Desh with a false positive rate of no more than 25%. This lead time sensitivity shows that while low false positives and false negatives are desired, it is important to maintain a suitable trade-off between achievable lead times and prediction errors in a system. Aarohi further illustrates the merits of a flexible parsing-based approach over ML or python-based frameworks during testing. Low prediction times (< 20 msecs) are achievable for sufficiently long test sequences (e.g., 3820 log messages), indicating the feasibility of completing proactive actions within the estimated lead time. Root cause analysis brings awareness in taking suitable mitigation actions when failures are imminent, with an integrated understanding of how nodes fail. This can aid in addressing prediction inaccuracies as well. Conforming to our hypothesis in Chapter 1.4, which states that

"The majority of node failures in HPC systems can be predicted via machine learning thereby identifying the precise node location and with sufficient lead time to engage in proactive actions for fault mitigation"

the feasibility of obtaining short lead times has been demonstrated by machine learning-based effective prediction. Trained log chains comprising of messages eventually leading to terminal log messages facilitate the prediction of potential node failures during inference.

6.2 Future Work

Together with the existing state-of-the-art [Ouy11; Gup15], the proposed solutions lay the foundations for proactive fault tolerance in computing systems. In deep learning, normal or healthy logs and abnormal or unhealthy logs are often trained with appropriate labeling. Usually, a single log message or a data point is considered an anomaly, unlike a sequence of phrases over time and space as in Desh. Desh uses both healthy and unhealthy logs in the first phase since it intends to learn the phrase sequences observed in the dataset as opposed to anomalies. In the second phase, Desh learns failure chains with time differences. Since our test data contains a sufficient fraction of phrases similar to these chains, the sequences similar to them are correctly identified as immanent failures, and the remaining dissimilar sequences are considered as healthy (not leading to failures). However, the second phase may not be effective based on an imbalance, e.g., in the training logs, which contain only unhealthy node logs, while the test logs only contain healthy data.

The efficacy of the second phase training depends on the amount of overlapping (common) phrases present in healthy and unhealthy node logs and the fraction of *learned* failure sequences seen during testing. To address this, the second phase training should have both healthy and unhealthy node phrase sequences, distinguished by appropriate labeling. Such labeled training in the second phase can enable more generic prediction for test sequences with different proportions of healthy and unhealthy messages. Future work should compare testing of (a) an LSTM model trained with both healthy and unhealthy node logs with (b) an LSTM model trained with only unhealthy node logs (as in Desh).

Currently, preliminary efforts are being made for single node deployment using Desh and Aarohi. Figure 6.2 explains the high-level design. In production clusters, logs arrive at a port and are archived through the logging daemons such as `rsyslogd`. There are two aspects to be investigated further. First, how to optimize offline training performance using the archived logs. The choice of deep learning optimizers w.r.t. the batch size, the accuracy obtained w.r.t. the training duration, their influence on prediction over time require further studies. Please note that while several solutions [You19; Nei16] exist achieving improved training times, our work requires extra attention in the context of generating failure chains. Second, for real-time prediction, storing the streaming logs from the port directly (instead of files) and processing the buffer continuously brings it closer to practice. This requires enhancing prior solutions (Desh++/Aarohi++) and evaluating the end-to-end performance to understand the hurdles if any. Additionally, analyzing the resource consumption during inference (e.g., CPU, memory) can give an estimate of the costs incurred to run such a predictor. These experimentations can help decipher the missing elements in using the developed solutions.

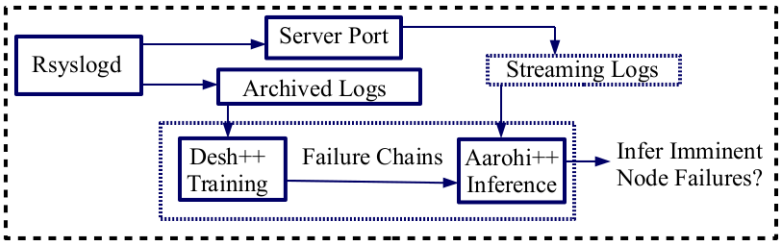

Figure 6.2 Real-Time Failure Prediction

Such ground work can help emanate stronger solutions with better lead time confidence. For sustained system resilience, the following directions should be investigated further:

- Performance logs have been analyzed thoroughly for anomaly detection in distributed systems [Dea12; Gua11]. Understanding the existence of correlations, if any, between performance logs (e.g., LDMS [Age14]) and system logs for proactive measures can help strengthen failure prediction in HPC. Can the association of application resilience [Mar15b; Fan17] and system failure characteristics [Mar14] together provide a better indication of system health?

Major failure characterizations are performed at the software layer [Bau16; Zhe12]. One may benefit from being more inclusive of system events considering diverse layers of the system stack. This can unveil a more consistent view of failures, which could remain salient otherwise (e.g., SEDC logs with power correlations are still unknown). Since applications impact compute node failures, more research is required to understand HPC applications and their resource usage traits in conjunction with the proposed failure prediction schemes such as TBP [Das18b].

- Another aspect is more focused on experimentation and deployment for using such solutions in the major production clusters. Addressing prediction inaccuracies such as false positives and negatives through a combination of mitigation actions either proactively or reactively (e.g., via live migration or checkpoint/restart) during lead times to failures is essential. Integrating streaming logs with dynamic retraining and the systemic assessment of predictor placement in the cluster with failure resolution costs (e.g., I/O overhead, resource usage) can help in understanding the benefit of adaptive fault tolerance for upcoming systems. The cost benefit evaluation for proactive actions over lead time versus their reactive counterpart is required eventually for their practical applications.

Such in-depth study can definitely help HPC systems and cloud computing systems to architect next generation production systems with improved reliability and provide insights to the research community to explore sustainable solutions for large-scale computing.

BIBLIOGRAPHY

[Aba16] Abadi, M. et al. "TensorFlow: A System for Large-Scale Machine Learning". *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 2016, pp. 265–283.

[Age14] Agelastos, A. et al. "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications". *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. 2014, pp. 154–165.

[Ahm17] Ahmad, S. et al. "Unsupervised real-time anomaly detection for streaming data". *Neurocomputing* **262** (2017), pp. 134–147.

[Aho86] Aho, A. V. et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[Alv16] Alvarez, G. P. R. et al. "Towards understanding job heterogeneity in hpc: A nersc case study". *Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 521–526.

[Amv18] Amvrosiadis, G. et al. "On the diversity of cluster workloads and its impact on research results". *USENIX Annual Technical Conference, ATC*. 2018, pp. 533–546.

[Arn07] Arnold, D. C. et al. "Stack Trace Analysis for Large Scale Debugging". *International Parallel and Distributed Processing Symposium IPDPS*. 2007, pp. 1–10.

[Ash18] Ashraf, R. A. et al. "Shrink or Substitute: Handling Process Failures in HPC Systems using In-situ Recovery". *arXiv preprint arXiv:1801.04523* (2018).

[Att12] Attariyan, M. et al. "X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software". *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2012, pp. 307–320.

[Aup12] Aupy, G. et al. "Impact of fault prediction on checkpointing strategies". *arXiv preprint arXiv:1207.6936* (2012).

[Aus18] Aussel, N. et al. "Improving Performances of Log Mining for Anomaly Prediction Through NLP-Based Log Parsing". *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS*. 2018, pp. 237–243.

[Bas16a] Baseman, E. et al. "Interpretable anomaly detection for monitoring of high performance computing systems". *Outlier Definition, Detection, and Description on Demand Workshop at ACM SIGKDD*. 2016.

[Bas16b] Baseman, E. et al. “Relational Synthesis of Text and Numeric Data for Anomaly Detection on Computing System Logs”. *15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016, Anaheim, CA, USA, December 18-20, 2016*. 2016, pp. 882–885.

[BG16] Bautista-Gomez, L. et al. “Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer”. *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE. 2016, pp. 645–655.

[Bau16] Bautista-Gomez, L. A. et al. “Reducing Waste in Extreme Scale Systems through Introspective Analysis”. *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 2016, pp. 212–221.

[Ber14] Berrocal, E. et al. “Exploring void search for fault detection on extreme scale systems”. *Cluster Computing*. 2014, pp. 1–9.

[Bir14] Birke, R. et al. “Failure analysis of virtual and physical machines: patterns, causes and characteristics”. *Dependable Systems and Networks (DSN)*. IEEE. 2014, pp. 1–12.

[BL06] Blei, D. M. & Lafferty, J. D. “Dynamic topic models”. *International Conference on Machine learning*. 2006, pp. 113–120.

[Ble03] Blei, D. M. et al. “Latent dirichlet allocation”. *Journal of machine Learning research* **3**.Jan (2003), pp. 993–1022.

[Bos16] Bosilca, G. et al. “Failure detection and propagation in HPC systems”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. 2016, pp. 312–322.

[BG13] Bosman, G. & Gruner, S. “Log File Analysis with Context-Free Grammars”. *International Conference on Digital Forensics*. 2013, pp. 145–152.

[Bou13a] Bouguerra, M. S. et al. “Failure prediction: what to do with unpredicted failures”. *28th IEEE international parallel and distributed processing symposium*. Vol. 2. 2013.

[Bou13b] Bouguerra, M. S. et al. “Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing”. *International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2013, pp. 501–512.

[Bra15] Brandt, J. et al. “New Systems, New Behaviors, New Patterns: Monitoring Insights from System Standup”. *Cluster Computing*. 2015, pp. 658–665.

[Bro92] Brown, P. F. et al. “Class-based n-gram models of natural language”. *Computational linguistics* **18**.4 (1992), pp. 467–479.

[C.15] C., P. S. G. et al. “Why Big Data Industrial Systems Need Rules and What We Can Do About It”. *ACM SIGMOD International Conference on Management of Data*. 2015, pp. 265–276.

[Cap09] Cappello, F. “Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities”. *The International Journal of High Performance Computing Applications, IJHCA* **23**.3 (2009), pp. 212–226.

[Cas] *Cassandra Bug 11050*. URL: `https://issues.apache.org/jira/browse/CASSANDRA-11050?attachmentSortBy=fileName`.

[Cat12] Catonsville, M. et al. “Inter-Agency Workshop on HPC Resilience at Extreme Scale” (2012).

[Blu] *CFDR Data*. URL: `https://www.usenix.org/cfdr-data`.

[Cha06] Chakravorty, S. et al. “Proactive fault tolerance in MPI applications via task migration”. *High Performance Computing* (2006), pp. 485–496.

[Cha12] Chalermarrewong, T. et al. “Failure prediction of data centers using time series and fault tree analysis”. *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2012, pp. 794–799.

[Che18] Chen, C. et al. “Detecting Data Center Cooling Problems Using a Data-driven Approach”. *APSys*. 2018.

[Che02] Chen, M. Y. et al. “Pinpoint: Problem Determination in Large, Dynamic Internet Services”. *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE. 2002, pp. 595–604.

[Che04] Chen, M. Y. et al. “Path-Based Failure and Evolution Management”. *Symposium on Networked Systems Design and Implementation NSDI*. 2004, pp. 309–322.

[Che17] Cheng, Y. et al. “A Survey of Model Compression and Acceleration for Deep Neural Networks”. *CoRR* **abs/1710.09282** (2017).

[Chi14] Chilimbi, T. M. et al. “Project Adam: Building an Efficient and Scalable Deep Learning Training System”. *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014.* 2014, pp. 571–582.

[Cho15] Chollet, F. et al. *Keras*. `https://github.com/keras-team/keras`. 2015.

[Cho17] Chothia, Z. et al. “Online Reconstruction of Structural Information from Datacenter Logs”. *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 344–358.

[Chu13] Chuah, E. et al. “Linking resource usage anomalies with system failures from cluster log data”. *International Symposium on Reliable Distributed Systems*. 2013, pp. 111–120.

[Coa13] Coates, A. et al. “Deep learning with COTS HPC systems”. *International Conference on Machine Learning*. 2013, pp. 1337–1345.

[Cos14] Costa, C. H. A. et al. “A System Software Approach to Proactive Memory-Error Avoidance”. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. 2014, pp. 707–718.

[Das18a] Das, A. et al. “Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC”. *High-Performance Parallel and Distributed Computing (HPDC)*. ACM. 2018.

[Das18b] Das, A. et al. “Doomsday: predicting which node will fail when on supercomputers”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC*. 2018, 9:1–9:14.

[Dea14] Dean, D. J. et al. “Perfscope: Practical online server performance bug inference in production cloud computing infrastructures”. *Symposium on Cloud Computing, (SoCC)*. ACM. 2014, pp. 1–13.

[Dea12] Dean, D. J. et al. “Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems”. *Proceedings of the 9th international conference on Autonomic computing*. ACM. 2012, pp. 191–200.

[Deb18] Debnath, B. et al. “LogLens: A Real-Time Log Analysis System”. *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. 2018, pp. 1052–1062.

[Di17] Di, S. et al. “LogAider: A tool for mining potential correlations of HPC log events”. *Cluster, Cloud and Grid Computing*. 2017, pp. 442–451.

[DM13] Di Martino, C. “One size does not fit all: Clustering supercomputer failures using a multiple time window approach”. *International Supercomputing Conference*. Springer. 2013, pp. 302–316.

[DM12] Di Martino, C. et al. “Assessing time coalescence techniques for the analysis of supercomputer logs”. *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE. 2012, pp. 1–12.

[Don15] Donahue, J. et al. “Long-term recurrent convolutional networks for visual recognition and description”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 2625–2634.

[DL16] Du, M. & Li, F. “Spell: Streaming Parsing of System Event Logs”. *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*. 2016, pp. 859–864.

[Du17] Du, M. et al. “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning”. *ACM Conference on Computer and Communications Security*. 2017, pp. 1285–1298.

[ES13] El-Sayed, N. & Schroeder, B. “Reading between the lines of failure logs: Understanding how HPC systems fail”. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. 2013, pp. 1–12.

[ESS14a] El-Sayed, N. & Schroeder, B. “Checkpoint/restart in practice: When ’simple is better’”. *Cluster Computing (CLUSTER)*. IEEE. 2014, pp. 84–92.

[ESS14b] El-Sayed, N. & Schroeder, B. “To checkpoint or not to checkpoint: Understanding energy-performance-I/O tradeoffs in HPC checkpointing”. *Cluster Computing (CLUSTER)*. IEEE. 2014, pp. 93–102.

[ES18] El-Sayed, N. & Schroeder, B. “Understanding Practical Tradeoffs in HPC Checkpoint-Scheduling Policies”. *IEEE Trans. Dependable Sec. Comput.* **15**.2 (2018), pp. 336–350.

[El-12a] El-Sayed, N. et al. “Temperature management in data centers: why some (might) like it hot”. *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS ’12, London, United Kingdom, June 11-15, 2012*. 2012, pp. 163–174.

[El-12b] El-Sayed, N. et al. “Temperature management in data centers: why some (might) like it hot”. *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS ’12, London, United Kingdom, June 11-15, 2012*. 2012, pp. 163–174.

[ES17] El-Sayed, N. et al. “Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations”. *International Conference on Distributed Computing Systems, ICDCS*. IEEE. 2017, pp. 1333–1344.

[Ell12] Elliott, J. et al. “Combining Partial Redundancy and Checkpointing for HPC”. *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21*. 2012, pp. 615–626.

[Eln08] Elnozahy, E. et al. “System resilience at extreme scale”. *Defense Advanced Research Project Agency, Tech. Rep* (2008).

[Ecp] *Exascale Computing Project (ECP*. URL: `https://www.exascaleproject.org/what-is-exascale/`.

[Fan17] Fang, B. et al. “LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures”. *High-Performance Parallel and Distributed Computing, HPDC*. Ed. by Huang, H. H. et al. ACM, 2017, pp. 117–130.

[Fu09] Fu, Q. et al. “Execution anomaly detection in distributed systems through unstructured log analysis”. *International Conference on Data Mining, (ICDM)*. IEEE. 2009, pp. 149–158.

[FX07] Fu, S. & Xu, C. “Exploring event correlation for failure prediction in coalitions of clusters”. *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*. 2007, p. 41.

[Fu14] Fu, X. et al. “Digging deeper into cluster system logs for failure prediction and root cause diagnosis”. *Cluster Computing*. 2014, pp. 103–112.

[Ful08] Fulp, E. W. et al. “Predicting Computer System Failures Using Support Vector Machines.” *WASL* **8** (2008), pp. 5–5.

[Gai11] Gainaru, A. et al. “Event log mining tool for large scale HPC systems”. *European Conference on Parallel Processing*. 2011, pp. 52–64.

[Gai12a] Gainaru, A. et al. “Fault prediction under the microscope: a closer look into HPC systems”. *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, p. 77.

[Gai12b] Gainaru, A. et al. “Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems”. *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS, Shanghai, China, May 21-25*. 2012, pp. 1168–1179.

[Gai13] Gainaru, A. et al. “Failure prediction for HPC systems and applications Current situation and open issues”. *International Journal of High Performance Computing Applications* **27**.3 (2013), pp. 273–282.

[Gai14] Gainaru, A. et al. *Navigating the blue waters: online failure prediction in the petascale era. Argonne National Laboratory Technical Report*. Tech. rep. ANL/MCS-P5219-1014, 2014.

[Gar18] Garg, R. et al. “Shiraz: Exploiting System Reliability and Application Resilience Characteristics to Improve Large Scale System Throughput”. *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. 2018, pp. 83–94.

[Gar14] Garraghan, P. et al. “An empirical failure-analysis of a large-scale cloud computing environment”. *High-Assurance Systems Engineering (HASE)*. IEEE. 2014, pp. 113–120.

[Ghi16] Ghiasvand, S. et al. “Lessons learned from spatial and temporal correlation of node failures in high performance computers”. *International Conference on Parallel, Distributed, and Network-Based Processing*. 2016, pp. 377–381.

[Gho18] Ghose, S. et al. “What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study”. *ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS*. 2018, p. 110.

[Gcp] *Google Cloud Platform*. URL: https://cloud.google.com.

[GF13] Guan, Q. & Fu, S. "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures". *International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2013, pp. 205–214.

[Gua11] Guan, Q. et al. "Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems". *Availability, Reliability and Security (ARES)*. IEEE. 2011, pp. 83–90.

[Gun16] Gunawi, H. S. et al. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages". *ACM Symposium on Cloud Computing*. 2016, pp. 1–16.

[Gun18] Gunawi, H. S. et al. "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems". *USENIX Conference on File and Storage Technologies, FAST*. 2018, pp. 1–14.

[Guo13] Guo, Z. et al. "Failure Recovery: When the Cure Is Worse Than the Disease". *14th Workshop on Hot Topics in Operating Systems, HotOS XIV*. 2013.

[Gup15] Gupta, S. et al. "Understanding and Exploiting Spatial Properties of System Failures on Extreme-Scale HPC Systems". *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*. 2015, pp. 37–44.

[Gup17] Gupta, S. et al. "Failures in large scale systems: long-term measurement, analysis, and implications". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*. 2017, 44:1–44:12.

[Hac09] Hacker, T. J. et al. "An analysis of clustered failures on large supercomputing systems". *Journal of Parallel and Distributed Computing* **69**.7 (2009), pp. 652–665.

[Had] *Hadoop Bug 1911*. URL: https://issues.apache.org/jira/browse/HADOOP-1911.

[Ham16a] Hamooni, H. et al. "LogMine: Fast Pattern Recognition for Log Analytics". *International Conference on Information and Knowledge Management*. 2016, pp. 1573–1582.

[Ham16b] Hamooni, H. et al. "LogMine: Fast Pattern Recognition for Log Analytics". *ACM International Conference on Information and Knowledge Management, CIKM*. 2016, pp. 1573–1582.

[He16] He, P. et al. "An evaluation study on log parsing and its use in log mining". *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE. 2016, pp. 654–661.

[He17] He, P. et al. “Drain: An Online Log Parsing Approach with Fixed Depth Tree”. *International Conference on Web Services, ICWS*. 2017, pp. 33–40.

[HS97] Hochreiter, S. & Schmidhuber, J. “Long short-term memory”. *Neural computation* **9**.8 (1997), pp. 1735–1780.

[Hua17] Huang, K.-Y. et al. “Mood detection from daily conversational speech using denoising autoencoder and LSTM”. *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE. 2017, pp. 5125–5129.

[Hua18] Huang, P. et al. “Capturing and Enhancing In Situ System Observability for Failure Detection”. *13th USENIX OSDI*. 2018, pp. 1–16.

[Hwa12] Hwang, A. A. et al. “Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design”. *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. 2012, pp. 111–122.

[IM17] Islam, T. & Manivannan, D. “Predicting Application Failure in Cloud: A Machine Learning Approach”. *Cognitive Computing (ICCC), 2017 IEEE International Conference on*. IEEE. 2017, pp. 24–31.

[Jha17a] Jha, S. et al. “Holistic Measurement-Driven System Assessment”. *IEEE International Conference on Cluster Computing, CLUSTER*. 2017, pp. 797–800.

[Jha17b] Jha, S. et al. “Resiliency of HPC Interconnects: A Case Study of Interconnect Failures and Recovery in Blue Waters”. *IEEE Transactions on Dependable and Secure Computing* (2017).

[Jha13] Jhawar, R. et al. “Fault tolerance management in cloud computing: A system-level perspective”. *IEEE Systems Journal* **7**.2 (2013), pp. 288–297.

[Jia15] Jia, R. et al. “Towards Proactive Fault Management of Enterprise Systems”. *International Conference on Cloud and Autonomic Computing*. 2015, pp. 21–32.

[Jia17] Jianwu, X. et al. *System failure prediction using long short-term memory neural networks*. US Patent App. 15/478,714. 2017.

[JZ16] Johnson, R. & Zhang, T. “Supervised and semi-supervised text categorization using LSTM for region embeddings”. *International Conference on Machine Learning*. 2016, pp. 526–534.

[Kas15] Kasikci, B. et al. “Failure sketching: a technique for automated root cause diagnosis of in-production failures”. *Symposium on Operating Systems Principles, SOSP*. 2015, pp. 344–360.

[Kli17] Klinkenberg, J. et al. “Data Mining-Based Analysis of HPC Center Operations”. *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. 2017, pp. 766–773.

[Kum18] Kumar, M. et al. “Understanding and Analyzing Interconnect Errors and Network Congestion on a Large Scale HPC System”. *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. 2018, pp. 107–114.

[Lan10a] Lan, Z. et al. “A study of dynamic meta-learning for failure prediction in large-scale systems”. *J. Parallel Distrib. Comput.* **70**.6 (2010), pp. 630–643.

[Lan10b] Lan, Z. et al. “Toward Automated Anomaly Identification in Large-Scale Systems”. *IEEE Trans. Parallel Distrib. Syst.* **21**.2 (2010), pp. 174–187.

[Lou10] Lou, J.-G. et al. “Mining Invariants from Console Logs for System Problem Detection.” *USENIX ATC*. 2010.

[Man16] Manousakis, I. et al. “Environmental Conditions and Disk Reliability in Free-cooled Datacenters”. *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. 2016, pp. 53–65.

[Mar93] Martinetz, T. M. et al. “’Neural-gas’ network for vector quantization and its application to time-series prediction”. *IEEE transactions on neural networks* **4**.4 (1993), pp. 558–569.

[Mar14] Martino, C. D. et al. “Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters”. *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. 2014, pp. 610–621.

[Mar15a] Martino, C. D. et al. “LogDiver: A Tool for Measuring Resilience of Extreme-Scale Systems and Applications”. *Workshop on Fault Tolerance for HPC at eXtreme Scale*. 2015, pp. 11–18.

[Mar15b] Martino, C. D. et al. “Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5, 000, 000 HPC Application Runs”. *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25*. 2015, pp. 25–36.

[McC09] McCallum, A. et al. “Factorie: Probabilistic programming via imperatively defined factor graphs”. *Advances in Neural Information Processing Systems*. 2009, pp. 1249–1257.

[Mik13] Mikolov, T. et al. “Efficient estimation of word representations in vector space”. *arXiv preprint arXiv:1301.3781* (2013).

[Nag12] Nagaraj, K. et al. “Structured comparative analysis of systems logs to diagnose performance problems”. *Networked Systems Design and Implementation*. 2012, pp. 26–26.

[Nag07] Nagarajan, A. B. et al. “Proactive fault tolerance for HPC with Xen virtualization”. *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007*. 2007, pp. 23–32.

[Nak11] Nakka, N. et al. “Predicting Node Failure in High Performance Computing Systems from Failure and Usage Logs”. *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*. 2011, pp. 1557–1566.

[Nei16] Neil, D. et al. “Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences”. *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*. Ed. by Lee, D. D. et al. 2016, pp. 3882–3890.

[Nie17] Nie, B. et al. “Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities”. *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2017 IEEE 25th International Symposium on*. IEEE. 2017, pp. 22–31.

[Oli08] Oliner, A. J. et al. “Alert detection in system logs”. *Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on*. IEEE. 2008, pp. 959–964.

[Ouy11] Ouyang, X. et al. “High Performance Pipelined Process Migration with RDMA”. *IEEE/ACM Cluster, Cloud and Grid Computing, CCGrid*. 2011, pp. 314–323.

[Pec11] Pecchia, A. et al. “Improving log-based field failure data analysis of multi-node computing systems”. *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE. 2011, pp. 97–108.

[Pit17] Pitakrat, T. et al. “Hora: Architecture-aware online failure prediction”. *Journal of Systems and Software* (2017).

[RM15] Rezaei, A. & Mueller, F. “DINO: Divergent Node Cloning for Sustained Redundancy in HPC”. *Cluster Computing*. 2015, pp. 180–183.

[Rig17] Rigo, A. et al. “Paving the way towards a highly energy-efficient and highly integrated compute node for the Exascale revolution: the ExaNoDe approach”. *Euromicro Conference on Digital System Design (DSD)*. IEEE. 2017, pp. 486–493.

[Sal10] Salfner, F. et al. “A survey of online failure prediction methods”. *ACM Computing Surveys (CSUR)* **42**.3 (2010), p. 10.

[SH15] Shatnawi, M. & Hefeeda, M. “Real-time failure prediction in online services”. *2015 IEEE Conference on Computer Communications, INFOCOM*. 2015, pp. 1391–1399.

[Sho16] Shohdy, S. et al. “Fault tolerant frequent pattern mining”. *High Performance Computing (HiPC)*. IEEE. 2016, pp. 12–21.

[Sni14] Snir, M. et al. “Addressing failures in exascale computing”. *The International Journal of High Performance Computing Applications, IJHCA* **28**.2 (2014), pp. 129–173.

[Sri15] Sridharan, V. et al. “Memory errors in modern systems: The good, the bad, and the ugly”. *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 297–310.

[SO08] Stearley, J. & Oliner, A. J. “Bad words: Finding faults in Spirit’s syslogs”. *Cluster Computing and the Grid*. 2008, pp. 765–770.

[Ste12] Stearley, J. et al. “A State-Machine Approach to Disambiguating Supercomputer Event Logs”. *Workshop on Managing Systems Automatically and Dynamically, MAD*. 2012.

[Str15] Strom, N. “Scalable distributed DNN training using commodity GPU cloud computing”. *INTERSPEECH 2015, 16th Annual Conference of the International Speech Communication Association, Dresden, Germany, September 6-10, 2015*. 2015, pp. 1488–1492.

[Tiw14] Tiwari, D. et al. “Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems”. *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. 2014, pp. 25–36.

[Tiw15] Tiwari, D. et al. “Understanding GPU errors on large-scale HPC systems and the implications for system design and operation”. *High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 331–342.

[Top] *Top 500 List*. URL: `https://www.top500.org/lists/top500/`.

[Nov] *Top 500 List*. URL: `https://www.top500.org/list/2017/11/`.

[Tun17] Tuncer, O. et al. “Diagnosing performance variations in HPC applications using machine learning”. *International Supercomputing Conference*. Springer. 2017, pp. 355–373.

[Tun18] Tuncer, O. et al. “Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning”. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[Vis16] Vishnu, A. et al. “Fault Modeling of Extreme Scale Applications Using Machine Learning”. *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. 2016, pp. 222–231.

[VN10] Vishwanath, K. V. & Nagappan, N. “Characterizing cloud computing hardware reliability”. *ACM Symposium on Cloud Computing, SoCC*. 2010, pp. 193–204.

[Wan08] Wang, C. et al. “Proactive process-level live migration in HPC environments”. *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*. 2008, p. 43.

[Wan17a] Wang, G. et al. “What Can We Learn from Four Years of Data Center Hardware Failures?” *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. 2017, pp. 25–36.

[Wan17b] Wang, H. et al. “Online Reliability Prediction via Long Short Term Memory for Service-Oriented Systems”. *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE. 2017, pp. 81–88.

[Wan15] Wang, K. et al. “A methodology for root-cause analysis in component based systems”. *International Symposium on Quality of Service (IWQoS)*. IEEE. 2015, pp. 243–248.

[WM06] Wang, X. & McCallum, A. “Topics over time: a non-Markov continuous-time model of topical trends”. *International Conference on Knowledge discovery and data mining*. 2006, pp. 424–433.

[Wat12] Watanabe, Y. et al. “Online failure prediction in cloud datacenters by real-time message pattern learning”. *Cloud Computing Technology and Science (CloudCom)*. IEEE. 2012, pp. 504–511.

[Xia16] Xiao, Y. et al. “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation”. *USENIX Security Symposium*. 2016, pp. 19–35.

[Xie17] Xie, B. et al. “Predicting Output Performance of a Petascale Supercomputer”. *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 2017, pp. 181–192.

[Xu09] Xu, W. et al. “Detecting large-scale system problems by mining console logs”. *Symposium on Operating systems principles*. 2009, pp. 117–132.

[Xu10] Xu, W. et al. “Detecting large-scale system problems by mining console logs”. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. Citeseer. 2010, pp. 37–46.

[Xu18] Xu, Y. et al. “Improving Service Availability of Cloud Systems by Predicting Disk Error”. *USENIX Annual Technical Conference, ATC*. 2018, pp. 481–494.

[You19] You, Y. et al. “Fast Deep Neural Network Training on Distributed Systems and Cloud TPUs”. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[YL16] Yu, L. & Lan, Z. “A scalable, non-parametric method for detecting performance anomaly in large scale computing”. *IEEE Transactions on Parallel and Distributed Systems* **27**.7 (2016), pp. 1902–1914.

[Yu11] Yu, L. et al. “Practical online failure prediction for blue gene/p: Period-based vs event-driven”. *Dependable Systems and Networks Workshop*. 2011, pp. 259–264.

[Yu12] Yu, L. et al. “Filtering log data: Finding the needles in the Haystack”. *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*. 2012, pp. 1–12.

[Yu16] Yu, X. et al. “CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs”. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’16, Atlanta, GA, USA, April 2-6*. 2016, pp. 489–502.

[Zha16a] Zhang, K. et al. “Automated IT system failure prediction: A deep learning approach”. *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. 2016, pp. 1291–1300.

[Zha18a] Zhang, Q. et al. “Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure”. *15th USENIX NSDI*. 2018, pp. 519–532.

[Zha18b] Zhang, S. et al. “PreFix: Switch Failure Prediction in Datacenter Networks”. *POMACS* **2**.1 (2018), 2:1–2:29.

[Zha16b] Zhao, X. et al. “Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle”. *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 2016, pp. 603–618.

[Zhe09] Zheng, Z. et al. “System log pre-processing to improve failure prediction”. *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2*. 2009, pp. 572–577.

[Zhe10] Zheng, Z. et al. “A practical failure prediction with location and lead time for blue gene/p”. *Dependable Systems and Networks Workshop*. 2010, pp. 15–22.

[Zhe11] Zheng, Z. et al. “Co-analysis of RAS Log and Job Log on Blue Gene/P”. *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 2011, pp. 840–851.

[Zhe12] Zheng, Z. et al. “3-Dimensional root cause diagnosis via co-analysis”. *9th International Conference on Autonomic Computing, ICAC’12, San Jose, CA, USA, September 16 - 20, 2012*. 2012, pp. 181–190.

[Zhu19] Zhu, J. et al. “Tools and benchmarks for automated log parsing”. *International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*. 2019, pp. 121–130.