FAILURE ANALYSIS, MODELING, AND PREDICTION FOR BLUEGENE/L

BY YINGLUNG LIANG

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Yanyong Zhang

and approved by

New Brunswick, New Jersey

October, 2007

ABSTRACT OF THE DISSERTATION

Failure Analysis, Modeling, and Prediction for BlueGene/L

by Yinglung Liang

Dissertation Directors: Professor Yanyong Zhang

The growing computational and storage needs of scientific applications mandate the deployment of extreme-scale parallel machines, such as IBM's BlueGene/L, a 64K dual-core processor system. One of the challenges of designing and deploying such systems in a production setting is the need to take failure occurrences into account. Once the large scale system equipped with a failure predictability, the fault tolerance and resource management strategies of the system can be improved significantly, and its performance can be highly increased.

This dissertation is based on the Reliability, Availability and Serviceability (RAS) events generated by IBM BlueGene/L over a period of 142 days. Using these logs, we performed failure analysis, modeling, and prediction. Filters are created to reveal the system failure behaviors, three preliminary models are identified for the failures, and finally, three failure predictors are established for the system. We heavily use data mining and time series analysis techniques for this dissertation. Our comprehensive evaluation demonstrates that our Bi-Modal Nearest Neighbor predictor greatly outperforms the other two (RIPPER and LIBSVM based), leading to an F-measure of 70% and 50% for a 12-hour and 6-hour prediction window size.

Acknowledgements

This dissertation would not have been possible if it were not for the guidance, support, help and friendship of many people.

It is my pleasure and honor working with my Prof. Yanyong Zhang. Her inspiring guidance, constant encouragement, and complete support during our research course led me to a Ph.D. degree. In technical wise, she has great sensitivities to unknown research problems. In personal wise, she has tolerance to my mistakes and sympathy to my difficulties while my parents were in sickness. She is a nice person for people around her to remember. I feel so lucky to be her student. I also highly appreciate and value my co-advisor Prof. Hui Xiong's help in many ways. He has been a constant source of guidance and encouragement over the past year. Eventually, they provide me role models of being an intellectual scholar and a persistent researcher to pursue another goal in a higher level.

Additionally, I am so thankful to Ramendra Sahoo from IBM T. J. Watson Research Center, and Morris Jette from Lawrence Livermore National Laboratory. Without their helps, my dissertation wouldn't be able to proceed smoothly. Also, I won't be able to forget Lynn Ruggiero, and Dianne Coslit. They are like angels to me. Once I ran into a financial difficulty in 2002, Lynn lifted me up by providing me a work opportunity as a grader. From there on, I were able to receive scholarship and work with my advisor Prof. Zhang until now. Lynn and Dianne are recognized by students for their kindness and gentleness.

Finally, I am so glad being a Christian, and I praise the Lord who strengthens me up in days and nights during this long and unpredictable Ph.D. study.

Dedication

To my teachers and my parents ...

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Meta-scale scientific and engineering applications have been playing a critical role in many aspects of the society, such as economies of countries, health development, and military/security. The large processing and storage demands of these applications have mandated the deployment of extreme-scale parallel machines, such as IBM's BlueGene/L which accommodates 64K dual-core processors, the fastest supercomputer on the TOP500 supercomputer list [12]. BlueGene/L is currently deployed at Lawrence Livermore National Laboratory (LLNL), hosting applications that span several thousand processors, in the domains such as hydrodynamics, molecular dynamics, and climate modeling.

As applications and the underlying platforms scale to this level, failure occurrence as well as its impacts on system performance and operation costs, are becoming a critically important concern to the research community. Specifically, failures are becoming a norm, rather than an exception. Firstly, transient hardware failures are increasing, not just in memory structures and communication paths, but also in combinational circuits [41, 43]. Higher chip integration densities and lower voltages (to reduce power consumption), are making circuits more susceptible to bit flips [64]. Secondly, permanent hardware device failures are also a growing concern, especially with high power consumption for these large scale systems leading to immense heat dissipation, which in consequence can accelerate the failure rates for different devices [50, 52], including the CPUs, memory systems, and disk drives. External events, such as cooling system failures, also play a crucial role in the reliability of these systems that need to provide continuous operation over extended periods of time. Thirdly, in addition to the hardware issues, the sophisticated software that is taking on more duties, can (i) contain bugs, (ii) be difficult to comprehend and analyze (and thus may not be used in the right way), and (iii) age in quality over time [25, 57], which can all again lead to application crashes and/or system downtime.

Failures can make nodes unavailable, thereby lowering system utilization. Further, failures can cause applications executing on the nodes (probably having run for a long time) to abort, thus wasting the effort already expended. A long running (over several days) application that spans a large (hundreds/thousands) number of nodes, may find it very difficult to make any forward progress because of the failures. Additionally, some failures, especially those in network subsystem, may affect multiple applications that happen to run side by side. Eventually, the lower utilization and availability impacts the response times and system throughput of all the jobs, thus putting at risk the main motivation behind deploying these large scale systems. Our earlier studies [42, 65] show 100% worsening of job performance, with a 1 failure per day assumption (and we assume these failures affect one job at a time). As a real example, LLNL has witnessed frequent L1 cache failures for long running jobs, and in order to finish these jobs, L1 cache has been disabled for jobs longer than 4 hours, which results in much prolonged execution times for these jobs.

In addition to lowering system performance and availability, failures can also greatly increase the system management costs. The system administrator may need to detect failure occurrence, diagnose the problem, and figure out the best sequence of remedial actions. On the hardware end, this may entail resetting a node, changing the motherboard/disk, etc., and on the software end it may require migrating the application, restarting the application, re-initializing/rejuvenating [57] a software module, etc. In addition to the time incurred by such operations during which the system (or at least the affected nodes) may be unavailable, personnel time needs to be allotted for this purpose. The resulting personnel involvement will increase the Total Cost of Operation (TCO), which is becoming a serious concern in numerous production environments [11, 29].

One of the challenges when designing and deploying these systems in a production setting is the need to take failure occurrences into account. Earlier work has shown that conventional runtime fault-tolerant techniques such as periodic checkpointing are not effective to the emerging systems. Instead, the ability to predict failure occurrences can help develop more effective checkpointing strategies and even more sophisticated resource management strategies so that in the presence of system failures, applications that span several thousand processors, in the domains such as hydro-dynamics, molecular dynamics, and climate modeling, are able to continue their progresses. Failure prediction, however, has long been regarded as a challenging research problem, mainly due to the lack of realistic and informative failure data from actual production systems.

To attack this challenge, my thesis is based on the Reliability, Availability and Serviceability (RAS) events generated by IBM BlueGene/L over a period of 142 days. Using these logs, we answered the critical question: whether the failures can be predictable? If so, what are the prediction models?

Along the path leading us to establish prediction models, we first found that the raw RAS logs can not be directly used. Such large lags contain a huge amount of redundancy temporally and spatially. In order to pinpoint unique FATAL failures, resolve the complexity of the system failure behavior, reveal the system failure patterns, and perform further system failure analysis, the RAS logs must be carefully filtered. In this thesis, we developed two filtering tools, the Spatial-Temporal filter and the Adaptive Semantic filter. With any of these filters, the logs are condensed 99.7%. Based on the filtered events, we established three preliminary prediction models, the first one based on time between failure (TBF) characteristics, the second one based on spatial skewness of the failures, and the third one based on the correlation of non-fatal and fatal events. Our results show the three models can capture 62.3% of the fatal events.

After studying the feasibility of failure prediction, we applied time series analysis and data mining techniques to build failure predictors. Time series analysis reveals the randomness of failure pattern. Moreover, AR, ARMA models are applied for building time-between-failure predictors. Next, we developed three systematic failure predictors using data mining techniques. They are RIPPER, SVM, and Nearest Neighbor methods. We first tried and evaluated a number of ways to select features that thoroughly describe and capture failure behaviors. Then, we exercised the three classification concepts to build online predictors. Finally, we evaluated the performance of the three predictors. We concluded that among the 3 predictors, Bi-Modal Nearest Neighbor predictor is the best early stage failure warning system that is able to issue alarms to the upcoming failure events with high prediction accuracy of .7039 F-Measure in average of all cross validations.

1.1 Filters

While designing and deploying a large scale system in a production setting, there is the need to take failure occurrences, whether it be in the hardware or in the software, into account. It needs to continuously monitor and log failures over extended periods of time, across thousands of compute processors, and the resulting event logs can be used for failure analysis and effective system management. However, the event logs can get voluminous because of the large temporal and spatial dimensions, and can also contain records which may not all be distinct. Consequently, filtering these logs towards isolating unique failures is crucial for subsequent analysis.

In this study, we first develop and test two filtering mechanisms using the event logs collected from IBM BlueGene/L, currently the fastest machine in the world, over a period of 142 days starting from August 2, 2005. The first technique is spatio-temporal filtering (STF), which involves three steps: extracting and categorizing failure events, coalescing event clusters from the same locations, and finally coalescing event clusters across different locations. Furthermore, based on the lessons learnt from the experience with the spatio-temporal filter, we develop an Adaptive Semantic Filtering (ASF) method, which exploits the semantic correlation between two events, and dynamically adapts the correlation threshold based on the temporal gap between the events. We have validated the ASF method using the failure logs collected from BlueGene/L. Our experimental results show that ASF can effectively remove redundant entries in the logs, and the filtering results can serve as a good base for future failure analysis studies. Using these two approaches, we can substantially compress these logs, removing over 99% of the 1,792,598 original entries, and more accurately portray the failure occurrences on this system.

1.2 Preliminary Failure Models

After the redundancy had been removed, failure analysis and modeling were the next research topics. As shown in earlier work that conventional runtime fault-tolerant techniques such as periodic checkpointing are not effective to the emerging systems. Instead, the ability to predict failure occurrences can help to develop more effective checkpointing strategies, and more sophisticated resource management strategies to have applications run without stop in the presence of system failures. However, failure prediction has long been regarded as a challenging research problem, mainly due to the lack of realistic and informative failure data from actual production systems. In this study, we have collected both RAS event logs and job logs from BlueGene/L over a period of 109 days (from August 2, 2005 to November 18, 2005). We have carefully studied the impact of fatal failure events on job executions. More importantly, we have investigated the characteristics of fatal failure events, as well as the correlation between fatal events and non-fatal events. Based on the observations, we have developed three simple yet effective failure prediction methods, which can predict around 80% of the memory and network failures, and 47% of the application I/O failures. In total, the three models can capture 62.3% of the fatal events.

1.3 Failure Predictors

Further, in order to develop an online failure predictor, we apply time series analysis and data mining techniques for failure analysis, modeling, and prediction. In the time series study, it reveals the randomness of the failure sequence, and AR, ARMA are used for time between failure prediction. Next, in the data mining study, we first show how the event data can be converted into a data set that is appropriate for running classification techniques. Then, we apply four classifiers on the data, including RIPPER (a rule-based classifier), Support Vector Machines (a hyperplane based), a traditional Nearest Neighbor method, and a customized Nearest Neighbor method (a Bi-Modal Nearest Neighbor classification). Finally, we show that the Bi-Modal Nearest Neighbor approach can effectively encapsulate expert knowledge to create an adaptable algorithm that can substantially outperform RIPPER, SVMs, and the traditional Nearest Neighbor method in terms of both coverage and precision. The results suggest that the Bi-Modal Nearest Neighbor approach can be used to alleviate the impact of failures.

Chapter 2

The Event Filters

Understanding the failure behavior in large scale parallel systems is crucial towards alleviating the problems addressed earlier. This requires continual online monitoring and analysis of events/failures on these systems over long periods of time. The failure logs obtained from such monitoring can be useful in a number of ways. First, the failure logs can be used by hardware and system software designers during early stages of machine deployment in order to get feedback about system failures and performance. It can also help system administrators for maintenance, diagnosis, and enhancing the overall health (up-time). Finally, it can be useful in fine-tuning the runtime system for checkpointing (e.g. modulating the frequency of checkpointing based on error rates), job scheduling (e.g. allocating nodes which are less failure prone), network stack optimizations (e.g. employing different protocols and routes based on error conditions), etc.

Raw event logs, however, cannot be directly used for the above purposes; redundant and/or unimportant information must be first removed. First, many recorded warnings do not necessarily lead to a failure, and such warnings need to be removed. Also, the same error could get registered multiple times in the log, or could get flagged in different ways (e.g. network busy and message delivery error). For instance, we have noticed that a single EDRAM error on BlueGene/L is record in 20 successive entries. Consequently, it becomes imperative to filter this evolving data and isolate the specific events that are needed for subsequent analysis.

This chapter presents our effort in filtering the system event/failure logs from the IBM BlueGene/L machine over a period of 142 days, and the design of two filters that we have developed. In the raw event logs from this system, we have over 1,792,598 events with 281,441 fatal failures. In the development of the first filter, spatio-temporal filter (*STF*), we first perform a temporal filtering step by eliminating event records at a node which are within twenty minutes of each other. This step brings down the fatal failures to 145,376. However, when we closely examine the log after the temporal filtering, we still find that not all events are independent/unique. We break down the events based on the system components - the memory hierarchy, the networks, the node cards, the input/output nodes, and the link cards - impacted by these failures. We then show how the event records for each of these components can further be filtered by removing the spatial redundancy among them. At the end of all such filtering, we isolate the number of actual errors over this 142 day period to 819, which represents a reduction of 99.710% of the fatal failures reported in the original logs.

While the STF method is able to correctly filter out many redundant fatal events, it is not suitable for online filtering, because it requires extensive domain knowledge and manual operations. To this end, we further propose an **A**daptive **S**emantic **F**iltering (ASF) method. The key idea behind ASF is to use semantic context of event descriptions to determine whether two events are redundant. ASF involves three steps. Firstly, it extracts all the keywords from event descriptions and builds a keyword dictionary. Secondly, it converts every event description into a binary vector, with each element representing whether the description contains the corresponding keyword. Using these vectors, we can compute the correlation between any two events. Thirdly, it determines whether two events are redundant based on their correlation as well as the temporal gap between them. Specifically, the choice of the correlation threshold is not uniform, but varies according to the time window between two events. For example, two events that are temporally close are considered redundant even with low correlation, but two far-away events are only considered redundant when their correlation is very high. Compared to STF and other existing filtering tools, ASF considers both semantic correlation and temporal information, and therefore, the filtering results more accurately capture the real-world situations. More importantly, ASF does not require much human intervention, and can thus be easily automated. Consequently, ASF is one big step forward towards self-managing online monitoring/analysis for large-scale systems.

After applying ASF on the failure logs from IBM BlueGene/L, we find that ASF is more accurate than STF in filtering fatal events due to its consideration of semantic correlation between events. Also, due to its low overhead, we can use ASF to filter non-fatal events as well. We find that ASF is quite effective for all the severity levels, with the resulting compression ratio always below 3%, and often below 1%. After merging filtering results for all severity levels, we find that events naturally form “clusters”, with each cluster starting with non-fatal events, followed by one or more fatal events. These clusters can help visualize how events evolve with increasing severity. Indeed, this can serve as a good basis for further analysis and investigations.

The rest of this chapter is organized as follows. We give an overview of the BG/L architecture, together with specifics on the system where the logs were collected in Section 2.1. Subsequently, the details of the two filtering algorithms are given in section 2.2 and 2.3. In section 2.3.3, we provide the ASF filtering results. The next section briefly summarizes related work. Finally, section 2.5 concludes with a summary of the results and identifies directions for future work.

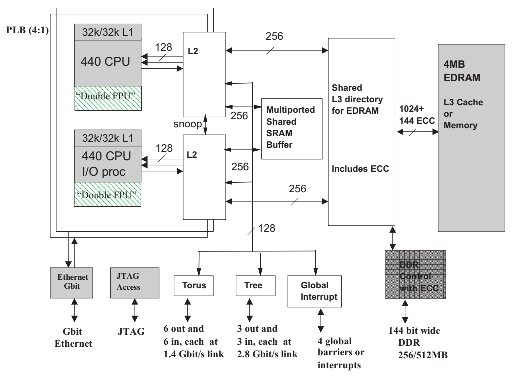
2.1 BG/L Architecture and Error Logs

In this chapter, we give an overview of the BG/L on which the event logging has been performed, together with details on the logs themselves.

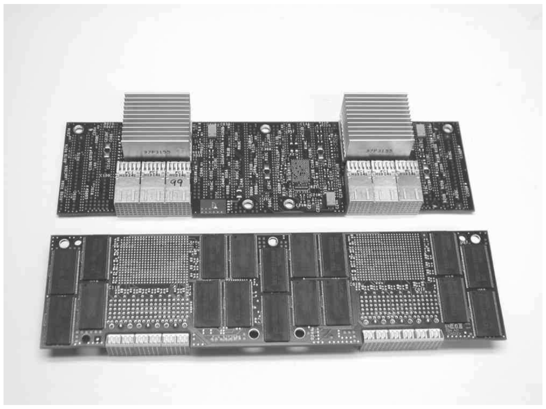
2.1.1 BG/L Architecture

In this study, we use event/failure logs collected from IBM BlueGene/L, housed at Lawrence Livermore National Laboratory (LLNL), which currently stands at number 1 in the Top 500 list of supercomputers [12]. The machine had been up since August 2, 2005 and had been primarily running a growing list of applications including hydrodynamics, quantum chemistry, molecular dynamics, and climate modeling. These applications usually require a large number of processors, and the granularity of job allocation on BlueGene/L is usually a midplane, which consists of 1K compute processors. The machine has 128 midplanes comprising 64K compute chips. The compute chip is an Application Specific Integrated Circuit (ASIC). It is built by using *system-on-a-chip* technology. All functions of a node, except for main memory, are integrated onto a single ASIC.

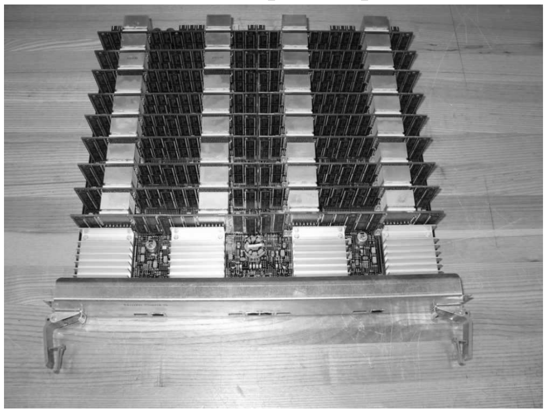
As shown in Figure 2.1 (a), each compute chip consists of two PPC 440 cores (processors) clocked at 700MHz, with a 32KB L1 cache and a 2 KB L2 (for prefetching) for each core. Associated with each core is a 64-bit "double" floating point unit (FPU) that can operate in Single Instruction Multiple Data (SIMD) mode. The 440 CPU core does not implement the necessary hardware to provide symmetric multiprocessor (SMP) support; therefore, the two cores are not L1 cache coherent. The L2 and L3 caches are coherent between the two cores, meaning that the cores share the 2 KB L2 cache and the 4MB EDRAM L3 cache. In addition, a shared fast SRAM array is used for communication between the two cores. L1 caches in the cores are parity protected, and so are most of the internal buses. The L2 caches and EDRAM are ECC protected. Each chip has the following network interfaces: (i) a 3-dimensional torus interface that supports point-to-point communication, (ii) a tree interface that supports global operations (barriers, reductions, etc.), (iii) a gigabit ethernet interface that supports file I/O and host interfaces, (iv) a control network that supports booting, monitoring and diagnostics through JTAG access, and (v) a network for broadcasts and combining operations, such as those used



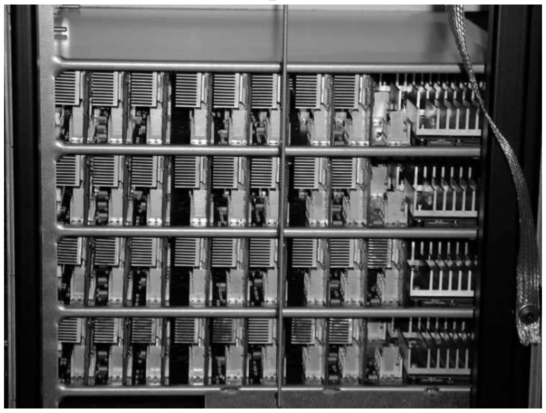
(a) Compute chip



(b) Compute card



(c) Node card



(d) Midplane

Figure 2.1: BG/L Hardware

in the MPI collective communication library.

A compute card (Figure 2.1 (b)) contains two such chips, and also houses either 512 MB or 1 GB of Synchronous Dynamic Random Access Memory (SDRAM) on the card. In this dissertation, we use the term *compute card* and compute node interchangeably. A node card shown in Figure 2.1 (c) contains 16 such compute cards, and a midplane (Figure 2.1 (d)) holds 16 node cards (a total of 512 compute chips or 1K processors). The BG/L has 32768 compute cards, 2048 node cards, and 128 midplanes, that are housed in 64 racks. A Input/Output (I/O) node is similar to the compute node with two differences. First, I/O nodes do not connect to the torus network. Second, I/O nodes always have Ethernet interfaces and an amount of memory usually larger than that of compute nodes. There are 4 such I/O cards (i.e., 8 I/O chips) for each midplane on the machine, i.e. 1 I/O chip for every 64 compute chips. All compute nodes go through the gigabit ethernet interface to these I/O chips for their I/O needs.

In addition, a midplane also contains 24 ASICs/link chips acting as midplane switches that are equipped in 4 link cards for connecting with other midplanes. When crossing a midplane boundary, BG/L's torus, global combining tree and global interrupt signals pass through these link chips. The BG/L, thus, has 3072 link chips totally. A midplane also has 1 service card that performs system management services such as distributing the system clock, verifying the heart beat of the nodes, monitoring errors, and providing other rack control functions. The card also connects individual 100MHz Fast Ethernet connections from link cards and node cards to a single Gigabit Ethernet link leaving the rack. This card has much more powerful computing abilities, and runs a full-fledged IBM DB2 database engine for event logging [39].

In most cases, a midplane is the granularity of job allocation, i.e., a job is assigned to an integral number of midplanes. Though rare, it is also possible to subdivide a midplane, and allocate part of it (an integral number of node cards) to a job. However, in this case, a job cannot run on node cards across different midplanes. Compute cards are normally shut down when they are not running any job. When a job is assigned, the card is reset and the network is configured before execution begins.

2.1.2 RAS Event Logs

Error events are logged through the Machine Monitoring and Control System (MMCS). There is one MMCS process per midplane, running on the service node. Moreover, there is a polling agent that runs on each compute chip. Errors detected by a chip are recorded in its local SRAM via an interrupt. The polling agent at some later point would pick up the error records from this SRAM and ship them to the service node using a JTAG-mailbox protocol. The frequency of the polling needs to be tuned based on the SRAM capacity, and speeds of the network and the compute nodes. The failure logs that we have obtained are collected at the frequency of less than a millisecond.

After procuring the events from the individual chips of that midplane, the service node records them through a DB2 database engine. These events include both hardware and software errors at individual chips/compute-nodes, errors occurring within one of the networks for inter-processor communication, and even errors such as temperature emergencies and fan speed problems that are reported through an environmental monitoring process in the backplane. Job related information is recorded in job logs.

We have been collecting *reliability*, *availability*, and *serviceability* (RAS) event logs from BlueGene/L since August 2, 2005. Up to the date of December 21, 2005, we have totally 1,792,598 entries. These entries are records of all the RAS related events that occur within various components of the machine. Information about scheduled maintenances, reboots, repairs, and job related information such as starting time, ending time, and job size is not included. Each record of the logs contains a number of attributes to describe the event. We name eight of them used in our studies as follows:

- *RECID* is the sequence number of an error entry, which is incremented upon each new entry being appended to the logs.

- *EVENT_TYPE* specifies the mechanism through which the event is recorded, with most of them being through RAS [24].

- *SEVERITY* can be one of the following levels - INFO, WARNING, SEVERE, ERROR, FATAL, or FAILURE - which also denotes the increasing order of severity. INFO events are more informative in nature on overall system reliability, such as "a torus problem has been detected and corrected", "the card status has changed", and "the kernel is generating the core", etc. WARNING events are usually associated with node-card/link-card/service-card not being functional. SEVERE events give more details about why these cards may not be functional (e.g."link-card is not accessible", "problem while initializing link/node/service card", and "error getting assembly information from the node card", etc.). ERROR events report problems that are more persistent and further pinpoint their causes ("Fan module serial number appears to be invalid", "cable x is present but the corresponding reverse cable is missing", "Bad cables going into the linkcard", etc.). All of these above events are either informative in nature, or are related more to initial configuration errors, and are thus relatively transparent to the applications/runtime environment. However, FATAL or FAILURE events (such as "uncorrectable torus error", "memory error", etc.) are more severe, and usually lead to application/software crashes. Note that, in this dissertation, we do not focus on application-domain software failures.

- *FACILITY* denotes the component where the event is flagged, which can be one of the following 10 components: LINKCARD, APP, KERNEL, HARDWARE, DISCOVERY, MMCS, MONITOR, CMCS, BGLMASTER, SERV_NET. Events with LINKCARD facility report problems with midplane switches, which is related to communication between midplanes. APP events are those flagged in the application domain of the computing chips. Most of these are reported by the I/O demon regarding invalid path names, wrong access permissions, severed links, etc. Events with KERNEL facility are those reported by the OS kernel domain of the computing chips, which are usually in the memory or network subsystem. These could include memory parity/ECC errors in the hardware, bus errors due to wrong addresses being generated by the software, torus errors due to links failing, etc. Events with HARDWARE facility are usually related to the hardware operations of the system (e.g. "node card power module is not accessible", "node card is not fully functional", etc). Events with DISCOVERY facility are usually related to resource discovery and initial configurations within the machine (e.g. "cannot get assembly information for a node card", "fan module is missing", etc), with most of these being at the INFO or WARNING severity levels. MMCS facility reports mmcs database server, ciodb server, idoproxydb server, and ido packet related issues, and majority of these issues are at INFO or ERROR severity levels. CMCS, BGLMASTER SERV_NET facility errors are again mostly at the INFO level, which report events in the operation of the CMCS, BGLMASTER and the service network. Finally, events with MONITOR facility are usually related to the power/temperature/wiring issues of link-card/node-card/service-card. Nearly all MONITOR events are in the FAILURE severity level.

- *EVENT_TIME* is the time stamp associated with that event.
- *JOB_ID* denotes the job that experiences this event. This field is only valid for those events reported by computing/IO chips.
- *LOCATION* of an event (i.e., which chip/service-card/ node-card/link-card experiences the error), can be specified in two ways. It can either be specified as (i) a combination of job ID, processor, node, and block, or (ii) through a separate location field. We mainly use the latter approach (location attribute) to determine where an error takes place.

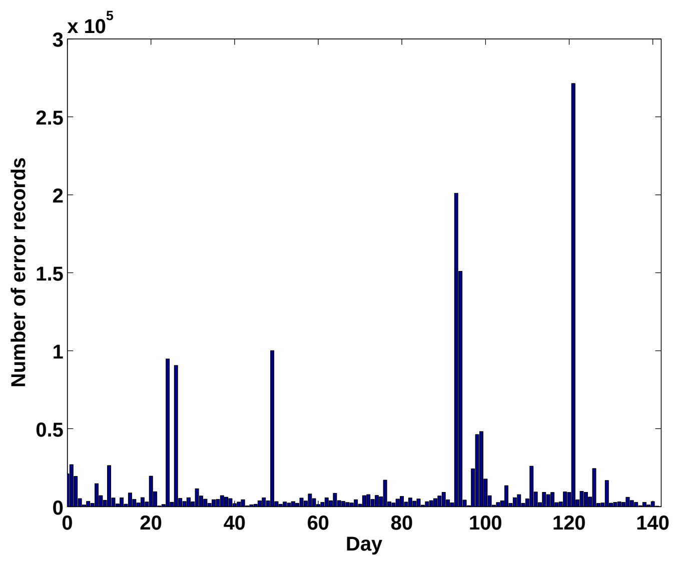


Figure 2.2: Temporal distribution of raw events in the log between August 2 and December 21, 2005

- *ENTRY_DATA* gives a description of the event.

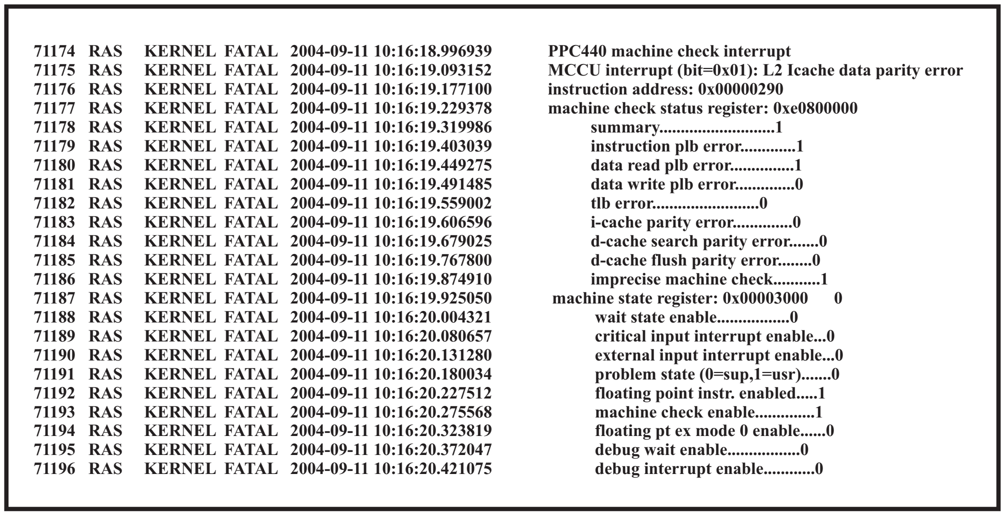
Between August 2, 2005 and December 21, 2005 (142 days), there are totally 1,792,598 events in the raw log. Figure 3.1 shows the temporal distribution of these events over this time period at a daily granularity.

2.2 A Spatial-Temporal Filter

While event logs can help one understand the failure properties of a machine to enhance the hardware and systems software for better fault resilience/tolerance, it has been recognized [17, 26, 30, 56] that such logs must be carefully filtered and preprocessed before being used in decision making since they usually contain a large amount of redundant information. As today's machines keep boosting their logging granularity, filtering is becoming even more critical. Further, the need to continuously gather these logs over extended periods of time (temporal) and across thousands of hardware resources/processors (spatial) on these parallel machines, exacerbates the volume of collected data. For instance, the logging tool employed by BG/L has generated 1,792,598 entries over a period of 142 days. The large volume of the raw data sets, however, should not be simply interpreted as a high system failure rate. Instead, it calls for an effective filtering tool to parse the logs and identify failure issues, if any, from the huge event records. In this section, we present our first filtering algorithm, which involves three steps: first extracting and categorizing failure events from the raw logs, then performing a temporal filtering step to remove duplicate reports from the same location, and finally coalescing failure reports across multiple locations. Using these techniques, we can substantially compress the generated error logs and more accurately portray the failure occurrences on BG/L for subsequent analysis.

2.2.1 Step I: Extracting and Categorizing Failure Events

Even though informational warnings and other non-critical errors may be useful for system diagnostics and other issues, our main focus is on isolating and studying the more serious hardware and software failures that actually lead to application crashes since those are needed to revise software/hardware revisions and design more effective fault resilience/tolerance mechanisms. Consequently, we are mainly interested in events at severity level of either FATAL or FAILURE, which are referred to as *failures* in this dissertation. As a result, we first screen out events with lower severity levels such as INFO, WARNING, SEVER, and ERROR. This step removes 1,509,449 event records from 1,792,598 total entries, leaving only 283,149 failures, which constitute 15.795% of the total event records. Next, we remove application level



```
71174  RAS   KERNEL  FATAL  2004-09-11 10:16:18.996939   PPC440 machine check interrupt
71175  RAS   KERNEL  FATAL  2004-09-11 10:16:19.093152   MCCU interrupt (bit=0x01): L2 Icache data parity error
71176  RAS   KERNEL  FATAL  2004-09-11 10:16:19.177100   instruction address: 0x00000290
71177  RAS   KERNEL  FATAL  2004-09-11 10:16:19.229378   machine check status register: 0xe0800000
71178  RAS   KERNEL  FATAL  2004-09-11 10:16:19.319986           summary..........................1
71179  RAS   KERNEL  FATAL  2004-09-11 10:16:19.403039           instruction plb error............1
71180  RAS   KERNEL  FATAL  2004-09-11 10:16:19.449275           data read plb error..............1
71181  RAS   KERNEL  FATAL  2004-09-11 10:16:19.491485           data write plb error.............0
71182  RAS   KERNEL  FATAL  2004-09-11 10:16:19.559002           tlb error........................0
71183  RAS   KERNEL  FATAL  2004-09-11 10:16:19.606596           i-cache parity error.............0
71184  RAS   KERNEL  FATAL  2004-09-11 10:16:19.679025           d-cache search parity error.......0
71185  RAS   KERNEL  FATAL  2004-09-11 10:16:19.767800           d-cache flush parity error........0
71186  RAS   KERNEL  FATAL  2004-09-11 10:16:19.874910           imprecise machine check...........1
71187  RAS   KERNEL  FATAL  2004-09-11 10:16:19.925050    machine state register: 0x00003000      0
71188  RAS   KERNEL  FATAL  2004-09-11 10:16:20.004321            wait state enable................0
71189  RAS   KERNEL  FATAL  2004-09-11 10:16:20.080657            critical input interrupt enable...0
71190  RAS   KERNEL  FATAL  2004-09-11 10:16:20.131280            external input interrupt enable...0
71191  RAS   KERNEL  FATAL  2004-09-11 10:16:20.180034            problem state (0=sup,1=usr).......0
71192  RAS   KERNEL  FATAL  2004-09-11 10:16:20.227512            floating point instr. enabled.....1
71193  RAS   KERNEL  FATAL  2004-09-11 10:16:20.275568            machine check enable.............1
71194  RAS   KERNEL  FATAL  2004-09-11 10:16:20.323819            floating pt ex mode 0 enable......0
71195  RAS   KERNEL  FATAL  2004-09-11 10:16:20.372047            debug wait enable................0
71196  RAS   KERNEL  FATAL  2004-09-11 10:16:20.421075            debug interrupt enable...........0
```

Figure 2.3: A typical cluster of failure records that can be coalesced into a single memory failure. For each entry, the following attributes are shown: id, type, facility, severity, timestamp, and description.

of events resulting from application failures or applications being killed by signals from the console (these entries usually have APP facility and their event descriptions start with "Applications killed by signal x"), since these are not failures caused by the underlying system (hardware or software). This step further brings down the log size from 283,149 to 262,043 entries.

Instead of lumping together failures from different components of the machine, we categorize them into six classes: (i) memory failures, denoting faults in any part of the memory hierarchy on all the compute nodes; (ii) network failures, denoting exceptions or faults from the network components of the compute nodes; (iii) input/output (I/O) failures, denoting errors within I/O operations including problems of connecting to external file systems such as Network File System (NFS), other computing resources, and I/O communications; (iv) link card failures denoting failures within the link cards/midplane switches; (v) node card failures, denoting problems with the operations of node cards such as power deactivated on node cards or Power_Good signal lost; and (vi) service card failures, denoting failures within the service cards. These six classes cover all the major components of the BG/L hardware. In later sections, we will show that different classes of failures have different properties. Among 262,043 total failures, we have 16,033 memory events, 10,593 network events, 233,386 I/O events, 1507 link card events, 452 node card events, and 72 service card events.

2.2.2 Step II: Compressing Event Clusters at a Single Location

When we focus on a specific location/component (i.e., a specific chip/node-card/link-card/service-card), we notice that events tend to occur in bursts, with one occurring within a short time window (e.g., a few seconds) after another. We call these an event *cluster*. It is to be noted that the entire span of a cluster could be large, e.g., a couple of days, because of a large cluster size. The event descriptions within a cluster can be identical, or completely different. An event cluster can be formed for different reasons. First, the logging interval can be smaller than the duration of a failure, leading to multiple recordings of the same event. Second, a failure can quickly propagate in the problem domain, causing other events to occur within a short time interval. Third, the logging mechanism sometimes records diagnosis information as well, which can lead to a large number of entries for the same failure event.

We give below some example clusters that we find in the BG/L logs:

- Failures in the memory hierarchy (e.g. parity for I/D-L2, ECC for EDRAM L3) typically lead to heterogeneous event clusters. Figure 2.3 illustrates such a cluster. Examining these entries carefully, we find that they are all diagnostic information for the same fatal memory event, namely, an L2 I-cache data parity error in this example. In fact, as soon as a memory failure is detected upon the reference to a specific address, the OS kernel performs a thorough machine check by taking a snapshot of the relevant registers. It records the instruction address that incurred the fault, information on which hardware resources incurred the failure(s) - the processor local bus (plb), the TLB, etc. - and a dump of status registers.

- Failures in the other components tend to form homogeneous clusters. For example, in a case of the I/O failure, the description "invalid or missing program image, Exec format error" appears 20480 times between timestamp 2005-08-12-23.20.01.496813 and timestamp 2005-08-12-23.58.41.334754 across 1024 I/O chips. Similar trends are also observed at network failures, but network failure clusters are usually smaller in size.

In order to capture these clusters and coalesce them into a single failure, we have used a simple threshold-based scheme. We first group all the entries from the same location, and sort them based on their timestamps. Next, for each entry, we compare its timestamp and job

ID with its previous record, and only keep the entry if the gap is above the threshold T_{th} or the job ID is unique. We would like to emphasize that, as discussed above, an event cluster contains failures of the same type; for instance, a network failure and a memory failure should not belong to the same cluster. Hence, when we cluster the events, if two subsequent events are of different types, no matter how temporally close they are, we put them into different clusters. Note that this scheme is different from the one employed in our earlier work [46] in that the earlier scheme only filters out entries with identical descriptions, which is insufficient in this study.

If a data set has n entries before filtering, and m entries after filtering, then the ratio of $\frac{n-m}{n}$, referred to as *compression ratio*, denotes how much the data set can be reduced by the filtering algorithm. The compression ratios are governed by the distributions of clusters for that data set and the choice of the threshold T_{th}. With a specific T_{th}, a data set with larger clusters has a higher compression ratio. At the same time, a larger T_{th} also leads to a larger compression ratio. Table 2.1 summarizes the number of remaining failure entries after applying this filtering technique with different T_{th} values. Specifically, $T_{th} = 0$ corresponds to the number of events before filtering. From the table, we have the following observations. First, even a small T_{th} can significantly reduce the log size, resulting in a large compression ratio. Second, different components have different compression ratios. For example, network failures have a small compression ratio, e.g., 17.173% with an 1-hour threshold, because they usually have small cluster sizes associated to a location. Third, when the threshold reaches a certain point, although the compression ratio still keeps increasing, the increasing rate of the compression ratio is decreasing until the compression ration reaches a constant. This is true because we constrain an event cluster only contains failures of the same job ID and at the same location. At the same time, a threshold larger than 1 hour is undesirable because the logging interval of BG/L

number of failures	memory	network	I/O	link card	node card	service card
T_{th} = 0	16,033	10,593	233,386	1507	452	72
T_{th} = 30 second	466	9,486	226,514	715	275	10
T_{th} = 2 minutes	443	9,486	160,390	575	273	10
T_{th} = 5 minutes	432	9,483	143,482	568	272	10
T_{th} = 20 minutes	427	9,285	135,626	558	272	10
T_{th} = 1 hour	417	8,762	120,897	546	272	10

Table 2.1: The impact of T_{th} on compressing different failures. T_{th} = 0 denotes the log before any compression.

memory failures	**network failures**	**I/O**
machine check interrupt	Error receiving packet on tree network	Error loading FILE
L2 dcache parity error	bad message header	Error reading message prefix
data TLB error interrupt	tree/torus link training failed	Lustre mount FAILED

link card failures	**node card failures**	**service card failures**
Temperature Over Limit on link card	I2C Operation failed	Fan Module failed
Link PGOOD error latched on link card	CONTROL Operation failed	Segmentation fault
Midplane Switch Controller Failed	Hardware monitor caught Broken Pipe	N/A

Table 2.2: Three types of failures for each subsystem

is significantly smaller than that, and a large temporal threshold may cover more than one failure clusters resulting in an over filtered results. As a result, by carefully studying the failure cluster patterns, we choose 20 minutes as the threshold to temporally coalesce event clusters at a location/component.

After compressing all the event clusters, we have 14 types of memory failures, 13 types of network failures, 11 types of I/O failures, 7 types of midplane switch failures, 9 types of node card failures, and 2 types of service card failures. For each component, we show the descriptions of three typical failure types in Table 2.2 in the decreasing order of occurrence frequency. (These failures will be further explained in later sections.)

2.2.3 Step III: Failure-Specific Filtering Across Locations

Let us now take a closer look at the resulting failure logs from the previous filtering step. The time-series dataset contains periods with multiple events per second and other periods with no events over many consecutive seconds. Hence, our analysis considers two different but related views of these time-series datasets, i.e., the number of failure records per time unit, or *rate process*, and the time between failures, or *increment process*, for the sequences of failure entries. Note that these two processes are the inverse processes of each other.

Some statistics of the raw rate processes (i.e., number of failure records within each subsystem, average number of records per hour, variance and maximum number of records per hour) and the raw increment processes (i.e., the average, variance, and maximum number of hours between failure records within each subsystem) are demonstrated in Tables 2.3 (a) and (b). It is noticed that even after clustering events by applying a 20-minute temporal threshold at each location (Step II), the number of failure records (146178 across all components) is still quite large especially for I/O and network failures. In this section, we set out to investigate the reason

	entire system	memory	network	I/O	link card	node card	service card
total failure records	146178	427	9285	135626	558	272	10
Average daily failure records	1029.4	3.0070	65.3873	955.1127	3.9296	1.9155	.0704
Variance of daily failure records	1308700	9.6808	63119	1272800	1846.7	468.7162	.4631
Maximum daily failure records	5152	16	2049	5150	512	258	8

(a) rate process

	entire system	memory	network	I/O
MTBF (hours)	0.0230	7.7042	.3474	.0247
Variance of times between failure records (hours)	1.2145e3	8.0843e5	1.4967e5	3.3795e3
Maximum time between failure records (hours)	51.7592	133.4164	371.8439	143.4325

	link card	node card	service card
MTBF (hours)	5.3010	7.8062	260.7104
Variance of times between failure records (hours)	8.7049e6	1.6305e7	1.4157e9
Maximum time between failure records (hours)	744.8197	913.3453	1.8820e3

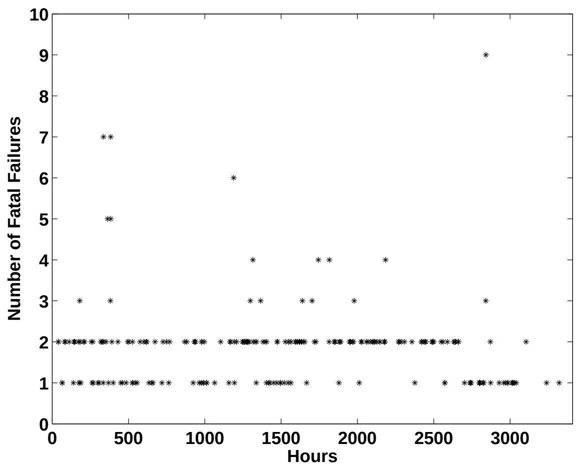
(b) increment process

Table 2.3: General statistics about failure logs after applying step II.

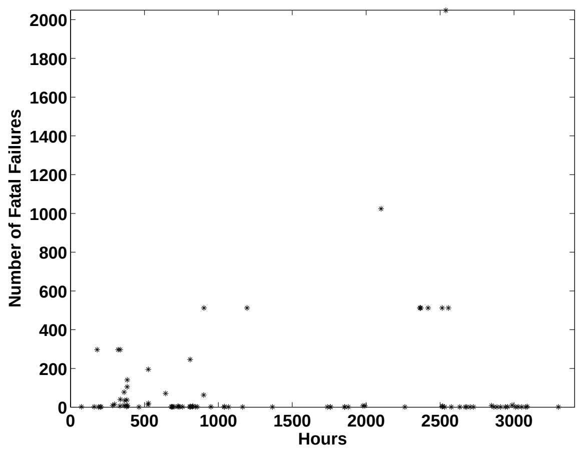
behind this phenomenon, and in the interest of space, we only focus on memory, network and I/O failures in the rest of this section because they are more prevalent than others.

In order to study this more closely, we plot the rate processes of failures in Figures 2.4(a)-(c) at an hourly granularity. The figures clearly show that some failure occurrences are still skewed - tens, hundreds, or even thousands of failures can be reported within an hour or within an interval of hours. For instance, out of the entire 3408-hour period, 22% of the network failures (2048 out of 9285) occur in a single hour (i.e., the 2539th hour), and 77.2% of the network failures take place within an interval of 10 hours. In the case of the I/O failures, there are 198 hours containing at least 100 I/O failures, and 98.26% of the I/O failures occurred in these 198 hours. It is also noticed that memory failures are less bursty than others. For example, as shown in Figure 2.4(a), only 18 out of 3408 data points (142 days) contain 3 or more failures. In these 18 hours, 79 out of 427 total memory failures are accumulated. The rest of the 348 memory failures are distributed in 212 data points.

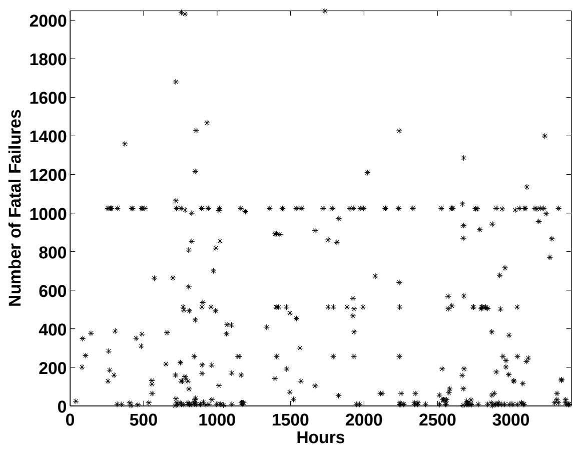
Figures 2.5(a)-(c) reiterate the above observations by plotting the cumulative distribution function (CDF) of the inter-failure times. Please note that the x-axes are in log scale to accommodate wide variances of the inter-failure times. As indicated in Figures 2.5(b)(c), 98.66% of the network failures follow their preceding network failures within 1 second, and 99.24% of the I/O failures follow their preceding I/O failures within 1 second. As for memory failures, Figure 2.5(a) and Figure 2.4(a) consistently demonstrate that memory failures are less bursty than others.



(a)memory failure records



(b) network failure records

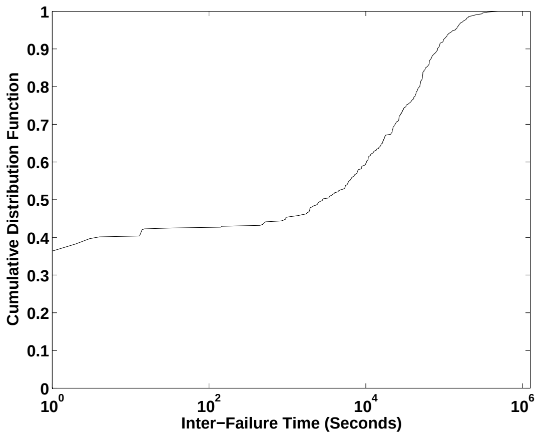


(c) I/O failure records

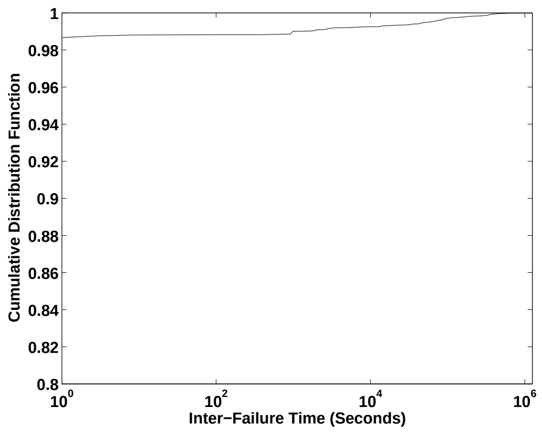
Figure 2.4: Rate processes of failure entries after applying filtering step II with T_{th} = 20 minutes.

One may wonder why many failures are occurring in bursts, given that we have already used a temporal threshold of 20 minutes to filter out event clusters in Step II. A quick investigation reveals that step II only eliminates duplicate records that occur at same locations, and it is quite possible that the large number of failures across locations/chips also refer to the same failure. In order to take a closer look at the spatial correlation of these failures, we plot the number of failures across the whole 128 midplanes.

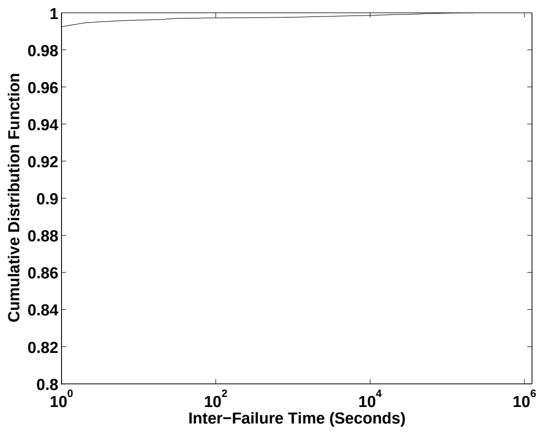
Network Failures: Figure 3.10(b) shows that there are two groups of adjacent midplanes report large numbers of network failures. They are midplanes with ID 92, 93, 94, 95, and midplanes with ID 100, 101, 102, 103. Figure 2.7 looks at the failures reported by these midplanes in a great detail. These plots indicate that node cards within the same midplane report a similar



(a) memory failure records



(b) network failure records

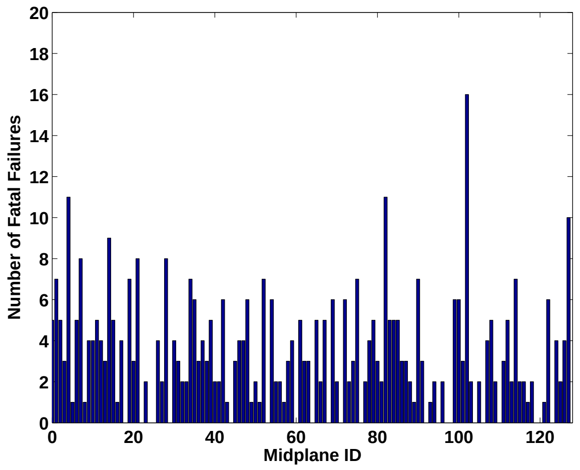


(c) I/O failure records

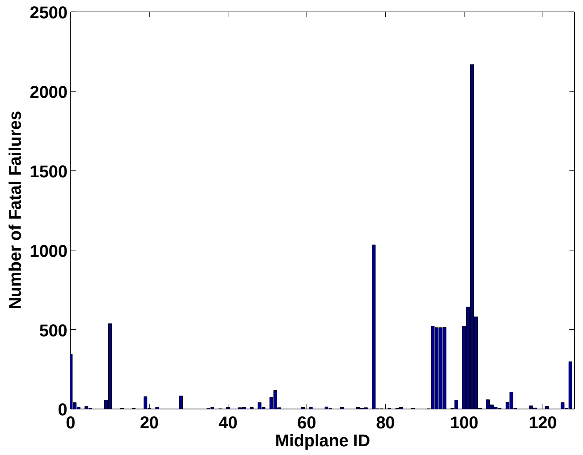
Figure 2.5: Inter-failure times after applying filtering step II with T_{th} = 20 minutes. The x-axes are plotted using log scale.

number of failures, and some times, even node cards from neighborhood midplanes report a similar number of failures. This suggests that chips within a midplane or from different midplanes may report same network problems, and we need to coalesce these redundant reports. A more direct evidence is presented in Figures 2.7(b) and (d), which show the time (on the x axis) and chips/locations (on the y axis) of all the network failures in these 8 midplanes. It is clear that many chips report failures almost at the same time. For example, in Figure 2.7(b), all 2048 chips in the 4 contiguous midplanes report network failures around $9.14 \cdot 10^6$ second. This explains why the inter-arrival time of the majority of network failures is very short, as depicted in Figure 2.5(b).

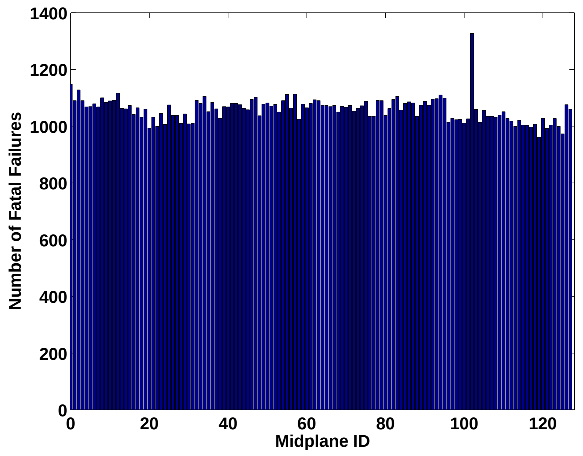
Filtering such events requires clustering the events from a number of nodes, within a short



(a) Memory failures



(b) Network failures

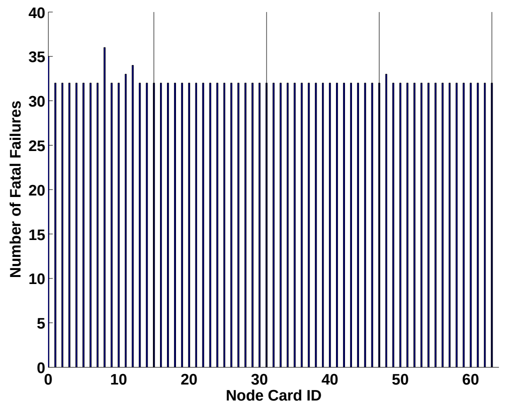


(c) I/O failures

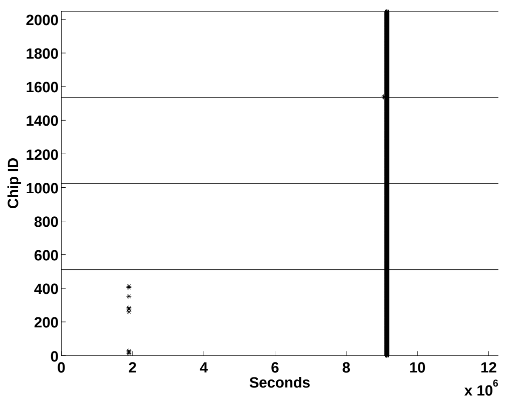
Figure 2.6: Failures at midplane level.

time interval, and with the same event description and job ID. Our filtering algorithm, referred to as spatial filtering algorithm, represents each failure record by a tuple <timestamp, job ID, location, description>, and it generates a table containing failure clusters in the form of <timestamp, job ID, description, cluster size>, where the attribute of cluster size denotes how many different nodes report this failure within a time threshold T_{net}. After applying this spatial filtering technique, we show portions of the resulting clusters for the network failures with T_{net} = 20 minutes in Table 2.4. As indicated in Table 2.4, the size of a network failure cluster may vary from a few to tens, hundreds, or thousands. The job ID attribute in some clusters is indicated by a "-" because those events do not associate with a job.

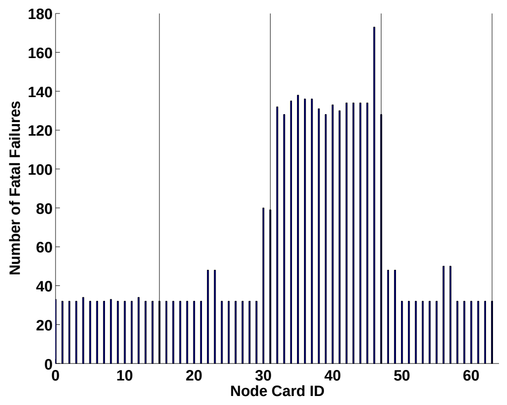
There are four typical network failures that are reported by a large number of chips within



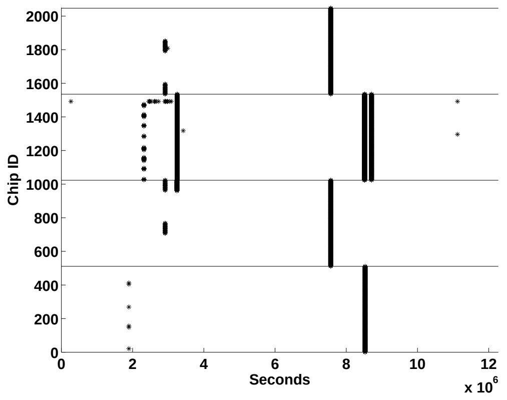
(a) Network failures across node cards for midplanes 92-95



(b) Detailed network failure occurrences (timestamp on x-axis and chip location on y-axis) for midplanes 92-95



(c) Network failures across node cards for midplanes 100-103



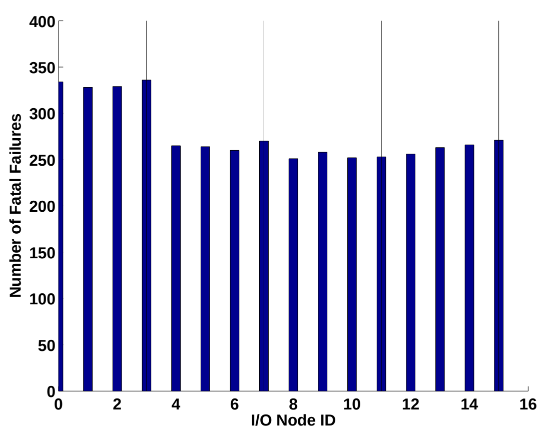
(d) Detailed network failure occurrences (timestamp on x-axis and chip location on y-axis) for midplanes 100-103

Figure 2.7: A closer look at network failures, midplane boundaries are marked by 4 vertical lines in (a) and (c), and by 4 horizontal lines in (b) and (d).

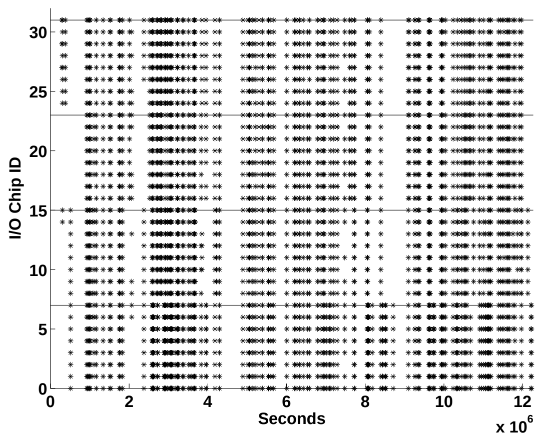
a short period of time: (1) rts tree/torus link training failed, (2) Error receiving packet on tree network, (3) bad message header, and (4) external input interrupt. Among these four, (1), (2), and (3) occur when the chips that are assigned to an application cannot configure the torus by themselves. There could be many causes of this problem, such as link failure, network interface failure, wrong configuration parameters, to name just a few. In such scenarios, the application will not even get launched. Failure (4) occurs when an uncorrectable torus error is encountered

timestamp	Job ID	Failure description	cluster size
2005-08-08 16:39:45	-	no ethernet link	2
2005-08-09 12:09:23	-	rts tree/torus link training failed	297
2005-08-15 23:57:13	-	bad message header	343
2005-09-04 17:12:22	38221	external input interrupt	241
2005-10-28 13:00:07	-	Error receiving packet on tree network	1024

Table 2.4: Network failure clusters across multiple locations.



(a) I/O failures across node cards



(b) Detailed I/O failure occurrences (timestamp on x-axis and chip location on y-axis)

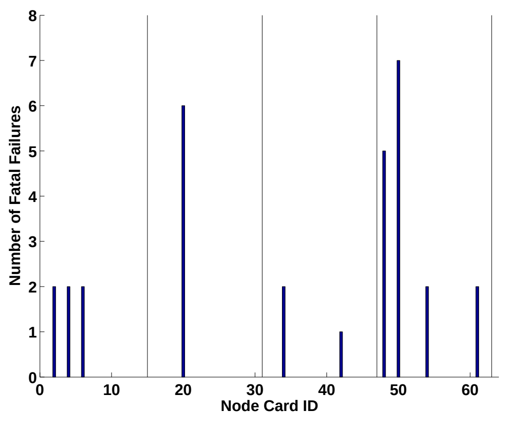
Figure 2.8: A closer look at I/O failures for midplanes 102-105. Midplane boundaries are marked by 4 vertical lines in (a), and by 4 horizontal lines in (b).

during execution, making the applications crash again. At the end of this filtering, 106 unique network failures are identified from the 142 days of RAS event collection.

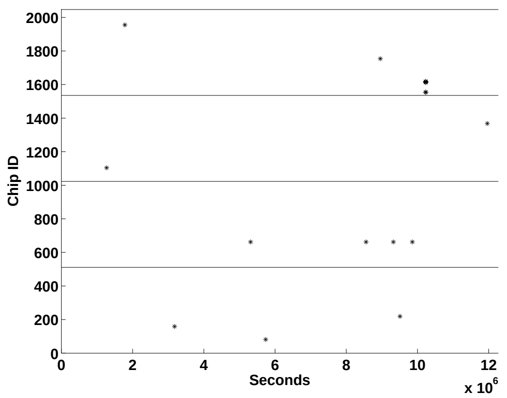
I/O Failures: I/O nodes perform I/O operations for the applications. Through an external Gigabit Ethernet switch, the I/O nodes are connected to the external file system, such as the Network File System (NFS), the Front End Nodes, Service Node, and other computing resources. In a fully loaded BG/L, an I/O node supports I/O communication services of 64 compute nodes; therefore, there are 4 I/O nodes (8 I/O chips) per midplane to support 256 compute nodes.

As depicted in Figure 3.10(c), each of the 128 midplanes report at least 961 I/O failures even after we applied the 20-minute temporal filtering mechanism at each I/O chip, and midplane 102 experienced the most failures, 1327. Figure 2.8 provides detailed views of I/O failures in 4 the midplanes 102, 103, 104, 105, which reported the largest number of I/O failures. Figure 2.8(a) shows that I/O nodes within a midplane report a similar number of failures. Further, Figure 2.8(b) shows almost all the I/O nodes within a midplane report failures about the same time, and this is also true across midplane boundaries. Clearly, those failures should be coalesced together by using the spatial filtering algorithm.

we applied the same spatial filtering scheme to obtain event clusters across multiple I/O nodes. The threshold here is again 20 minutes. Several typical failure clusters are listed in Table 2.5. The spatial filtering scheme efficiently brings down the number of I/O failures from



(a) Memory failures across node cards.



(b) Detailed memory failure occurrence (time on x-axis, and at which chip on y-axis).

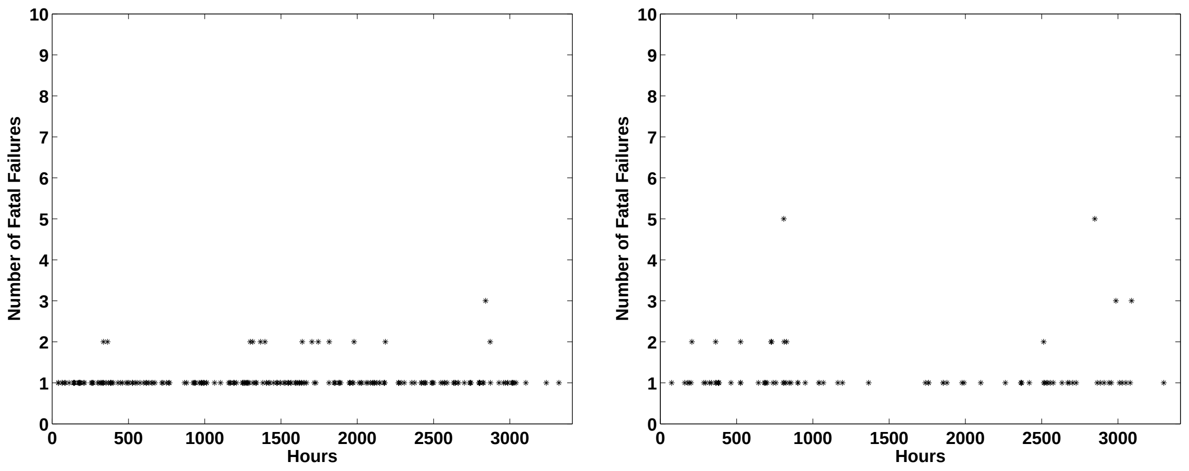
Figure 2.9: A closer look at memory failures for midplanes 99-102. Midplane boundaries are marked by 4 vertical lines in (a), and by 4 horizontal lines in (b).

135,626 to 403.

Memory Failures: Unlike the other two types of failures, temporal filtering is efficient in coalescing memory failures. For example, we only have 427 memory failures (97.34% compression ratio) after temporal filtering. As a result, we hypothesize that the spatial redundancy among memory failures is rather low. We observe that only 226 out of 2048 node cards (11% of the total node cards) ever report memory failures, and these 226 node cards are scattered in 106 midplanes. Clearly, only a few node cards in these 106 midplanes experienced failures in the 142 days of RAS event collection period. Figure 3.10 confirms our hypothesis: figure 3.10(a) shows that 77.36% of the failed midplanes contain at most 5 failures, and that only 3.77% of the failed midplanes report more than 10 failures. Further, we did not observe correlations between adjacent midplanes. Figure 2.9 provides a zoom-in view of failure occurrences (down to node card level) within midplanes 99, 100, 101, and 102, and the results again shows that memory failures are more randomly distributed spatially. In fact, in our earlier work [35], we found spacial redundancy among memory failures from a BlueGene/L prototype, and most of these

timestamp	Job ID	Failure description	cluster size
2005-08-05 13:07:01	-	Error: unable to mount filesystem	349
2005-08-13 13:53:56	-	LOGIN chdir(pwd) failed: No such file or directory	1024
2005-08-31 22:54:11	-	Lustre mount FAILED	1720
2005-11-07 04:31:32	-	Error loading FILE, invalid or missing program image	1024
2005-12-18 12:30:59	58467	Error reading message prefix, Link has been severed	1024

Table 2.5: I/O failure clusters across multiple locations.



(a)Memory failures (b) Network failures

Figure 2.10: Time series of the Memory and Network failure datasets.

failures were software failures such as de-referencing a null pointer, jumping to an invalid location, accessing out-of-bounds, etc. As these failures got fixed and removed from the prototype machine, the memory failures from BlueGene/L do not exhibit much spatial redundancy.

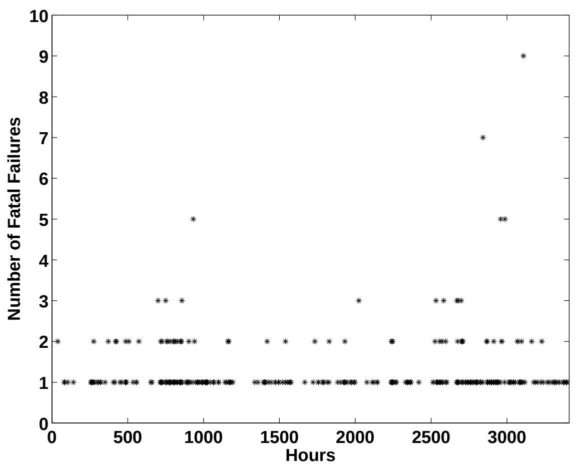
Despite the low spatial redundancy, we applied the same spatial filtering algorithm to memory failures. After this step, the number of memory failures is only reduced from 427 to 244.

2.2.4 Empirical Failure Trend Analysis

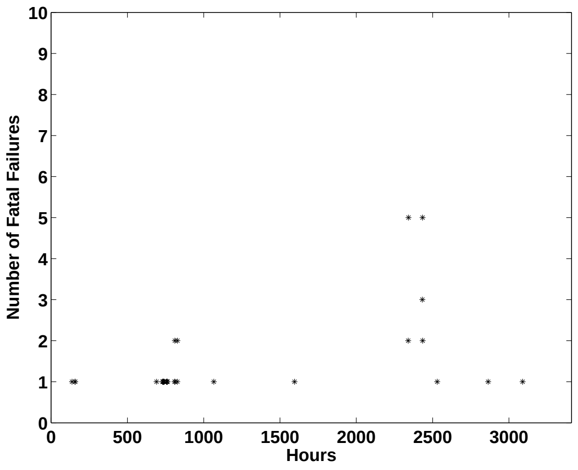
After discussing the two proposed filtering techniques, we next present some statistics of the failures (after filtering) that have occurred between August 2, 2005 and December 21, 2005.

In total, we have 244 memory failures, 106 network failures, 403 I/O failures, 45 link card failures, 18 node card failures, and 3 service card failures over the period of 142 days (3408 hours). In this section, we focus on the first three types of failures because the other failure counts are small. Figures 2.10(a), (b) and Figures 2.11(a) present the time series of these failures at an hourly granularity. It is noticed that after filtering, only a few hours contain more than one failure. For instance, there are 229 out of 3408 hours ever reporting memory failures, and among these 229 hours, only 14 hours contain more than one memory failure. Despite the small number of failures in every hour, we still observe that failures are likely to crowd together.

Some statistics about failure occurrences with respect to the day of a week, and the hour of

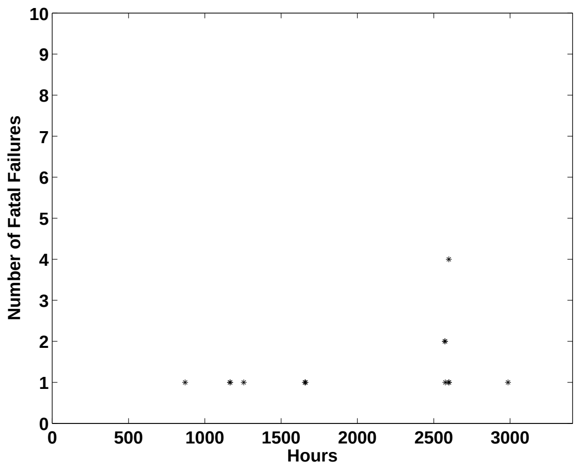


(a) I/O failures

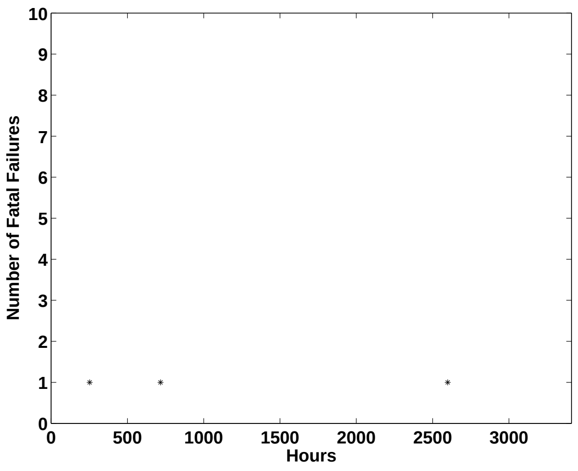


(b) Link Card failures

Figure 2.11: Time series of the I/O and Link Card failure datasets.



(a) Node Card failures



(b) Service Card failures

Figure 2.12: Time series of the Node Card and Service Card failure datasets.

a day are presented in Tables 2.6 (a) and (b). From Table 2.6(a), we can see that the number of failures, across different types, are more or less evenly distributed across the weekdays. On the other hand, Table 2.6(b) shows there are fewer failures during the midnight, namely, between 8 PM and 4 AM in the morning.

The temporal/spatial correlations of failures are shown in Figure 2.13, Figure 2.14, and Figure 2.15. In these plots, a failure occurrence is represented by both its time, shown on the x-axis, and its location, shown on the y-axis. Except for Figure 2.13(a), the y-axis in Figure 2.13(b) and Figure 2.14(a) is drawn by the 128 midplane IDs, for as depicted earlier, the network and the I/O failures cause all nodes in a midplane to report failure message, and memory failures are

Number of Failures	Total	Mon.	Tue.	Wed.	Thu.	Fri.	Sat.	Sun.
Memory Failures	244	34	40	28	28	31	37	46
Network Failures	106	19	23	18	17	7	5	17
I/O Failures	403	66	72	47	65	69	43	41
Link Card Failures	45	12	3	0	9	16	0	5
Node Card Failures	18	4	0	1	5	7	0	1
Service Card Failures	3	0	0	1	0	2	0	0
Total Failures	819	135	138	95	124	132	85	110

(a) Number of failures during each day of a week.

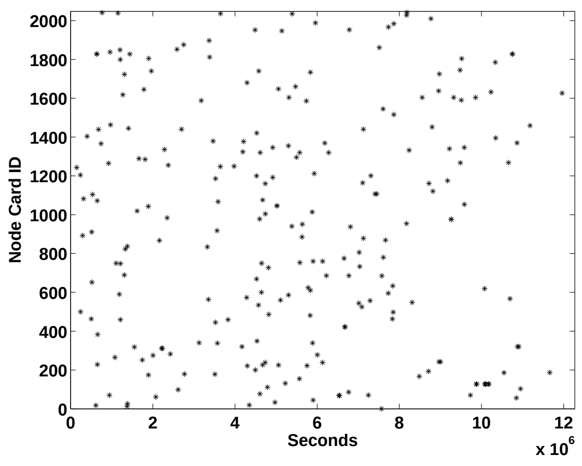
Number of Failures	Total	00:00 - 03:59	04:00 - 07:59	08:00 - 11:59	noon - 15:59	16:00 - 19:59	20:00 - 23:59
Memory Failures	244	38	38	40	42	48	38
Network Failures	106	5	18	19	20	29	15
I/O Failures	403	19	85	96	101	57	45
Link Card Failures	45	0	0	16	16	11	2
Node Card Failures	18	1	12	3	2	0	0
Service Card Failures	3	0	1	0	1	0	1
Total Failures	819	63	154	174	182	145	101

(b) Number of failures during each period of a day

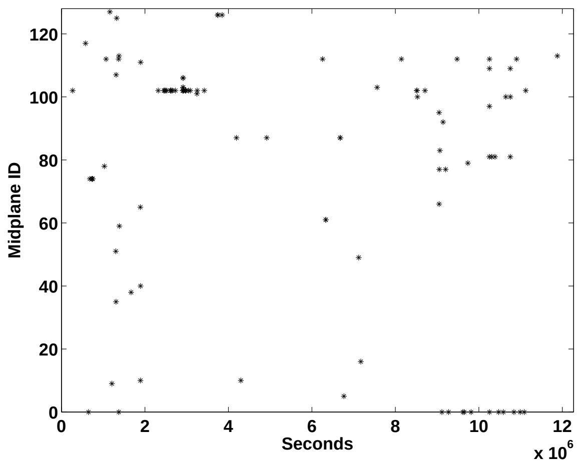
Table 2.6: Failure distribution with respect to different day of a week, and different time of a day.

bounded within a node card. Therefore, the y-axis in Figure 2.13(a) is in node card granularity and the y-axes of Figure 2.13(b) and Figure 2.14(a) are in midplane granularity. It is noticed that a more solid point on the plots represent a few occurrences of failures in a short time frame. The scattered dots in Figure 2.13(a) reveal that the probability distribution of memory failures is close to the uniform distribution. And, from Figure 2.13(b), it shows network failures exhibit more temporal locality. Further, in Figure 2.14(a), the dense dots crowded not only horizontally but also vertically. It says if a midplane fails at I/O communication, it is not only likely for it to fail again shortly but also probably for it's neighborhood midplanes to fail in short as well.

The link card failures also contain interesting failure patterns. In Figure 2.14(b), failures form a vertical line and a horizontal line. A few failures scattered around the two lines. The horizontal line is formed by 30 failures in midplane 102, and the vertical line is formed by 10 failures occurring within 107.7 minutes. There are 45 link card failures scattered in 14 midplanes, and 3 midplanes experience more than one failures. The probability for a midplane to fail again is .2143. In node card failures depicted in Figure 2.15(a), there are 18 failures scattered in 11 midplanes, and 5 midplanes reporting more than one failures. As it is, if a midplane experiences one node card failure, then the probability for it to report another link card failure is .4545 (5 out of 11). In the case of service card failures, after coalesced the redundant failures, 3 failure clusters are identified. Two of them contain invalid location ID. Therefore, instead of plotting the single service card failures in a 128-midplane space, system-wise failures

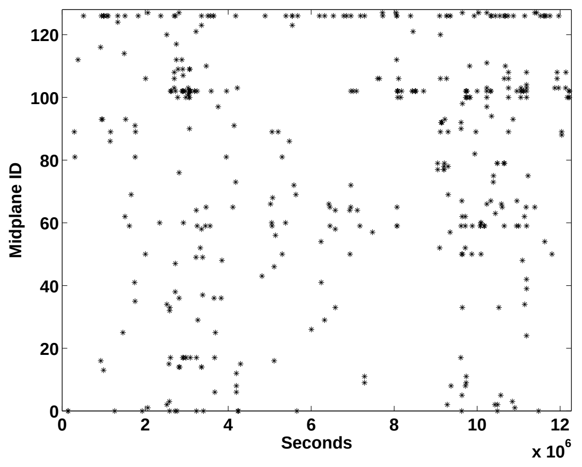


(a) Memory failures

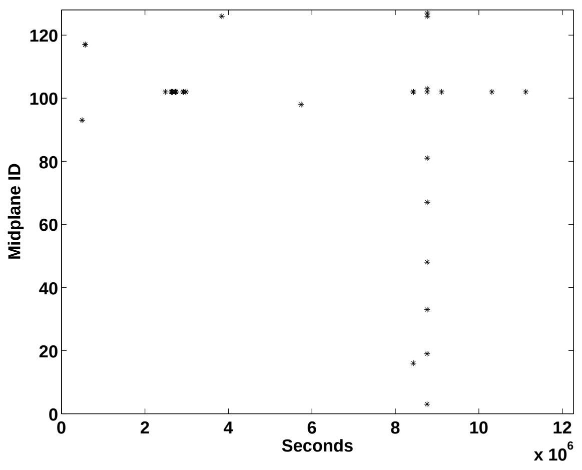


(b) Network failures

Figure 2.13: Time and location of Memory and Network failures.



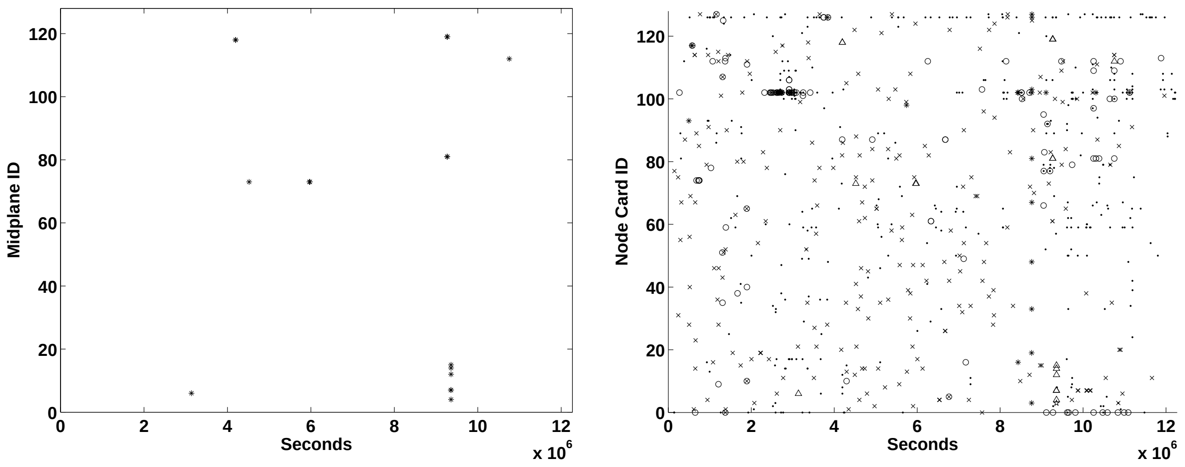
(a) I/O failures



(b) Link Card failures

Figure 2.14: Time and location of I/O and Link Card failures.

are depicted in Figure 2.15(b). In the figure, locations of memory failures are presented in the midplane level (104 midplanes ever report failures, and 71 out of 104 midplanes contribute more than one failures to the 244 total memory failures). As depicted in Figure 2.15(b), a cross marker indicates a memory failure, a circle marker indicates a network failure, a dot marker indicates an I/O failure, an asterisk maker indicates a link card failure, an upward pointing triangle marker indicates a node card failure, and a five-pointed pentagram indicates a service card failure. There are 123 out of 128 midplanes ever reported a failure, and 113 out of the 123 midplanes experienced more than one failures. The probability for a failed midplane to experience another failure is pretty high, i.e. .9187. The crowded complex in the figure indicate



(a) Node Card failures (b) System failures

Figure 2.15: Time and location of Node Card and total system failures.

multiple classes of errors occurred across a number of midplanes and in a short period of time.

We often observe that, after a midplane encounters a failure, it will keep encountering the same failure for a few more times, with these occurrences not far away from each other. We would like to point out that these temporally close records are from different applications because these network failures make the application crash. Furthermore, before the next application can start its execution, the involved compute chips must be reset. The fact that these failures are reported by multiple applications whose scheduled slots are close to each other (because these records are not far from each other) indicate that these failures cannot be addressed by reboots, and before it is diagnosed and fixed, it will lead to a sequence of application crashes, resulting in a burst of failure records of the same failure.

This bursty behavior can also be observed in memory failures (Figure 2.13(b)). As mentioned earlier, only chips within 22 node cards (out of 128 total node cards) ever have memory failures. Further, we find that, once a node card starts reporting memory failures, it usually reports the same failure several times within a short time window. This is because of the same reason as in the case of network failures: it takes some time to diagnose these failures, and before that, a sequence of applications that are scheduled on that node card will encounter the problem and crash. However, we would like to point out that, for both network failures and memory failures, some of them are only reported once. These failures are usually caused by the applications, either because the application contains a bug which tries to access an invalid address, or because the application tries to configure the torus using the wrong parameters.

The observations discussed above actually provide good indications on how such failure analysis studies can be used to enhance the run time health of such large scale parallel systems. For instance, when the reboots (or application crashes) do not change the failure behaviors of some nodes, instead of scheduling more jobs onto these nodes, it may be better for one to take these nodes offline and perform a thorough check on them. On the other hand, if the same application keeps crashing, it may be due to a software bug in the application. Exploiting these issues are part of our ongoing work.

2.3 An Adaptive Semantic Filtering Method

While STF can effectively compress BlueGene/L data logs, it has the following drawbacks. First, it requires extensive domain knowledge, both when categorizing fatal events and when adopting suitable threshold values. Second, it requires manual operations. For example, upon the addition of a new event entry, a human operator is needed to categorize it into different types. As another example, after each step, manual operations are needed to process the partial results to enable operations in the next step. Third, STF only employs simple thresholding-techniques, which cannot handle many tricky situations, and may lead to incorrect results.

To address these challenges, we next propose an **A**daptive **S**emantic **F**iltering (ASF) method, which exploits the semantic context of the event descriptions for the filtering process.

2.3.1 Description of ASF

ASF involves the following three steps: (1) building a dictionary containing all the keywords that appear in event descriptions, (2) translating every event description into a binary vector where each element of the vector represents whether the corresponding keyword appears in the description or not, and (3) filtering events using adaptive semantic correlation thresholds.

Keyword Dictionary: The keyword dictionary is the base for developing the ASF method. The keywords in the dictionary should capture the semantic context of all the events. Building the dictionary is an iterative process. In each iteration, we examine an event entry, identify its keywords, and append new keywords into the dictionary. In order to identify the keywords of an event description, we have adopted the following removal/replacement rules:

1. Remove punctuation, equal signs, single quotes, double quotes, parentheses (including the content in the parentheses).
2. Remove indefinite articles `a`, `an`, and definite article `the`.
3. Remove words such as `be`, `being`, `been`, `is`, `are`, `was`, `were`, `has`, `have`, `having`, `do` or `done`.
4. Remove prepositions such as `of`, `at`, `in`, `on`, `upon`, `as`, `such`, `after`, `with`, `from`, `to`, etc.

5. Replace an alphabetic-numeric representation of a rack, a midplane, a link card, or a node card by the keyword `LOCATION`.
6. Replace an eight-digit hexadecimal address by the keyword `8DigitHex_Addr`.
7. Replace a three-digit hexadecimal address by the keyword `3DigitHex_Addr`.
8. Replace an eight-digit hexadecimal value by the keyword `8DigitHex`.
9. Replace a two-digit hexadecimal value by the keyword `2DigitHex`.
10. Replace a numeric number by the keyword `NUMBER`.
11. Replace binary digits by the keyword `BinaryBits`.
12. Replace a register, e.g. r00, r23, etc., by the keyword `REGISTER`.
13. Replace a file directory or an image directory by the keyword `DIRECTORY`.
14. Replace uppercase letters by the corresponding lowercase letters.
15. Replace present participles and past participles of verbs by their simple forms, e.g. `Failing`, `Failed` being replaced by `Fail`.
16. Replace a plural noun by its single form, e.g. `errors` being replaced by `error`.
17. Replace week days and months by the keyword `DATE`.

After processing all 1,792,598 entries in the raw logs from August 2, 2005 to December 21, 2005, we have identified 667 keywords. One of the advantages of this method is that upon the arrival of new data, we only need to process the new entries as described above, without affecting the earlier data in any way.

Correlation Computation: Following the construction of the keyword dictionary, we next convert each event description into a binary vector for the purpose of semantic correlation calculation. Suppose there are N keywords. Then the vector will have N elements, with each element corresponding to one keyword. In this way, assigning vectors to event descriptions becomes straightforward: 1 denoting the description includes the associated keyword, and 0 denoting otherwise. In order to make this step more automatic, we can choose a reasonably large value for N so that adding new logs will not require re-doing the translations for earlier logs, even when these new logs may introduce new keywords. This approach is further supported by the observation that the number of raw events may be huge, but the number of keywords stays more or less constant after it reaches a certain level.

After generating a binary vector for event record, we can then compute the correlation between any two events using the ϕ correlation coefficient [44], the computation form of Pearson's Correlation Coefficient for binary variables.

		B		Row Total
		0	1	
A	0	$P_{(00)}$	$P_{(01)}$	$P_{(0+)}$
	1	$P_{(10)}$	$P_{(11)}$	$P_{(1+)}$
Column Total		$P_{(+0)}$	$P_{(+1)}$	N

Figure 2.16: A two-way table of item A and item B.

For a 2 $\times$ 2 two-way table as shown in Figure 2.16, the calculation of the ϕ correlation coefficient reduces to

$$\phi = \frac{P_{(00)}P_{(11)} - P_{(01)}P_{(10)}}{\sqrt{P_{(0+)}P_{(1+)}P_{(+0)}P_{(+1)}}}, \tag{2.1}$$

where $P_{(ij)}$ denotes the number of samples that are classified in the i-th row and j-th column of the table. Furthermore, we let $P_{(i+)}$ denote the total number of samples classified in the ith row, and $P_{(+j)}$ the total number of samples classified in the jth column. Thus, we have $P_{(i+)} = \sum_{j=0}^{1} P_{(ij)}$ and $P_{(+j)} = \sum_{i=0}^{1} P_{(ij)}$. In the two-way table, N is the total number of samples.

Adaptive Semantic Filtering: ASF tells whether a record is redundant or not based on its semantic context. An intuitive semantic correlation based approach would regard two records with a high correlation between their descriptions as redundant. A closer look at the logs, however, reveals that in addition to the correlation coefficient, the interval between two records also plays an important role in determining whether these two records are redundant. For example, if two events are very close to each other, even though their correlation may be low, they may still be triggered by the same failure. On the other hand, two records that are far away from each other, though their descriptions may be exactly the same, are more likely to report unique failures. As a result, it is insufficient to adopt a simple thresholding technique solely based on the correlation coefficients between events. Instead, we propose to adopt an adaptive semantic filtering mechanism, which takes into consideration both semantic correlation and temporal information between events to locate unique events. We thus believe that, by doing so, we can

```
The Adaptive Semantic Filter Algorithm
Input:    $TTH$: a set of time thresholds; $CTH$: a set of correlation thresholds.
          $dimsn$: total number of keywords in the dictionary; $IFile$: raw logs.
Output:   $OFile$: The filtering result.
AdaptiveFilter($TTH, CTH, dimsn, IFile, OFile$)
 1.   for each cEvent in IFile /*Events are sorted by time.*/
 2.       extract eventTime, JobID, eventDescription; put the cEvent into the JobID symbolic
          reference table;
 3.       for each anchorEvent in the JobID symbolic reference table.
 4.           if the eventTime of the anchorEvent is outside the largest time threshold in TTH
 5.               delete anchorEvent from the JobID symbolic reference table.
 6.           else
 7.               corr= compCorr(anchorEvent, cEvent); timeDiff = eventTime - anchorTime;
 8.               while(CTH and TTH Not Null)
 9.                   if(corr $>$ CTH.curr and timeDiff $<$ TTH.curr)
10.                       filter the current event record and break;
11.               end while;
12.       end for;
13.   end for;
```

Figure 2.17: The Adaptive Semantic Filtering (ASF) Algorithm.

combine the advantage of STF and the advantage of a simple semantic filtering technique. The principle of the adaptive semantic filtering is that as the time gap between records increases, two records must have a higher correlation coefficient to be considered redundant, and after the time gap reaches a certain level, even two records with a perfect correlation coefficient will be considered unique.

Figure 2.17 outlines the Adaptive Semantic Filtering (ASF) algorithm, which first sorts the log entries in time sequence and stores them into the jobID symbolic reference table as shown in Line 2 and Line 3. Then, for each anchor event in the jobID table, if the timestamp of the current anchor event is greater than the largest time threshold, the ASF algorithm deletes this anchor event from the jobID table; otherwise, in line 8, the algorithm computes the semantic correlation between the current event and the anchor event. Line 9 obtains the temporal gap between the current event and the anchor event. If either the temporal or the correlation criteria is satisfied, the current record is filtered out as indicated in Line 12. The above process is iterated for every event record.

2.3.2 Comparing ASF Versus STF

First, we would like to compare the effectiveness of the new semantic filter ASF against STF. It is noticed that STF is expected to accurately extract unique events from voluminous raw logs because it has involved extensive domain knowledge in the organization of the hardware platform, as well as the logging procedure, and because it has involved a considerable amount of manual effort from these domain experts. As shown in Section 2.2, STF can bring down the number of FATAL records from 281441 to 819, which only constitutes 0.2899% of the FATAL events in the raw log.

ASF adopts different correlation coefficient threshold values according to the intervals between subsequent event records. Specifically, we take the viewpoint that two records that are temporally close to each other are likely to be correlated, and therefore, should be coalesced into one event. As a result, we adopt a lower correlation threshold for shorter intervals between subsequent records. On the other hand, two records that are far apart from each other should only be considered correlated when the semantic correlation between them is high, which suggests that we should adopt a higher threshold for events with larger time intervals between them.

In order to develop suitable threshold values, we have partitioned the data sets into two halves, the first half being training data while the second half being test data. On the training data, we have applied different correlation coefficient and interval pairs, and chosen the following values which have produced similar results as those from STF:

$$C_{th} = \begin{cases} 1 & \text{if } T \in [20, \infty) \\ 0.9 & \text{else if } T \in [10, 20) \\ 0.8 & \text{else if } T \in [5, 10) \\ 0.7 & \text{else if } T \in [1, 5) \\ 0.0 & \text{else if } T \in [0.5, 1) \\ -1.0 & \text{else if } T \in [0, 0.5) \end{cases}, \tag{2.2}$$

where T denotes the interval between the current record and the previous record, and the time unit is one minute. Equation 2.2 specifies correlation coefficient threshold values for different intervals. For example, if the gap between the current record and the previous record,

T, is greater than or equal to 20 minutes, then the current record will be kept in the filtering results if the semantic correlation between them is less than or equal to 1.0. (Since 1.0 is the maximum correlation coefficient, all events that are longer than 20 minutes apart from their preceding events will be kept.) As another example, Equation 2.2 specifies that if T is less than 30 seconds, then the current event will be filtered out if the semantic correlation between itself and its previous event is greater than -1.0. In another word, all the events that occur within 30 seconds after their previous events will be filtered out.

After we extract the parameters in Equation 2.2 from the training data, we have applied them to the test data to examine whether they are only specific to the training data or they can be used to the test data as well. Fortunately, we find that these values are effective for all the data after careful inspection.

Using the chosen parameters, ASF can condense the fatal failures from 281441 records to 835 records (after remove software failures), while STF produces 819 records after filtering. Though these two numbers are rather close, we have observed three typical scenarios in which these two filters yield different results, and that among these three, two cases demonstrate ASF produces better filtering results than STF. These four scenarios are listed below (each record contains the timestamp, job ID, location, and entry data fields):

- *Advantage I: ASF can efficiently filter out semantically-correlated records whose descriptions do not exactly match each other.* Records that are semantically correlated should be filtered out (if they are reasonably close to each other), even though they do not have identical descriptions. ASF can easily achieve this because it considers semantic correlation between events. STF, on the other hand, compares event descriptions word by word, which can be problematic because many highly-correlated records do not have identical descriptions. For example, STF has produced the following records in its result:

  [ST1]2005-11-15-12.07.42.786006 - R54-M0-N4-I:J18-U11 ciod: LOGIN chdir(/xxx/xxx) failed: No such file or directory

  [ST2]2005-11-15-12.07.42.858706 - R64-M1-N8-I:J18-U11 ciod: LOGIN chdir(/xxx/xxxx/xx) failed: No such file or directory

  [ST3]2005-11-15-12.07.42.779642 - R74-M0-NC-I:J18-U11 ciod: LOGIN chdir(/xxx/xxxx/xxx) failed: No

```
such file or directory
```

In this example, all three events occur at the same time, but at different locations, and they correspond to the same fatal failure that affects all three locations. However, in the spatial filtering phase, STF only filters out records if their descriptions are the same. As a result, it has kept all three entries. This problem, however, can be avoided by ASF because ASF considers semantic correlation instead of exact word-by-word match. Hence, the result from ASF only contains one entry:

```
[AS1]2005-11-15-12.07.42.786006    -    R54-M0-N4-I:J18-U11    ciod: LOGIN chdir(/xxx/xxx) failed: No such
file or directory
```

This example emphasizes the importance of semantic correlations in filtering error logs.

- *Advantage II: ASF can prevent non-correlated events from being filtered out.* The previous example shows that ASF can filter out semantically correlated events even when their descriptions are not identical. Similarly, ASF can also prevent non-correlated events from being blindly filtered out just because they are close to each other. This is because STF, in its temporal filtering phase, simply treats all the events that are more than 20 minutes apart as unique while all the events that are less than 20 minutes apart as redundant. Compared to STF, ASF employs a much more sophisticated mechanism, which not only exploits correlation coefficient between two events, and the threshold for the correlation coefficient also adapts to the gap between the events. As a result, if the gap between two events (from the same location) is less than 20 minutes, STF will filter out the second event, but ASF will only do so if their correlation coefficient is above a ceratin level.

  As an example, ASF has produced the following sequence:

```
[AS1]2005-09-06-08.45.57.171235    -    R62-M0-NC-I:J18-U11    ciod: LOGIN chdir(/home/xxx/xx/run) failed:
Permission denied

[AS2] 2005-09-06-08.49.34.442856    -    R62-M0-NC-I:J18-U11    ciod: Error loading “/home/xxxxx/xxx/xxxx/xxxx”:
program image too big, 1663615088 > 532152320
```

In the above sequence, ASF chooses to keep both records because the semantic correlation between them is less than 0.0, and according to parameters in Equation 2.2, they are unique events. On the other hand, STF condenses the same example scenario to the only entry because the gap between these two events is 4 minutes:

```
[ST1]2005-09-06-08.45.57.171235    -    R62-M0-NC-I:J18-U11    ciod: LOGIN chdir(/home/xxx/xx/run) failed:
Permission denied
```

Comparing the results produced by both filters, we can easily tell that both entries need to be kept because each corresponds to a different problem in the system.

- *Disadvantage: ASF may filter out unique events that occur with 30 seconds from each other.* According to Equation 2.2, ASF filters out events when they occur within 30 seconds of each other. Though in most cases, events that are so close to each other are highly correlated, there are some rare cases where different types events may take place at the same time, e.g. a memory failure and a network failure occurring at the same time. STF can avoid such problems by categorizing failures before filtering. For example, STF has produced the following sequence of records:

```
[ST1]2005-08-05-09.11.35.447278    -    R33-M1-NC-C:J13-U11    machine check interrupt

[ST2]2005-08-05-09.11.59.393092    -    R54-M1-N8-I:J18-U01    ciod: Error reading message prefix after
LOAD_MESSAGE on CioStream socket to xxx.xx.xx.xxx:xxxxx: Link has been severed
```

Since these two failures, one being memory failure and the other application I/O failure, occur within 24 seconds from each other, ASF compresses them into one entry:

```
[ST1]2005-08-05-09.11.35.447278    -    R33-M1-NC-C:J13-U11    machine check interrupt
```

Fortunately, this problem of ASF does not affect the filtering results much because the likelihood of having two failures within 30 seconds is very low. In fact, we have checked the log carefully, and found that the above example is the only case where two distinct events occur so close to each other. Even in such cases, the problem will be further alleviated by the fact that the production BlueGene/L usually runs one job at a time, which spans all the processors of that machine. As a result, the adverse effect of compressing

	Info	Warning	Severe	Error	Failure	Fatal
before filtering	1,367,531	17,121	15,749	109,048	1,708	281,441
after filtering	11,044	343	148	146	53	1,147
compression ratio	.9919	.9800	.9906	.9987	.9690	.9959

Table 2.7: The number of events at all severity levels before and after filtering by ASF.

failures that occur at the same time is negligible because they hit the same job anyway.

2.3.3 The BlueGene/L RAS Event Analysis

In addition to filtering fatal events, it is also equally important to filter other non-fatal events, since such information can depict a global picture about how warnings evolve into fatal failures, or about how a fatal failure is captured by different levels of logging mechanisms. STF, however, cannot be used to filter non-fatal events due to its complexity, especially when the number of non-fatal events (1,509,449 in our case) is substantially larger than that of FAILURE and FATAL events (283,149). Fortunately, this void can be filled by the introduction of ASF, which involves much less overhead and can thus yield an automatic execution.

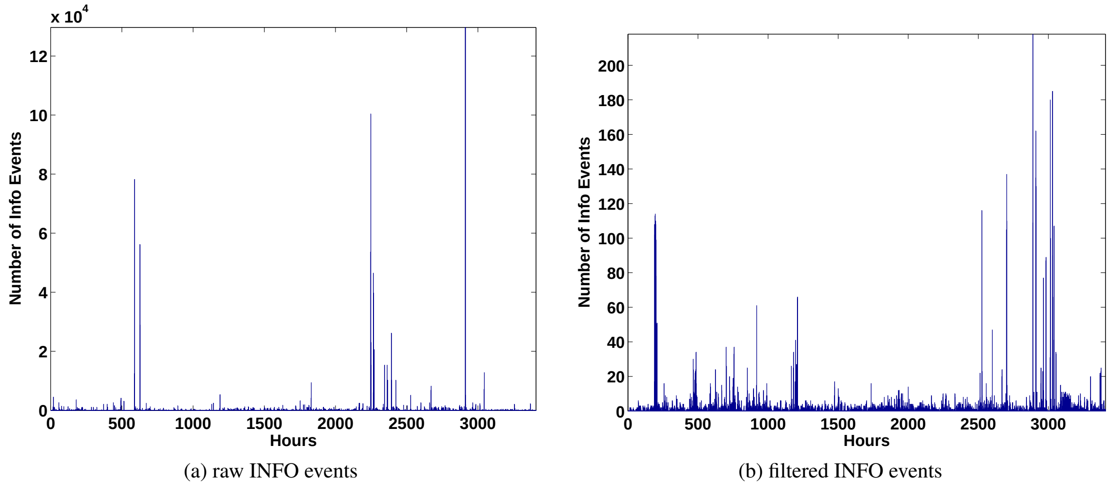
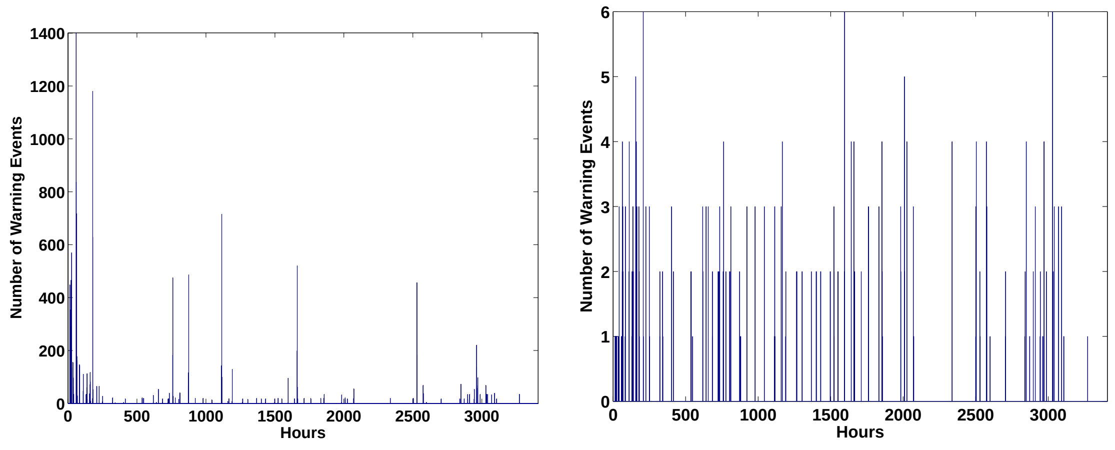


Figure 2.18: The number of INFO events in each hour (between 08/02/05 and 12/21/05) before and after filtering.

Table 2.7 summarizes the filtering results for events at all severity levels. These numbers show that ASF is quite effective in filtering all types of events, achieving average compression ratios above 98.76% (many compression ratios are above 99%). More detailed statistics are presented in Figure 2.18, Figure 2.19, Figure 2.20, Figure 2.21, Figure 2.22, and Figure 2.23.

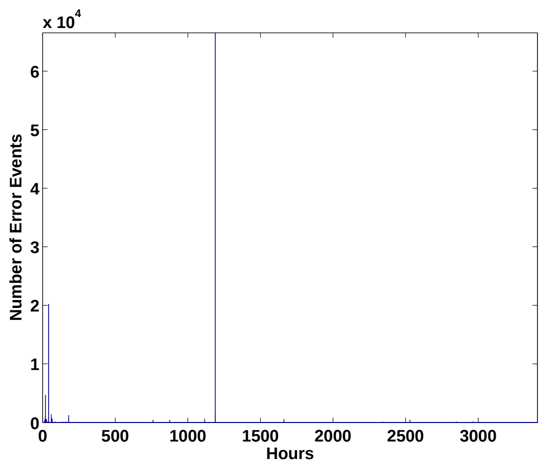


(a) raw WARNING events (b) filtered WARNING events

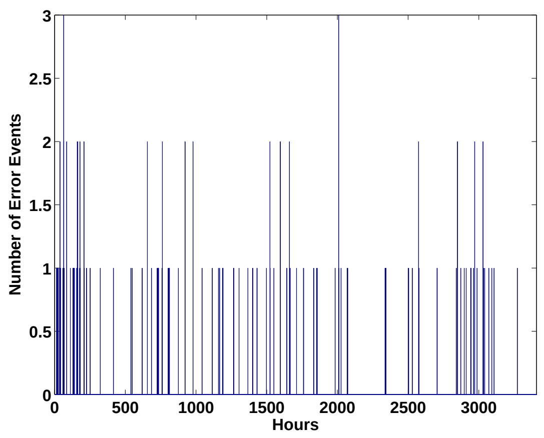
Figure 2.19: The number of WARNING events in each hour (between 08/02/05 and 12/21/05) before and after filtering.

Specifically, the 12 plots in Figure 2.18, Figure 2.19, Figure 2.20, Figure 2.21, Figure 2.22, and Figure 2.23 show the number of events in every hour from August 2, 2005 to December 21, 2005 for all six severity levels, before filtering and after filtering. We have two main observations from these plots. First, ASF can remove the spikes in the raw logs. For example, in the raw logs, during some hours there are several thousand, or even several hundred thousand records, especially for INFO and SEVERE severities. It is obvious that most of these records are redundant, and a filter must identify this redundancy. The results show that ASF has achieved this objective. After filtering, except INFO events (we do sometimes observe tens, or even a couple of hundred of INFO events, but this is due to the nature of such events), the resulting logs only have a few events per hour, which can serve as a good basis for further research such as failure analysis and failure prediction. The second observation we have is that ASF can preserve the trend of the raw logs, while greatly reducing the absolute numbers. For example, the raw logs have large numbers of WARNING, SEVERE and ERROR events in the first 250 hours (refer to Figures 2.19(a), Figures 2.20(a), and Figures 2.21(a)), and the final logs also show the same trend for these types of events (refer to Figures 2.19(b), Figures 2.20(b), and Figures 2.21(b)). Similarly, there are bursts of FAILURE events from hours 2400 to 2600 in the raw logs (refer to Figure 2.22(a)), and the corresponding burst is also observed in the final logs (refer to Figure 2.22(b)).

After filtering events of all severity levels, we can next merge them in time order, and study

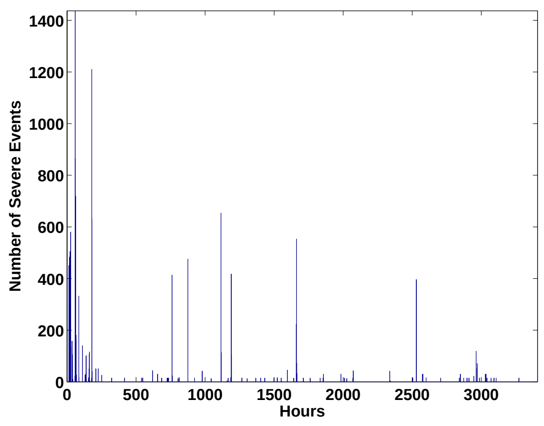


(a) raw SEVERE events

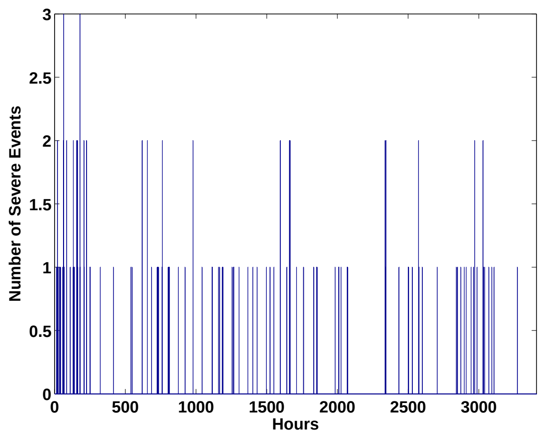


(b) filtered SEVERE events

Figure 2.20: The number of SEVERE events in each hour (between 08/02/05 and 12/21/05) before and after filtering.



(a) raw ERROR events

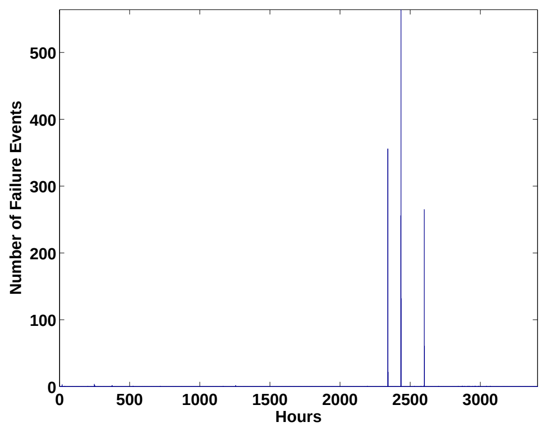


(b) filtered ERROR events

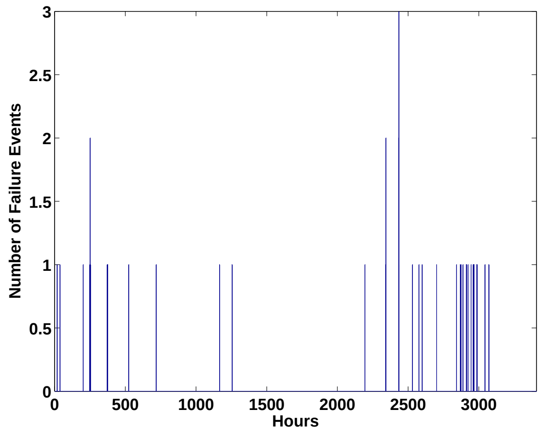
Figure 2.21: The number of ERROR events in each hour (between 08/02/05 and 12/21/05) before and after filtering.

how lower-severity events evolve to a FAILURE or FATAL event, which can terminate job executions and cause machine reboots. We would note that, the investigation of detailed rules about what non-fatal events will lead to fatal failures, and in what fashion, is well beyond the scope of this dissertation. In this dissertation, we argue that such studies are made possible by the introduction of ASF.

After merging events with different severity levels, we observe that they form natural "clusters" consisting of multiple non-fatal events and one or more fatal events following them. These clusters clearly show that how events evolve in their severity. An example cluster is shown in Table 2.8. This sequence starts with an INFO event that informs the system controller was

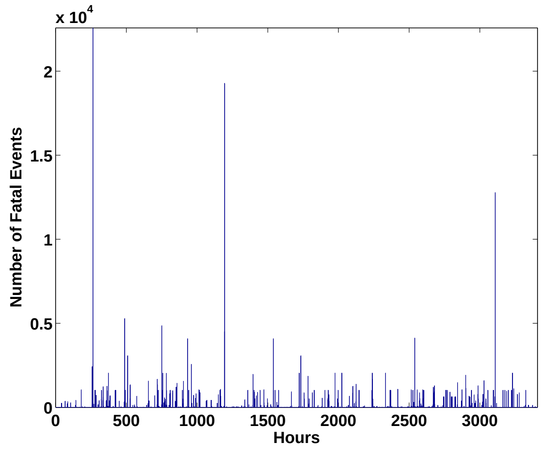


(a) raw FAILURE events

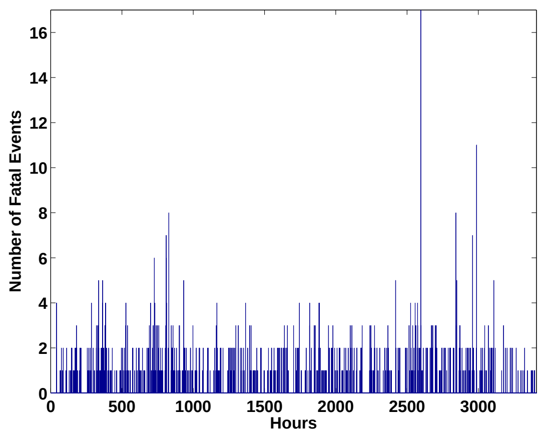


(b) filtered FAILURE events

Figure 2.22: The number of FAILURE events in each hour (between 08/02/05 and 12/21/05) before and after filtering.



(a) raw FATAL events



(b) filtered FATAL events

Figure 2.23: The number of FATAL events in each hour (between 08/02/05 and 11/18/05) before and after filtering.

starting. Thirty seconds later, all the node cards on midplane R63-M0 were restarted, as suggested in the following WARNING event, and another two minutes later, another WARNING message points out that one of the node cards on that midplane, R63-M0-N6, was not fully functional. At almost the same time, a SEVERE event and an ERROR event were recorded, which give more detailed information about the node card malfunction. The SEVERE event reports that the assembly information could not be obtained for the same node card, and the ERROR event reports several major status parameters of the node card, such as “PGOOD is not asserted”, “MPGOOD is not OK”, etc. About 4 hours later, a SEVERE event reports that one of the link cards’ power module U58 was not accessible from the same midplane, and about 2

facility	severity	timestamp	location	entry data
CMCS	INFO	2005-11-07-08.40.12.867033	-	Starting SystemController UNKNOWN_LOCATION
HARDWARE	WARNING	2005-11-07-08.40.48.975133	R63-M0	EndServiceAction is restarting the NodeCards in midplane R63-M0 as part of Service Action 541
DISCOVERY	WARNING	2005-11-07-08.42.07.610916	R63-M0-N6	Node card is not fully functional
DISCOVERY	SEVERE	2005-11-07-08.42.07.769056	R63-M0-N6	Can not get assembly information for node card
DISCOVERY	ERROR	2005-11-07-08.42.07.797900	R63-M0-N6	Node card status: no ALERTs are active. Clock Mode is Low. Clock Select is Midplane. Phy JTAG Reset is asserted. ASIC JTAG Reset is asserted. Temperature Mask is not active. No temperature error. Temperature Limit Error Latch is clear. PGOOD IS NOT ASSERTED. PGOOD ERROR LATCH IS ACTIVE. MPGOOD IS NOT OK. MPGOOD ERROR LATCH IS ACTIVE. The 2.5 volt rail is OK. The 1.5 volt rail is OK.
HARDWARE	SEVERE	2005-11-07-12.28.05.800333	R63-M0-L2	LinkCard power module U58 is not accessible
MONITOR	FAILURE	2005-11-07-14.11.44.893548	R63-M0-L2	No power module U58 found found on link card
HARDWARE	SEVERE	2005-11-07-14.38.38.623219	R63-M0-L2	LinkCard power module U58 is not accessible

Table 2.8: An example event sequence that reveals how INFO events evolve into FAILURE events.

hours later, the power module U58 was reported totally un-found by a FAILURE event. After the FAILURE event, the midplane needs to be repaired by a system administrator before it can be used again.

The ability to locate such sequences is important for studying failure behavior and predicting failures. This was impossible without a good filtering tool. In our example above, there are only 8 records, but they correspond to a much longer sequence in the raw logs, with 572 records. We would note that, it is very difficult, if not at all impossible, to keep track of event occurrences from a 572-entry sequence.

2.4 Related Work

Though there has been previous interest in monitoring and filtering system events/failures (e.g. [16, 37, 53, 55]), there has been no prior published work on failures in large scale parallel systems spanning thousands of nodes. While some supercomputing centers do monitor and keep track of hardware/software failures on their systems, this data has rarely been made available to the research community due to proprietary or confidential reasons.

Collection and filtering of failure logs has been examined in the context of much smaller scale systems. Lin and Siewiorek [37] found that error logs usually consist of more than one failure process, making it imperative to collect these logs over extended periods of time. In [16], the authors make recommendations about how to monitor events and create logs by using one of the largest data sets at that time, comprising 2.35 million events from 193 VAX/VMS systems. The authors point out that data sets with poor quality are not very helpful, and can lead to wrong conclusions. Lack of information in the logs (e.g. power outages), errors in the monitoring system (e.g. in the timestamps), and the difficulty of parsing and collecting useful patterns is reiterated by this study. It has been recognized [17,26,30,56] that it is critical to coalesce related events since faults propagate in the time and error detection domain. The tupling concept developed by Tsao [56], groups closely related events, and is a method of organizing the information in the log into a hierarchical structure to possibly compress failure logs [17].

Tang et al. [53, 55] studied the error/failure log collected from a VAXcluster system consisting of seven machines and four storage controllers. They found that 98.8% of the errors are recovered, and 46% of the failures are caused by external errors. Using a semi-Markov failure model, they further pointed out that failure distributions on different machines are correlated rather than independent. In their subsequent work [54], Tang and Iyer pointed out that many failures in a multicomputer environment are correlated with each other, and they studied the impact of correlated failures on such systems. Xu et al. [62] performed a study of error logs collected from a heterogeneous distributed system consisting of 503 PC servers. They showed that failures on a machine tend to occur in bursts, possibly because common solutions such as reboots cannot completely remove the problem causing the failure. They also observed a strong indication of error propagation across the network, which leads to the correlation between failures of different nodes. These studies use the empirical (marginal) distribution obtained from their respective datasets to build failure models. Kalyanakrishnam et al. [33] studied the error logs collected from a commercial cluster to explore the causes of reboots, and they found that most of reboots are caused by software failures and many of the reboots cannot really solve the problem. Heath et al. [27] collected failure data from three different clustered servers, ranging from 18 workstations to 89 workstations. They assumed the times between failures are independent and identically distributed, and fit the failure data using a Weibull distribution with a shape parameter less than 1. In addition, they used a property of this Weibull distribution – nodes that just failed are more likely to fail again in the near future – to motivate a new resource management strategy. In our previous study [46], we reported failure data for a large-scale heterogenous AIX cluster involving 400 nodes over a 1.5 year period and studied their statistical properties. In another previous work of ours [35], we proposed STF, and used STF to filter RAS event logs that were collected from a 8192 processor BlueGene/L prototype at IBM Rochester, over a period of 84 days starting from August 26, 2004. Recently, Schroeder and Gibson studied the failure logs that were collected from high performance systems at Los Alamos National Laboratory [49].

2.5 Concluding Remarks

Parallel system event/failure logging in production environments has widespread applicability. It can be used to obtain valuable information from the field on hardware and software failures, which can help designers make hardware and software revisions. It can be also used by system administrators for diagnosing problems in the machine, scheduling maintenance and downtimes. Finally, it can be used to enhance fault resilience and tolerance abilities of the runtime system for tuning checkpointing frequencies and locations, parallel job scheduling, etc. There has been no prior work, to our knowledge, which has published failure data of large scale parallel machines.

With fine-grain event logging, the volume of data that is accumulated can become unwieldy over extended periods of time (months/years), and across thousands of nodes. Further, the idiosyncracies of logging mechanisms can lead to multiple records of the same events, and these need to be cleaned up in order to be accurate for subsequent analysis. In this dissertation, we have embarked on a study of the failures on the 128K-processor BlueGene/L machine. We have presented two filtering algorithm. The first one is spatio-temporal filering (STF), which has three steps: first extracting and categorizing failure events from the raw logs, then performing a temporal filtering to remove duplicate reports from the same location, and finally coalescing failure reports across a number of locations. Based on the lessons learnt from the experience with STF, we also develop an Adaptive Semantic Filtering (ASF) method, which exploits the semantic correlation as well as the temporal information between events to determine whether they are redundant. The ASF method involves three steps: first building a keyword dictionary, then computing the correlation between events, and finally choosing appropriate correlation thresholds based on the temporal gap between events. Compared to STF, ASF (1) produces more accurate results, (2) incurs less overhead, and (3) avoids frequent human intervention. Using these filtering tools, we can extract 1,147 unique fatal failures from 1,792,598 raw events.

Chapter 3

Failure Analysis and Preliminary Modeling

It has been recognized that preventing failures from occurring is very challenging, if at all possible [11]. Instead, we take the viewpoint that runtime fault-tolerant measures that can mitigate the adverse impacts of failures are in an urgent need. Checkpointing [65] is such a technique that can allow the failed jobs to start from a saved point, rather than restarting from the beginning. Though checkpointing techniques have been widely used in conventional systems, they are not as effective in large-scale parallel systems such as BlueGene/L because the overheads of checkpointing overshadow the gain: checkpointing a job that involves tens of thousand tasks may take at least half an hour. In our earlier study [65], we find that checkpointing at regular intervals simply does not improve the performance; a large interval may miss many failures, while a small interval may incur high checkpointing overheads. Instead, we find that the capability of predicting the time/location of the next failure, though not perfect, can considerably boost the benefits of runtime techniques such as job checkpointing or scheduling [42, 65].

Failure prediction, however, has long been considered a challenging research problem. One of the main reasons is the lack of suitable data from realistic systems. To address this void, we have obtained event logs containing all the RAS (reliability, availability, and serviceability) data since 08/02/05 as well as job logs containing all the job execution information since 08/31/05, both from BlueGene/L. After carefully preprocessing these data using a three-step filtering tool [36], we extract all the failure events that can lead to job terminations, and categorize them into memory failures, network failures, and application I/O failures. We first investigate the consequences of these failures on job executions, and find that failures usually cause many jobs running side by side, or fewer jobs spanning a large number of processors, to terminate. After looking at the consequences of failures, we conduct in-depth studies to explore the predictability of these failure events. Firstly, we find that 50% of the network failures and

35% of the application I/O failures occur within a window of half an hour after the preceding failures. Secondly, we find that network failures exhibit strong spatial skewness, with 6% of the nodes encountering 61% of the failures. Thirdly, we find that jobs that report non-fatal events are very likely followed by a fatal failure event. These observations are evidence that we can effectively predict failures, which can in turn be used to develop efficient runtime fault-tolerant strategies. In total, we have developed three prediction algorithms based on the bursty nature of failure occurrence, spatial skewness, and preceding non-fatal events. We have evaluated the effectiveness of these algorithms carefully, and found they are able to capture a large fraction of failures.

The rest of this chapter is organized as follows. Section 3.1 provides an overview of BlueGene/L and describes the logs used in the study. The impact of failure events on the jobs that run on BlueGene/L is studied in Section 3.2. In section 3.3, we discuss the temporal and spatial characteristics of failure events, and develop two prediction strategies based upon these characteristics. Following the failure characteristics, in Section 3.4, we examine the relationship between fatal events and non-fatal events, and develop another prediction scheme. The related work and concluding remarks are shown in Sections 3.5 and 3.6 respectively.

3.1 BlueGene/L Architecture, RAS Event Logs, and Job Logs

In this study, the RAS event logs are collected from IBM, the supercomputer vendor, through their RAS monitoring system, while the job logs are obtained from LLNL, the hosting site, through their resource management tool SLURM [6, 63].

3.1.1 RAS Event Logs

BlueGene/L has 128K PowerPC 440 700MHz processors, which are organized into 64 racks. Each rack consists of 2 midplanes, and a midplane (with 1024 processors) is the granularity of job allocation. A midplane contains 16 node cards (which houses the compute chips), 4 I/O cards (which houses the I/O chips), and 24 midplane switches (through which different midplanes connect). RAS events are logged through the Machine Monitoring and Control System (CMCS), and finally stored in a DB2 database engine. The logging granularity is less than 1 millisecond. More detailed descriptions of the BlueGene/L hardware and the logging mechanism can be found in Section 2.1.

Raw RAS Event Logs

We have been collecting *reliability*, *availability*, and *serviceability* (RAS) event logs from BlueGene/L since August 2, 2005. Up to the date of November 18, 2005 (109 days), we have totally 1,318,137 entries. These entries are records of all the RAS related events that occur within various components of the machine. Figure 3.1 shows the temporal distribution of these events over this time period at a daily granularity. Information about scheduled maintenances, reboots, and repairs is not included. Each record of the logs contains a number of attributes to describe the event. The relevant attributes are described in Section 2.1.2.

RAS Data Preprocessing

The raw logs contain an enormous amount of entries, many of which are repeated or redundant. Before we can use these logs to study the failure behavior of BlueGene/L, we must first filter the failure data and isolate unique failures. Filtering failure data has traditionally been a challenging task [36]. Further complicating the process is the fact that the BlueGene/L logging

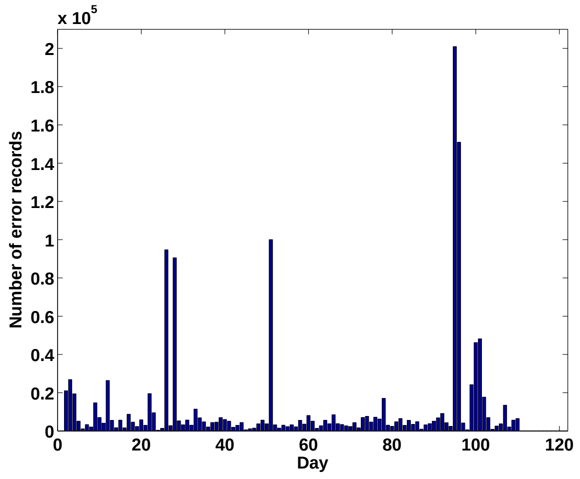


Figure 3.1: Temporal distribution of raw events in the log between August 2 and November 18, 2005

mechanism operates at much finer granularity in both temporal (e.g. the logging interval is less than 1 millisecond) and spatial (hundreds of thousand of processors) domains than earlier machines. In addition, the nature of parallel applications calls for unique filtering techniques that are not needed for sequential applications. To address these challenges, we suitably modify the filtering tool we developed in our earlier work [36], which involves the following three steps:

1. *Extracting and Categorizing Failure Events.* In this step, we extract all the events whose severity levels are either FATAL or FAILURE, referred to as *failures*, because these events will lead to application crashes, and thus significantly degrade the performance. Further, we classify failures into the following categories according to the subsystem in which they occurs: (i) memory failures, (ii) network failures, (iii) application I/O failures, (iv) midplane switch failures, and (v) node card failures. Failures are classified into these five types based on the ENTRY_DATA field.

2. *Temporal Compression at a Single Location.* Failure events from the same location often occur in bursts, and we call such bursts as *clusters*. Some clusters are homogeneous, with their failures having identical values in the ENTRY_DATA field; others are heterogeneous and their failures usually report different attributes of the same event. For example, a memory failure cluster is heterogeneous because each entry reports a unique system state upon the occurrence of a memory failure [36]. Therefore, in the second step, we

Log size with T_{th} =	Memory	Network	APP-IO	Midplane Switch	Node Cards
0	8,206	10,554	178,292	166	96
30 sec	267	9,418	178,015	83	6
1 min	251	9,418	173,491	52	6
5 min	246	9,415	102,442	30	4
30 min	241	9,219	89,333	22	4
1 hour	237	8,705	81,834	17	4

(a) Number of failure events after temporal filtering using different T_{th}

Log size with S_{th} =	Memory	Network	APP-IO	Midplane Switch	Node Cards
0	246	9,415	101,196	30	4
30 sec	217	139	331	30	4
1 min	217	139	318	30	4
5 min	215	139	299	30	4
30 min	208	114	237	22	4
1 hour	199	105	225	17	4

(b) Number of failure events after spatial filtering using different S_{th}

Table 3.1: Filtering thresholds

need to coalesce a cluster into a single failure record. Identifying such clusters from the log, requires sorting/grouping all the failures according to the associated subsystem (i.e. memory, network, or application/IO), the location, and the job ID, and using a suitable threshold T_{th}. Hence, failures that occur within the same subsystem and are reported by the same location and the same job, belong to a cluster if the gaps between them are less than T_{th}. Table 3.1 (a) presents the number of remaining failure records after filtering with different T_{th} values. In this exercise, we set the threshold value to 5 minutes, as also suggested by previous studies [17, 30, 36].

3. *Spatial Compression across Multiple Locations.* A failure can be detected/reported by multiple locations, especially because BlueGene/L hosts parallel jobs. For example, all the tasks from a job will experience the same I/O failure if they access the same directory. For another example, a network failure is very likely detected by multiple locations. As a result, it is essential to filter across locations, which we call *spatial filtering*. Spatial filtering removes failures that are close to each other (gaps between them below the threshold S_{th}), with the same entry data, from the same job, but from different locations. Similarly, Table 3.1 (b) presents the number of remaining failure records after spatial filtering with different S_{th} values. Choosing the appropriate value for S_{th} is rather straightforward since the resulting failure count is not very sensitive to the threshold value. Then we

simply choose 5 minutes.

After applying the three-step filtering algorithm, we can identify unique failures within the boundary of a job. Here, we do not attempt to coalesce failures that are experienced by different jobs because relevant information is missing for this purpose. Among memory, network, and app-IO failures, we find that temporal filtering is effective for memory failures, while spatial filtering is more effective for network and app-IO failures. This is because tasks from a parallel job are more likely to detect same I/O failures or network failures.

3.1.2 Job Logs

The BlueGene/L system is used to run a growing list of applications including hydrodynamics, quantum chemistry, molecular dynamics, and climate modeling. These applications usually require a large number of processors, and the granularity of job allocation on BlueGene/L is a midplane, which consists of 1K processors.

SLURM [6, 63] is used as the resource manager on this BlueGene/L system. It manages a queue of pending jobs, allocates exclusive access to resources for execution of the jobs, and initiates the jobs. A typical job submission specifies the resource requirements (time limit and midplane count) plus a script to be executed. Once the job is allocated resources, the midplane and I/O nodes are re-booted as needed. When resources are ready for use, the job script is executed. Every job submitted to SLURM is recorded upon its termination.

SLURM has several fault-tolerant mechanisms. For example, it may need to reboot nodes for particular applications, and if a boot fails, SLURM repeats the operation for up to three times or at least for five minutes before the job terminates with a FAILED state. The nodes which can not be booted are removed from service until a system administrator resolves the problem. SLURM also monitors the state of resources (midplanes and link-cards). Any resource found in an ERROR state is removed from service until a system administrator resolves the problem. Both of these mechanisms prevent SLURM from trying to run applications on faulty resources, which could quickly abort every job in the queue.

Raw Job Logs

We have obtained the job log from LLNL during the duration between 8/31/2005 and 11/23/2005, total 85 days. (Note that this period is different from the period for RAS event logs.) There are totally 136,890 entries in the job log, and each entry has the following relevant attributes:

- *JobId* is the ID for each job entry. These numbers are sometimes recycled, and thus are not unique.
- *UserId* is the ID of the user who has submitted the job. Since the original information of this field is confidential, it has been mapped to numbers before the log is released to us, and the mapping from the original user ID to the number is unique. Hence we can still use these numbers to correlate jobs from the same user.
- *JobState* is the status of the job upon termination. It has three values: COMPLETED, CANCELLED, and FAILED. COMPLETED indicates that the job was allocated resources and the job script ran to completion with an exit code of zero. CANCELLED indicates the job was explicitly cancelled by either its user or the system administrator, possibly before initiation. FAILED indicates that the job could either not be initiated for some reason or it terminated with a non-zero exit code. Most of the times, a non-zero exit code would indicate an abnormal job termination, and there are exception cases when the job run consists of more than one mpirun executions.
- *StartTime* is the time when the job starts execution, in the format of MM/DD-HH:MM_SS.
- *EndTime* is the termination time, no matter whether the job ends normally or abnormally. It has the same format as StartTime.
- *NodeList* is the list of midplanes that are allocated to this job.
- *NodeCount* is the number of midplanes the job requires. Usually this field is the size of the NodeList field.

Filtering Job Logs

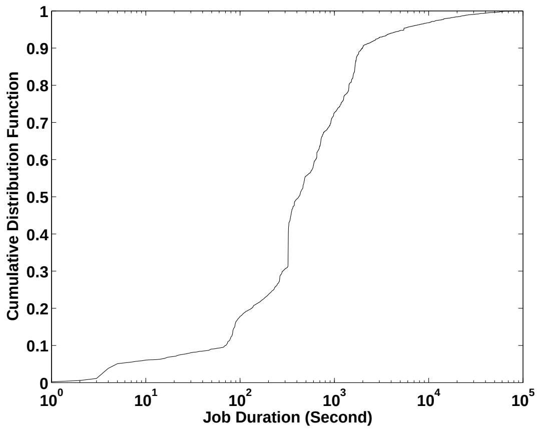
Unlike RAS event entries, each job entry is unique, and there are no redundant job entries. Filtering job logs is thus much simpler, mainly involving the removal of those entries with *null* NodeList and 0 NodeCount. These jobs were never allocated resources or began execution due to some flaw in the submission request such as a request to run a non-executable file or invalid time limit. The large number of failures can be attributed to the use of script to submit jobs, potentially resulting in hundreds of bad requests at a time. After filtering, we reduce the log size from 136,890 entries to 9393 entries, including 1,692 FAILED jobs, 3,937 CANCELLED jobs, and 3,762 COMPLETED jobs. Some statistics about the remaining job entries are shown in Figures 3.2 (a)-(e). Figure 3.2 (a) shows that most of the jobs are rather short: more than 90% of the jobs are less than 1 hour, and among them, a fairly large number of jobs (40% or so) last between 5 minutes and 15 minutes. Figure 3.2 (b) shows that the time between arriving jobs (i.e. job inter-arrival time) is generally rather short. More than 95% of the jobs arrive within a window of 1 hour after their previous jobs, and more than 85% of the jobs arrive within a window of 5 minutes. In particular, we observe that 50% of the job arrivals occur within 1 second from each other. A closer look at the raw job log reveals that there are bursts of job arrivals, and each of these bursts consist of multiple jobs, often 128, from the same user, starting from almost the same time. These bursts are not a normal operating mode, but meant for "stress" test, which are more often at the beginning of the system operation. Moving our attention to job size (specified by the NodeCount field) distributions shown in Figure 3.2(c), we only observe a small number of job size values: 1, 2, 4, 8, 16, 32, 64, 127, and 128, with the unit being a midplane. Most of these values are 2's exponentials. Most of the jobs require only one midplane (6525 out of 9393), and the second most popular job size is 128 midplanes (808 out of 9393).

Another important workload metric is CPU utilization, which measures how busy the system is. Suppose each job i has duration t_i and size s_i. Suppose that T denotes the entire operation duration (from 8/31/05 to 11/23/05 in our case), n the total number of jobs submitted to the system during the duration T, and N denotes the entire system size (128 in our case).

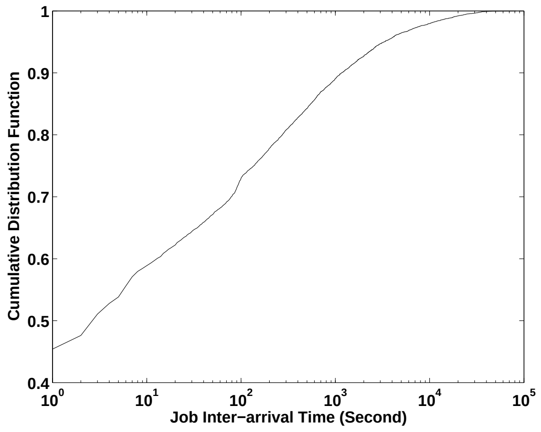
Then the overall CPU utilization is calculated as

$$U = \frac{\sum_{i=1}^{n} s_i \times t_i}{N \times T}.$$

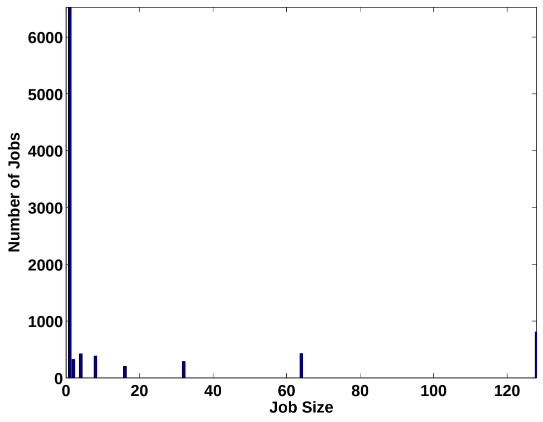
Hence, BlueGene/L has the overall utilization of 0.657 during the observed period. Similarly, we can also calculate the utilization for specific midplanes within specific durations. For example, Figure 3.2 (d) shows the utilization for each midplane during the entire duration, and Figure 3.2 (e) shows the system utilization during each day. Despite the fact that many jobs require a small number of midplanes, different midplanes have similar utilization. This is because the job allocator has done a good job at balancing the load across different midplanes. On the other hand, utilization varies considerably with time, as shown in Figure 3.2(e). This is because the job arrival pattern is not uniform with time.



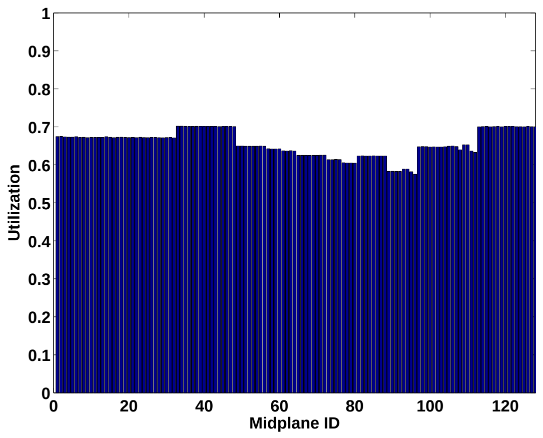
(a) job duration distribution



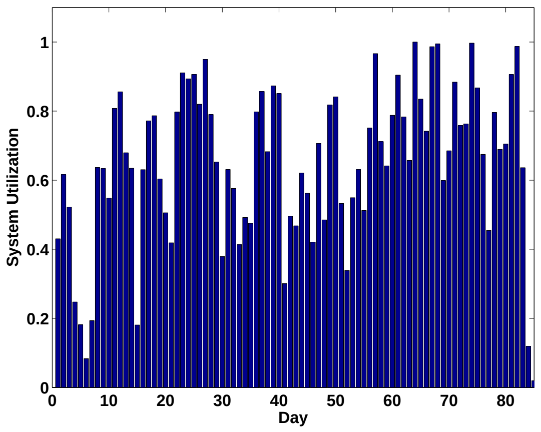
(b) job inter-arrival time distribution



(c)job size distribution

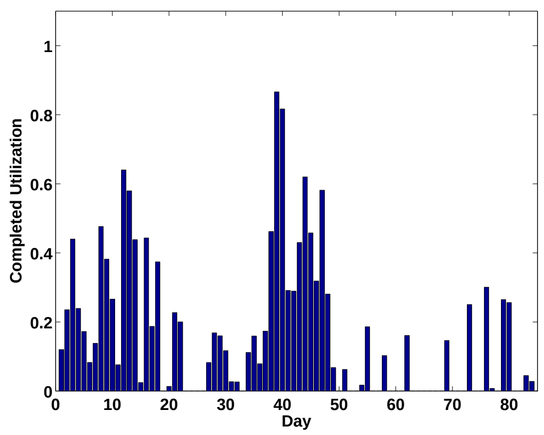


(d)utilization across different midplane

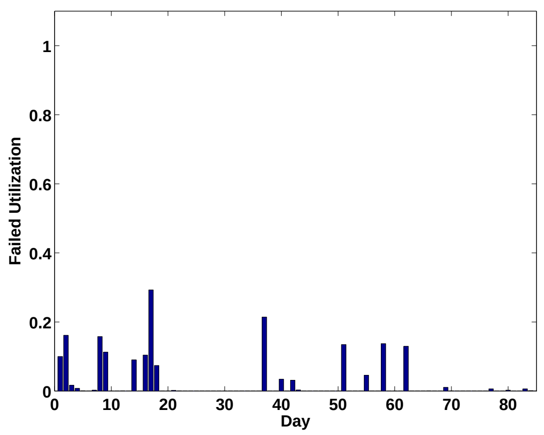


(e) utilization of each day

Figure 3.2: Job statistics



(a) effective utilization per day



(b) failed utilization per day

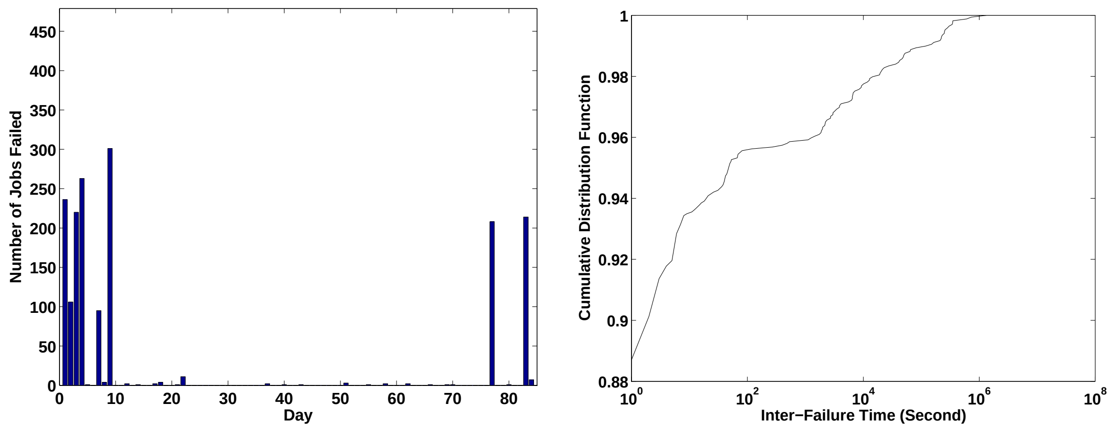
Figure 3.3: Effective utilization versus failed utilization

3.2 Impact of Failures on Job Executions

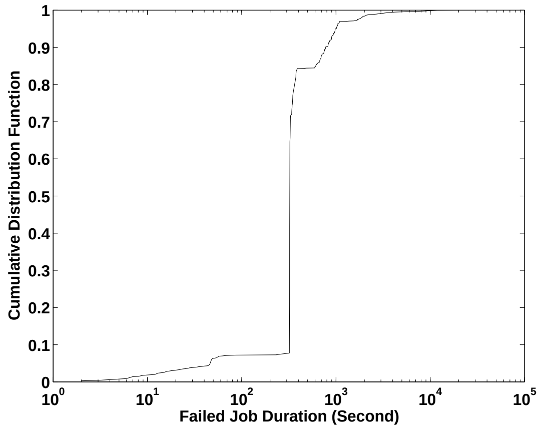
During the period between 08/31/05 and 11/23/05, 9393 jobs were submitted to and executed on BlueGene/L, among which 1692 aborted due to hardware/software failures. Please note that there is a discrepancy between the number of failed jobs, obtained from the job logs, and the number of failures after filtering, obtained from the RAS event logs, because these two logging mechanisms operate at different levels of system operation, with RAS logging being lower level than job logging.

Figures 3.3 (a)-(b) show the effective utilization and failed utilization every day during the period between 08/31/05 and 11/23/05. Effective utilization is contributed by jobs with COMPLETED status, while failed utilization shows how much CPU resource is wasted for failed jobs. Both effective utilization and failed utilization exhibit considerable variation with time. Over the entire period, the overall failed utilization is 0.02, and the overall effective utilization is 0.17.

Figures 3.4 (a)-(c) present the statistics for the failed jobs. Figure 3.4(a) shows that the number of failed jobs varies significantly from day to day. During the period of 85 days, 72.46% of the failed jobs occur within a window of 9 days. This temporal imbalance can be in part explained by Figure 3.4(b) showing the CDF of the time between consecutive failed jobs. More than 89% of the failed jobs failed within 0 or 1 second after the previous failed job. As we explained in Section 3.1.2, users often launch 128 jobs simultaneously, each requiring



(a) number of failed jobs per day (b) time between failed jobs



(c) duration of failed jobs

Figure 3.4: Statistics of failed jobs

one midplane, and if a failure occurs during their executions, then multiple jobs will fail at the same time, resulting in small inter-failure times less than 1 second. After clustering the failed jobs based on their EndTime and NodeCount fields, we only find 74 such clusters. Some of the clusters contain many jobs with one midplane, and others contain a small number of jobs (usually 1 or 2), but each job with many midplanes (128 or 64). Therefore, although 1692 jobs failed over the period of 85 days, the real effect is not as severe as the case in which these jobs fail more uniformly. Figure 3.4(c) shows an interesting phenomenon that 80% of failed jobs had executed for 5 minutes before they terminated. This is because that if the allocated midplanes have problems, the resource management tool of BlueGene/L will keep retrying to boot the blocks for 5 minutes.

Figure 3.5 offers a closer picture of the failed jobs. A job that fails at time t, spanning across n midplanes $m_1, \dots, m_n$, is represented by n dots $(t, m_1), \dots, (t, m_n)$ in this figure. From this

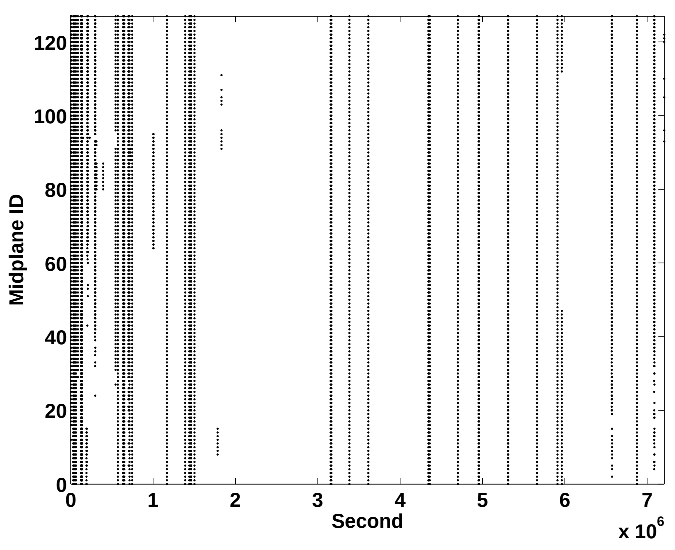


Figure 3.5: The time and midplane for each failed job. The x-axis values are normalized by subtracting the timestamp of the first failed job from every timestamp.

figure, we observe many vertical lines, and each vertical line either represents the failure of a job that has many midplanes, or represents many failures of small jobs that run side by side. Considering the fact that it is very unlikely for many different failures to occur at the same time, we hypothesize that each vertical line may indicate a failure, which may affect multiple jobs. In addition, we observe that several vertical lines may be very close to each other, indicating that consecutive jobs may be affected by the same failure.

3.3 Failure Prediction Based on Failure Characteristics

Before we report our observations, we would like to emphasize that each failure event in our study does not necessarily correspond to a *unique* physical failure in the system hardware or software. Instead, several failure events, especially those with the same entry data and temporally close to each other, may just be that the same failure is encountered by subsequent jobs. Therefore, the observations do not only reflect the system's failure behaviors, but also the interplay between the failure behaviors and the usage patterns (e.g. jobs' arrive/execution times). This vagueness is due to the lack of exact duration information for each failure. However, we emphasize that it is extremely difficult to pinpoint the real root of each failure event, let alone its actual duration, so we do not isolate the impact of job execution on the failure pattern.

3.3.1 Temporal Characteristics

	Entire System	Memory	Network	Midplane Switch	App IO	Node Card
Total failure count	687	215	139	30	299	4
Average daily failures	6.3028	1.9725	1.2752	0.2752	2.7431	0.0367
Variance of daily failures	31.0279	2.4715	7.1643	1.4050	13.5445	0.0542
Maximum of daily failures	37	7	16	8	23	2

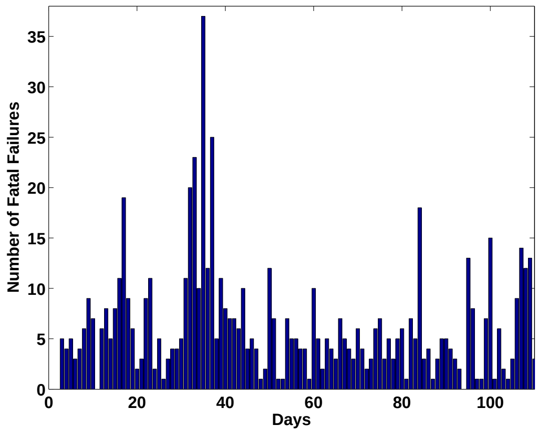
(a) Rate processes of failure events

	Entire System	Memory	Network	Midplane Switch	App IO	Node Card
MTBF (day)	0.1554	0.4963	0.7552	2.0951	0.3577	10.9503
Variance of TBF	6.3038e3	3.2903e4	2.9676e5	2.9576e6	4.5406e4	9.0541e6
Maximum of TBF	2.1566	4.1466	15.4935	22.2213	5.9538	20.4931

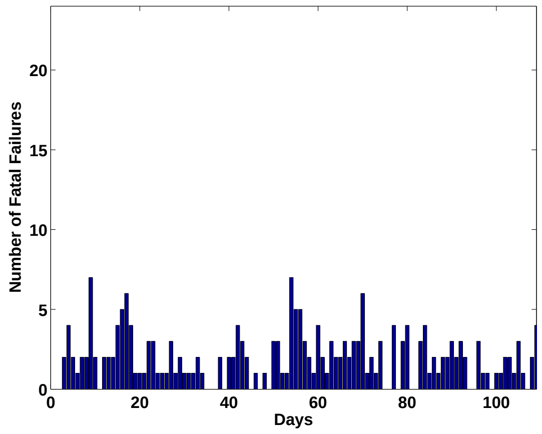
(b) Increment processes of fatal failures

Table 3.2: Failure event statistics, from 8/2/05 to 11/18/05, total 109 days

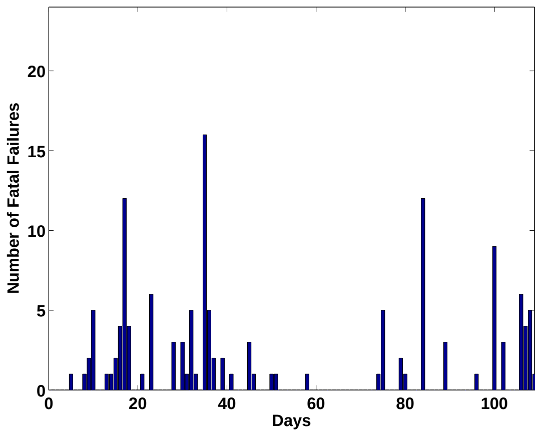
Tables 3.2 (a) and (b) summarize the rate process (i.e. the average, variance, and maximum number of failures within each subsystem per day) and the increment process (i.e. the average, variance, and maximum number of days between subsequent failures within each subsystem) of the fatal failures. From these two tables, we observe that memory, application I/O, and network failures are significantly more dominant than the other two types of failures. As a result, in the rest of this chapter, we focus our attention on these three types of failures and ignore the midplane switch and node card failures. Among these three types of failures, we observe that memory failures are more evenly spread out than the other two, i.e. they have a lower variance of number of failures per day, and a smaller variance of TBF (time between failures).



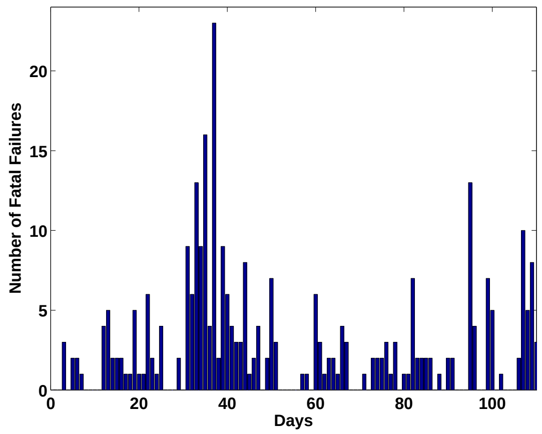
(a) entire system



(b) memory failures



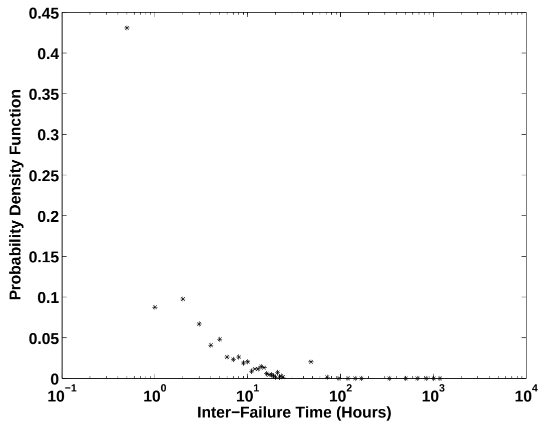
(c) network failures



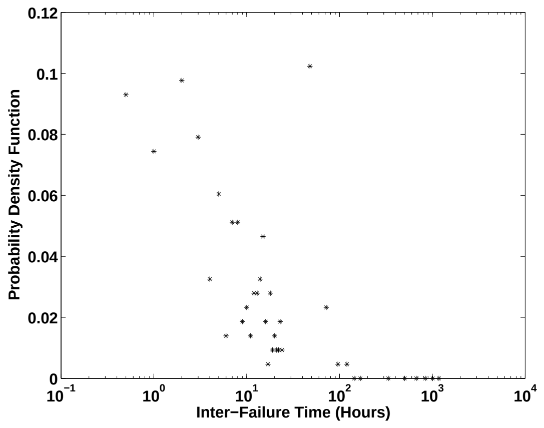
(d) application I/O failures

Figure 3.6: Temporal distribution of failure events from 8/2/05 to 11/18/05. The four plots show the number of failure events observed every day during the duration of 109 days.

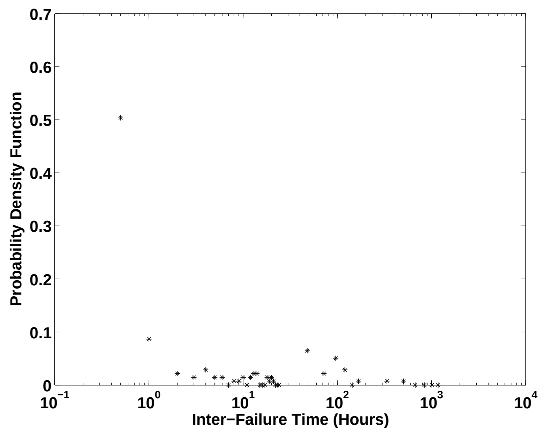
Figure 3.6 and Figure 3.7 depict the two aspects of temporal characteristics of failure events: time series of failure occurrence, i.e. number of failures observed every day during the log duration, and the probability density function (PDF) of the TBF distribution. These results are in agreement with our earlier observation that memory failures are not as bursty as the other two types of failures. Figure 3.6 (b) shows that memory failures almost occur every day, and further, that the number of failures per day does not vary considerably. Figure 3.7 (b) shows that memory failures, though not bursty, do not exhibit periodic occurrence as well; not a single TBF value dominates, but many TBF values are possible and have comparable likelihoods. Unlike memory failures, both network and application I/O failures occur in bursts. As a result, small TBF values are more popular than larger ones. For example, 50% of the network failures occur within half an hour after the previous failures, and 35% of the application



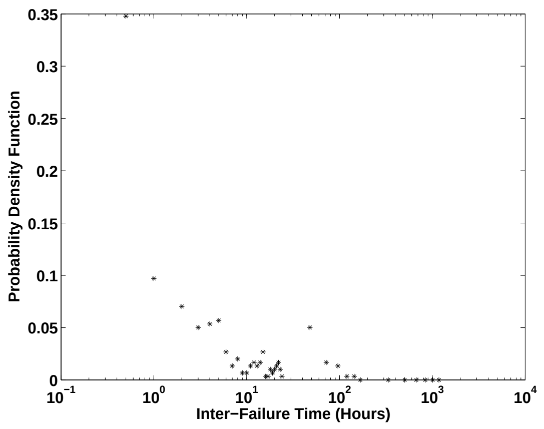
(a) entire system



(b) memory failures



(c) network failures



(d) application I/O failures

Figure 3.7: Temporal distribution of failure events from 8/2/05 to 11/18/05. The four plots show the probability density function (PDF) of the TBF during the duration of 109 days.

I/O failures occur within half an hour after the previous failures. In addition, another 10% the network and application I/O failures occur within a window of between half an hour and an hour after the preceding failures. A possible reason is that it is harder to pinpoint network as well as application I/O failures than memory failures because they tend to involve more hardware components. This hypothesis is supported by the results presented in Table 3.1 (a) and (b), which show that a network or application I/O failure is usually reported by many locations simultaneously (e.g. 139 network failures lead to 9,415 records; 299 application I/O failures 102,442 records), while a memory failure is usually reported by only one or few locations (e.g. 215 memory failures only have 246 records). As a result, a network or application I/O failure may hit several consecutive jobs, or jobs that are running side by side simultaneously, resulting in small TBF values.

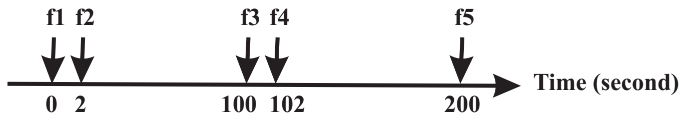
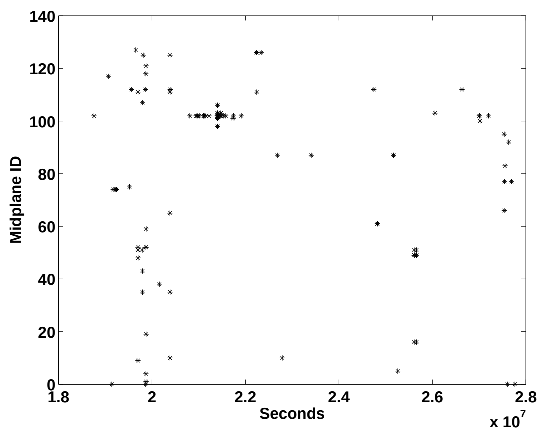
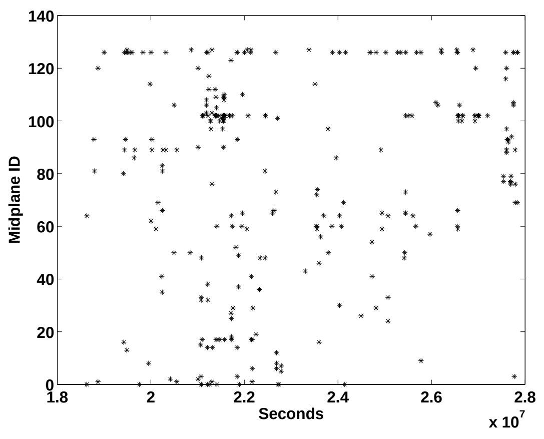


Figure 3.8: Prediction algorithm based on TBF

Prediction Based on TBF: Based on the above observation, we can naturally develop a simple failure prediction strategy for network and application I/O failures: as soon as such a failure is reported, the system should be closely monitored for a period of time since more failures are likely to occur in the near future. However, the tricky issue here is that if the next failure is too close to the current one, say within a window of a few seconds, then predicting its occurrence is not very useful. For instance, in the example scenario shown in Figure 3.8, although f_1 can be used to predict the occurrence of f_2, the gap between them is only 2 seconds, making the prediction less useful since few meaningful actions can be taken in such a short time frame. On the other hand, f_1 can be used to predict the occurrence of f_3 and f_4, and both predictions are useful. As a result, in this example, we can use this simple prediction strategies to make effectively predict the following three failures: f_3, f_4, and f_5.



(a) network failures



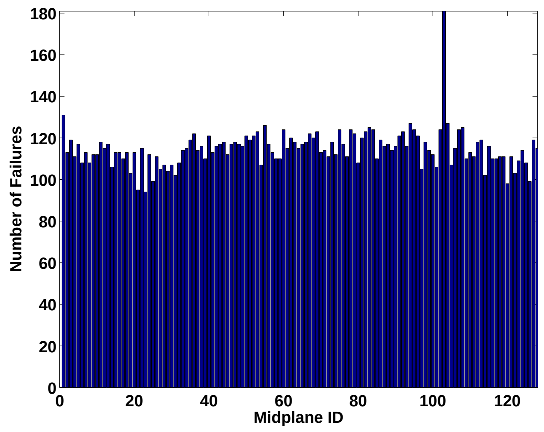
(b) application I/O failures

Figure 3.9: The time/location of each failure. Consecutive failures likely to occur on the same or neighboring midplane(s).

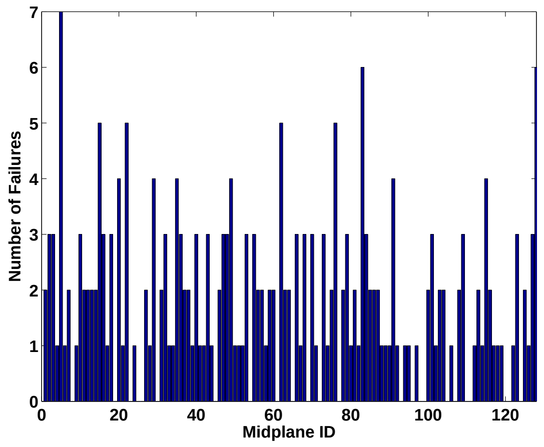
We have run the proposed prediction algorithms against the network failures and application I/O failures. In the experiments, we assume that a failure can be predicted by another failure that occurs within a window between 5 minutes and 2 hours before its own occurrence. The rationale of choosing this window duration is that predicting a failure that will occur within

5 minutes is not very useful, and that monitoring the system for more than 2 hours incurs a high overhead. Using this window size, we find that we can predict 52 network failures out of 139 (37%), and 143 application I/O failures out of 299 (48%). Furthermore, if we lump all the failures together, and use this strategy, then we can predict 370 failures out of 687 (54%). To further support the feasibility of this strategy, Figures 3.9 (a) and (b) reveal that subsequent failures tend to occur on the same midplane or neighboring midplane(s).

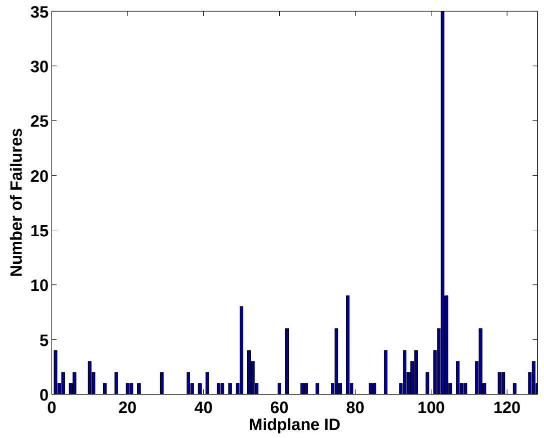
3.3.2 Spatial Characteristics



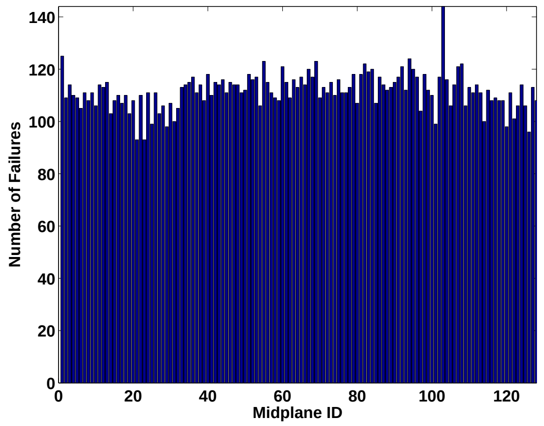
(a) entire system



(b) memory failures



(c) network failures



(d) application I/O failures

Figure 3.10: Number of failures for each midplane during the period between 08/02/05 and 11/18/05. In these results, we have performed spatial filtering within a midplane, by removing redundant records from different nodes within a midplane, but not across midplanes.

After considering the temporal characteristics of failures, we next look at their spatial characteristics, i.e. how the failures are distributed across the 128 midplanes. Figures 3.10 (a)-(d) present the number of failures that have occurred on each midplane during the entire period. In order to study the spatial distribution of failures, we do not perform the normal spatial filtering algorithm, in which all the failure records (1) whose entry data are the same, (2) whose job IDs are identical, (3) whose timestamps are within a window of S_{th} from each other, and (4) whose locations are different, are coalesced into one single failure record. Instead, we adopt a *partial spatial filtering* algorithm, which coalesces redundant failure records only within the boundary of a midplane. Those failure records, though satisfying the conditions (1)-(3), but from different midplanes, then stay uncompressed. Hence, summing up the number of failures for each midplane after using the partial spatial filtering algorithm, results in a larger number than what is reported in earlier sections due to the redundancy between midplanes. Specifically, using the partial spatial filtering algorithm can lead to 230 memory failure records (versus 215 after using the normal algorithm), 180 network failure records (versus 139 after using the normal algorithm), and 14303 application I/O failure records (versus 299 using the normal algorithm).

Figure 3.10 shows that failures from different components demonstrate different spatial behaviors. Like the case in temporal distribution, memory failures are fairly evenly distributed across all the midplanes; 104 out of 128 midplanes have reported failures, and the maximum number of memory failures a midplane has is 7. This observation indicates that all the midplanes have a similar probability of failing in their memory subsystem, and that it is hard to predict memory failures based on the spatial characteristics. Similarly, we also observe that every midplane has a comparable number of application I/O failures (refer to Figure 3.10(d)). This observation, however, is not because all the midplanes have uniform probabilities of failing in the I/O subsystem, but due to the fact that an application I/O failure can be detected/reported by many midplanes simultaneously. Therefore, it is not very feasible to predict application I/O failures using their spatial distribution.

Network failures show more pronounced skewness in the spatial distribution. Among the 128 midplanes, 61 of them have network failures, and midplane 103 alone experiences 35 failures, 26% of the total network failures. In addition, 6% of the midplanes encounter 61% of the failures. Because of this skewness, we propose to develop a simple prediction strategy for network failures.

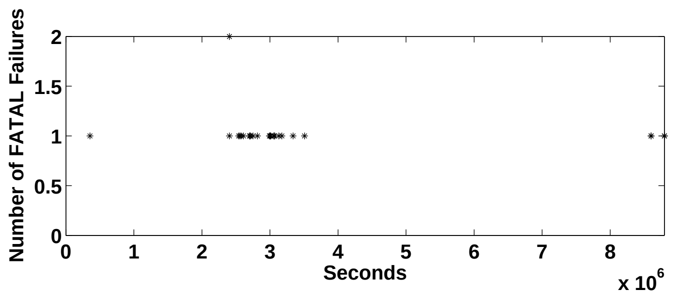


Figure 3.11: The time series of failure occurrence on midplane 103

Failure Prediction Based on Spatial Skewness: For network failures, we can focus on those midplanes that have reported more failures than others because they are likely to have even more failures in the future. Figure 3.11 shows the time series of failure occurrence on midplane 103, which has the most number of failures (35) among all the midplanes. Clearly, most of the failures on midplane 103 are close to each other temporally as well. This observation has two implications on the failure prediction: (1) this simple prediction strategy is very promising since most of the failures are clustered together; and (2) the hotspot midplane may change with time, and we need to dynamically choose the appropriate hotspot.

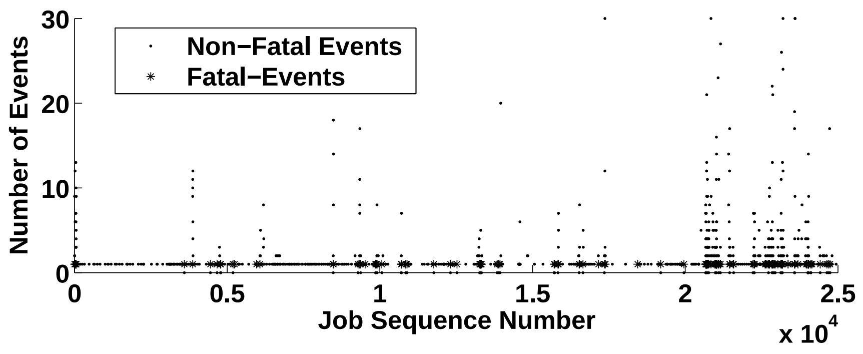


Figure 3.12: The number of non-fatal and fatal events for each job. On the x-axis, we obtain the job sequence number by subtracting the minimum JOB_ID from the original JOB_ID field of the raw log.

3.4 Predicting Failures Using the Occurrence of Non-Fatal Events

After studying the temporal/spatial characteristics of the BlueGene/L fatal events, we next investigate the correlation between fatal events and non-fatal events. Such correlation may lead to efficient ways of predicting the occurrence of fatal events, and thus minimizing the adverse impact of such events on system performance.

Since exploring the correlation in detail requires an enormous amount of effort simply due to the volume of the non-fatal events, we first conduct a quick experiment to evaluate the likelihood of such correlation. For this purpose, we take those fatal events (after filtering) with a valid JOB_ID field, and search the raw logs to examine whether the same job has also reported non-fatal events before this fatal event. Please note that the jobs from the RAS event logs are different from the jobs from the job logs because they are logged at different levels on the execution stack. To differentiate these two, we refer to the jobs from the RAS event logs as *MPI jobs*. This experiment reveals that, among 134 MPI jobs that are terminated by a fatal memory failure, 82 reported one or more non-fatal events; among 34 MPI jobs that are terminated by a fatal network failure, 15 reported one or more non-fatal events. (There are more MPI jobs that failed due to memory or network failures, but not all of them have a valid JOB_ID field.) This observation shows that it is promising to use the occurrence of non-fatal events to predict the occurrence of fatal events.

Motivated by the observation from the quick experiment, we next conduct a more detailed study to extract the occurrence pattern of both the fatal and non-fatal events. Again, we limit our

n	no. of jobs with n non-fatal events (x)	no. of failures within a window of 5 jobs after these jobs (y)	y/x (%)
$[40, \infty)$	4	1	25
$[20, 40)$	9	2	22.22
$[10, 20)$	30	8	26.67
$[2, 10)$	257	53	20.62
1	1543	74	4.70

(a) The correlation between non-fatal events and fatal events

Number of failures with a valid JOB_ID field	168
Number of failures that follow non-fatal events (within a window of 5 jobs)	138
Number of failures that follow another failure (within a window of 5 jobs)	16
Number of stand-alone failures	14

(b) Some statistics about fatal failures

Table 3.3: Exploring the correlation between non-fatal events and fatal failures based on the job_ID field

investigation to events that have a valid JOB_ID field. We first filter all the non-fatal evens using the same three-step filtering algorithm as described in Section 3.1.1. Then we mix the filtered fatal and non-fatal events, and count the number of events each MPI job has encountered. (An MPI job has at most one fatal event.) In this process, we lump together memory events and network events because most of the application I/O events do not have a valid Job_ID field. Figure 3.12 plots the number of non-fatal/fatal events reported by each MPI job, and the x-axis is normalized by subtracting the minimum MPI job ID from every MPI job ID. The figure shows that large bursts of non-fatal events are likely followed by fatal failures. More specifically, we observe that, if an MPI job experiences frequent occurrence of non-fatal events, either itself or MPI jobs immediately following it may be terminated by a fatal failure.

Tables 3.3 (a) and (b) present detailed statistics about the correlation between the occurrence of non-fatal and fatal events. There are 168 fatal memory/network failures that have a valid JOB_ID field, among which only 14 failures are stand-alone, i.e. not within a window of 5 MPI jobs after any job with non-fatal events or fatal events. 16 failures occur within a window of 5 MPI jobs after another fatal failure, and 138 failures are within a window of 5 MPI jobs after an MPI job with non-fatal events. Further, according to Table 3.3 (a), these 138 fatal failures are more likely to occur after MPI jobs that experience more non-fatal events (after filtering). For example, if an MPI job reports more than 40 non-fatal events, then there is a 25% chance that a fatal failure will occur within a window of 5 MPI jobs after it. On average, we observe that if an MPI job experiences 2 or more non-fatal events after filtering, then there is a 21.33% chance that a fatal failure will follow. For MPI jobs that only have 1 non-fatal event, this probability drops to 4.7%. Given that only 300 MPI jobs out of 24,942 (1.2%) have two or more non-fatal events during their lifetimes, it is very reasonable to develop a simple prediction strategy: if an MPI job has observed two non-fatal events, then a fatal failure may occur to this job, or the following four MPI jobs. This prediction strategy only incurs very little overhead, and it can predict 65 out of 168 fatal failures. In addition, if one is willing to pay slightly more cost, by checking the following 5 MPI jobs after observing one non-fatal event (1843 MPI jobs have one or more non-fatal events out of 24,942 total MPI jobs), then one can predict 82% of the total fatal failures. Furthermore, if one can also monitor 5 MPI jobs after a fatal failure, then 16 more fatal failures can be caught, corresponding to 9.5% of total failures. This extra overhead is negligible, since the number of fatal failures is low compared to the total number of MPI jobs.

An interesting phenomenon is that after filtering, only a few types of non-fatal events remain, all with the severity level of INFO. These events are “instruction cache parity error corrected”, “critical input interrupt”, “ddr: activating redundant bit steering”, “ddr: unable to steer”. Furthermore, these events appear before both memory failures and network failures. This will further simplify our prediction strategy because we only need to track very few types of INFO events.

3.5 Related Work

Understanding (and possibly anticipating) the failure properties of real systems is essential when developing pro-active fault-tolerance mechanisms. There has been prior work on monitoring and predicting failures for specific components in computer systems. Storage is one such subsystem which has received considerable attention because of its higher failure rates, and their goals are somewhat similar to PROGNOSIS (except in a different context). S.M.A.R.T. is a recent technology, that disk drive manufacturers now provide, to help predict failures of storage devices [28]. SIGuardian [5] and Data Lifeguard [3] are utilities to check and monitor the state of a hard drive, and predict the next failure, to take pro-active remedies before the failure. More recently, a Reliability Odometer [51] has been proposed for processors to track their wear-and-tear and predict lifetimes.

Moving to parallel/distributed systems, Tang et al. [53–55] studied the error/failure log collected from a small (seven machines) VAXcluster system, to show that most errors are recoverable and the failures on different machines are correlated. A confirmation of the error propagation across machines was noted by [62] on a heterogeneous cluster of 503 PC servers. A more recent study [27] collected failure data from three different clustered servers (between 18-89 workstations), and used Weibull distribution to model inter-failure times. Both these studies [27, 62] found that nodes which just failed are more likely to fail again in the near future. At the same time, it has also been found [57] that software related error conditions can accumulate over time, leading to system failing in the long run.

In addition to examining just the inter-failure times, a more careful examination of all system events can provide better insights/predictions. Consequently, several studies [13, 59, 60], including ours [47], have examined system logs either in an online or offline fashion, to identify *causal events* that lead to failures. The imposed workload can also have a high correlation on the failure properties of real systems as pointed out in many studies [38,40]. Cross-correlations between the workload and system events can thus be useful to develop better failure prediction models.

However, there is a void in event/failure data of a large scale parallel system which can be used to not only develop fault prediction models, but also for designing/evaluating runtime solution strategies, that this study attempts to fill using event/failure logs from the BlueGene/L system.

3.6 Concluding Remarks and Future Directions

Frequent fault occurrences and their consequences on system performance and management costs are a rapidly growing concern for large-scale parallel systems, such as IBM BlueGene/L. While fault avoidance has been shown impossible, fault-tolerance has not made much progress either, mainly due to the high overheads involved in runtime techniques, such as checkpointing, and the complete lack of hints about when and where the next failure will occur. Though it is widely believed that failures are hard to predict, as the scale of the system/application increases, and the logging mechanism evolves, it is worthwhile to revisit the feasibility of predicting failure occurrences.

This study has tackled this challenge by looking at RAS event logs and job logs from BlueGene/L over a period of 100 days. It finds strong correlations between the occurrence of a failure and several factors, including the time stamp of other failures, the location of other failures, and even the occurrence of non-fatal events. Based on these correlations, three simple yet powerful prediction schemes have been designed, and their effectiveness have been demonstrated through analysis and empirical results. With these prediction schemes deployed online, one is able to effectively predict failures in the future, and possibly take remedial actions to mitigate the adverse impacts that these failures would cause. Also, this study shows that these prediction schemes can be implemented at a very low runtime cost.

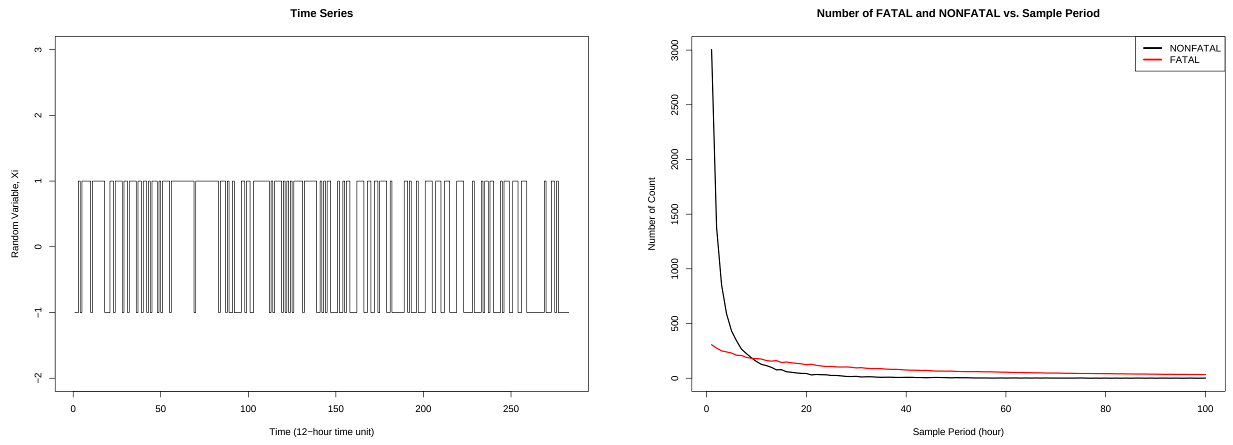
Chapter 4

Time Series Analysis

As described in the earlier sections, each RAS event contains a time stamp, and the events are sorted in increasing time order. Before the RAS logs can be used efficiently in time series study, they have to be transformed into a sequence of random variables in time order. An intuitive transformation is done as follows. We break the time into time windows and observe events in each window. If a time window contains no FATAL event, that window is marked as NONFATAL; otherwise, it is marked as FATAL. From a random process point of view, the FATAL or NONFATAL label for each time window is a suitable random variable, and a sequence of FATAL and NONFATAL labels observed and recorded from an extended period of time is an appropriate time series of random variables. Furthermore, we assign numeric values to represent these labels. Specifically, in this study, value 1 is assigned to the FATAL label, and value -1 is assigned to NONFATAL label. By following this assignment, a random variable in the random process may have a mean close to zero and a standard deviation σ close to 1, and the random variable can be presented as follows.

$$X_i = \begin{cases} 1, & x_i = FATAL, \quad (4.1) \\ -1, & x_i = NONFATAL, \quad (4.1') \end{cases}$$

where i is a nature number (positive integer). Then, the time series of the random variables, X_1, X_2, …, is represented by $\{X_i\}$, where $i = 1, 2, \ldots$. Consequently, the FATAL or NONFATAL label is an outcome of the random variable, and the sequence of the labels obtained from the 142 days of RAS logs is a sample sequence of the time series of the random variables $\{X_i\}$. For convenience, "sample $\{X_i\}$" is used to represent the sequence of the labels and their corresponding numeric values in the later sections. In the time series study, we apply discrete time series analysis and modeling techniques for our time series $\{X_i\}$, and we apply AR (autoregressive), and ARMA (autoregressive moving-average) models for failure predictions.



(a) time series of 12-hour sampling (b) outcomes vs. sample period

Figure 4.1: An instance of the time series and the effect of the sample periods to the number of FATAL/NONFAILURE outcomes

Based on the study interesting failure behaviors are revealed from the sample $\{X_i\}$, and the predictability of AR, ARMA models are evaluated for the time series $\{X_i\}$.

4.1 Cumulative Sum of the Random Variables

A general approach to time series analysis is first to plot it in time order. By looking at the time series diagrams, we may observe some valuable information such as trend and/or seasonal components [15]. Therefore, we follow this thumb rule by plotting a number of time series derived from different time window durations. We have exercised data sampling of 1-hour period (outcome of a random variable), 2-hour period, 3-hour period, . . ., up to 100-hour period that generate 100 discrete time series from the 142 days of RAS logs. After obtaining the 100 time series, we choose a number of them to plot in diagrams. Unfortunately, it is extremely difficult to find any possible trend or periodic component from these plots. Figure 4.1(a) presents an example of these time series plots. In the figure, the sample period is 12 hours, and the duration is the full 142 days of the RAS logs. After studying all the plots, we conclude that all of the plots share the same properties - no trend and no periodic components in the 142 days of logs, and that the increase of the sample period causes the ratio of the NONFATAL to FATAL numbers to decrease from 9.817 to 0 as plotted in Figure 4.1(b). The ratio 0 means all the sample outcomes are FATAL, and the ratio 9.817 suggests that on average, a FATAL outcome is observed for

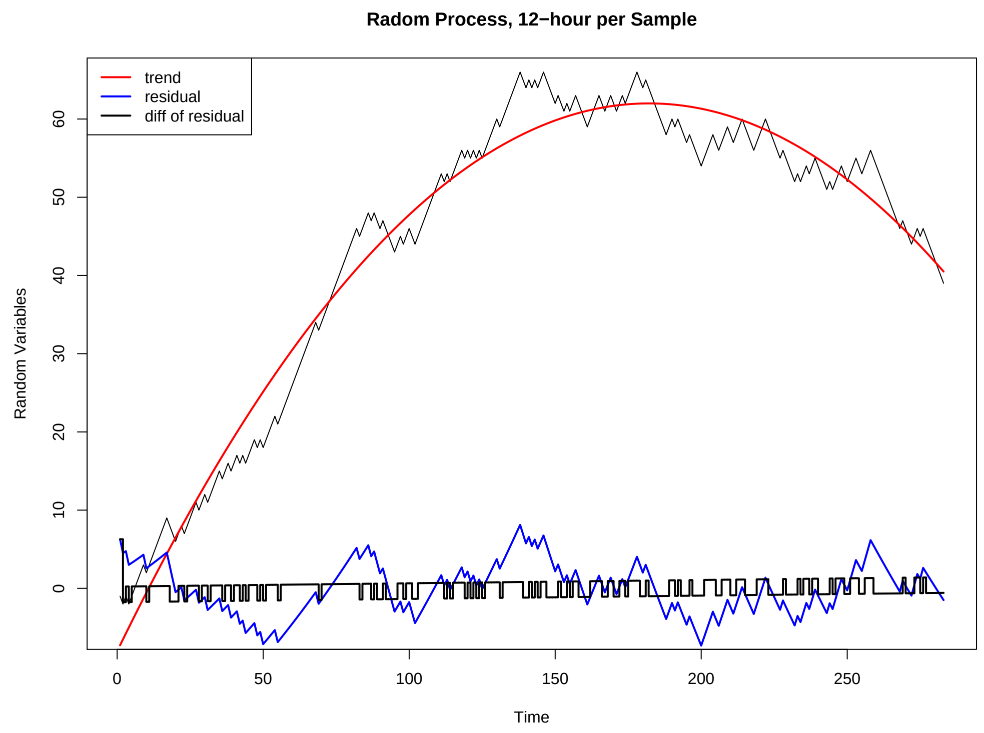


Figure 4.2: Analysis of the cumulative sum

every 10 (9.817 is close to 10) NONFATAL outcomes are drawn. The ratio 0 is first observed from the 77-hour sample period, and at a 9-hour sample period, the ratio $186/182 = 1.022$ is closest to 1. If we assume that the outcomes of FATAL or NONFATAL of the random variables in the time series are equally likely, then we should choose the sample period of 9-hour or around 9-hour. Since we consider 12-hour is a reasonable time window size for a system administrator to take actions in response to system failures. Time series generated by 12-hour sampling period is ,therefore, used to demonstrate our studies of time series analysis, modeling, and prediction in this chapter.

Even though it is difficult to identify a trend or a periodic component from the time series plots, we still can calculate the autocorrelation function (ACF) of the time series to identify the correlation (or dependence) between the random variables. Figure 4.3(a) depicts the sample ACF of the 12-hour sampling window with maximum lag of 60. Apparently, with 12 sample autocorrelations falling outside of the bounds $\pm 1.96/\sqrt{n}$ (where 1.96 is the .975 quantile of the standard normal distribution, and n is the total number of samples, 283 in this case), the time series is definitely not a sequence of iid (independent and identically distributed) noise [15]. Conventionally, the ACF of time series $\{X_i\}$ at lag h is represented by $\rho_X(h)$. Theoretically, if $\{X_i\}$ is iid noise, and expectation of X_i^2, $E[X_i^2] = \sigma^2 < \infty$ (we assume they are true in our random variables), then $\rho_X(h) = 0$ for $h > 0$, and $\rho_X(h) = 1$ for $h = 0$. As a result, one would also expect the corresponding sample autocorrelations of iid noise are close 0. In fact, for iid noise with finite variance, the sample autocorrelations $\hat{\rho}(h), h > 0$, are approximately IID $N(0, 1/n)$ for n large, where N stands for normal distribution [15]. If the time series derived from 12-hour sampling was a sequence of iid noise, approximately 95% of the sample autocorrelations would have fallen between the bounds $\pm 1.96/\sqrt{n}$. Therefore, if there were less than 3 sample autocorrelations falling outside of the bounds, then we would claim the sequence of the 12-hour samples were iid noise. However, as it is indicated there are 12 sample autocorrelations exceeding the bounds, and 12 is far more than 3. As a result, the ACF plots asserts that dependence exists in the 12-hour sampling time series depicted in Figure 4.3(a). Then, how do we find the dependence among the random variables?

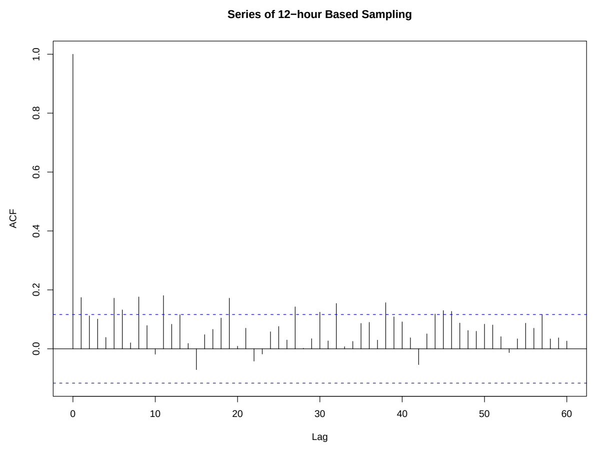
After a careful study and several trying, we identify a trend in the cumulative sum of the time series $\{X_i\}$ as depicted in Figure 4.2, and we define a new random process $\{S_i\}$ for the cumulative sum of the time series $\{X_i\}$ as follows.

$$\begin{cases} S_1 = X_1, & (4.2) \\ S_i = X_i + S_{i-1}. \ where\ i = 2, 3, ... & (4.2') \end{cases}$$

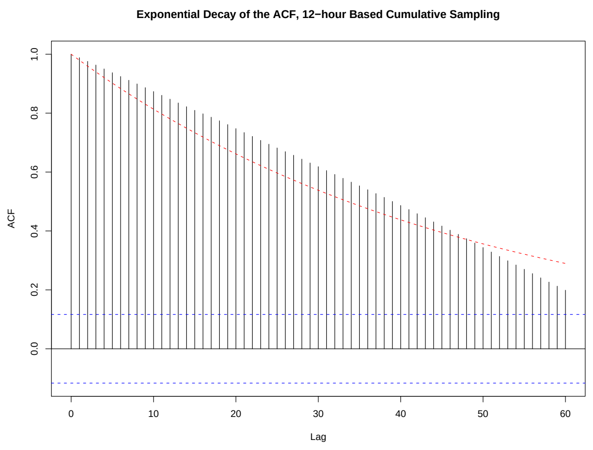
As depicted in Figure 4.2, in the first 180 sample sums, the new time series $\{S_i\}$ increases roughly in a quadratic order, then it falls in the same quadratic order for the rest of the sample sums. In Figure 4.2, a red parabola line presents the quadratic order, and indicates the trend of $\{S_i\}$. The parabola is defined by the following quadratic polynomial.

$$p(i) = -8.048940 + 0.769230i - 0.002112i^2 \tag{4.3}$$

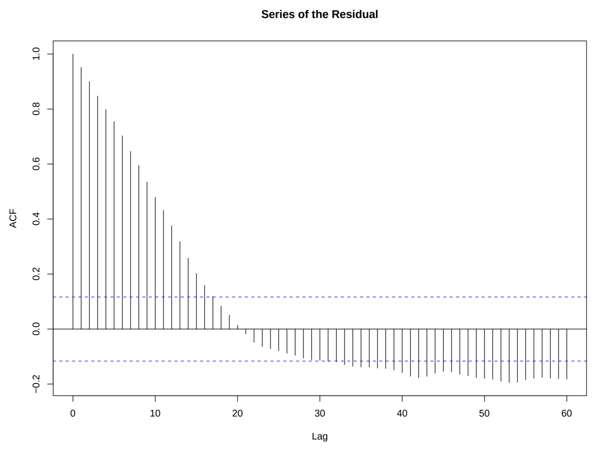
Figure 4.3(b) depicts the ACF of $\{S_i\}$. A red dotted concave up line in the figure indicates an exponentially falling line. We observe that the shape of the ACF of $\{S_i\}$ is somewhat looks like a shape of an ACF from a random walk. Next, we want to remove the parabola shaped trend from the time series $\{S_i\}$, and obtain the residual $\{R_i\}$. The blue line in Figure 4.2 depicts the



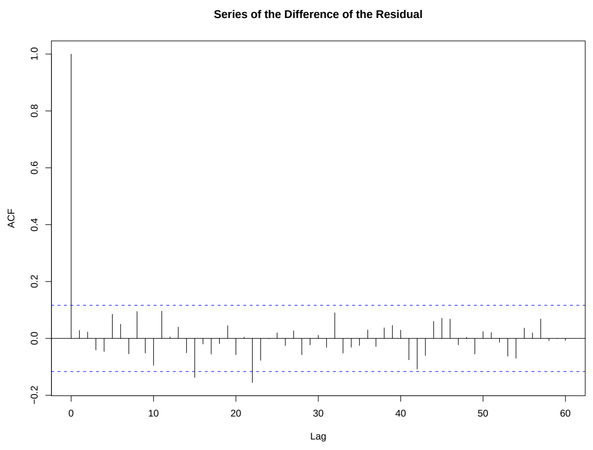
(a) ACF of time series



(b) ACF of the cumulative sum



(c) ACF of the residual



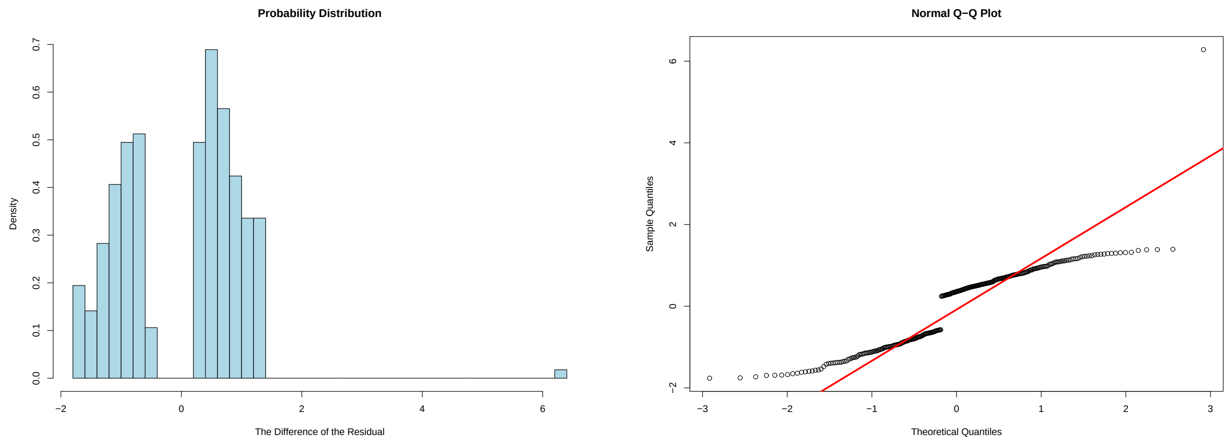
(d) ACF of the difference of the residual

Figure 4.3: ACF plots.

residual, and Figure 4.3(c) plots its ACF. Since the ACF of the residual decreasing rapidly from positive to negative looks so much like an ACF from a random walk, the likeliness prompts us to find the difference of the residual that cumulatively sums up to $\{R_i\}$. The difference of $\{R_i\}$, a new time series $\{D_i\}$, is defined as follows.

$$\begin{cases} D_1 = R_1, & (4.4) \\ D_i = R_i - R_{i-1}.\ where\ i = 2, 3, ... & (4.4') \end{cases}$$

The black line fluctuates around the level of zero in Figure 4.2 indicates the sample sequence of the time series $\{D_i\}$, and Figure 4.3(d) plots its ACF. Clearly, Figure 4.3(d) tells us



(a) sample pdf (b) qqnorm

Figure 4.4: The iid noise study.

the time series $\{D_i\}$ is a sequence of iid noise for only two of its sample autocorrelations exceed the bounds $\pm 1.96/\sqrt{n}$. A time series study should stop at a series of iid noise as described in the book [15]. It is understandable that there is nothing can be done from an iid noise.

This study suggests that our original time series $\{X_i\}$ can be obtained by taking difference of a sum of a random walk and a parabola shaped trend. Moreover, the correlation found in Figure 4.3(a) is nothing but a trend. The trend can be visualized and quantified by a parabola (a quadratic polynomial) in the cumulative sum of the time series $\{X_i\}$. After the trend is removed from $\{X_i\}$, only iid noise are left.

Next, we want to study the iid noise a little more. We want to see how close between the distribution of the iid noise and a normal distribution. The sample probability density function (pdf) of $\{D_i\}$ is plotted in Figure 4.4(a). Figure 4.4(b) depicts a normal quantile-quantile (QQ) plot of the value in $\{D_i\}$, and the red line in the figure is a normal QQ plot that passes through the first and third quartiles.

Except the gap around 0, Figure 4.4(a) shows a bell shape distribution, and again, in Figure 4.4(b), except the gap at 0, the normal QQ plot of $\{D_i\}$ is very close to the normal QQ line. Basically, Figure 4.4(a)(b) echoes each other, the sample pdf of $\{D_i\}$ is close to normal distribution except some missing values at the neighborhood of 0. What happens to the values around the neighborhood of 0? It is noticed that we assign 1 or -1 to the samples of the time series $\{X_i\}$. The gap inherits the characteristics (the gap between 1 and -1) of the way we

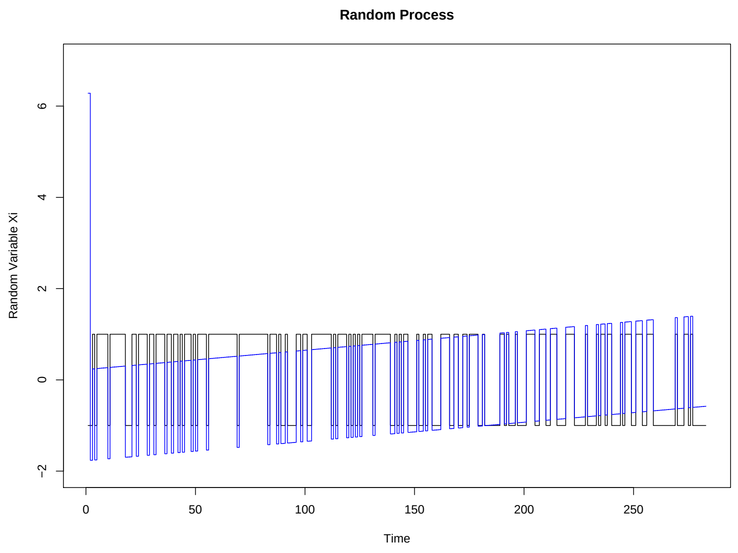


Figure 4.5: The iid noise study.

assign our random variables.

Finally, Figure 4.5 plots our sample $\{X_i\}$ in black and sample $\{D_i\}$ in blue together. Removing the trend from sample $\{X_i\}$ results in a tilt, and some slop and randomness on the continuous 1 and continuous -1 on the sample $\{X_i\}$. What does the trend mean? From Figure 4.2, the sample $\{S_i\}$ tells us that the increasing rate of the cumulative FATAL number is considerably larger than the increasing rate of the cumulative NONFATAL number (resulting in the positive slop on the parabola) at the first 180 samples, and then the slop changes from positive to negative. The number of cumulative FATAL samples is less than the number of cumulative NONFATAL samples. The over all picture is that at the beginning, BlueGene/L was overwhelmed by FATAL failures, then gradually, the overwhelmed situation got better and better. Eventually, after 90 days (180 samples), the number of FATAL failures were remarkably decreased, and NONFATAL periods dominate the on going operation of the machine.

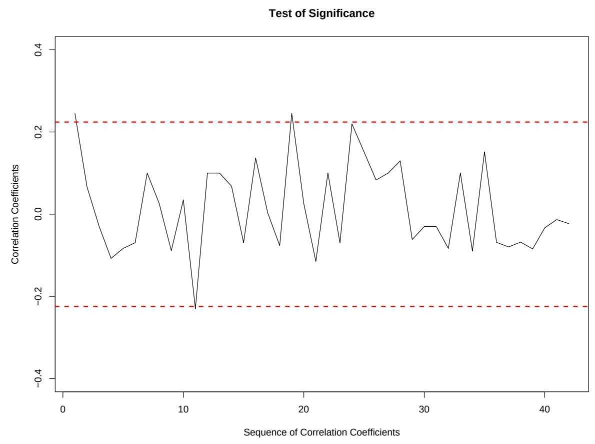
4.2 Correlation and Test of Significance

In this section, we apply the idea of "test of significance" from the world of statistics [4, 7] to our time series $\{X_i\}$, and correlation coefficients [1] are used as a measure to the significance test. In the application, a correlation coefficient is a single number that describes the degree of relationship between two sequences of random variables. And, significance test is performed to determine if the sample distribution of one sample sequence is statistically differs from the sample distribution of the other sample sequence [4, 7].

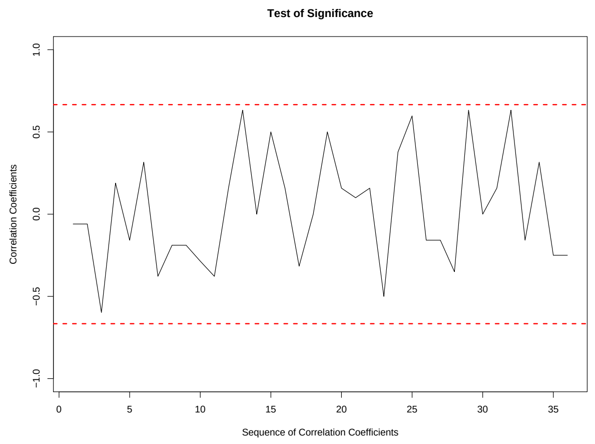
Basically, a question is interesting to us, that is "does the failure experience gained from the past help in failure prediction on the future?" If it does, how much it helps? We try to quantify the significance of the help, and determine how much the past helps to tell the future. The next paragraph describes the way we prepare the data for this study.

The idea for the data preparation is simple. If a FATAL or NONFATAL outcome of a random variable is generated each hour, we observe the FATAL and NONFATAL sequence to an extended period of time, and eventually, the longest time period we can observe is 142 days (the total length of the RAS log). If we choose the extended period of observation equal to 12 hours and a random sample generated every hour, after 12 hours, we start a new observation. We found that some of the observation periods do not contain any FATAL outcome. In order to find some possible periodic failure components, and even a failure trend, we need to extend the observation time window from 12 hours to some extended period of time, so that there exists at least one FATAL outcome and at least one NONFATAL outcome in each of the extended observation period. Then, the sample sequences of the random variables from each of the extended observation periods can be compared, and the correlation between neighborhood sample sequences can be calculated, so that the degree of relationship between any neighborhood sequences is determined.

In this section, two different cases are studied, one case based on a random variable generated every hour, and the other case base on a random variable generated every 10 hours. It is found that 77-hour is the extended period for the case of a random variable generated per hour to have at least one FATAL outcome, and 90-hour is the extended period for the case of a random variable generated every 10 hours to have at least one FATAL outcome. The same Pearson Product Moment Correlation (called Pearson's correlation for short) used in Adaptive Semantic Filter (ASF) is used here to measure the correlation between two neighborhood sample sequences of random variables. As described earlier, this measure ranges from 1 through 0 to -1. Correlation of 1 represents a perfect positive linear relationship, correlation of 0 represents no linear relationship, and correlation of -1 represents a perfectly negative linear relationship. When computed in two sample sequences of random variables, the correlation is designated by



(a) 77-hour period, 1-hour per sample



(b) 90-hour period, 10-hour per sample

Figure 4.6: study of significance test.

the letter r [9].

According to the test of significance in basic statistics [7], first we need to define our null hypothesis and alternative hypothesis, and they are defined as follows.

Null hypothesis, H_0: $r <> 0$, two neighborhood sample sequences extracted from the time series $\{X_i\}$ are highly correlated (meaning similar or no difference). More specifically speaking, for significance level equal to 0.05, between any two of the neighborhood sequences of random variables, no more than 5 out of 100 correlations are less than their critical value, or bigger than their negative critical value. (two-tailed test)

Alternative hypothesis, H_a: $r = 0$, two neighborhood sample sequences of random variables extracted from the time series $\{X_i\}$ are uncorrelated.

To the case of 77-hour observation period (one random variable generated per hour), from the 142 days of RAS log, we have 43 total sample sequences (each sample sequence corresponding to a 77-hour observation period) and therefore, 42 correlation coefficients for all the neighborhood sequences. These 42 correlation coefficients in time order are plotted in Figure 4.6(a). As specified in the null hypothesis, from a statistics table of critical values of testing significance [2], with significance level of $\alpha = 0.05$, degree of freedom (*df*) equal to 75 (*df*=N-2, where N is the number of random variables in the observation period), the critical value of correlation is 0.2242. The red dotted lines in Figure 4.6(a) indicate the bounds of ± 0.2242. In the figure, it shows clearly that only 3 out of 42 (7.1429%) correlations exceed the bounds of

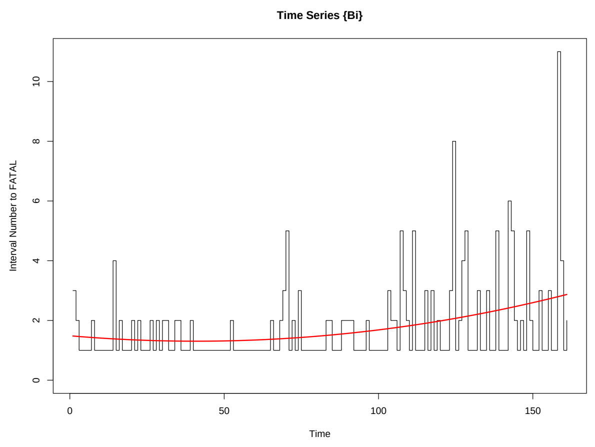
± 0.2242, and 92.8571% of the correlations stay in the boundaries. As a result, it is safe enough for us to reject the null hypothesis.

To the case of 90-hour observation period (one random variable generated per 10-hour), from the 142 days of RAS log we have 37 total sample sequences (each sample sequence corresponding to a 90-hour observation window) and 36 correlation coefficients for all the neighborhood sequences. These 42 correlation coefficients in time order are plotted in Figure 4.6(b). As specified in the null hypothesis, from a statistics table of critical values of testing significance, with significance level of $\alpha = 0.05$, degree of freedom (*df*) equal to 7 (*df*=N-2, where N is the number of random variables in the observation period, N equal to 9 in this case), the critical value of correlation is 0.666 [2]. The red dotted lines in Figure 4.6(b) indicate the bounds of ± 0.666. In the figure, it shows clearly that none of 36 (0.0%) correlations exceed the bounds of ± 0.666, and 100% of the correlations fall inside the boundaries. Therefore, again, it is safe enough for us to reject the null hypothesis.

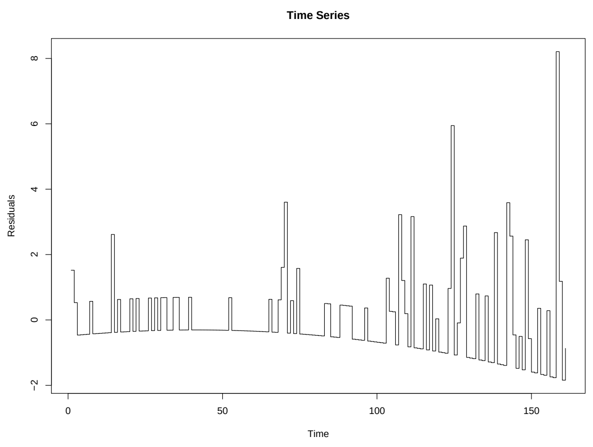
In addition to the low correlation coefficient relationship, as indicated in Figure 4.6(a)(b), to both cases, the traces of the correlation coefficients are fluctuating around 0. In most of the cases, a positive correlation is followed by a negative correlation and vice versa. These fluctuating lines indicate a one time positive and next time negative correlation relationship. This tells us that knowing a failure sequence in one observation window hardly helps anything in prediction a failure sequence in next observation period. As a result, according to the above study, we reject the null hypothesis H_0 and in favor of the alternative hypothesis H_a.

4.3 Time Between Failure

In this section, we focus on characteristics of the time between failures. This study is based on the time series $\{X_i\}$ defined earlier, and again, for the same reasons, 12-hour per random variable is chosen to generate $\{X_i\}$. The new time series derived from $\{X_i\}$ that fully captures the characteristics of the time between failures is $\{B_i\}$. The value of the random variable B_i is defined by “1 plus the number of consecutive NONFATAL samples before a FATAL sample”. For instance, the following sequence of sample $\{X_i\}$, “1, -1, -1, 1, 1, -1, 1, -1, -1, -1, 1, -1, -1, -1, -1, 1, 1”, will produce “3, 1, 2, 4, 5, 1” in samples of $\{B_i\}$, where in the



(a) time between failure sample series



(b) Residuals

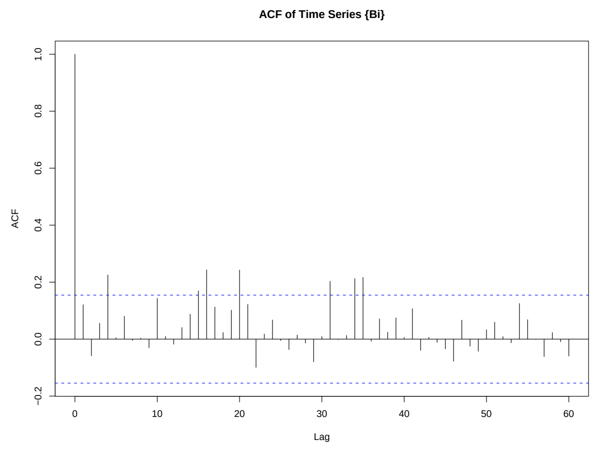
Figure 4.7: Time between failure study.

samples of $\{X_i\}$, 1 represents FATAL and -1 represents NONFATAL same as in the previous sections. Figure 4.7(a) plots the samples of the time series $\{B_i\}$. The red line in the figure is a trend of the time series quantified by a quadratic polynomial that derived from least square approximation. This concave up parabola echoes the trend found in the previous section that the interval between failures increases in a quadratic order. Figure 4.7(b) depicts the time series after taking out the trend from the sample $\{B_i\}$. Figure 4.8 plots the ACF of the sample $\{B_i\}$ in (a) and its residuals after removing the trend in (b). In the ACF figures, it shows clearly that without the trend, the residuals (removing the parabola trend from the sample $\{B_i\}$) is still highly correlated. As a result, the following autoregressive (AR) process approximation and autoregressive moving-average (ARMA) process approximation are directly applied on the sample $\{B_i\}$ without taking out of the trend.

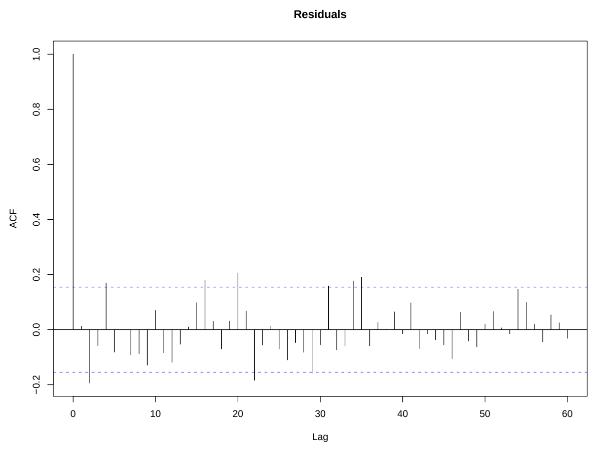
4.3.1 Autoregressive Model for Interval Between Failure Prediction

The tool used in this study is the ar function in R statistics software package. The Akaike Information Criterion (AIC) is used to choose the order of the autoregressive model [10]. The formula used in the AR model is as follows.

$$(x_i - m) = a_1(x_{i-1} - m) + a_2(x_{i-2} - m) + \ldots + a_p(x_{i-p} - m) + e_i \tag{4.5}$$



(a) ACF of the sample $\{B_i\}$



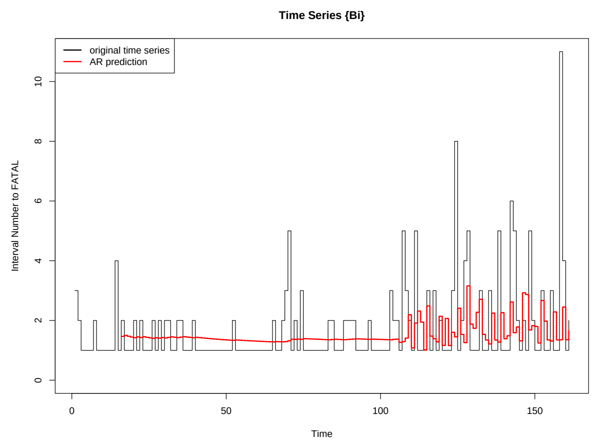
(b) ACF of the residuals

Figure 4.8: ACF study.

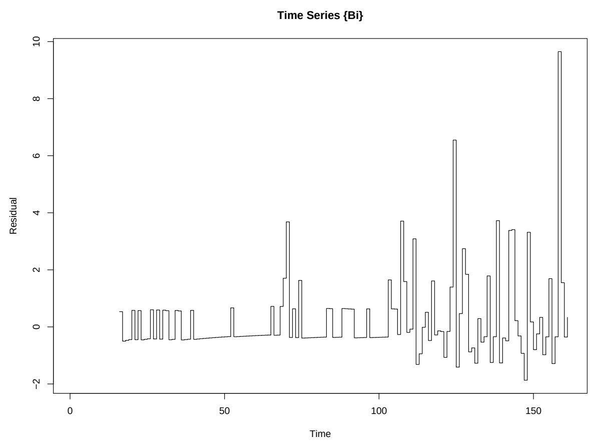
where m is the sample mean (sample 1 to sample $i-1$), and e_i is the error term. Starting from the first 15 samples $b_1, b_2, .., b_{15}$, the 16th random variable b_{16} is estimated by Equation 4.5, then b_{16} together with the first 15 samples is used to estimate the 17th random variable b_{17}, and so on. This calculation steps continue recursively until the last random variable b_n is estimated. The AR predicted time series together with the sample $\{B_i\}$ is plotted in Figure 4.9(a). In the figure, the red line is the AR prediction. The residuals obtained by removing the AR prediction from the sample $\{B_i\}$ is depicted in Figure 4.9(b). Figure 4.9(c) plots the ACF of the residuals. With 5 autocorrelations exceed the bounds $\pm 1.96/\sqrt{n}$, clearly, the residuals are not likely iid noise. Figure 4.9(d) plots the histograms of the residuals. The histograms indicate 86 residuals are between -0.5 and 0.5. It is reasonable to consider the 86 residuals equal to 0 during AR prediction which means out of 146 ($161 - 15 = 146$) predict values, AR can successfully predicts 86 failure intervals.

4.3.2 Autoregressive Moving-average Model for Interval Between Failure Prediction

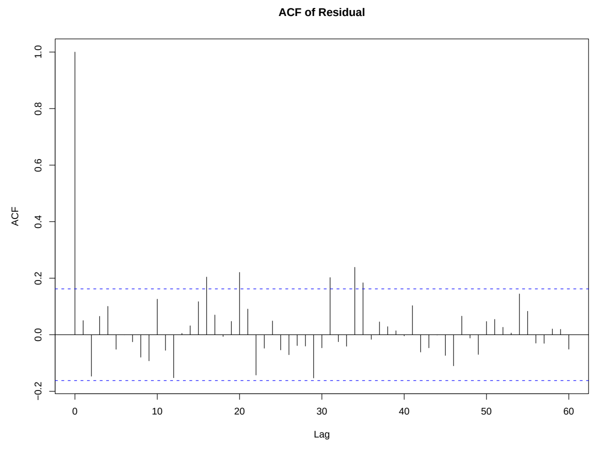
Same as in the previous subsection, the tool used in this ARMA modeling is based on the arma function resided in the R statistics software package. The following parametrization is used for the ARMA(p,q) model [10]:



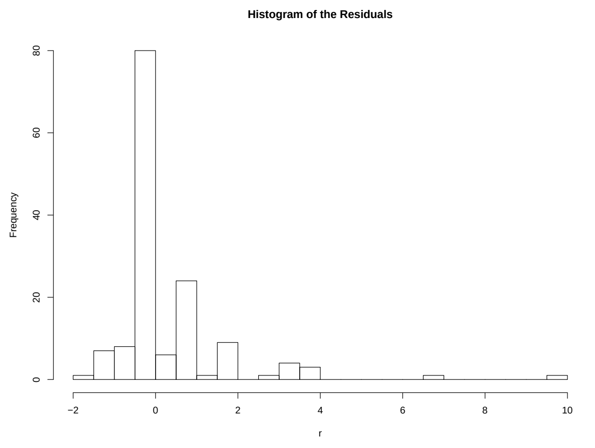
(a) AR



(b) residuals



(c) ACF of the residuals



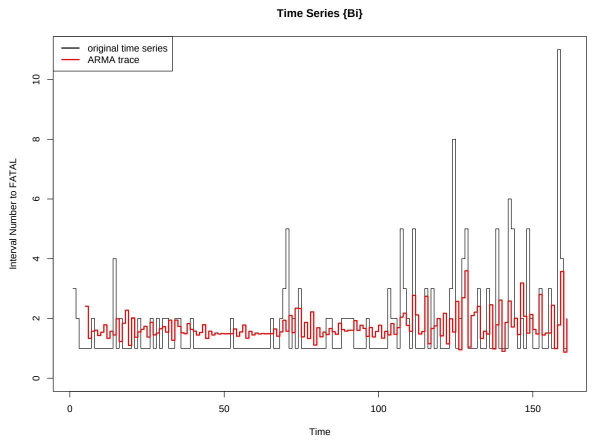
(d) histograms of the residuals

Figure 4.9: AR study.

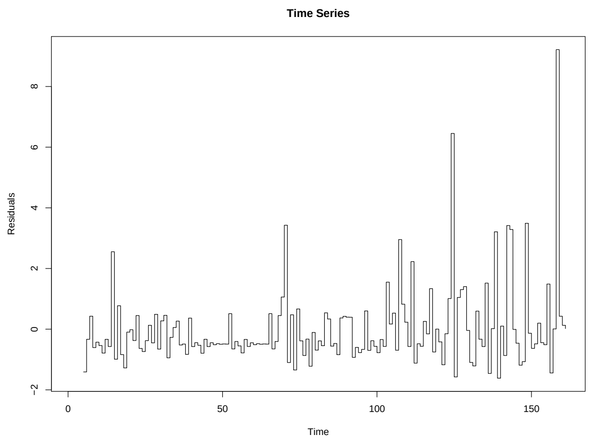
$$x_i = a_0 + a_1 \cdot x_{i-1} + a_2 \cdot x_{i-2} + \ldots + a_p \cdot x_{i-p} + b_1 \cdot e_{i-1} + b_2 \cdot e_{i-2} + \ldots + b_q \cdot e_{i-q} + e_i \quad (4.6)$$

where a_0 is set to 0 if no intercept is included. Unlike the order (or complicity) in the AR model that is determined by AIC, before ARMA function can be used, the (p, q) parameters must be determined first. As a result, an offline study is conducted to assess the (p, q) parameters that lead to optimal results. The whole sample $\{B_i\}$ is fed to arma function, and every possible combination of (p, q) from (1, 1) to (10, 10) (p running from 1 to 10, and for each p, q running from 1 to 10) is given to the function. Due to the randomness found in the earlier sections, and the empirical results we obtained, it is not likely for p or q bigger than 10 to have a ARMA process close to the sample $\{B_i\}$. Clearly, from each (p, q) pair, a ARMA process is obtained. The experiment is next to subtract the ARMA process from the sample $\{B_i\}$ to obtain residuals. By calculating the ACF of the residuals, the randomness of the residuals is determined. From our experiment, we found that with (p, q) equal to (4, 1), the ACF of the residuals are likely iid noise. Figure 4.10(a) plots the sample $\{B_i\}$ together with the ARMA process. Figure 4.10(b) depicts the residuals after removing the ARMA trace from the sample $\{B_i\}$. The ACF of the residuals is demonstrated in Figure 4.10(c). Clearly, with only 3 autocorrelations exceeding the bounds, the ACF suggests the residuals are consisted of iid noise. Figure 4.10(d) shows the histogram of the residuals. Again if we consider the residuals between -0.5 and 0.5 are matches between the sample $\{B_i\}$ and the ARMA trace, then the ARMA process can produce 70 matches out of 157 data points. Next, after the orders of the ARMA model is determined, an ARMA model based predictor can be built.

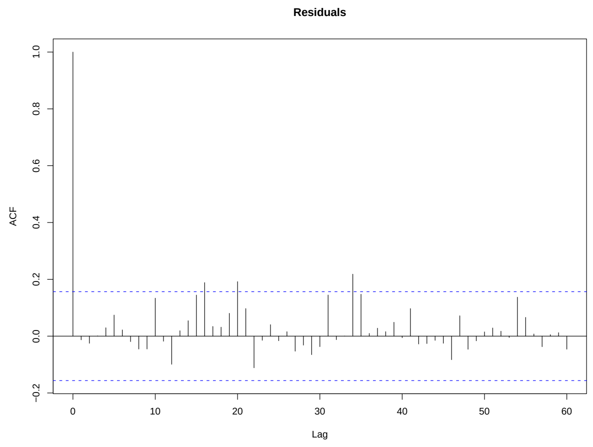
In order to obtain converge results and avoid too many warning messages from the arma function in R, the first 55 samples in $\{B_i\}$ are used to generate the arma(4, 1) coefficients. These coefficients are used to predict the outcome of the 56th random variable. After the prediction, the outcome of the 56th random variable together with the first 55 sample outcomes are fed to the arma(4, 1) function again. The coefficients from the arma(4, 1) function with 56 sample outcomes are then used to predict the outcome of the 57th random variable. As a result, the arma(4, 1) function is called 106 times recursively to produce 106 estimated outcomes for the 106 random variables. Figure 4.11(a) plots the sample $\{B_i\}$ together with the ARMA prediction results. Figure 4.11(b) depicts the residuals after removing the ARMA prediction trace from the sample $\{B_i\}$. The ACF of the residuals is demonstrated in Figure 4.11(c). Clearly, the ACF suggests the residuals are consisted of iid noise. This results agree with the results from the ARMA offline study that chooses arma(p, q) equal to arma(4, 1). Figure 4.11(d) shows the histogram of the residuals. Again if we consider the residuals between -0.5 and 0.5 are matches between the sample $\{B_i\}$ and the ARMA trace, then the ARMA process can produce 71 matches out of 106 data points. This results are remarkable. However, the (p, q) orders of the ARMA models are significant in prediction. They define the correlation between the current random variable and its preceding random variables. Without the offline study, it is impossible for us to know these relationship, and the prediction based on the ARMA model



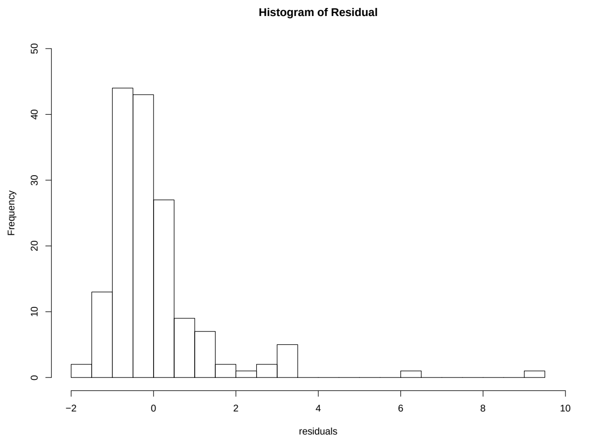
(a) ARMA trace



(b) residuals



(c) ACF of the residual



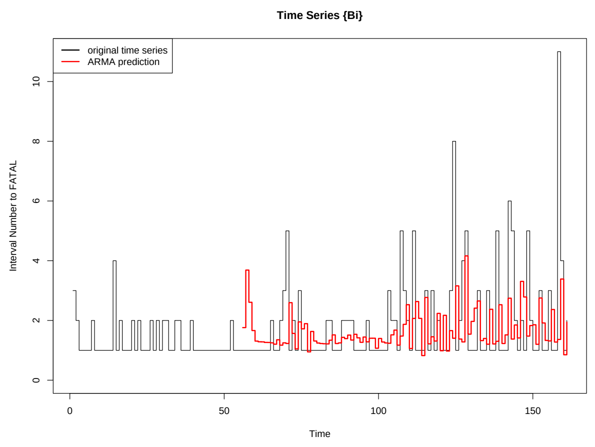
(d) histograms of the residual

Figure 4.10: ARMA study.

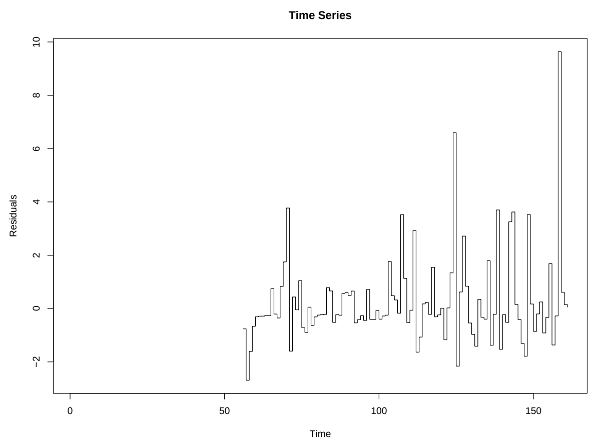
will not be this good.

4.4 Conclusion

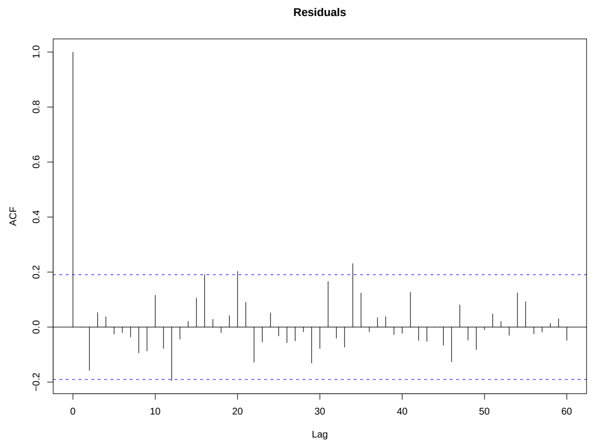
Generally speaking, in order to utilize the time series analysis and modeling techniques in failure analysis and prediction, the RAS event sequence (after filtering) has to be customized to NONFATAL and FATAL sequence. From a broad overview, the sequence does provide some valuable insides such as failure trend and correlation between two neighborhood sequences of random variables. However, by customizing the RAS event sequence, some valuable event information is lost. For instance, the six event severity information, exact number of NONFATAL/FATAL events and the distribution (mean variance, etc.) of the events in an observation



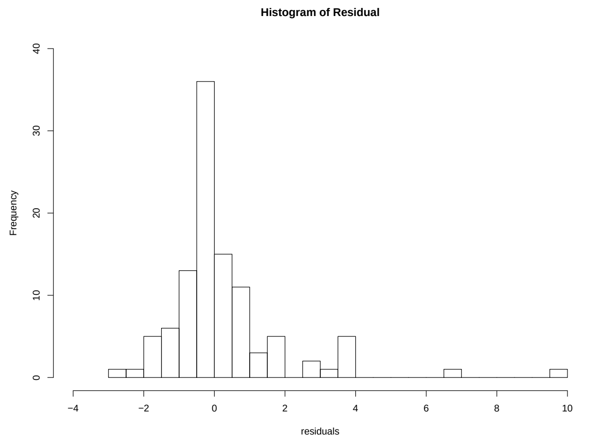
(a) ARMA



(b) residuals



(c) ACF of the residual



(d) histograms of the residual

Figure 4.11: ARMA study.

period, and event entry data information are all lost. Also, considering BlueGene/L is a machine running hundred of thousand compute node in parallel, 142 days of data collection is a relatively short period of time, and BlueGene/L was a newborn giant in these 142 days of data collection, time series modeling is really too much for the 142 days of data. Moreover, hundred of thousand compute nodes running in parallel introduce too much randomness for the simple FATAL/NONFATAL sequence to catch and, therefore, to fully describe the failure behaviors. Besides, the 142 days of event data collection hardly convey any significant failure pattern for time series analysis to discover and for time series modeling to catch. Additionally, in the 142 days of event data collection period, the machine itself was still in a trouble shooting stage (in the sense of personnel training and hardware setup), and was not yet stabilized. Considering all these factors, it is not a surprise for time series modeling doesn’t work out in failure event prediction. However, surprisingly, the ARMA model for TBF prediction is astonishingly good.

Chapter 5

Failure Modeling and Prediction - A Data Mining Approach

Failure prediction is a challenging research problem, mainly due to the fact that the number of fatal events (in a level of hundreds) is far exceeded by the number of nonfatal events (in a level of millions). In the field of data mining and machine learning, this problem is known as rare class analysis problem; that is, mining rare classes in the data sets that contain imbalanced class distributions. It is well accepted that rare class analysis problems are challenging. Another important obstacle that undermines the development of efficient failure predictors is the lack of real-world data and consequently the lack of the thorough understanding of their properties. Despite these challenges, there have been several attempts [47, 48, 58] to derive realistic failure prediction models. While these studies have demonstrated reasonable prediction accuracy, they failed to prove that their prediction methodology is of practical use. For instance, some of them focus on long term prediction based on seasonal system utilization, without pinpointing the occurrence times of the failures [58]. As another example, some of the earlier studies tried to predict whether there will be failure in the next 30 seconds [48], and within such a short notice, no remedial measures can be taken, especially for large-scale systems. To address these issues, in this study, we derive our prediction models from the failure logs collected from IBM BlueGene/L over a period of 142 days. To emphasize both prediction accuracy and prediction utility, we partition the time into fixed windows (each window is one or several hours), and attempt to predict whether there will be fatal event in every window based on the event patterns in the preceding windows.

Our prediction effort consists of two main parts. First, we need to extract a set of features that can accurately capture the characteristics of failures. Feature selection is not only important but also challenging. Often, we cannot directly use the attributes from the event logs as features, but we have to compose features using second-order, or even third-order information from the logs, which requires expert domain knowledge. In particular, we select features based on our previous study [34] on BlueGene/L event characteristics. After establishing features, we exploit three classifiers including RIPPER (a rule-based classifier), Support Vector Machines (SVMs), and a Nearest Neighbor based method for predicting failure events. While we used off-the-shelf tools for the first two methods, we devised our own nearest neighbor predictor which utilizes two distance thresholds. Our evaluation results show that, though RIPPER and SVMs are well-known classification tools, our customized nearest neighbor predictor can substantially outperform them in both coverage and precision. We also discuss the feasibility of employ the nearest neighbor prediction to improve the runtime fault-tolerance of IBM BlueGene/L.

The rest of the chapter is organized as follows. In Section 5.1, we provide our overall prediction methodology and feature selection algorithm. Next, we discuss the details of the three classifiers in Section 5.2, and the detailed evaluation results in Section 5.3. Finally, we present the concluding remarks in Section 5.5.

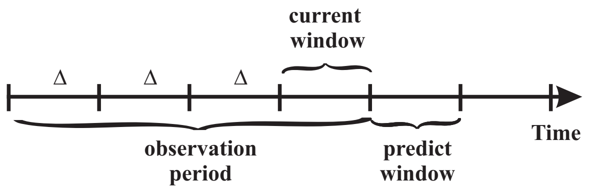


Figure 5.1: The illustration of our prediction methodology. We use Δ to denote the duration of a time window.

5.1 Problem Definition and Methodology Overview

This section specifically presents our prediction methodology for the failures and the feature selection method for the methodology. Section 5.1.1 defines what to predict, and Section 5.1.2 establishes how to select the feature set. Generally speaking, features that are selected and derived from the original RAS logs serve as a key to distinguish between multiple interested classes. In our studies, we have FATAL and NONFATAL classes. A FATAL class indicates that within an interval of time, there contains at lest one FATAL event. On the other hand, a NONFATAL class label is assigned to a time interval from there no FATAL event is found. An ideal feature set can provide a clear cut between classes and completely distinguishes them. Nevertheless, a mal-defined feature set conveys ambiguities and makes class prediction impossible. For instance, warm blood is an excellent feature selected to distinguish between mammal and reptile. However, it is not a sufficient feature to classify mammal and bird for both of them are warm blood. As a result, feature selection is essential to classification work, and once a feature set is selected, the predictability of a classifier built on it is about to be determined. As what to predict and how to predict have been determined, the rest of the work is to choose a better classification tool (classifier). Yet, feature selection is a very difficult work and an ideal feature set may not exist to a given data set. Therefore, in the real world data set, most of the time, feature selection is a best-effort work, and after a long term research, our feature set has been evolved and improved the prediction accuracy to 0.7039 F-measure for BlueGene/L RAS events.

5.1.1 Prediction Methodology

The first question that arises when designing a predictor is what information should the predictor predict. For example, several related studies tried to estimate the mean time between failures [46], which offers a coarse prediction granularity, while on the other extreme of the spectrum, one could attempt to predict at what time the next failure will occur. There are problems with both of these two approaches. The mean time between failures, though easy to estimate, cannot be used to develop remedial actions to mitigate the impact of fatal events. On the other hand, it is well accepted that predicting the exact occurrence time for each failure is almost impossible. Further complicating the problem is the large number of RAS events contained in our data sets, which makes it extremely hard to build such a fine-granularity model. In order to keep the prediction both feasible and useful, we specify our prediction objective between these two extremes – we break the time axis into fixed windows, each with duration Δ, and the goal of the predictor is *to predict whether a failure will occur in the next window or not*.

After determining what to predict, the next question is how to predict, i.e. based on what information the predictor should make the decision. Our earlier investigation has revealed that the runtime health of a parallel system can be correlated to several internal and external events, which is somewhat analogous to human health [34]. Just as there could be symptoms to the onset of a disease, there could be pre-cursor events leading up to a failure. Consequently, we take the viewpoint that there might be certain event patterns prior to the occurrence of fatal events. For examples, certain warning messages might be observed before a fatal event occurs; a large burst of warning messages might be recorded before a fatal event occurs; or, if the period during which there are no fatal events is long enough, then the likelihood of a fatal event occurring in the near future may increase. As a result, in this study, in order to predict whether a failure will occur within a time window, we need to look at the characteristics of the events within the previous time windows.

Figure 5.1 illustrates the basic idea of our prediction methodology. As shown in the figure, we seek to predict whether there will be fatal events in the next Δ interval, which we call *prediction* window, based on the events in the *observation period* (with duration $T = 4\Delta$ in this example), which usually consists of several windows. Though we consider the event characteristics during the entire observation period, we consider events in the current window (the one immediately preceding the prediction window) more important. In this thesis, we attempt to forecast whether fatal events will occur in the prediction window; we classify the prediction window as FATAL if fatal events are predicted to occur during the period, and NONFATAL if no fatal event is predicted.

The prediction window size, Δ, is the most important parameter for our predictor. It directly impacts the predictability of the system as well as the utility of the prediction results. For example, for a very small Δ value, say 15 minutes, it may be easy to predict whether a failure will occur in the next window (based on some precursor events), but a large system such as BlueGene/L cannot take any remedial action within such a short time frame. As in the other extreme, a very large prediction window, say 1 week, may also make prediction easy (there will definitely be a failure every week), but a prediction at such a coarse granularity is not very helpful. In the rest of this thesis, we adopt four prediction window sizes, 1, 4, 6, and 12 hours.

5.1.2 Building the Feature Library

Carefully selected features, either directly extracted from the raw data sets or indirectly derived from the data sets, serve as the key to successfully distinguishing multiple classes. The prediction power of a predictor is largely dependent on the quality of its feature set. In our study, we attempt to label each time window as either FATAL or NONFATAL. A well defined feature set can describe each class well, and thus provide a clear cut between these two classes, while a poorly defined feature set can make classification almost impossible. We note that, feature selection is not only important, but also very challenging, mainly due to the inherent difficulty of the problem as well as the large volume of the data sets. To tackle this challenge, we first rigorously study the characteristics of the failure events (see [34]), and then select the features accordingly.

In order to forecast whether the next time window will encounter fatal events, we identify the following features from those events that occurred during the observation period:

1. The first group of features represent the number of events (at all 6 severity levels) that occurred during the current time window. Specifically, we have *InfoNum*, *WarningNum*,

SvrNum, *ErrNum*, *FlrNum*, and *FatNum*. For example, *FatNum* is used to signify how many fatal events occurred during the current time window. In total, we have 6 features in this group.

2. The second group of features represent the number of events (at all 6 severity levels) accumulated over the entire observation period. Specifically, we have *AcmInfoNum*, *AcmWarningNum*, *AcmSvrNum*, *AcmErrNum*, *AcmFlrNum*, and *AcmFatNum*. For example, *AcmFatNum* is used to denote the number of fatal events accumulated over the entire observation period. In total, we have 6 features in this group.

3. The third group of features describe how the events (at all 6 severity levels) are distributed over the observation period. To this end, we break down a time window (with duration Δ) into a number of smaller sampling intervals (with duration δ), and count the number of events in each sampling interval over the entire observation period. For instance, if we have $\Delta = 1$ hour, $\delta = 10$ minutes, and the observation period of 5Δ, then we will have 30 sampling intervals in total, and therefore, 30 samples for each severity level. Each of these features is named to include the following three parts: (1) its severity level, (2) the keyword "Sample_", and (3) the index of the corresponding sampling interval. For example, *FatalSample_23* signifies the number of FATAL events in the $23rd$ sampling interval. In addition to the individual sample values, we also report the mean and variance of the samples for each severity level, and name these features as *InfoSampleMean*, *InfoSampleVariance*, etc. As a result, the total number of features we have in this group is $6(\frac{n\Delta}{\delta} + 2)$.

4. The fourth group of features are used to characterize the inter-failure times. There is only one feature that belongs to this group, i.e. the elapsed intervals since last FATAL interval, which we call *ItvNum2Fatal*.

5. The last group of features describe how many times each entry data phrase occurred during the current period. Entry data phrases are entry data keywords after we take out numbers, punctuation, file names, directory names, etc. from the entry data. The detailed method of extracting entry data phrases from entry data can be found in our earlier work [34]. In our study, there are totally 529 phrases from 1,792,598 RAS events.

Therefore, we have 529 features in this group, each corresponding to one phrase, and these features are named in the format of *Phrase_245*, where 245 is the phrase ID.

After we establish the above raw feature sets, we next need to preprocess these features to make them suitable for the later prediction study. Feature preprocessing consists of two steps: first normalizing the numeric feature values and then calculating each feature's significance major. For a numeric feature value v that belongs to feature V, we calculate the normalized value as follows:

$$v' = \frac{v - median(V)}{std(V)}. \tag{5.1}$$

After normalizing each feature value, we next calculate feature V's significance major $SIM(V)$ as follows:

$$SIM(V) = |\frac{mean(V_F) - mean(V_{NF})}{std(V)}|. \tag{5.2}$$

Here, we partition all the values of feature V into two parts: V_F containing the feature values of the FATAL class (i.e. these features collected from an observation period followed by a FATAL window), and V_{NF} containing the feature values of the NONFATAL class (i.e. these features collected from an observation period followed by a NONFATAL window). A feature will have a high significance major if its values differ significantly between FATAL records and NON-FATAL records. In the prediction study, we sometimes only use features whose significance index values are above a certain threshold (T_{SIM}).

After calculating the significance major values for all the features, we found that only a few features have non-negligible significance major values. For example, under the setting with $\Delta = 4$ hours, $\delta = 30$ minutes, $T = 28$ hours, only 7 features (out of 890 raw features) have a significance major higher than $T_{SIM} = 2.5$. These seven features are *FatalSample_51*, *FatalSampleMean*, *FatalSampleVariance*, *CurrFatalNumber*, *AcmFatalNum*, *itvNum2Fatal*, and *Phrase_322* (the 322nd phrase is "Lustre mount Fail"). These 7 features suggest that statistics about FATAL events are more important than those of other events.

After building the feature library using the above steps, we next feed these features into prediction tools to study the predictability of BlueGene/L RAS events.

5.2 Prediction Techniques

In this study, we applied three well-known data mining algorithms to construct a rule-based, a hyperplane-based, and a similarity-based online predictors. These three algorithms are RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [21], SVM (Support Vector Machine) [14], and Nearest Neighbor classifier. We exercised the predictors on the BlueGene/L RAS event logs, with the intention of predicting whether the next window will witness fatal events based on event characteristics during the previous time windows. For the first two predictors, we directly feed the feature sets to the off-the-shelf software tools and collect prediction results, while for the third technique, we devised the prediction algorithm to form a Bi-Modal Nearest Neighbor predictor that directly extracts the feature sets from the RAS event flow and makes prediction in real-time.

5.2.1 RIPPER

Our understanding of rule-based data mining classification algorithms guides us to choose RIPPER (as a representative to the rule-based classifiers) for our rule-base online predictor. We chose RIPPER, because (a) the data is very difficult to be separated linearly, (b) most of the feature attributes are continuous, and (c) the data has unbalanced class distribution (RIPPER has been shown able to predict rare classes well [31]). Ripper can handle all of these properties quite well. Furthermore, it produces a relatively easily interpretable model in the form of rules allowing us to pinpoint key features and feature correlations leading to failures, and to assess whether the model reflects reality well or if it is merely coincidental. An additional benefit is that classification is computationally inexpensive in general [20], and RIPPER is a relatively fast classification algorithm [21]. The drawback of RIPPER is its greedy optimization algorithm and its tendency to over fit the training data at times. This drawback did not set us back too much for the problem we encountered the most is not over fitting but few or no rule is found from training data. Besides, over fitting problem can be easily located by doing cross validation on data.

When using RIPPER, we first feed the feature values as described in Section 5.1.2 to RIPPER as input, and RIPPER will output a set of classification rules. Some example rules are

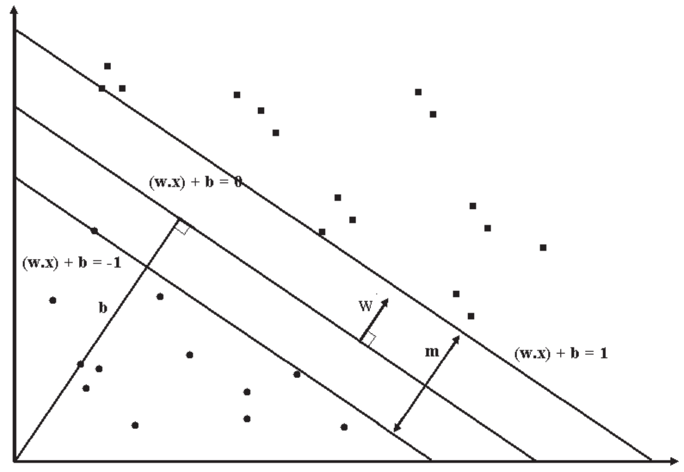


Figure 5.2: The illustration of our predictor. We use Δ to denote the duration of a time window.

provided below:

1. FATAL :- Phrase_216>=97, InfoSample_55<=22 (46/9)
2. FATAL :- Phrase_213>=3, InfoNum<=1669, InfoSample_98<=1 (37/20)
3. FATAL :- InfoSample_105>=34 (12/6)

Finally, rules induced from the training data will be applied on test data to make prediction.

5.2.2 Support Vector Machines

The second classifier we employed in this study is Support Vector Machines (SVMs), which are a set of generalized linear classifiers. The primary advantage of SVMs is their fast execution speed in prediction (after pass the training stage). Also, in many real-word cases [23], SVMs have best classification performance. A drawback of SVMs is that the training cost can be expensive if there are a large number of training samples. For example, in our study, it usually took 10 hours to run the training data that contain 2890 records each with 986 features. In these cases, execution can become slow, especially for SVMs with a nonlinear kernel.

In this study, we chose LIBSVM [18] with the Radial Basis Function (RBF) kernel.

SVM is a hyperplane classifier [14]. A function

$$f : \mathcal{R}^N \longrightarrow \{\pm 1\} \tag{5.3}$$

is estimated and obtained from training data. Each data point consists of an N-dimensional vector **x**, and an associated class label y. As indicated in Figure 5.2, assuming two linearly separable classes, we can have a hyperplane,$(\mathbf{w} \cdot \mathbf{x}) + b = 0$, which separates the circle class and the square class. **w** is a vector perpendicular to the hyperplane. By properly choosing the vector **w** and scalar b, the biggest margin between these two classes is defined by the two lines, $(\mathbf{w} \cdot \mathbf{x}) + b = 1$ and $(\mathbf{w} \cdot \mathbf{x}) + b = -1$. Apparently, the distance between the two lines (the margin width) can be calculated by the following formula.

$$m = \|\lambda \mathbf{w}\| = \frac{2}{\mathbf{w} \cdot \mathbf{w}} \sqrt{\mathbf{w} \cdot \mathbf{w}} = \frac{2}{\sqrt{\mathbf{w} \cdot \mathbf{w}}} \tag{5.4}$$

Practically, the hyperplane is built based on the training data, and serves as a prediction boundary to the test data.

Given an **x** from the test data, classification by SVM involves calculating $f(\mathbf{x})$ and comparing its value with zero.

$$f(\mathbf{x}) = (\mathbf{w} \cdot \mathbf{x}) + b \tag{5.5}$$

If $f(\mathbf{x})$ is larger than zero, then it is classified as a square class; otherwise, it is classified as a circle.

In cases where the two classes are not linearly separable, we need to introduce a function ϕ to transform **x** to a higher dimensional space. In this way, by solving the following optimization problem, the two classes are linearly separable with a reasonable error and with a maximal margin in this higher dimensional space [14], [22].

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^T\mathbf{w} + \mathcal{C}\sum_{i=1}^{l} \xi_i, \tag{5.6}$$

subject to

$$y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \tag{5.7}$$

where $\xi_i \geq 0$ denotes the error term, $\mathcal{C} > 0$ denotes the penalty of the error term, $i = 1 \ldots l$ denotes the sequence of training data points, and the kernel function is defined by $K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$. The kernel function used in this thesis is Radial Basis Function (RBF) which is suggested by LIBSVM [18].

$$K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma \parallel \mathbf{x}_i - \mathbf{x}_j \parallel^2), \gamma > 0. \tag{5.8}$$

```
Itv12HrAcm3Smp20Min
Accuracy = 51.2195% (21/41) (classification)
Mean squared error = 1.95122 (regression)
Squared correlation coefficient = 0.0137063 (regression)
Scaling training data...
Cross validation...
Best c=512.0, g=3.0517578125e-05 CV rate=65.9574
Training...
Output model: Itv12HrAcm3Smp20Min.data.model
Scaling testing data...
Testing...
Output prediction: Itv12HrAcm3Smp20Min.test.predict
```

Table 5.1: A LIBSVM command-line output example.

LIBSVM [18] is the top one SVM tool recommended by an SVM organization [8]. It defines the following steps to obtain an optimal solution for Equation 5.6 and 5.7.

1. To prepare data to be SVM-compliant;
2. To conduct simple scaling on data;
3. To consider the RBF kernel,
   $K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma \parallel \mathbf{x}_i - \mathbf{x}_j \parallel^2)$;
4. To find the optimal parameter $\mathcal{C}$ and γ using cross-validation, and train the training set using these parameters;
5. To test on test data.

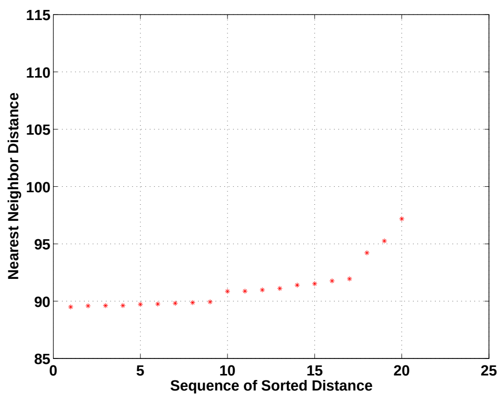
By following the recommended procedures calling the scaling, griding, training, prediction functions with the automated python script, easy.py, provided by LIBSVM, we collected a LIBSVM command-line outputs as an example in Table 5.1.

After following the above steps, LIBSVM classifies the test data, and notifies the user whether the classification is a hit or a miss. Based on that, we calculate the F-measure, precision, and recall accordingly.

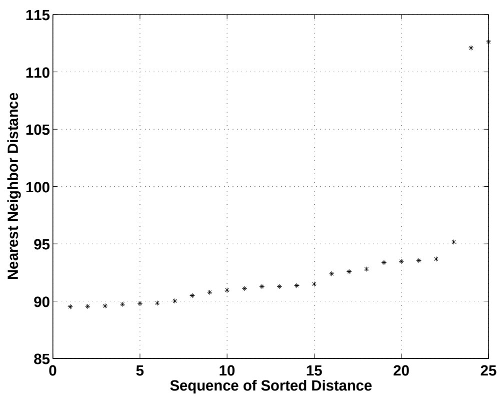
5.2.3 A Bi-Modal Nearest Neighbor Predictor

In addition to using off-the-shelf prediction tools, we also designed a nearest neighbor predictor.

For our nearest-neighbor prediction, we partition the data sets into three parts: *anchor* data, *training* data and *test* data. From the anchor data, we identify all the FATAL intervals in



(a) FATAL distance map



(b) NONFATAL distance map

Figure 5.3: The distance maps with $\Delta = 12$ hours, $T = 36$ hours and no sampling.

d_1/d_2	f_{FF}	f_{FN}	f_{NF}	f_{NN}	p (%)	r (%)	F (%)
89.8/91.4	13	15	7	10	46.43	65.0	54.17
89.9/91.4	15	17	5	8	46.88	75.0	57.69
90.0/91.4	16	17	4	8	48.48	80.0	60.38
90.1/91.4	16	18	4	7	47.06	80.0	59.26
90.0/91.5	15	16	5	9	48.39	75	58.82

Table 5.2: Prediction results using different (d_1, d_2) values on the training data in Figure 5.3.

which at least one FATAL event occurred, and compute the feature values in the corresponding (preceding) observation periods. These feature values serve as a base for distance computation, and are organized into a two-dimensional matrix where rows correspond to each FATAL interval and columns correspond to each feature. We call this matrix the *anchor matrix*, and each row of this matrix a *feature vector*.

After establishing the anchor matrix from the anchor data, we next process all the FATAL features from the training data, and calculate the nearest distance between each FATAL feature vector and all the rows in the anchor matrix. The resulting distance values form a FATAL nearest distance map. Similarly, we can also prepare the NONFATAL feature vectors and thus the NONFATAL nearest distance map. Examining these two maps carefully, we can characterize the properties of the nearest neighbor distributions for both FATAL and NONFATAL classes, and based on these properties (which hopefully are distinct), we can make prediction on the test data.

In order to understand the working of the nearest neighbor predictor, let us look at an example first. Figures 5.3 (a) and (b) show the FATAL and NONFATAL nearest neighbor maps when we have $\Delta = 12$ hours, $T = 36$ hours, and no sampling. In both figures, we sort all the distances and plot them in the increasing order. From Figure 5.3, our first observation is that the distance distributions from both classes look rather alike. For example, in both classes, most of the distances are between 89.5 and 94, i.e. 16 out of 17 FATAL distances in this range, and 23 out of 27 NONFATAL distances in this range. This similarity makes it impossible to adopt a single threshold to differentiate these two classes. Fortunately, after examining the figures more carefully, we find a sparse region in FATAL distance distribution (Figure 5.3(a)), i.e. [90, 91], and meanwhile, many NONFATAL distances are within this range. This observation suggests that we can employ two distance thresholds (d_1 and d_2 with $d_1 < d_2$), and adopt the following prediction rule:

$$r_{nnp} : ((distance < d_1) \vee (distance > d_2)) \longrightarrow FATAL.$$

We also observed the above trend in other parameter settings, and therefore, our nearest neighbor predictor adopts this simple prediction rule across all the settings. Since this nearest neighbor predictor utilizes two distance thresholds, we refer to it as *bi-modal nearest neighbor* predictor (BMNN, in short). Traditional nearest neighbor prediction, on the other hand, only uses the lower bound threshold, and it classifies an event as FATAL as long as the nearest distance is smaller than the threshold.

The performance of BMNN is largely dependent on the values of d_1 and d_2. Here, we use the training data to learn the appropriate values for these two thresholds; for a given setting, we vary their values, and choose the pair which leads to the best prediction results for the training data. Later on, we apply the same threshold values learnt from the training data to the test data. For the example shown in Figure 5.3, Table 5.2 summarizes the prediction results with a range of (d_1, d_2) values. Based on the results shown in Table 5.2, we choose $d_1 = 90.0$ and $d_2 = 91.4$, as they lead to the highest F-measure (60%), and more importantly, the highest recall value (80%).

5.2.4 Bi-Modal Nearest Neighbor Algorithm Details

The Bi-Modal Nearest Neighbor real-time failure predictor created in this dissertation is an instance of nearest neighbor classifier. It measures how close and how far apart the distance between current event statistics and the fatal event statistics that have occurred before. The prediction is based on the distance measurement, and the strategy is to mark the unusual (extremely close or extremely far away) events as FATAL. If the distance is less than a lower bound threshold or is larger than an upper bound threshold, a FATAL prediction is made for the following time interval. The lower bound and upper bound thresholds are obtained from the training data.

In order to utilize the nearest neighbor classification technique, several off-line studies are required. They are feature normalization, feature significance selection, and training data FATAL interval collection. First, the feature median and feature standard deviation need to be determined in order to normalize a numeric feature value. Based on the following Equation 5.9 the normalized value is calculated.

$$NormalizedFeatureValue = \frac{FeatureValue - FeatureMedian}{FeatureStandardDeviation} \tag{5.9}$$

Second, the feature significance to the nearest neighbor classification is measured by the following formula, Equation 5.10.

$$FeatureSignificance = \left| \frac{FatalClassFeatureMean - NonfatalClassFeatureMean}{FeatureDomainStandardDeviatioin} \right| \tag{5.10}$$

The algorithm memorizes features that have feature significance bigger than a significance threshold, and passes the feature indices to the Bi-Modal Nearest Neighbor failure predictor.

Third, fingerprints of event statistics in the training data that lead to FATAL failures are collected for the Bi-Modal Nearest Neighbor failure predictor to make a basis for prediction.

As the BMNN real-time predictor starts to work, it looks for a number of input parameters and files on the command line. Table 5.3 lists the input parameters and their descriptions.

Basically, Bi-Modal Nearest Neighbor failure predictor is a self-learning real-time predictor comprises an event bucket, an event sample bucket, an ItvNum2Fatal counter, a state printing unit, correctness checking unit, a learning unit, and a prediction unit. The details of the predictor is presented in the following subsections, and since the design of the two buckets plays an essential role to the predictor, they are introduced first.

Input Parameters	Description
Itv	prediction interval
AcmItvNum	accumulate interval number
SampleItv	sample interval
HowClose	indicating how close the distance that the nearest neighbor predictor will choose, 0 indicates the nearest neighbor, 1 indicates the second nearest neighbor, 2 the third nearest neighbor, and etc..
FatalEventFingerprints	feature sets leading to FATAL failures extracted from a training data set
SignificantFeatureIdx	a file containing indices of features obtained from the significance measurement program
LwBoundThreshold	the distance lower bound threshold
UpBoundThreshold	the distance upper bound threshold
EntryDataPhrases	a file containing all entry data phrases
OutputFileName	an output file containing prediction results
TestingLogs	the testing RAS event logs

Table 5.3: A list of input parameters to the Bi-Modal Nearest Neighbor failure predictor.

The Event Buckets

According to the input parameters, Itv and SampleItv, two logical buckets, event bucket and event sample bucket, are created and initiated. The purpose of the event bucket is to hold the statistics of events occurring in the time interval defined by Itv. Meanwhile, the event sample bucket holds the statistics of the events occurring in the time interval defined by SampleItv. As the first event coming in to the predictor, the time stamp of it is extracted and marked as bucket bottoms to the event bucket and the event sample bucket. The top of the event bucket is then defined by the sum of its bottom plus Itv, and the top of the event sample bucket is then defined by the sum of its bottom plus SampleItv. The event statistics defined in the event bucket are the number of INFO events, the number of WARNING events, the number of SEVERE events, the number of ERROR events, the number of FAILURE events, the number of FATAL events, and the number of each entry data phrase occurred in the Itv time interval. The event statistics defined in the event sample bucket are the numbers of the 6 severity levels (addressed above) of events occurred in the SampleItv time interval. The event bucket statistics stored in the event bucket are poured out to an event state array (EvtStateAry) and to 6 event accumulate arrays (that associated to the six severity levels) when the times tamp of an incoming event is bigger than the top of the event bucket. As this happens, the following actions are taken.

- *EventBucketBottom=EventBucketTop;*
- *EventBucketTop=EventBucketBottom+Itv;*

The length of the 6 event accumulate arrays is defined by the input parameter AcmItvNum and they are used to store the event statistics at the 6 severity levels. As the 6 event accumulate arrays are full, the oldest event statistics are retired.

A similar rule is applied to the event sample bucket top, bottom adjustment and event statistics retirement. As a time stamp of an incoming event is bigger than the time marked at the top of the event sample bucket, the event statistics stored in the event sample bucket is pushed into 6 event sample arrays that are holding the event statistics distributions of the sex severity levels of events. Then, the top and the bottom of the event sample bucket are adjusted accordingly.

- *EventSampleBucketBottom=EventSampleBucketTop;*
- *EventSampleBucketTop=EventSampleBucketBottom+SampleItv;*

The length of the 6 arrays is the number of samples during the time interval of Itv times AcmItvNum divided by SampleItv. It is expressed in a formula as follows.

$$EventSampleArryLength = SampleNumber = \frac{Itv \times AcmItvNum}{SampleItv} \tag{5.11}$$

It becomes apparent that the sample interval (SampleItv) must be chosen in a way so that SampleNumber is an integer. The six event sample arrays are assigned to store the statistics of the events at the 6 severity levels, and the event statistics are put in time order in the arrays. As the 6 arrays are full, the oldest event statistics are retired. The in-order storage and element retirement scheme are implemented by 6 shift registers. New event statistics are put on the right of the shift registers and the shift registers are shifted to the right.

The Algorithm

With the above background details about how event bucket and event sample bucket work with the RAS event flow, the prediction algorithm is ready to be presented. The prediction is triggered by an incoming event with a time stamp bigger than the time stamp marked on the top of the event bucket. As it happens, we call that the event bucket is full. Once the prediction is made, the correctness of the prediction is checked by the occurrence of a FATAL event in the next event bucket. As a result, until the next event bucket is full, the correctness is not checked. Therefore, three major action are taken when the event bucket is full. First, the event statistics along with the event sampling statistics are printed out to the corresponding storage. Second, from the current event bucket statistics, the correctness of the previous prediction is checked. Last, a prediction for the next event bucket is made based on the current bucket statistics. A better way to describe the prediction algorithm is to list the actions it takes when the event bucket is full. The actions are listed as follows.

- Adjusting the interval-number-to-fatal parameter (itvNum2Fat): if the event bucket contains no FATAL event, itvNum2Fat is increased by 1; otherwise, itvNum2Fat is reset to 0.
- Printing state:
  - pushing the event statistics stored in the event sample bucket to the 6 event sample arrays, calculating means and variances of the statistics stored in the arrays, and printing the statistics together with their means and variances to the event state array.
  - printing the event statistics stored in the event bucket to the event state array.
  - pushing the event statistics stored in the event bucket to the 6 event accumulate arrays, summing up the statistics stored in the 6 arrays individually, and printing the 6 accumulate event statistics to the event state array.
  - printing ItvNum2Fat to the event state array.
  - printing the entry data phrase statistics to the event state array.
- According to the HowClose parameter, calculating the Euclidean distance between the event state array and the FATAL class event state arrays that are collected along the progress of RAS event flow and are extracted from the training data. This distance is named event state distance (EvtStateDst).
- Setting the class label (ClassLabel): If the event bucket contains a FATAL event, the class label is set to FATAL; otherwise, the class label is set to NONFATAL.
- Checking the correctness of prediction: (This step is skipped if the loop is first time entered, the predict state (pdcState) is set to UNKNOWN initially)

  - if the predict state (pdcState) is FATAL and ClassLabel is FATAL, then do
    * the PdcFatalActFatal (predict FATAL and actually FATAL) counter plus 1
    * calling SetPdcRule (set prediction rule) subroutine and passing the label “PdcFatalActFatal”, previous event state distance (PrvEvtStateDst), and previous event state array (PrvEvtStateAry) to the subroutine
    * printing PrvEvtStateDst, ClassLabel, and pdcState to the output file
  - else if the pdcState is FATAL and ClassLabel is NONFATAL, then do
    * the PdcFatalActNFatal (predict FATAL and actually NONFATAL) counter plus 1
    * calling SetPdcRule (set prediction rule) subroutine and passing the label “PdcFatalActNFatal”, previous event state distance (PrvEvtStateDst), and previous event state array (PrvEvtStateAry) to the subroutine
    * printing PrvEvtStateDst, ClassLabel, and pdcState to the output file
  - else if the pdcState is NONFATAL and ClassLabel is FATAL, then do
    * the PdcNFatalActFatal (predict NONFATAL and actually FATAL) counter plus 1
    * calling SetPdcRule (set prediction rule) subroutine and passing the label “PdcNFatalActFatal”, previous event state distance (PrvEvtStateDst), and previous event state array (PrvEvtStateAry) to the subroutine
    * printing PrvEvtStateDst, ClassLabel, and pdcState to the output file
  - else if the pdcState is NONFATAL and ClassLabel is NONFATAL, then do
    * the PdcNFatalActNFatal (predict NONFATAL and actually NONFATAL) counter plus 1
    * calling SetPdcRule (set prediction rule) subroutine and passing the label “PdcNFatalActNFatal”, previous event state distance (PrvEvtStateDst), and previous event state array (PrvEvtStateAry) to the subroutine
    * printing PrvEvtStateDst, ClassLabel, and pdcState to the output file

- Calling mkPredict (make prediction) subroutine and passing EvtStateDst and EvtStateAry to the subroutine, so that pdcState is set to FATAL or NONFATAL

- Setting the previous event state distance (PrvEvtStateDst) equals to the event state distance (EvtStateDst) found in the third step
- Setting the previous event state array (PrvEventStateAry) equals to the event state array (EvtStateAry)
- Cleaning up (resetting) EvtStateDst, and EvtStateAry
- Adjusting the limits of the event bucket top and bottom by executing the following instruction:
  - EventBucketBottom=EventBucketTop;
  - EventBucketTop=EventBucketBottom+Itv;

The purpose of SetPdcRule subroutine is to adjust prediction rules used by mkPredict subroutine. This function provides the predictor an opportunity to learn from its prediction experience. Currently, this function is disabled.

The mkPredict subroutine sets pdcState to FATAL or NONFATAL based on EvtStateDst, and the input parameters, LwBoundThreshold and UpBoundThreshold. The prediction rule is that if EvtStateDst is less than LwBoundThreshold or bigger than UpBoundThreshold, then pdcState is set to FATAL; otherwise, pdcState is set to NONFATAL.

The nearest neighbor optimal threshold finder (nearNeiOptTholdFinder) is a program that takes in the output file from the Bi-Modal Nearest Neighbor failure predictor and determines the optimal lower bound threshold (LwBoundThreshold) and optimal upper bound threshold (UpBoundThreshold) for the input file. It is recalled that the first column of the predictor output file is the event state distance (EvtStateDst). The nearNeiOptTholdFinder program first sorts the distance column in increasing order, and then collects middle points between each distance pair. From the midpoints of the distances, each possible combination of lower bound, upper bound threshold pair is tested and a corresponding F-Measure is generated. The optimal LwBoundThreshold, and UpBoundThreshold are determined by a pair of midpoint combination that leads to biggest F-Measure. After the optimal LwBoundThreshold and UpBoundThreshold are determined from the training data, the predictor is activated again with the thresholds and testing data as its inputs.

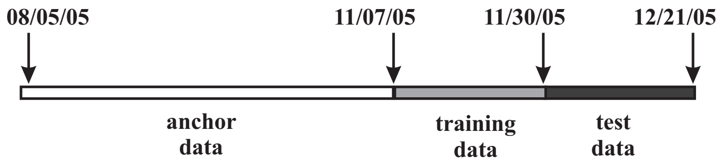


Figure 5.4: How to split the failure logs into anchor data, training data, and test data.

5.3 Experimental Results

In this section, we discuss our evaluation of the predictors and present the detailed prediction results.

5.3.1 Experimental Data Sets

Evaluating the predictors first requires partitioning raw event logs into two parts, training data and test data (in the case of nearest neighbor predictor, three parts: anchor, training and test data). Random sampling [45], a popular technique to extract test data, selects random records from the original data set. Though random sampling is widely used in many domains, it is not suitable for our study. Specifically, since our data have both temporal (the sequence number of an event entry) and spatial (the location of an event entry) dimensions, we can potentially apply random sampling in both of these two dimensions, leading to temporal sampling and spatial sampling. Let us next look at the applicability of these two methods more closely. Temporal sampling is not a viable solution because our predictor relies on the temporal continuity between intervals – we use the features of the previous interval to predict whether there will be failure(s) in the current interval. Temporal sampling will break this continuity. On the other hand, spatial sampling involves partitioning the data spatially, i.e. along the midplane boundary, and selecting events from one or more midplanes as test data. This is not desirable because our earlier studies [34] have shown that different midplanes exhibit distinctively different characteristics. Hence, it is very hard, if at all possible, to predict failures of a midplane using failures on another midplane.

In this study, we choose to horizontally (along the time axis) partition the event logs into large consecutive blocks. For RIPPER and SVMs, we partition the event logs into two consecutive parts which are training and test data respectively, while for BMNN, we partition the

test case	RIPPER & SVM		BMNN		
	training (17weeks)	test (3weeks)	anchor (14weeks)	training (3weeks)	test (3weeks)
T(1)	1-17	18-20	1-14	15-17	18-20
T(2)	1-16,20	17-19	1-13,20	14-16	17-19
T(3)	1-15,19-20	16-18	1-12,19-20	13-15	16-18
T(4)	1-14,18-20	15-17	1-11,18-20	12-14	15-17
T(5)	1-13,17-20	14-16	1-10,17-20	11-13	14-16
T(6)	1-12,16-20	13-15	1-9,16-20	10-12	13-15
T(7)	1-11,15-20	12-14	1-8,15-20	9-11	12-14
T(8)	1-10,14-20	11-13	1-7,14-20	8-10	11-13
T(9)	1-9,13-20	10-12	1-6,13-20	7-9	10-12
T(10)	1-8,12-20	9-11	1-5,12-20	6-8	9-11
T(11)	1-7,11-20	8-10	1-4,11-20	5-7	8-10
T(12)	1-6,10-20	7-9	1-3,10-20	4-6	7-9
T(13)	1-5,9-20	6-8	1-2,9-20	3-5	6-8
T(14)	1-4,8-20	5-7	1,8-20	2-4	5-7
T(15)	1-3,7-20	4-6	7-20	1-3	4-6

Table 5.4: The distribution of the 15 test cases for the predictors.

event logs into three parts. Figure 5.4 illustrates one way of partitioning the event log for the nearest neighbor predictor.

In order to conduct a comprehensive evaluation of the predictors, we need to run them against multiple test cases. In this study, we constructed 15 test cases, $T(1)$, ..., $T(15)$. Our event log consists of failure events collected over a 20-week period (In fact, the log duration is 20 weeks and 2 days, and we merged the last two days into the last week), and our test data always contain events from a three-week period. In the 15 test cases, the test data start from week 18, 17, ..., and week 4 respectively. As far as RIPPER and SVMs are concerned, the data from the remaining 17 weeks are regarded as training data. In the case of BMNN, we need to further split these 17-week data into two parts, events from the immediately preceding 3 weeks are the training data, and the remaining 14-week data are the anchor data. The details about each of the 15 test cases are listed in Table 5.4. Please note that test data are the same for all the classifiers across the 15 test cases.

		Predicted Class	
		FATAL	NONFATAL
Actual Class	FATAL	f_{FF} (TP)	f_{FN} (FN)
	NONFATAL	f_{NF} (FP)	f_{NN} (TN)

Table 5.5: The confusion matrix

Δ (x hour)	**δ (x min)**	**T (x Δ)**	**# of FATAL windows**	**# of NONFATAL windows**
1	10	12	306	3088
4	30	7	240	608
6	60	5	210	355
12	20	3	161	121

Table 5.6: The sampling periods and observation periods for under different prediction window sizes (column 2 and 3), and the total number of FATAL and NONFATAL (column 4 and 5) with different prediction window sizes.

5.3.2 Evaluation Methodology

Since detecting FATAL intervals is more important than detecting NONFATAL intervals, we use *recall* and *precision* to measure the effectiveness of our predictors. Using the confusion matrix as shown in Table 5.5, we define these two metrics as below.

$$Precision, p = \frac{f_{FF}}{f_{FF} + f_{NF}} \tag{5.12}$$

$$Recall, r = \frac{f_{FF}}{f_{FF} + f_{FN}} \tag{5.13}$$

A predictor with a high precision commits fewer false positive errors, while a predictor with a high recall commits fewer false negative errors. Finally, we also use a combined metric, F measure.

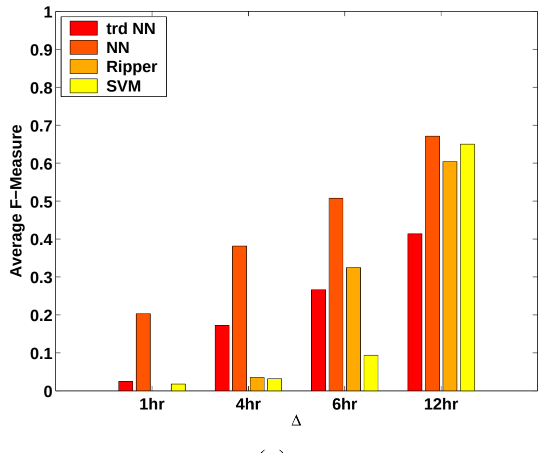
$$F = \frac{2rp}{r + p} = \frac{2f_{FF}}{2f_{FF} + f_{NF} + f_{FN}} \tag{5.14}$$

A high F-measure ensures that both precision and recall are reasonably high.

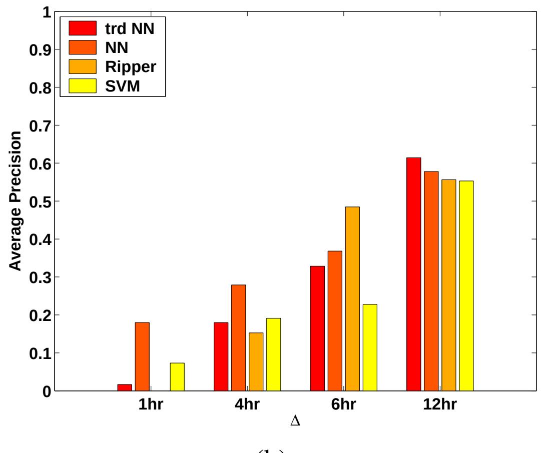
5.3.3 Comparison of the Predictors

In the first set of experiments, we compare the performance of the following predictors: RIPPER, SVM, BMNN, and traditional nearest neighbor predictor. We have run the four predictors over all 15 test cases that are listed in Table 5.4, and we report the average F-measure (across all 15 test cases), the average precision, and the average recall in Figure 5.5.

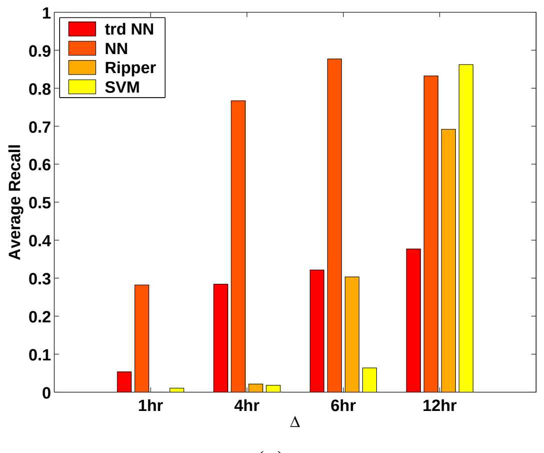
Regardless of which predictor we use, they all take feature data derived from raw logs as input. Nonetheless, we can render different versions of feature data by tuning several parameters, i.e. the significance major threshold T_{sim}, the sampling period δ, and the observation period T. We found that both RIPPER and SVMs prefer directly using raw feature data without applying



(a)



(b)

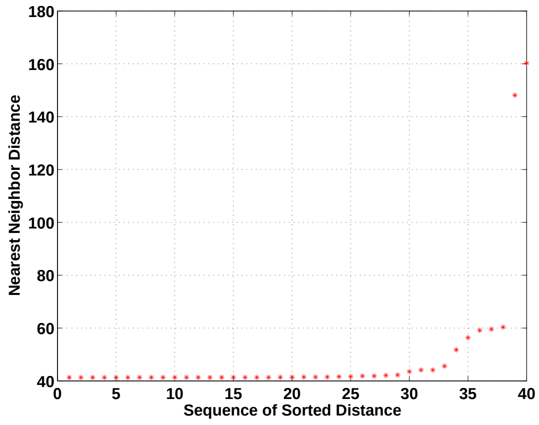


(c)

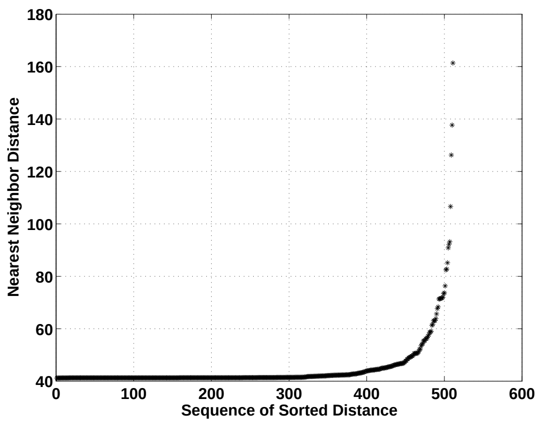
Figure 5.5: (a) The average F-measure, (b) the average precision, and (c) the average recall with varying prediction window size.

any significance major filtering, i.e. $T_{sim} = 0$. On the other hand, we had $T_{sim} = .25$ for nearest neighbor predictors. Incidentally, the same value was also recommended in [45]. As far as the sampling period and the observation period are concerned, we employed the same values for all four predictors, and their values are summarized in Table 5.6 (the second and third columns).

From the results in Figure 5.5, our first observation is that the prediction window size Δ has a substantial impact on the prediction accuracy. With $\Delta = 12$ hours, RIPPER, SVM, and BMNN can perform reasonably well: F-measure higher than 60%, precision higher than 50%, and recall higher than 70%. As prediction window becomes smaller, the prediction difficulty rapidly increases, leading to much degraded performance, especially for RIPPER and SVMs. The F-measure for these two predictors is less than 5% when Δ is 4 hours or shorter (0 in



(a) FATAL distance map



(b) NONFATAL distance map

Figure 5.6: The distance maps with $\Delta = 1$ hour, $T = 12$ hours, and no sampling.

some cases), which is hardly of any practical use for our application. The traditional nearest neighbor also fares poorly. Fortunately, our nearest neighbor predictor, BMNN, managed to sustain a much slower degradation. Even with $\Delta = 1$ hour, its F-measure is above 20%, and its recall is as high as 30%. This result is not surprising because our prediction scenario becomes closer to a real-time prediction scenario as prediction window decreases. It is also analogous to the target-shooting scenario in real life where a smaller target is much harder to target at (which corresponds to a smaller prediction window in our case).

The impact of window size on the prediction accuracy can be partially explained by the statistics shown in Table 5.6, i.e. the total number of FATAL windows (the fourth column) and NONFATAL windows (the fifth column) in the data set. With $\Delta = 12$ hours, the ratio of FATAL to NONFATAL intervals is 1.33, while with $\Delta = 1$ hour, this ratio becomes 0.099. It is always easier to classify two balanced classes.

From Figure 5.5, we also observe that the predictors usually result in a higher recall than precision. This is particularly important in the domain of failure prediction because missing a failure has a more serious outcome than over-predicting.

Discussion: It is well known that rare class classification is a challenging problem. Here, we would like to reiterate this point, in the domain of failure prediction, using some detailed statistics we have collected in this study.

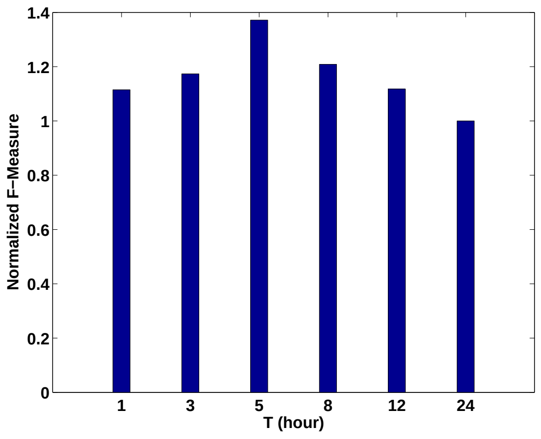
First, RIPPER, which is known to be good at rare class analysis, performs poorly with smaller prediction windows. When we have $\Delta = 1$ hour, RIPPER failed to extract any hypothesis for 2 test cases, resulting in an F-measure of 0 in these cases. Even for the other 13 test cases for which RIPPER extracted hypotheses, these hypotheses were rather random and of little help in prediction. These hypotheses at best led to an F-measure of 15% on the training data! On the test data, across all the 15 test cases, RIPPER's true positive number was 0, leading to an F-measure of 0 for all these cases. Second, SVM only performs marginally better than RIPPER for short prediction windows, e.g. its $F-$measure is less than 2%. Among all 15 test cases, SVM has zero true positive number for 11 cases, and for the other 4 cases, the true positive numbers are 4 (our of 61 failures), 1 (out of 43 failures), 1 (out of 31 failures), and 1 (out of 28 failures) respectively.

Finally, BMNN experienced the same level of difficulty. Figure 5.6 shows the FATAL and NONFATAL nearest distance maps with $\Delta = 1$ hour, $T = 12$ hours, and no sampling. From the figure, it is clear that it is very difficult to differentiate these two maps from each other. First of all, we observe that most of the FATAL distances are either included in the NONFATAL distances, or very similar to some of NONFATAL distance values. Even though there is a sparse region in the FATAL distance map, [60, 148], the density of the NONFATAL distances in that range is also very low, i.e. 27 out of 511 NONFATAL distances are between 60 and 148. Such a distribution makes it hard to set thresholds d_1 and d_2.

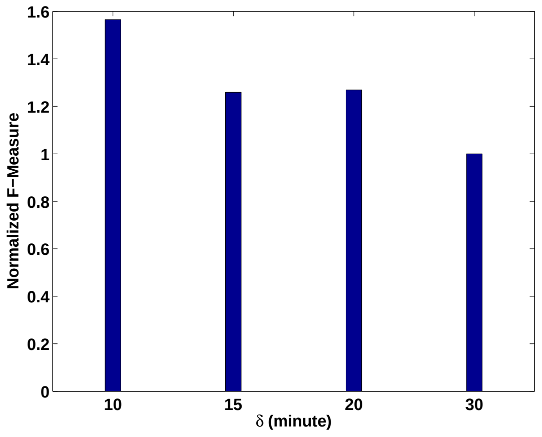
5.3.4 A Closer Look at BMNN

Figure 5.5 shows that our nearest neighbor classifier greatly outperforms the other classifiers, especially for smaller prediction window sizes. In this section, we perform an in-depth study of BMNN, to understand how to tune its parameters for better results.

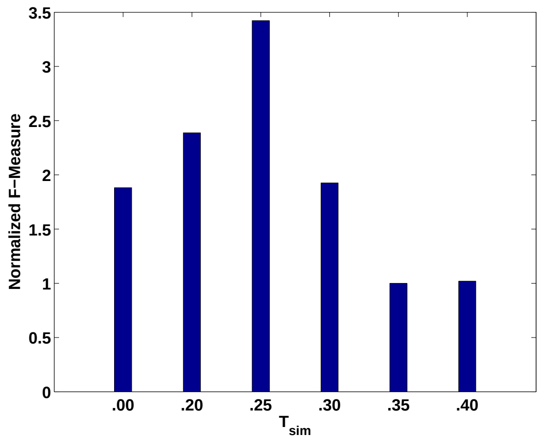
The Impact of Observation Period (T): First, let us look at the impact of the observation period. Usually, the observation period is a multiple of the prediction window, i.e. $T = k\Delta$, and by varying the value of k, we can study how BMNN fares with different observation periods. Figure 5.7(a) shows the F-measure of our nearest neighbor predictor when we varied T as 1, 3, 5, 8, 12, 24 hours, while keeping $\Delta = 1$ hour, $\delta = 10$ minutes, and $T_{sim} = .25$. In order to accentuate the performance difference with varying observation period, we normalized the



(a)



(b)



(c)

Figure 5.7: The impact of parameter values on the performance BMNN: (a) observation period (T), (b) sampling period (δ), and (c) significance major threshold (T_sim). In all three plots, we have $\Delta = 1$ hour. In all three plots, we report the normalized F-measure, by dividing the actual F-measure in each case with the minimum F-measure value in all the cases.

F-measure values with respect to the worst case.

As we expected, looking at a longer period before the prediction window is better than just looking at the previous interval alone, especially for a small prediction window size such as 1 hour. A more interesting observation is that, when the observation period is too long, the prediction accuracy actually degrades. For example, an observation period of 12 hours is worse than just 1 hour. This can be explained by the fact that looking at too long a period may make the most influential pre-cursor events buried in the large number of events during that period.

In this particular setting, an observation period of 5 hours is better than others. However, we would like to note that the purpose of this study is not to identify the best observation period duration, but to reveal the trend – there is a "sweet spot" for the observation period. Our experiments show that this sweet spot varies with Δ.

The Impact of Sampling Interval (δ): Next, we look at the impact of sampling intervals. We have conducted a set of experiments to study how our nearest neighbor predictor performs with different sampling periods, and show the results in Figure 5.7(b). In this set of experiments, we varied δ as 10, 15 20, 30 minutes, while keeping $\Delta = 1$ hour, $T = 12$ hours, and $T_{sim} = .25$. The results show that for a given observation period, a smaller sampling interval is always advantageous. This is because for a given observation period, the duration of the events is fixed, and more frequent sampling can provide us a more detailed picture about the event patterns. As a result, a shorter sampling interval is always preferred.

The Impact of Significance Major Threshold (T_{sim}): The significance major threshold also plays an important role in determining the prediction accuracy. Figure 5.7(c) presents the normalized F-measure values of BMNN when we varied T_{sim} as 0.0, .20, .25, .30, .40, while keeping $\Delta = 1$ hour, $\delta = 10$ minutes, and $T = 12$ hours. The case with $T_{sim} = 0$ corresponds to the situation where we use the raw feature set without filtering them using a significance major threshold. A higher T_{sim} value will filter out more features, leaving fewer features that are used for prediction.

Comparing Figures 5.7 (a) and (c), we find that significance major threshold has a larger impact than observation period does: the ratio between the best and the worst F-measure can be as high as 3.5. Just as the way observation period impacts F-measure, F-measure first increases with significance major threshold, but after a certain point, it starts to drop. This is because prediction based on either too many features or too few features is not as accurate as prediction based the right amount of features. In this set of experiment, F-measure is the highest when T_{sim} = .25, and we also observed that this threshold value works the best for other Δ values as well.

Which Nearest Neighbor to Use? One potential concern with the nearest neighbor predictor is that it is not robust against noise. Therefore, instead of always picking the nearest neighbor, a common practice is to employ the i-th nearest neighbor where $i > 1$, or to employ the median of the distances.

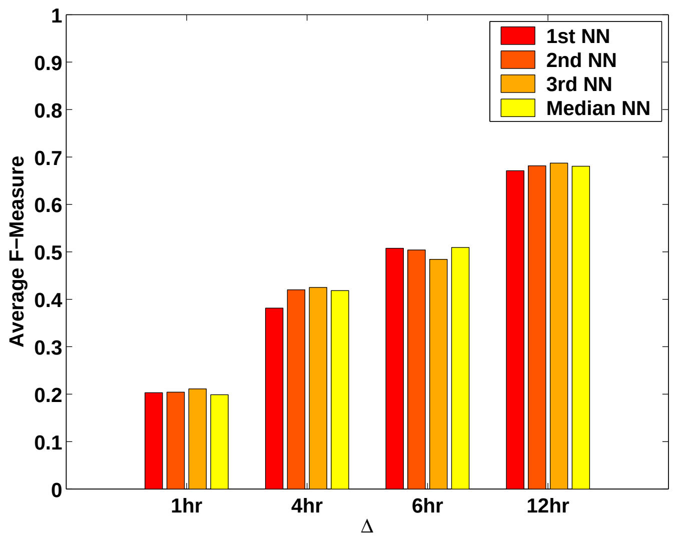


Figure 5.8: Average F-measure of BMNN using the first, second, and third nearest neighbor, as well as the median distance.

We conducted a set of experiments to look at whether choosing a different nearest neighbor can noticeably affect the performance. We report the average $F-$measure across 15 test cases in Figure 5.8 for the following cases: (1) choosing the nearest neighbor, (2) choosing the second nearest neighbor, (3) choosing the third nearest neighbor, and (4) choosing the median of the distances. The results show that the difference between these four alternatives is negligible – the maximum F-measure difference in Figure 5.8 is only 4.36%. This actually suggests that in our nearest neighbor predictor, choosing which neighbor is not an important issue, which can greatly simplify the predictor configuration.

5.3.5 Implications of the Prediction Results on Fault Tolerance

After presenting the detailed performance results, we next discuss the implications of these results on runtime fault-tolerance of IBM BlueGene/L. The first lesson learnt from this study is that a prediction window from 12 to 6 hours balances the prediction accuracy (above 50%) and prediction utility the best. A prediction window of 6 or 12 hours may appear too long to be useful for small systems that run short-running jobs, but we believe they are quite proper for systems like BlueGene/L. Jobs that are designed for BlueGene/L usually run for days, or even months, and more importantly, checkpointing a job on the system takes 30 minutes. Next, let us look at one example that illustrates how the prediction can help a system's runtime fault-tolerance. Let us suppose that the prediction window is 6 hours, and that we predict the next window will have failures. Then the system can initiate a checkpointing immediately, and the likelihood of checkpointing completing before the failure will be reasonably high. The second lesson is that a high recall value is more important than a high precision value. This is because even if we over-predict, the system can still determine whether a checkpointing is necessary based on the amount and the importance of the work completed since last checkpointing. On the other hand, the outcome of under-predicting is much more serious.

5.4 Related Work

In this section, we broadly classify the related work in two categories: (1) failure prediction in computer systems, and (2) rare class classification.

Predicting failures in computer systems has received some attention in the last couple of decades. However, most of the prediction methods focused only on prediction accuracy, but not the practical usage of the prediction. For example, Vilalta et al. [58] presented both long-term failure prediction based on seasonal trends such as CPU utilization or disk utilization and short-term failure prediction based on events observed in the system. While these two methods can provide valuable insights, they cannot help runtime fault tolerance because long term prediction does not tell when the failure will occur while short term prediction only predicts failures a few minutes before their occurrence. Similarly, in [47], Sahoo et al. applied rule-based classification method on failure logs to find error patterns preceding a critical event, but failed to show that there is an appropriate gap between these errors and the critical event. Another example can be found in [48], in which Salfner et al. attempted to predict whether there will be a failure in the next 30 seconds. We note that not many practical measures can be taken within such a short time frame.

At the same time, the data mining community has witness a number of studies on rare class analysis. For instance, Joshi et al. [32] discussed the limitations of boosting algorithms for rare class modeling and proposed PNrule, a two-phase rule induction algorithm, to carefully handle the rare class cases [31]. Other algorithms developed for mining rare classes include SMOTE [19], RIPPER [21], etc. A survey paper was given by Weiss [61]. However, we note that, these research efforts merely focused on the algorithm-level improvement of the existing classifiers for rare class analysis on generic data, but not on event log data in a specific application domain. Indeed, as previously described, IBM BlueGene/L event log data have uniquely challenging characteristics which thus requires domain expert knowledge to transform data into a form that is appropriate for running off-the-shelf classifiers. More importantly, we would like to point out that due to these characteristics, off-the-shelf tools fail to classify BlueGene/L event data.

5.5 Concluding Remarks

As failures become more prevalent in large-scale high-end computing systems, the ability to predict failures is becoming critical to ensure graceful operation in the presence of failures. A good failure prediction model should not only focus on its accuracy, but also focus on how easily the predicted results can be translated to better fault tolerance. To address this need, we collected event logs over an extensive period from IBM BlueGene/L, and attempted to develop a prediction model based on the real failure data. Our prediction methodology involves first partitioning the time into fixed intervals, and then trying to forecast whether there will be failure events in each interval based on the event characteristics of the preceding intervals.

Our prediction effort addressed two main challenges: feature selection and classification. We carefully derived a set of features from the event logs. We then designed a customized nearest neighbor classifier, and compared its performance with standard classification tools such as RIPPER and SVMs, as well as with the traditional nearest neighbor based approach. Our comprehensive evaluation demonstrated that the nearest neighbor predictor greatly outperforms the other two, leading to an $F-$measure of 70% and 50% for a 12-hour and 6-hour prediction window size. These results indicate that it is promising to use the nearest neighbor predictor to improve system fault-tolerance.

References

[1] Computing Pearson's Correlation Coefficient. http://davidmlane.com/hyperstat/A51911.html.

[2] CORRELATION COEFFICIENT, Critical values for Testing Significance . http://www.met.rdg.ac.uk/cag/stats/corr.html.

[3] Data Lifeguard. http://www.wdc.com/en/library/2579-850105.pdf.

[4] Research Methods Knowledge Base. http://www.socialresearchmethods.net/kb/statcorr.php.

[5] SIGuardian. http://www.siguardian.com.

[6] SLURM: Simple Linux Utility for Resource Management. http://www.llnl.gov/linux/slurm.

[7] Statistics 101-103 Course Information, Department of Statistics, Yale University. http://www.stat.yale.edu/Courses/1997-98/101/stat101.htm.

[8] Support Vector Machines. http://www.svms.org.

[9] Testing the Population Correlation Coefficient from Dr. Hossein Arsham, Merrick School of Business, University of Baltimore . http://home.ubalt.edu/ntsbarsh/zero/correlation.htm.

[10] The R Project for Statistical Computing. http://www.r-project.org/.

[11] The UC Berkeley/Stanford Recovery-Oriented Computing (ROC) Project. `http://roc.cs.berkeley.edu/`.

[12] TOP500 List 11/2004. http://www.top500.org/lists/2005/11/basic.

[13] H.R. Berenji, J. Ametha, and D.A. Vengerov. Inductive Learning For Fault Diagnosis. In *Proceedings of the 12th IEEE International Conference on Fuzzy Systems*, 2003.

[14] B. Boser, I. Guyon, and V. Vapnik. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, 1992.

[15] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting, 2nd Edition*. Springer, 2002.

[16] M. F. Buckley and D. P. Siewiorek. Vax/vms event monitoring and analysis. In *FTCS-25, Computing Digest of Papers*, pages 414–423, June 1995.

[17] M. F. Buckley and D. P. Siewiorek. Comparative analysis of event tupling schemes. In *FTCS-26, Computing Digest of Papers*, pages 294–303, June 1996.

[18] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at *http://www.csie.ntu.edu.tw/ cjlin/libsvm*.

[19] N.V. Chawla, K.W. Bowyer, L.O. Hall, and W.P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of AI Research*, 16:321–357, 2002.

[20] William W. Cohen. Fast Effective Rule Induction. In *ICML*, 1995.

[21] W.W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995.

[22] C. Cortes and V. Vapnik. Support-vector network. In *Machine Learning 20*, pages 273–297, 1995.

[23] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines (and other kernel-based learning methods)*. Cambridge University Press, 2000.

[24] M. L. Fair, C. R. Conklin, S. B. Swaney, P. J. Meaney, W. J. Clarke, L. C. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber. Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990. *IBM Journal of Research and Development*, 48(3/4), 2004.

[25] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing Completion Time of a Program by Checkpointing and Rejuvenation. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 252–261, May 1996.

[26] J. Hansen. *Trend Analysis and Modeling of Uni/Multi-Processor Event Logs*. PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1988.

[27] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *Proceedings of the ACM SIGMETRICS 2002 Conference on Measurement and Modeling of Computer Systems*, pages 217–227, 2002.

[28] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan. Improved disk-drive failure warnings. *IEEE Transactions on Reliability*, 51(3):350–357, 2002.

[29] IBM. Autonomic computing initiative, 2002. http://www.research.ibm.com/autonomic/index_nf.html.

[30] R. Iyer, L. T. Young, and V. Sridhar. Recognition of error symptons in large systems. In *Proceedings of the Fall Joint Computer Conference*, 1986.

[31] M.V. Joshi, R.C. Agarwal, and V. Kumar. Mining needle in a haystack: Classifying rare classes via two-phase rule induction. In *SIGMOD*, pages 91–102, 2001.

[32] M.V. Joshi, R.C. Agarwal, and V. Kumar. Predicting rare classes: Can boosting make any weak learner strong? In *KDD*, 2002.

[33] M. Kalyanakrishnam and Z. Kalbarczyk. Failure data analysis of a lan of windows nt based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[34] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2006. To Appear.

[35] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta. Filtering Failure Logs for a BlueGene/L Prototype. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 476–485, 2005.

[36] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[37] T. Y. Lin and D. P. Siewiorek. Error log analysis: Statistical modelling and heuristic trend analysis. *IEEE Trans. on Reliability*, 39(4):419–432, October 1990.

[38] J. Meyer and L. Wei. Analysis of workload influence on dependability. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 84–89, 1988.

[39] J. Milano, G.L. Mullen-Schultz, and G. Lakner. Blue Gene/L: Hardware Overview and Planning. Technical report.

[40] S. Mourad and D. Andrews. On the Reliability of the IBM MVS/XA Operating System. In *IEEE Trans. Software Engineering*, October, 1987.

[41] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.

[42] A. J. Oliner, R. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware Job Scheduling for BlueGene/L Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[43] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2004.

[44] H. T. Reynolds. *The Analysis of Cross-classifications*. The Free Press, New York, 1977.

[45] Richard J. Roiger and Michael W. Geatz. *Data Mining: A Tutorial Based Primer*. Addison-Wesley, 2003.

[46] R. Sahoo, A. Sivasubramaniam, M. Squillante, and Y. Zhang. Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 389–398, 2004.

[47] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical Event Prediction for Proactive Management in Large-scale Computer Clusters . In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 2003.

[48] F. Salfner, M. Schieschke, and M. Malek. Predicting Failures of Computer Systems: A Case Study for a Telecommunication System. http://samy.informatik.hu-berlin.de/ schiesch/papers/salfner06predicting.pdf.

[49] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance-computing systems. In *The 2006 International Conference on Dependable Systems and Networks*, June 2006.

[50] K. Skadron, M.R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2003.

[51] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. A reliability odometer - lemon check your processor! In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI), The Wild and Crazy Idea Session IV*, 2004.

[52] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Processor Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, June 2004.

[53] D. Tang and R. K. Iyer. Impact of correlated failures on dependability in a VAXcluster system. In *Proceedings of the IFIP Working Conference on Dependable Computing for Critical Applications*, 1991.

[54] D. Tang and R. K. Iyer. Analysis and modeling of correlated failures in multicomputer systems. *IEEE Transactions on Computers*, 41(5):567–577, 1992.

[55] D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modelling of a VAXcluster system. In *Proceedings International Symposium on Fault-tolerant Computing*, pages 244–251, 1990.

[56] M. M. Tsao. *Trend Analysis and Fault Prediction*. PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1983.

[57] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. In *Proceedings of the ACM SIGMETRICS 2001 Conference on Measurement and Modeling of Computer Systems*, pages 62–71, June 2001.

[58] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461–474, 2002.

[59] R. Vilalta and S. Ma. Predicting Rare Events in Temporal Domains. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM02)*, pages 13–17, 2002.

[60] G. M. Weiss and H. Hirsh. Learning to predict rare events in event sequences. In *Proceedings of the Fourth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1998.

[61] G.M. Weiss. Mining with rarity: a unifying framework. *ACM SIGKDD Explorations*, 6(1):7–19, 2004.

[62] J. Xu, Z. Kallbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. *Technical Report CRHC 9808 University of Illinois at Urbana- Champaign*, 1999.

[63] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages LNCS (2862):44–60, 2003.

[64] J.F. Zeigler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.

[65] Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. Sahoo. Performance Implications of Failures in Large-Scale Cluster scheduling. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

Vita

Yinglung Liang

1995	BS in Electrical & Computer Engineering; The University of Alabama at Birmingham, USA.
1998	MS in Electrical & Computer Engineering; The University of Alabama at Birmingham, USA.
2007	Ph.D. in Electrical & Computer Engineering; Rutgers University, The State University of New Jersey, USA.

1995-1998	Research Assistant, Department of Electrical & Computer Engineering; The University of Alabama at Birmingham, USA
1998-1999	Functional Integrated Tester, Lucent Technologies, Homedel, NJ, USA
1999-2001	Software Developer, Telcordia Technologies, Piscataway, NJ, USA
2002-2007	Graduate Assistant, WINLAB, Rutgers University, NJ, USA.

Publications

2007	*Failure Prediction in IBM BlueGene/L Event Logs.* Y. Liang, Y. Zhang, H. Xiong, R. Sahoo, and A. Sivasubramaniam, in Proceedings of IEEE International Conference on Data Mining (ICDM), 2007. (acceptance rate: 19.2%).
2007	*An Adaptive Semantic Filter for BlueGene/L Failure Log Analysis.* Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, in Proceedings of Third International Workshop on System Management Techniques, Processes, and Services (SMTPS), 2007.
2006	*BlueGene/L Failure Analysis and Prediction Models.* Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, R. Sahoo, in the Proceedings of IEEE, Dependable System and Network (DSN ’06), pp. 425-434. (acceptance rate: 18.4%)
2005	*Filtering Failure Logs for a BlueGene/L Prototype.* Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, M. Gupta, in the Proceedings of IEEE, Dependable System and Network (DSN ’05), pp. 476-485. (acceptance rate: 21.07%)
2007	*Filtering Failure Logs for a BlueGene/L.* Y. Liang, Y. Zhang, H. Xiong, A. Sivasubramaniam, R. Sahoo, under review for IEEE, Transactions on Parallel and Distributed System (TPDS ’07).

2007 *Enhancing Failure Analysis for IBM BlueGene/L: A Filtering Perspective*. Y. Liang, Y. Zhang, H. Xiong, R. Sahoo, under review for IEEE Transactions on Dependable and Secure Computing.

2007 Failure Analysis, Modeling, and Prediction for BlueGene/L,. Yinglung Liang, Ph.D. dissertation, Rutgers University, the State University of New Jersey, 2007

1998 Performance Analysis of Telepathology in an ATM network. Yinglung Liang, M.S. thesis, The University of Alabama at Birmingham, 1998

1995 Hardware Turing Machine. Yinglung Liang, B.S., individual senior design project, The University of Alabama at Birmingham, 1995