Filtering Failure Logs for a BlueGene/L Prototype

Yinglung Liang† Yanyong Zhang† Anand Sivasubramaniam‡ Ramendra K. Sahoo§ Jose Moreira§ Manish Gupta§
†ECE Dept. ‡CSE Dept. §Exploratory Systems Software Dept.
Rutgers University Penn State University IBM T. J. Watson Research Center
{ylliang,yyzhang}@ece.rutgers.edu anand@cse.psu.edu {rshaoo,jmoreira,mgupta}@us.ibm.com

Abstract

The growing computational and storage needs of several scientific applications mandate the deployment of extreme-scale parallel machines, such as IBM's BlueGene/L which can accommodate as many as 128K processors. In this paper, we present our experiences in collecting and filtering error event logs from a 8192 processor BlueGene/L prototype at IBM Rochester, which is currently ranked #8 in the Top-500 list. We analyze the logs collected from this machine over a period of 84 days starting from August 26, 2004. We perform a three-step filtering algorithm on these logs: extracting and categorizing failure events; temporal filtering to remove duplicate reports from the same location; and finally coalescing failure reports of the same error across different locations. Using this approach, we can substantially compress these logs, removing over 99.96% of the 828,387 original entries, and more accurately portray the failure occurrences on this system.

1 Introduction

The growing computational and storage demands of applications continue to fuel the research and development of high-end computer systems, whose capabilities and scale far exceed those available in the market today. Many of these applications play critical roles in impacting economies of enterprizes and even countries, health and human development, military/security, and in enhancing the overall quality of life. It is widely recognized that parallelism in processing and storage is essential to meet the immense demands imposed by many of these applications (e.g. protein folding, drug discovery, weather modeling, national infrastructure simulations, etc.), and there is a pressing need to accelerate the deployment of large scale parallel systems with several thousand processors. IBM's BlueGene/L [3, 15, 6, 5] is a recent commercial offering to meet these demands, with two deployments of this system (on a smaller scale than the maximum 128K processors allowed by this architecture) already making it to the top 10 of the Top 500 Supercomputers list [2].

While performance, and the usability issues (such as programming/tuning tools) to some extent, have been the primary targets of investigation traditionally, there is a growing problem - the occurrence of failures - which is demanding equal attention [17, 18, 20, 29, 22, 4, 13, 7, 21, 27, 11, 25, 23, 28]. Although fault-aware design has gained importance for uniprocessor and small-scale systems, the problem escalates to a much higher magnitude when we move to the large scale parallel systems. In addition to the multiplicative factor in the error occurrences, the complex inter-dependencies between the processors (e.g. in the way the applications communicate, in the way the processors are allocated to jobs, etc.) exacerbate the error rates of these systems [11, 25, 23, 28] containing thousands of processors. Rather than treat errors/failures as an exception, we need to recognize that they are commonplace on these systems.

Understanding the failure behavior of large scale parallel systems is crucial towards alleviating these problems. This requires continual online monitoring and analysis of events/failures on these systems over extended periods of time. The data obtained from such analysis can be useful in several ways. The failure data 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 from the field (for hardware/software revisions). It can help system administrators for maintenance, diagnosis, and enhancing the overall health (uptime). They can isolate where problems occur, and take appropriate remedial actions (replace the node-card, replace the disks/switches, reboot a node, select points for software rejuvenation, schedule down-times, etc.). 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.

Even if there have been production environments collecting these system events/failures, there are no published results to date on such data in the context of large scale parallel machines. More importantly, the monitoring and logging of these events need to be continuously done over extended periods of time, spanning several months/years. The consequence of such monitoring is the voluminous amount of information that can accumulate because of the temporal (over long periods of time) and spatial (on each of the thousands of nodes) dimensions that are involved. Further, there are (i) several extraneous (and non-critical) events that can get recorded, (ii) the same event could get recorded in multiple ways at a node, and (iii) the same event could be detected in different ways across the nodes. Consequently, it becomes imperative to filter this evolving data and isolate the specific events that are needed for subsequent analysis. Without such filtering, the analysis can lead to wrong conclusions. For instance, the same software error can materialize on all nodes that an application is running on, and can reduce the MTBF than what is really the case.

To our knowledge, this is the first paper to present system event/failure logs from an actual 8192-processor BlueGene/L system over a period of 84 days, and describes our experiences in filtering these events towards identifying the specific failures to be useful in subsequent analysis. In the raw event logs from this system, we have over 828,387 events with 211,997 fatal failures. We first perform a temporal filtering step by eliminating event records at a node which are within 2 minutes of each other. This step brings down the fatal failures to 9,150. 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 torus network, the node cards, the service cards, and the midplane switches - impacted by these failures. We then show how the event records for each of these components can further be filtered by a closer examination of the nature of the errors and how the errors get reported/propagated. At the end of all such filtering, we isolate the number of actual errors over this 84 day period to 311[1], which represents a reduction of 99.96% of the fatal failures reported in the original logs. Such filtering is extremely important when accumulating and processing the failures in a continuous fashion for online decision making.

The rest of this paper is organized as follows. The next section briefly summarizes related work. Subsequently, we give an overview of the BG/L architecture, together with specifics on the system where the logs were collected in Section 3. The details on the filtering steps are given in section 4. Finally, section 5 provides the concluding remarks.

2 Related Work

Though there has been previous interest in monitoring and filtering system events/failures (e.g. [16, 8, 25, 23]), there has been no prior published work on failures in large scale parallel systems spanning thousands of nodes. Collection and filtering of failure logs has been examined in the context of much smaller scale systems. Lin and Siewiorek [16] 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 [8], the authors make recommendations about how to monitor events and create logs. It has been recognized [26, 9, 14, 12] that it is critical to coalesce related events since failures propagate in the time and error detection domain. The tupling concept developed by Tsao [26], groups closely related events, and is a method of organizing the information in the log into a hierarchical structure to possibly compress failure logs [9].

Tang [25, 23] studied the error/failure log collected from a VAX cluster system consisting of seven machines and four storage controllers. 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 [24], 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 [28] 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. In our previous study [19], 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.

3 BG/L Architecture and Error Logs

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

3.1 BG/L Architecture

In this study, we use event/failure logs collected from a 8192-processor BlueGene/L DD1 prototype, housed at IBM Rochester, which currently stands at number 8 in the Top 500 list of supercomputers [2]. The machine has been up since May, 2004, and has been primarily running parallel scientific applications. This prototype has 4096 compute chips, with each chip containing two processors.

Each compute chip consists of two PPC 440 cores (processors), with a 32KB L1 cache and a 2 KB L2 (for prefetching) for each core. The cores share a 4MB EDRAM L3 cache. In addition, a shared fast SRAM array is used for communication between the two cores. L1 caches in

[1] Please note that these numbers do not reflect the MTBF of actual BlueGene machine [1] because the prototype is not as mature as the actual deployment in terms of both hardware and software.

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, and (iv) a control network that supports booting, monitoring and diagnostics through JTAG access.

A compute card contains two such chips, and also houses a 256MB or 512MB DRAM for each chip on the card. In this paper, we use the term *compute card* and compute node interchangeably. A node card contains 16 such compute cards, and a midplane holds 16 node cards (a total of 512 compute chips or 1K processors). The BG/L prototype has 2048 compute cards, 128 node cards, and 8 midplanes, that are housed in 4 racks. I/O cards, each containing two chips and some amount of memory (usually larger than that of compute nodes), are also housed on each midplane. There are 4 such I/O cards (i.e., 8 I/O chips) for each midplane on the prototype, 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 midplane switches ("link chips") to connect 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 prototype, thus, has 192 link chips totally. A midplane also has 1 service card that performs system management services such as verifying the heart beat of the nodes, and monitoring errors. This card has much more powerful computing abilities, and runs a full-fledged IBM DB2 database engine for event logging.

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.

3.2 System Error 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. However, 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 also recorded in job logs.

We have been collecting failure logs since August 26, 2004 until the present. The raw logs contain all the events that occur within different components of the machine. Information about scheduled maintenances, reboots, and repairs is not included. Each record of the logs describes an event using several attributes as described below:

- *Record ID* is the sequence number for an error entry, which is incremented upon each new entry being appended to the logs.
- *Event time* is the time stamp associated with that event.
- *Event type* specifies the mechanism through which the event is recorded, with most of them being through RAS [10].
- *Event 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", "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 on why these cards may not be functional (e.g."link-card is not accessible", "problem while initializing link/node/service card", "error getting assembly information from the node card", etc.). ERROR events report problems that are more persistent and further pin-point 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. Our primary focus in this study is consequently on FATAL and FAILURE events.

- *Facility* attribute denotes the component where the event is flagged, which can be one of the following: LINKCARD, APP, KERNEL, DISCOVERY, MMCS, or MONITOR. LINKCARD events report problems with midplane switches, which is related to communication between midplanes. APP events are those flagged in the application domain of the compute chips. Many of these are due to the application being killed by certain signals from the console. In addition, APP events also include network problems captured in the application code. Events with KERNEL facility are those reported by the OS kernel domain of the compute chips, which are usually in the memory and network subsystems. 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 DISCOVERY facility are usually related to resource discovery and initial configurations within the machine (e.g. "service card is not fully functional", "fan module is missing", etc), with most of these being at the INFO or WARNING severity levels. MMCS facility errors are again mostly at the INFO level, which report events in the operation of the MMCS. 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 FATAL or FAILURE severity levels.
- *Location* of an event (i.e., which chip/node-card/service-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.

Between August 26, 2004 and November 17, 2004 (84 days), there are totally 828,387 events in the raw log.

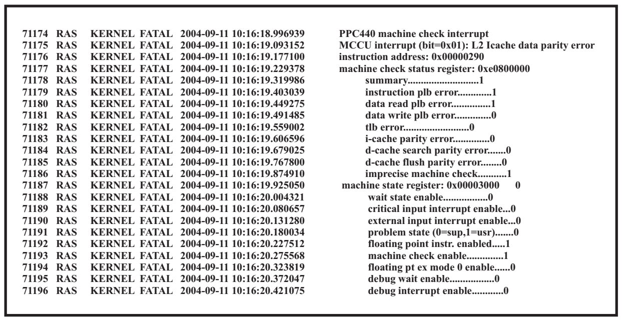
4 Filtering and Preprocessing Techniques

While event logs can help one understand the failure properties of a machine to enhance the hardware and systems software for better failure resilience/tolerance, it has been recognized [26, 9, 14, 12] 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 the thousands of hardware resources/processors (spatial) on these parallel machines, exacerbates the volume of data collected. For example, the logging tool employed by the BG/L prototype has generated 828,387 entries over a period of 84 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 isolate unique failure records. In this section, we present our 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 the BG/L prototype for subsequent analysis.

4.1 Step I: Extracting and Categorizing Failure Events

While 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 what are needed for software/hardware revisions and designing more effective failure 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 paper. As a result, we first screen out events with lower severity levels. This step removes 616,390 error records from 828,387 total entries, leaving only 211,997 failures, which constitute 14.7% of the total error records. Next, we remove those application level events due to 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 211,997 to 190,775 entries.

Instead of lumping together failures from different components of the machine, we categorize them into five classes: (i) memory failures, denoting failures in any part of the memory hierarchy on all the compute nodes; (ii) network failures, denoting exceptions within the torus when application processes running on the compute nodes exchange messages; (iii) node card failures, denoting problems with the operations of node cards; (iv) service card failures, denoting errors with service cards; and (v) midplane switch failures denoting failures with midplane switches or their links. These five classes cover all the major components of the BG/L hardware, or at least those components that are included in the error events. It is to be noted that our original plan was to include compute nodes as one of BG/L's components instead of memory, but we use memory in this paper because nearly all the failures we have observed so far on a compute node are within the memory hi-



```
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 1. 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.

erarchy. In later sections, we will show that different classes of failures have different properties. Among 190,775 total failures, we have 16,544 memory events, 157,162 network events, 10,500 node card events, 6,195 service card events, and 374 midplane switch events.

4.2 Step II: Compressing Event Clusters at a Single Location

When we focus on a specific location/component (i.e., a specific chip/node-card/service-card/midplane-switch), 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. It is quite likely that all events in the cluster are referring to the same failure, and this can arise because of the following 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-L1, ECC for EDRAM L3) typically lead to event clusters. Figure 1 illustrates such a cluster for a specific compute card. In this figure, each entry is represented by the following attributes: id, type, facility, severity, timestamp, and description. 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 error 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 failure(s), information on which hardware resources incurred the failure(s) - the processor local bus (plb), the TLB, etc. - and a dump of status registers. In our examination, we have found 10 such typical memory failure clusters, and we need to record just a single failure for each occurrence of a cluster.
- A node card failure cluster usually comprises of multiple temperature errors as well as power errors, with one power error almost immediately following a temperature error. This is because whenever a temperature exception (e.g., temperature being above a threshold) is detected on a node, the node card will shut down its power, resulting in the report of a power failure as well. This sequence gets recorded many times until it is finally resolved. The cluster size varies significantly throughout the log, ranging from a few entries to a few thousand entries, depending on how soon the problem is fixed in the system. Clearly, we can use a single temperature error (e.g. due to a broken fan) to replace such a cluster.
- Failures in the other three components, i.e., network, service card and midplane switches, exhibit clusters as well. Events within these clusters usually have identical descriptions. For example, the description “Service Card Power Error” appears 3162 times between timestamp 2004-09-14 14:05:05.440647 and timestamp 2004-09-15 17:18:29.862598 for the same service card, with an average time-stamp interval between records of 30 seconds. This is because the same failure is continuously recorded until the cause of the problem is fixed. Similar trends are also observed with network failures and midplane switch failures, though each failure type may have different cluster sizes. For instance, midplane switch failure clusters are usually very small, with many comprising a single entry.

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 with the previous record’s timestamp, and only keep the entry if the gap is above the clustering threshold T_{th}. 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 form clusters, 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 [19] in that the earlier scheme only filters out entries with identical descriptions, which is insufficient in our case.

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 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, midplane switch failures have a small compression ratio, e.g., 13% with a 5-minute threshold, because they usually have small cluster sizes. Third, when the threshold reaches a certain point, though the compression ratio still keeps increasing, the improvement is not that significant, especially because we ensure an event cluster only contains failures of the same type. At the same time, a threshold larger than 5 minutes is undesirable because the logging interval of BG/L prototype is significantly smaller than that, and it may cause unrelated events to fall in a cluster. As a result, we choose 2 minutes as the threshold to coalesce event clusters. *Note that such coalescing is done only for the events at the same location/component.*

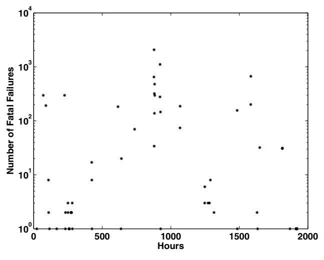
After compressing all the event clusters, we have 10 types of memory failures, 13 types of torus failures, 2 types of node card failures, 2 types of service card failures, and 8 types of midplane switch failures. At the end of this filtering step, we have brought down the number of failures to 9150, with the individual breakdown for each component given by the entry for T_{th}= 2 minutes in Table 1.

4.3 Step III: Failure-Specific Filtering Across Locations

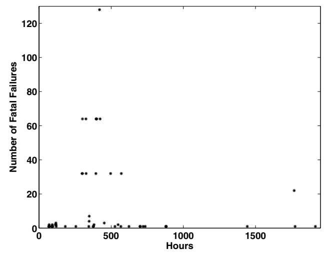
Let us now take a closer look at the results from the previous filtering step. The time-series dataset contains periods with multiple events per second and other periods with no events over many consecutive seconds. Hence, our analysis considers two different but related views of these time-series datasets, 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 inverse to each other.

T_{th} (s)	memory	network	node card	service card	midplane switch
0	16,544	157,162	10,500	6,195	374
1	9,442	32,152	3,361	5,455	374
30	787	13,038	438	424	371
60	764	11,193	8	17	358
120	714	8,063	8	14	351
300	645	7,541	6	12	324

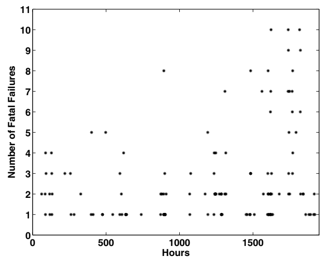
Table 1. The impact of T_{th} on compressing different failures. T_{th} = 0 denotes the log before any compression.



(a) network failure records



(b) memory failure records



(c) midplane switch failure records

Figure 2. Rate processes of failure entries after applying filtering step II. T_{th} = 2 minutes.

Statistics for 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 raw increment processes (i.e., the average, variance, and maximum number of hours between failure records within each subsystem) are provided in Tables 2 (a) and (b). We observe that despite removing event clusters (Step II) from the same location, the number of failure records (9150 across all components) is still quite large. This is particularly true of failures in the torus/tree networks, which constitute around 88% of the total failures, where Step II is less effective than for the other components.

In order to study this more closely, we plot the time series data for three of the components (the network, memory system, and midplane switches) in Figures 2(a)-(c) at an hourly granularity. We see that the failure occurrences are still (despite doing the time threshold based filtering in step II) highly skewed, i.e. some intervals contain considerably more records than others. For example, 26% of the total torus failures (2076 out of 8063) are reported during one hour (i.e., the 878th hour) out of the entire 1921-hour period. Besides, we also notice that an interval of 6 hours, starting from hour 877 and ending at hour 883, has 49.5% of the total failure records. Turning our attention to Figure 2(b), we find that memory failures are also reported

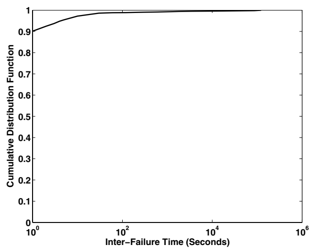
	entire system	network	memory	node card	service card	midplane switch
Total number of failure records	9150	8063	714	8	14	351
Average number of failure records per hour	4.7631	4.1973	0.3717	.0042	.0073	.1827
Variance of number of failure records per hour	3816	3791	22.5816	.0114	.0114	0.8359
Maximum number of failure records per hour	2077	2076	128	4	2	10

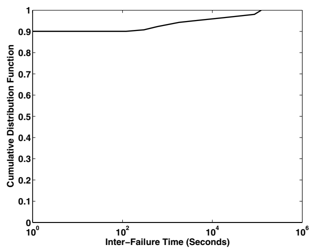
(a) rate process

	entire system	network	memory	node card	service card	midplane switch
MTBF (hours)	0.2075	0.2355	2.5892	147.0024	92.5513	5.3143
Variance of times between failure records (hours)	34997	99405	2.4426e6	2.8165e8	1.5197e8	1.1920e6
Maximum time between failure records (hours)	141.5822	190.5575	560.8161	765.5364	703.4128	160.1083

(b) increment process

Table 2. General statistics about failure logs after applying step II.


(a) Network failure records


(b) Memory failure records

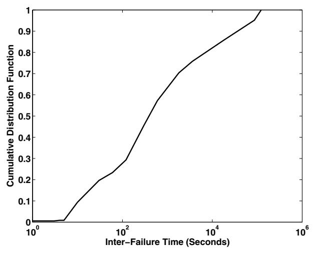
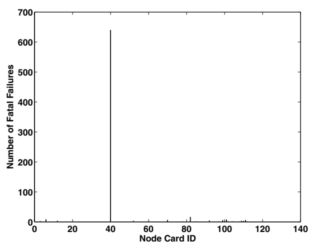

(c) Midplane switch failure records

Figure 3. CDF of inter-failure times after applying filtering step II. T_{th} = 2 minutes. The x-axes are plotted using log scale.


(a) Memory failure records across node cards

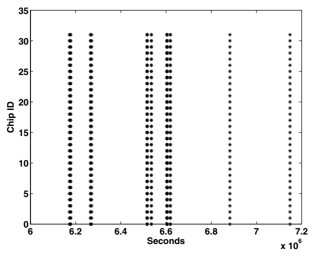

(b) All 32 chips within node card 40 report failures at the same time

Figure 4. A closer look at memory failures.

in bursts. For example, 93% of total memory failure entries fall in the interval between 299th hour and 570th hour. Compared to the other two components, midplane switch failures are more evenly distributed temporally.

Figures 3(a)-(c) reiterate the above observations by plotting the cumulative distribution function (CDF) of the inter-record times of the network, memory and midplane switch failures. Note that the x-axes are in log scale because the inter-failure times show wide variances. For both network and memory subsystems, 90% of the failure records are within 1 second of the preceding entry, with the midplane switch failures being more spaced out.

One may wonder why these failures are occurring in bursts, given that we have already used a temporal threshold of 2 minutes to filter out event clusters in Step II. Note that Step II only eliminates duplicate records that occur at the same location. It is possible that multiple locations/compute-chips could be detecting and recording the same errors. One cannot simply use a temporal threshold to filter out all of them since it is possible that they may be detecting different failures. It is important to dig deeper into the records for the different components to filter the events if they are indeed duplicates, and we discuss these issues for each component below.

4.3.1 Memory Failures

Note that the BG/L prototype contains 4,096 compute chips spread over 128 node cards, of which only chips of 22 node cards ever report memory failures (Figure 4(a)). Among these 22 node cards, chips within node card 40 report 90% of the failures. In addition, from Figure 4(b), we find that all 32 chips from node card 40 often report failures at almost the same time (with a few seconds apart of each other). Since it is extremely unlikely for 32 chips to encounter the same hardware failure at the same time - especially because these chips do not share memory, this strongly suggests that some of these memory failures may be caused by software bugs, e.g. de-referencing a null pointer, jumping to an invalid location, accessing out-of-bounds, etc., instead of hardware failures. The same bug in the software is causing a manifestation of this error on the different nodes running this application, and should thus be reported only once.

The challenge, however, lies in that it is difficult to tell whether an event is caused by a hardware failure or a software bug by just looking at its description. For example, an instruction TLB error can occur either because there is a hardware problem (similar to the i-L2 parity error that we illustrated in Figure 1) or because the program is trying to jump to an invalid instruction address. We take the viewpoint that a failure is more likely to be caused by software bugs if the following two conditions are true: (1) many compute chips (that are running the same program) report the same problem within a short interval, and (2) the instruction/data address at which the failure occurs should be the same across different chips (the probability of these two occurring in the case of a hardware error is extremely low).

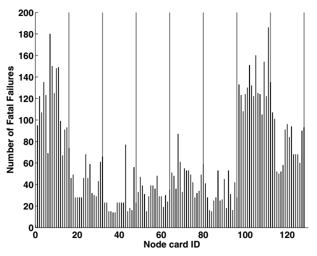
Filtering such events caused by the same error requires clustering the events that occur on different chips, within a short interval, and with the same event descriptions. Our filtering algorithm represents each failure record by a three-attribute tuple <timestamp, location, description>, and it outputs a table containing failure clusters in the form of <timestamp, description, cluster size>, where the cluster size field denotes how many different chips report this failure within a time threshold T_{mem}. When a new entry is scanned, we first use a hash structure to determine whether or not the description has appeared before. If this is the first time we see this description, it becomes the cluster head, and we insert its timestamp and description in the table, with the corresponding cluster size field set to 1. If the description has appeared before, we check if it falls into any cluster by comparing its timestamp with the cluster head which has the latest timestamp amongst all those having the same description. If the gap between these two timestamps is less than the threshold, it belongs to that cluster, and we will not add this entry to the table, only incrementing the cluster size by 1. If the gap is larger than the threshold, then this entry becomes a new cluster head, and we record its information in the result table. After applying this technique, we show portions of the resulting clusters for the memory failures with T_{mem} = 60 seconds in Table 3.

Next, we need to decide whether a cluster in Table 3 is a software failure or a hardware failure, by evaluating the two conditions mentioned above. Our algorithm marks those clusters having sizes larger than 1 as those caused by program bugs because the instruction addresses or data addresses at which the failures occur are exactly the same across the reporting chips.

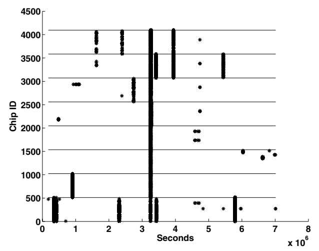
Our discussions with the BG/L logging experts confirm the validity of this approach. Some of the common clusters (which are also shown in Table 3) that we have found in the logs include: instruction storage interrupts and instruction TLB errors which can be caused by programs trying to jump to an instruction address that does not exist; data storage interrupts which can be caused by programs trying to de-reference an invalid address; program interrupts which can be caused by programs trying to execute an illegal instruction; and bus errors (denoted as L3 major error in the table) due to illegal addresses going out on the bus. After coalescing the software failures, we have totally 55 hardware failures, and 21 software failures over 84 days.

ID	Failure description	Cluster size
2	PPC440 instruction TLB error interrupt	32
5	PPC440 instruction storage interrupt	32
8	PPC440 data storage interrupt	32
12	PPC440 program interrupt	32
17	L3 major internal error	19

Table 3. Clusters of Memory failures.



(a) Network failure records across node cards. The 8 vertical lines mark the midplane boundaries.



(b) Many chips within the same midplane report the same network failure within a few minutes. The 8 horizontal lines mark the midplane boundaries.

Figure 5. A closer look at network failures.

4.3.2 Network Failures

Figure 5(a) shows how network failures are distributed across all 128 node cards. Note that these 128 node cards reside on 8 different midplanes, with node cards 1-16 on midplane 1, node cards 17-32 on midplane 2, and so on. In Figure 5(a), the midplane boundaries are marked by the eight vertical lines. An interesting observation is that the failure count statistics are correlated to the midplanes where they occur. This suggests that chips within the same midplane may be recording the same network problems, and we need to filter our multiple records of the same error. A more direct indication of this behavior is presented in Figure 5(b), which shows the time (on the x axis) and location (on the y axis) of all the network failures. It is clear that many chips report failures roughly at the same time and these chips are usually from the same midplane. In this figure, the midplane boundary is marked by the 8 horizontal lines.

Unlike the memory hierarchy (which is not shared) where the probability is extremely remote that the same hardware error simultaneously manifests on multiple chips, it is quite possible for the different compute chips to detect and report the same hardware failure in the (shared) network. If there exists a hardware problem in the network, such as a bad link, or a bad switch, this problem can be detected by any host that tries to access those shared resources. In order to accurately portray the number of network failures within the system, instead of counting one failure for each record, we need to coalesce them into one single failure. As a result, we use the same clustering technique as discussed in the context of memory failures to locate the redundant records generated by multiple chips. Partial results after applying this technique are shown in Table 4. In this table, the threshold that is used to form clusters is T_{net} = 15 minutes. Please note that this threshold is larger than the one used for the memory subsystem (i.e., 1 minute) because some network failures take a long time before they are fixed. Further, the same failures can be reported several times from the same midplane within a few minutes apart of each other possibly by different applications (the application crashes when encountering these failures), while in the memory subsystem, as soon as the application crashes due to a program bug, the same failure is unlikely to repeat within a short time frame.

There are three typical types of network failures that are usually reported by a large number of chips at the same time: (1) rts tree/torus link training failed, (2) BIDI Train Failure, and (3) external input interrupt:uncorrectable torus error. Among these three, both (1) and (2) 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 (3) occurs when an uncorrectable torus error is encountered, making the applications crash again.

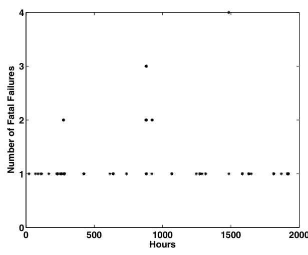
After the filteration with this clustering, the location of a network failure is no longer denoted by a chip ID, but by the midplane ID because a network failure can be encountered by any chip within that midplane. For those clusters that involve two midplanes (because the corresponding application is spread over two midplanes), we assume that the failure occurs within both midplanes. At the end of this filtering, we have 69 network failures over the 84 day period.

4.3.3 Midplane Switch Failures

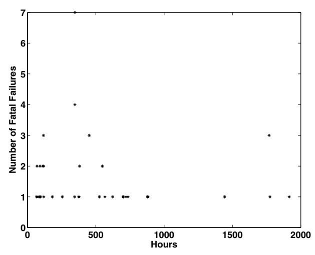
Midplane switches inter-connect midplanes, and are programmed by the control logic of the machine. These events usually report certain ports cannot be connected to or cleared. Whenever the control programs encounter such problems, it indicates there is a problem with the hardware - either the switch itself, or the link connecting the switches. If the switch has problems, then only the program that runs on the switch will report the failure; otherwise, several switches will be affected. As a result, we can use the same clustering algorithm (as descried in Section 4.3.1) to form the clusters, and coalesce a failure cluster into a single failure. However, as shown in Figure 2(c) and Figure 3(c), midplane switch failures are usually not clustered, with most cluster sizes being 1 or 2, and only very few going up to cluster sizes of 5 or 6. We would like to emphasize that Figure 2(c) shows the number of records per hour, but the threshold we use to determine the clusters is T_{mps} = 15 minutes, similar to the one used in the network subsystem. After coalescing these clustered events, we still have 152 midplane switch failures.

Failure description	cluster size
BIC_NCRIT interrupt (unit=0x02 bit=0x00)	1
rts tree/torus link training failed	297
(RAS) BiDi Train Failure	2071
external input interrupt:uncorrectable torus error	342
rts tree/torus link training failed	32

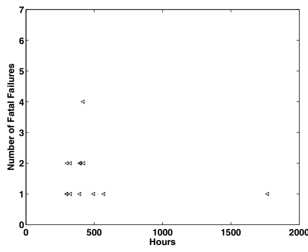
Table 4. Network failure clusters.



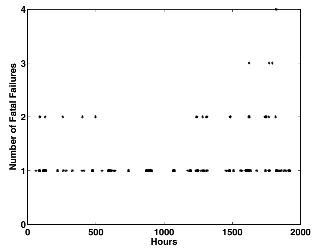
(a) Network failures



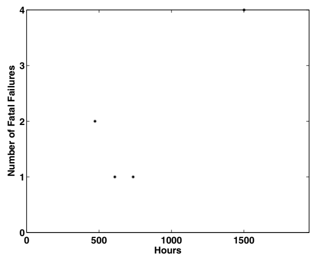
(b) Hardware memory failures



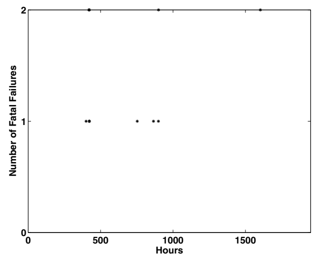
(c) Software memory failures



(d) Midplane switch failures



(e) Node card failures



(f) Service card failures

Figure 6. Time series of the failure datasets.

4.4 Applying the 3-step Filtering Algorithm

After discussing the 3-step filtering technique, we have 61 network failures, 76 memory failures (among which 55 are hardware failures, and 21 software failures), 152 midplane switch failures, 8 node card failures, and 14 service card failures over the period of 1921 hours. Figures 6(a)-(f) present the time series of failures at an hourly granularity.

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. 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 paper, we have embarked on a study of the failures on a 8192 processor BlueGene/L prototype at IBM Rochester, which current stands at #8 in the Top-500 list. We have presented a 3-step filtering algorithm: 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 different locations. Using this approach, we can considerably compress error logs by removing 99.96% of the 828,387 entries recorded between August 26, 2004 and November 17, 2004 on this system. Such filtering is being performed on a weekly basis.

References

[1] Blue Gene/L System Software Update. http://www-unix.mcs.anl.gov/ beckman/bluegene/SSW-Utah-2005/BGL-SSW13-LLNL-exper.pdf.

[2] TOP500 List 11/2004. http://www.top500.org.

[3] N. Adiga et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of Supercomputing (SC2002)*, November 2002.

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface – SCSI vs. ATA. In *Proceedings of the Conference on File and Storage Technology (FAST)*, March 2003.

[5] BlueGene/L Workshop. http://www.llnl.gov/asci/platforms/bluegene/.

[6] IBM Research, BlueGene. http://www.research.ibm.com/bluegene/.

[7] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[8] 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.

[9] 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.

[10] 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.

[11] R. L. Graham, S. E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *Proceedings of the International Conference on Supercomputing*, 2002.

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

[13] G. Herbst. IBM's Drive Temperature Indicator Processor (Drive-TIP) Helps Ensure High Drive Reliability. In *IBM White Paper*, October 1997.

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

[15] E. Krevat, J. G. Castanos, and J. E. Moreira. Job Scheduling for the BlueGene/L System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] J. Xu, Z. Kallbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. *Technical Report CRHC 9808 UIUC*, 1999.

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