FAILURE AVOIDANCE TECHNIQUES FOR HPC SYSTEMS BASED ON FAILURE PREDICTION

BY

ANA GAINARU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Abstract

A increasingly larger percentage of computing capacity in today's large high-performance computing systems is wasted due to failures and recoveries. Moreover, it is expected that high performance computing will reach exascale within a decade, decreasing the mean time between failures to one day or even a few hours, making fault tolerance a major challenge for the HPC community. As a consequence, current research is focusing on providing fault tolerance strategies that aim to minimize fault's effects on applications. By far, the most popular and used techniques from this field are rollback-recovery protocols. However, existing rollback-recovery techniques have severe scalability limitations and without further optimizations the use of current protocols is put under serious questions for future exascale systems. A way of reducing the overhead induced by these strategies is by combining them with failure avoidance methods. Failure avoidance is based on a prediction model that detects fault occurrences ahead of time and allows preventive measures to be taken, such as task migration or checkpointing the application before failure. The same methodology can be generalized and applied to anomaly avoidance, where anomaly can mean anything from system failures to performance degradation at the application level. For this, monitoring systems require a reliable prediction system to give information on when failures will occur and at what location. Thus far, research in this field used ideal predictors that do not have any implementation in real HPC systems.

This thesis focuses on analyzing and characterizing anomaly patterns at both the application and system levels and on offering solutions to prevent anomalies from affecting applications running in the system. Currently, there is no good characterization of normal behavior for system state data or how different components react to failures within HPC systems. For example, in case a node experiences a network failure and is incapable of generating log messages, the failure is announced in the log files by a lack of generated messages. Conversely, some component failures may cause logging a large numbers of notifications. For example, memory failures can result in a single faulty component generating hundreds or thousands of messages in less than a day. It is important to be able to capture the behavior of each event type and understand what is the normal behavior and how each failure type affects it. This idea represents the building block of a novel way of characterizing the state of the system in time by analyzing the properties of each event described in different system metrics, considering its own trend and behavior. The method introduces the integration between signal processing concepts and data mining techniques in the context of analysis for large-scale systems. By shaping the normal and faulty behavior of each event and of the whole system, appropriate models and methods for descriptive and forecasting purposes are proposed. After having an accurate overview of the whole system, the thesis analyzes how the prediction model impacts current fault tolerance techniques and in the end integrates it into a fault avoidance solution. This hybrid protocol optimizes the overhead that current fault tolerance strategies impose on applications and presents a viable solution for future large-scale systems.

Behind any PhD student there is a strong team of technical and creative people. This thesis is dedicated to my amazing fantastic four: Adi, Cici, Cri and Dan.

Acknowledgments

First and foremost I want to thank my advisor, professor Marc Snir, for encouraging my research and for helping me grow as a research scientist. It has been an honor to work with him. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. I am also very grateful to professor Franck Cappello for his scientific advice and knowledge and many insightful discussions and suggestions. They have both been tremendous mentors for me. I would also like to thank professors William Gropp, William Kramer and Andrew Chien for serving as my committee members and contributing to one of my proudest moments. I would like to thank them for their comments and suggestions which made my thesis stronger.

A special thank you is reserved for the people from the National Center for Supercomputing Applications who have contributed immensely to my personal and professional time at UIUC. The group has been a source of friendships as well as good advice and collaboration. Professor William Kramer has been a mentor and an excellent example on how to be a great scientist. He has supported me not only by providing a research assistantship for over five years, but also by giving me the moral support and the freedom I needed to finish this thesis. I would also like to thank Cristina Beldica, who has been helpful in providing advice many times during my graduate school career and who represents an excellent role model as a successful woman scientist in HPC. My thesis would not have been as strong if not for all the contradictory discussions with the people from the Blue Waters project: Jeremy Enos, Mike Showerman and Joshi Fullop.

During my time at UIUC I met many amazing people who have helped and taught me immensely and who have become my friends along the way: Leonardo Bautista Gomez, Amina Guermouche, Matthieu Dorier, Thomas Ropars, Bogdan Nicolae, Mohamed Slim Bouguera, Vincent Baudoui and

Guillaume Aupy. The joy and enthusiasm everyone has for HPC and research in general was contagious and motivational. Their presence has made conferences and collaborations become vibrant and full of energy and had a tremendous positive effect on the overall quality of my research.

The hybrid preventive and proactive checkpointing solution discussed in this dissertation would not have been possible without the FTI tool developed by Leonardo Bautista Gomez and the help given by the HPC group of professor Satoshi Matsuoka at the Tokyo Institute of Technology. I have greatly appreciated their collaboration and have been impressed with their work ethic and efficiency.

In my later work of studying the effect of I/O congestion on application performance degradation, I am particularly indebted to the people from ENS Lyon. Yves Robert, Anne Benoit and Guillaume Aupy made significant contributions, especially to the scheduling and theory part of the work. I would also like to thank the I/O group at Argonne National Laboratory for inspirational discussions with us regarding our experiments on this topic.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This research was done in the context of the INRIA-Illinois Joint Laboratory for Petascale Computing. The work was also supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

My time at UIUC was made enjoyable in large part due to the many friends and groups that became a part of my life. I am grateful for the time spent playing board games with Alex, Laura, Ben, Yun, Ryan and many others; for my climbing buddies and our memorable trips into the mountains: Santiago, Manish, Andrew; for Elena and Zach who were always there for me. A special thanks goes to Kevin, my first and only house mate, who put up with my messiness.

Lastly, I would like to thank everyone in my family for supporting me in all my pursuits and for being my best friends. I have an amazing family, whose members are unique and complement each other in a wonderful way: my mother, Cici, is creative and has the best advice for any situation; my father, Dan, is technical and analytical and I can always bounce ideas with him; my sister, Cri, is artistic and independent and has been invaluable in de-stressing me between deadlines; and my husband, Adi, who has been my rock during the past 5 years, always putting things in perspective with his analytical thinking and great sense of humor.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The last few years have been a fertile ground for the development of many scientific and data-intensive applications in all fields of science and industry. These applications provide an indispensable means of understanding and solving complex problems through simulation and data analysis. As large-scale systems evolve towards post-petascale computing to accommodate applications increasing demands for computational capabilities, many new challenges need to be faced, among which fault tolerance is a crucial one [2, 3]. At the scale of today's large scale systems, fault tolerance is no longer an option, but a necessity. With failure rates predicted in the order of tens of minutes for the exascale era [4] and applications running for extended periods of time over a large number of nodes, an assumption about complete reliability is highly unrealistic. Because processes from scientific applications are, in general, highly coupled, even more pressure is put on the fault tolerance protocol since a failure to one of the processes will normally lead to the failure of the entire application.

By far, the most popular fault tolerance technique to deal with application failures is the Checkpoint-Restart strategy. However, checkpointing and restarting has a cost in time and energy. Some projections [5] estimate that, with the current technique, the time to checkpoint and restart may exceed the mean time to interrupt of state-of-the-art future generation supercomputers. This new projection means that multiple errors must be handled by current fault tolerance protocols. Moreover, the current approach is developed to apply the same fault tolerance method to all types of faults (permanent node crash, detected transient errors, network errors, file system failures) and for the whole duration of the execution. However, not all faults require the same expensive checkpoint-restart approach.

Since exascale supercomputers, which are expected by 2023-2025, will exhibit much more complexity and many more faults than todays supercomputers, it is imperative that we design fault tolerance systems with as low overhead as possible. Moreover, the instability of exascale systems with their diversity of faults and the limitations of "one size fits all" fault tolerance approaches, creates the need to develop a set of dedicated solutions in addition to the current general purpose ones.

Currently, there are several fault tolerant methods implemented at different levels of the software stack, from ECC implemented at the hardware level to more complex application level recovery methods. A complement to the classical checkpoint-restart approach is failure avoidance, by which the occurrence of a fault is predicted and preventive measures are taken. Failure avoidance uses the information received by a failure predictor to facilitate proactive fault tolerance mechanisms such as preventive job migration or proactive checkpoint. In general, failure prediction is based on the observation that there is a Fault-Error-Failure propagation graph as shown in Figure 1.1 (page 4). The fault generates a number of errors that, frequently, can be observable at the system level either by generating notifications in the event and activity log files or by changing values for several performance metrics. Some faults, however, might change the function of the system without being observable in the logs or metrics. The propagation chain ends with the failure which is observed at the application level and usually is represented by either a failover, a recovery action or an application interruption. Or, in some cases, the error could propagate and generate other effects like performance degradation.

Over the years, different methods have been developed that deal with failure prediction in the HPC community [6], methods that have been used extensively on different HPC systems and that present a variety of results. There are two levels of failure prediction in literature: component level and system level failure prediction. The first level assumes methods that observe components with their specific parameters and domain knowledge and define different approaches that give best prediction results for each. One example of this type is to compare the execution of good components with failed ones [7, 8]. The second level is represented by system failure prediction, in which monitoring daemons observe different system parameters (system logs, scheduler logs, performance metrics, etc) and investigate the existence of correlations between different events. There are numerous methods, starting with simple brute force extraction of rules between non-fatal events and failures [6] with more sophisticated techniques. In [9], the authors are using a meta-learning predictor to chose between a rule-based method and a statistical method depending on which gives better predictions for a corresponding state of the system. Other research include Support Vector Machines (SVM) [10], Hidden-Markov chains [11] or Bayesian networks [12].

Our experience with HPC systems has shown me that different system components exhibit different types of syndromes, both during normal operation and as they approach failure. Our key observation is errors are often predicted by changes in the frequency or regularity of various events. For this purpose, we investigate the integration of signal processing concepts and data mining techniques in the context of failure analysis for large-scale systems. By shaping the normal and faulty behaviour of each event, and of the whole system, we are able to propose appropriate models and methods for describing the chains of events occurring before failures. The results show that conventional signal processing techniques can create clear markers for changes in events behavior. Moreover, machine learning techniques become much more efficient when applied to the derived markers, rather than to the original signal.

We develop a prototype implementation of preventive checkpointing coupled with periodic multi-level checkpointing. The preventive checkpointing save the state of the application in each node's memory, providing fast checkpoint, which is necessary in order to act quickly between the failure prediction and the moment of the failure. Our method improves the performance of classical fault tolerance techniques when dealing with failures in petascale systems and the results show the potential of using such an approach on future exascale systems.

1.2 Focus areas

In order to address the challenges presented in the previous subsection, in this thesis we address the following research questions:

- Characterizing the behaviour of HPC systems in failure free situations

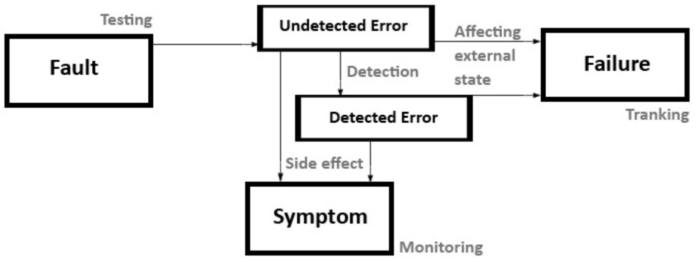


Figure 1.1: Failure Fault propagation

and how different types of failures affect this behaviour.

- Developing efficient methods for online analysis of the data generated by HPC systems for descriptive and predictive purposes.
- Developing forecasting methods that take into consideration the topology of HPC systems for predicting the locations along which a failure propagates.
- Characterizing failure precursors and their correlation with different levels of failures.
- Translating the failure prediction at the application level by correlating system and application events and metrics with application failures and performance degradation.
- Developing hybrid methods of preventive and proactive fault tolerance methods dependent of the type of failures affecting the applications.

This thesis introduces a new layer between the system and the fault tolerance protocols running beneath the applications. Figure 1.2 on page 5 presents an overview of our contribution. The layer we introduce is able to characterize the failures affecting the system and crashing applications. By monitoring the state of the system at all times, our modules are able to find patterns that lead to failures and to trigger alerts whenever the patterns are seen in real time. These alerts act as input to fault tolerance protocols that can then save the application's state or migrate the application before the failure actually affects it.

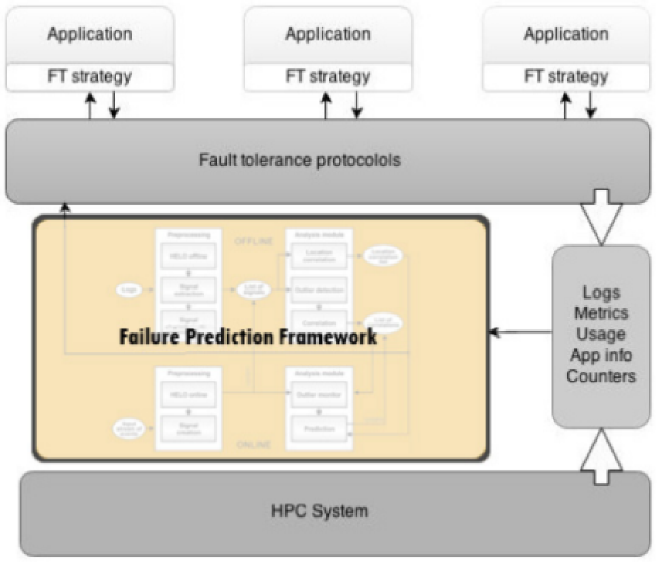


Figure 1.2: Overview of the research statement

1.3 Outline of the thesis

This section presents an overview of the thesis by presenting a summary of the contents for each chapter of the thesis. This thesis is structured in 6 additional chapters.

Chapter 2 introduces the context of the thesis, presents the state-of-the-art in event analysis and prediction methods, as well as in current fault avoidance theoretical techniques for todays large scale applications.

Chapter 3 presents a thorough analysis of several HPC systems current and past and introduces a characterization of the behavior of each. In this chapter we investigate the statistical properties of events and failures and their propagation behavior and we highlight the underlining differences and similarities between different HPC systems. Based on this information we propose new methods based on signal analysis concepts that automatically extract from log files the behavior of each event and create a universal global view of the system.

Chapter 4 focuses on proposing an accurate failure prediction methodology that includes the time and location of future failures. By combining signal analysis with data mining we were able to find patterns between events described in the log files and failures. We investigate the influence of different parameters on the overall prediction results and we highlight the best possible combination. Secondly, we investigate the feasibility of online failure prediction methods on a petascale machine. We are looking at a completely online approach based on the hybrid method for the Blue Waters system. This chapter also presents a location propagation methodology that uses network topology information. Finally, we show the differences between larger and smaller HPC systems that influence the prediction results and highlight possible future research directions.

Chapter 5 presents an implementation of a hybrid fault tolerance protocol that combines failure prediction with a multi-level checkpointing strategy. We show the overhead of such an approach and study the system utilization achieved for several HPC applications. This chapter also studies the benefit of our hybrid implementation on current and future exascale system by proposing several theoretical mathematical models.

Chapter 6 discusses on-going research related to application level failure prediction and specific predictors. We analyze the statistical characteristics of application crashes as well as the effect different failure types have on applications. We then focus on filesystem failures and the way they propagate to the applications. We show preliminary results for a new predictor focused on performance and environmental metrics used to extract patterns related to filesystem failures.

Chapter 7 is the last chapter of the thesis, it presents the conclusions of this work and includes a list of thesis contributions. The chapter also presents future research directions.

1.4 Definition of terms

The absence of consistent definitions and metrics for supercomputer reliability, availability and serviceability has been a problem for the community in the past [13]. In order to avoid this problem, the workshop organized by the Institute for Computing Sciences on August 2012 proposed a taxonomy of terms to be used as standard. The definitions that were proposed were based almost entirely on [14].

We will enumerate the most important ones in this section. For a complete list of metrics and definitions, consult the workshop's report [15] or the initial paper of A. Avizienis at al [14].

Figure 1.3 shows the propagation chain from faults to failures in a system. A fault represents the cause of an error, like a stuck bit or alpha particles. An error is the part of total state that might lead to a failure and the failure is a transition to incorrect function. Faults can be active or inactive, meaning actually causing errors or not. In general, a fault is local to one component while errors and failures may propagate from one component to another. In case of failures this propagation is called cascading failures. The distinction between hard and soft faults is not a strict one. Faults may be due to complex combination of internal state and external conditions that occur rarely and are difficult to reproduce. Memory errors can be classified into soft errors, which randomly corrupt bits but do not leave physical damage; and hard errors, which corrupt bits in a repeatable manner because of a physical defect.

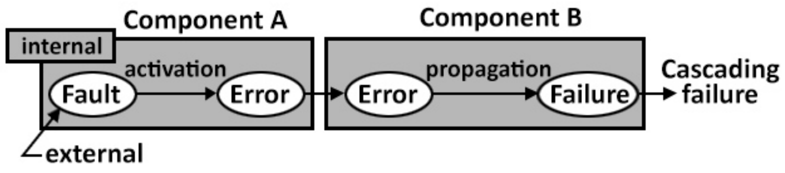


Figure 1.3: Error propagation and cascading failures

Error identification identifies the presence of an error but does not necessarily identify which part of the system state is incorrect, and what fault caused this error. By definition, every fault causes an error. Almost always, the fault is detected by detecting the error the fault caused. Therefore, fault detection or error detection refer often to the same thing. Latent or silent errors are errors that are not detected.

There are several means of dealing with faults: 1) Forecast is used to estimate the future number, future incidence and likely consequinces of faults; 2) Prevention is used to prevent fault occurences; 3) Removal is used to reduce the fault number and severity; 4) Tolerance is used to avoid failures in the presence of faults.

Time to Failure (TTF) is the interval between the end of last failure and the beginning of next failure. Time between Failures (TBF) is the interval between the beginnings of two consecutive failures. Time to Repair (TTR) is synonymous with Unscheduled Downtime and it represents an unplanned service outage or an incorrect service. In current supercomputers, several repairs can be done without any downtime for the system. Scheduled Downtime represents a planned service shutdown during which system upgrades and configuration changes are taking place. Production uptime is characterized by a correct operation service. Formally:

MTBF (Mean time between failures) = $\frac{TotalTime}{NumFailures}$
MTTF (Mean time to failure) = $\frac{Uptime}{NumFailures}$
MTTR (Mean time to repair) = $\frac{UnscheduledDowntime}{NumFailures}$

We use two metrics for measuring the results of a predictor, namely precision and recall. Precision is seen as a measure of fidelity and represents the proportion of correctly predicted failures to the total number of predictions made. Recall is the ratio of corrected predictions to all the existing failures in the system and represents a measure of completeness. The lead time represents the time between when a prediction is triggered and when the failure occurs. The lead time can be used by fault avoidance techniques to take a proactive action before the failure manifests in the system.

The components (such as memory, CPU, disk, and the network) of such systems arguably have different failure dynamics in terms of time and space. Some components may fail randomly and frequently while others may fail in a correlated fashion though rarely. Most modern supercomputers concurrently use multiple heterogeneous types of networks, storage and processors (such as GPUs, general purpose processors, and FPGAs). This potentially makes the failure dynamics even more diverse.

Chapter 2

State of the art on fault tolerance for HPC systems

Understanding the failure behavior of large scale parallel systems is crucial for achieving high utilization of large systems. The process requires continual online monitoring and analysis of all events generated in the system including normal notifications, performance metrics and failures 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 correct product defects. It can help system administrators for maintenance, diagnosis, and enhancing the overall system 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 predicting failures in these systems so that different algorithms could take preventive measures.

There is a significant number of research papers and tools in the area of event analysis in HPC systems that are used for many reasons, from system characterization to failure prediction or root cause analysis. This chapter discusses the challenges of event analysis in large scale systems and presents an overview of solutions proposed in the course of the last several years, while discussing their advantages and shortcomings.

2.1 Observations

The design of extreme-scale platforms that are expected to become available in 2022 will represent a convergence of technological trends and the boundary conditions imposed by over half a century of algorithm and application software development. The precise details of these new designs are not yet known. However, there are numerous papers that look at the configurations and properties of existing systems and make predictions regarding the future of HPC systems. Others present statistical information about failures and events by analyzing generated log files and gathered performance/environmental metrics.

There are several exascale/petascale reports that focus on resiliency and programming models for future exascale systems. The DARPA white paper on system resilience at extreme scale from 2008 [16] points out that current high end systems waste in average 20% of its computing capacity on failure and recovery. The paper outlines possible research in order to bring this number down to 2%. The DOD/DOE report issued from 2009 [17] identifies resilience as a major emerging issue for HPC. It proposes research in five thrust areas: theoretical foundations, enabling infrastructure, fault prediction and detection, monitoring and control and end-to-end data integrity. The paper published in the International Journal of High Performance Computing Applications in 2009 [18] describes the challenges resiliency faces in the exascale era and possible directions in order to address these needs. The DOD/DOE report [2] issued in 2012 identifies six high priorities: fault characterization, detection, fault-tolerant algorithms, fault-tolerant programming models, fault tolerant system services and tools. The DOE workshop from 2012 [3] describes the required HPC resilience for critical DOE mission needs and details what HPC resilience research is already being done at the DOE national labs and what is expected to be done by industry and other groups. Also, the workshop focused on determining what fault management research is a priority for DOE's Office of Science and NNSA over the next five years. The exascale report from March 2013 [15] gathered the main points discussed at the workshop organized by the Institute for Computing Sciences on August 2012. The report analyzes the state of resiliency for HPC and proposes three design approaches: 1) business as usual where the global checkpoint/restart is used; 2) system-level resilience where vendors do not provide sufficiently low SDC rates at an acceptable acquisition and operation cost and a combination of hardware and software technologies is needed to hide the increased failure rates from the application; 3) application-level resilience for which there is an assumption that application codes will need to be modified in order to handle the increased failure rate. The paper makes recommendations for each alternative in order for them to become solutions for future systems.

The International exascale Software Project (IESP) Workshop [19], held in Kobe, Japan on the 12th-13th of April 2012 discusses what will be the major obstacles that the climate community will face at exascale and proposes and evaluates possible ways to overcome these obstacles. The focus of the workshop is on node-level performance, scalability and resilience.

Continuous availability of HPC systems has become a primary concern with the continuing increase of system size. Understanding the behavior of failures in current systems is increasingly important in order to design more reliable systems. To this extent, failure data analysis of current HPC systems can serve three purposes. First, it can highlight dependability bottlenecks and might serve as a guideline for designing more reliable systems in the future. Second, real data can be used to drive numerical evaluation of performance models and simulations, which are an essential part of reliability engineering. Third, these models can be used to predict resource availability, which is useful for characterization and scheduling. The knowledge of resources reliability can reduce performance loss due to unexpected failures, and QOS (Quality of Service) either by Reliability-aware Scheduling, where a system schedules jobs with different sizes on nodes based on the node's reliability or by Reliability-aware Checkpointing, where the optimal checkpointing interval can be computed based on the reliability of a set of nodes.

There are several papers that study the statistics of reliability/failure data, including the root cause of failures, the mean time between failures, and the mean time to repair. Work on characterizing failures in computer systems differs in the type of data used; the type and number of systems under study; the time of data collection; the system's maturity (time since installation); and the number of failure or error records in the data set. Most of these statistics are based on reliability, availability and serviceability (RAS) data mainly provided by major HPC laboratories in the USA: Los Alamos National Laboratory(LANL), National Energy Research Scientific Computing Center (NERSC), Pacific Northwest National Laboratory, Sandia National Laboratory (SNL), Laurence Livermore National Laboratory (LLNL), and National Center for Supercomputing Applications (NCSA).

Most studies divide failures into two categories: software and hardware, each having different sub-categories. Reliability monitoring and analysis considers failures that affect a single node; failures affecting a group of nodes; failures that may affect applications or important services; and operator er-

Hardware failures	Software failures
Failures that affect group of nodes	
Interconnect; Power Supply	Scheduler; FS; Cluster Management Software
Individual node failure	
Processor; Memory; Mother Board; Disk	OS; Client Daemon

Table 2.1: Failure categories used in the HPC field

rors. Table 2.1 presents examples of what is used in literature.

Table 2.2 presents a summary of the papers presenting failure statistics for different machines. Depending on the study and on the analyzed system, the table shows a wide range of results. There is not a consistent main root cause of failures among all systems, nor a consistent MTBF or MTTR.

Some of the studies give additional information to what is presented in the table. Schroeder et. al. [24], demonstrates that the number of failures per processor in different systems is rather stable over the course of time interval between 1996 to 2004. They also find that average failure rates differ wildly across systems, ranging from 20-1000 failures per year, mainly because of their size. The normalized failure rates show significantly less variability, in particular across systems with the same hardware type.

In a paper from 2011, Zheng et. al. [25] analyze the Blue Gene/P at Argonne National Laboratory and by using both RAS and job logs, they filter out failures that do not affect any jobs. By characterizing only the failures that lead to application interruptions they make a couple of interesting observations which might influence the fault tolerance protocol used by different applications. For example they observe the probability of job interruption is high if there exist historical records of application-related interruptions. Moreover, most application errors tend to be reported in the first hour. Therefore it is inadvisable to introduce checkpointing early in the execution period if the job has historical records of application-related interruptions. The failures remaining after filtering follow a Weibull distribution as do all failures analyzed together. Although both distributions have decreasing hazard rates, the value of the distribution's shape parameter after the job-related preprocessing is much higher than that for all failures. The MTBF after job-related filtering is about three times larger than that

System	MTBF	Root cause analysis	Citation
A cluster of 12 SGI Origin 2000 (1500 CPUs) A PC cluster (1000 CPUs) A cluster of 162 Itanium dual CPUs	MTTI of 1 day, less than 1 hour and about 6 hours respectively	Software was at the origin of most outages (59-84%)	[20] 2005
22 HPC systems including a total of 4,759 machines and 24k processors from LANL during 1995-2005	Around 8h	Over 50% of failures due to hardware, software only around 20%	[21] 2007
Blue Gene/L during 6 months	More than 10h	-	[22] 2008
Blue Gene/L (131k CPUs) Red Storm (11k CPUs) Thunderbird (9k CPUs) Spirit (1k CPUs) Liberty (512 CPUs)	-	Software caused 64% of failures, while hardware only 18%	[23] 2007
22 different systems at LANL, mostly large clusters of SMP and NUMA nodes, over a period of 10 years	-	Hardware is the main cause of failures (from 30% to more than 60%). Software is the second largest contributor, with 5% to 24%	[24] 2007
Blue Gene/P	Job level MTBF is three times larger than that system level MTBF	-	[25] 2011
Blue Gene/L, Blue Gene/P, SciNet, Google	-	Over a third DRAM errors are hard errors, most commonly in the form of repeating errors on the same physical address	[26] 2012
Platium, Titan	6h	Software represent the cause of most failures with 84% for Platium and 60% for Titan	[27] 2013

Table 2.2: Different studies and their results on failure characterization

without job related filtering.

Tsai et. al. [28] present a study that uses data collected from a population of over 50,000 customer deployed disk drives to examine the relationship between soft errors and failures, in particular failures manifested as hard errors. They observe soft errors alone cannot be used as a reliable predictor for all hard errors. However, in those cases where soft errors do accurately predict hard errors, sufficient warning time exists for preventive actions.

In [26], Hwang et. al. analyze data on DRAM errors collected on a diverse range of production systems in total covering nearly 300 terabyte-years of main memory. The authors provide a detailed analytical study of DRAM error characteristics, including both hard and soft errors and show that a large fraction of DRAM errors in the field can be attributed to hard errors. The paper also provides a detailed analytical study of their characteristics and proposes future directions to increase the reliability of systems.

After observing that each node may have a different failure distribution, the study from [1] uses the failure trace obtained from prominent HPC platform to study and compare how well they are distributed by different distributions, such as Exponential, Weibull and Lognormal. Their results indicate that Weibull distribution results in a better reliability model and that if node failures possess a degree of dependency, the system becomes less reliable.

Recently, systems have started to have heterogeneous nodes, which may have different failure rates. The study from 2013 [1] has shown that even homogeneous nodes present different rates. Figure 2.1, taken from this paper, shows an average MTTF per node of 7,514 hrs with a variance of 23,110 hours depending on the node ID that is analyzed. Thus, failure rate and reliability varies with location.

Most studies that analyze distribution fitting show that the Weibull distribution to be a good fit [24, 23, 29, 30]. In [24] the authors takes into considerations two types of view: the first as seen by an individual node in which they study the time between failures that affect only this particular node and the second as seen by the whole system in which they study the time between subsequent failures that affect any node in the system. They found that failures that affect the whole system best fit the Weibull (with a hazard rate function decreasing, having the Weibull shape parameter of 0.78) and gamma distributions, while the lognormal and exponential fits are significantly worse. Individual node failures are best fit by the lognormal

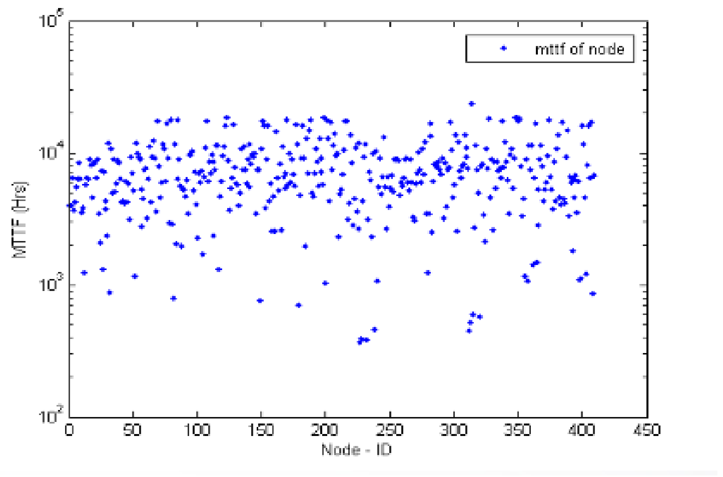


Figure 2.1: MTTFs of identical nodes over 4 year time. Image from [1]

distribution, followed by the Weibull and the gamma distribution. Two studies [21, 31] report correlations between workload and failure rate where [21] reports a correlation between the workload intensity and the failure rate. In our work published at SC 2011 [32] we show that the failure distribution changes depending on the failure type under study. We present some of these results in the following chapter.

Three of the studies analyze the mean time to repair [22, 24, 21]. In [24] the authors discovered that the lognormal distribution is the best fit, both visually as well as measured by the negative log-likelihood. The Weibull distribution and the gamma distribution are weaker fits than the lognormal distribution, but still considerably better than the exponential distribution. As expected the exponential distribution is a very poor fit given the high variability in the repair times. Both [24, 21] conclude that hardware type has a major effect on repair times. While systems of the same hardware type exhibit similar mean and median time to repair, repair times vary significantly across systems of different type. Unfortunately the study reports that repair times has different values for different types of system without characterizing this difference. In [33] , the authors analyze the Blue Waters system and show that failures with software root causes were responsible for 53% of the total node repair hours, although they constituted only 20% of the total number of failures. Hardware root causes, however, despite causing 42% of all failures, resulted in only 23% of the total repair time.

2.2 Failure detection methods

There are several general methods used in the literature in order to detect failures, from modeling the hardware system and finding outliers in this model, to implementing fault detection at the application and programming model levels. We will briefly present a few examples from each method.

One method used extensively in the past was to measure the system behavior and compare it to the expected normal behavior. An event is categorized as a failure in case of a significant deviation. Most examples from this category use performance metrics [8, 34]. Zheng et. al. [8] record system performance metrics every interval from various components in the system, and then aggregate all system information into a single large matrix. By normalizing and performing principal component analysis (PCA) they are able to determine anomalies.

Another similar methodology models the components and their interactions and then monitor the model. Most examples are using pattern recognition [35, 36] algorithms to model the system. Others include context free grammars [37] and mathematical equations [38]. Yamanishi et. al. [36] focuses on network failure symptom detection and event correlation discovery by modeling the system with a Hidden Markov Model (HMM). Chen et. al. [37] uses the runtime paths that requests follow as they move through the system as their core abstraction based on which they characterize component interactions. Automated statistical analysis of multiple paths allows to detect and put a diagnosis on complex failures. In [38] the authors apply deterministic function approximation techniques to characterize the functional relationships between the target function and some work metrics as input data. The focus is to develop deterministic models for approximating the leading indicators of aging and an automated procedure for statistical testing of their correctness. They automated modeling can be applied in server-type applications whose performance degrades depending on the work done since last rejuvenation, for example the number of served requests. In [39], the authors present an online failure detection method for clouds by using metric distributions rather than individual metric thresholds. The study uses entropy as a measurement that captures the degree of dispersal or concentration of such distributions, aggregating raw metric data across the cloud stack to form entropy time series. Their algorithm has three phases:

metric collection, entropy time series construction and entropy time series processing: spike detection, signal processing or subspace method in order to find anomalous patterns which are indications of anomalies in monitored systems.

A current approach for detecting node failure is to use heartbeats as a way of constantly monitoring a system. There are hardware health monitoring methods (e.g., IPMI an open standard hardware management specification that defines a set of common interfaces to hardware and firmware). For example, some network failures partition the file system into two or more groups of nodes that can only see the nodes in their group. These types of failures can be easily detected through a hardware heartbeat protocol. Software health monitoring systems [40] are also implemented on several large-scale systems by using timeouts to detect node problems.

Rani et. al. [40] propose a fault tolerant approach that provides the ability to detect and self-recover parallel runtime environment in cases of compute node failure. Their solution consists of a lightweight heartbeat protocol (BHB) that addresses the scalability issues in system monitoring and failure detection. Their focus are common fault tolerance issues in large scale systems, especially due to permanent component failure. Varma et. al. [41] develops a scalable approach to reconfigure the communication infrastructure after node failures. Their solution is a decentralized (peer-to-peer) protocol that maintains a consistent view of active nodes in the presence of faults. The failure detection method is based on a time out mechanism.

Application level failure detection is in general used for large servers or for large web application. For example, in [42], the authors present an application-generic framework for using statistical learning techniques to detect and localize likely application-level failures in component-based Internet services. In the HPC field, Hoefler et. al. [43] proposes a study of generic fault detection capabilities at the MPI level. The authors implement multiple detectors at various layers of the software stack: at the MPI communication layer and a separate one as stand-alone processes across nodes.

2.3 Prediction methods

Over the years, approaches on failure prediction have been developed in relation to reliability theory and preventive maintenance [44, 45, 24]. Models evolved by trying to incorporate several factors into the distribution, for example the manufacturing process [46] or code complexity [47]. However, all these methods are tailored to long-term predictions and do not work appropriately for online failure prediction.

More recent methods for short-term failure prediction are typically based on runtime monitoring as they take into account a current state of the system. There are two levels of online failure prediction in literature: component level and system level failure prediction. The first level assumes methods that observe components (hard drive, mother board, DRAM, etc) with their specific parameters and domain knowledge and define different approaches that give best prediction results for each [26]. One example of this type of approach is to compare the execution of good components with failed ones. A couple of studies from different fields that fit in this category are [7, 48]. For the HPC community, one example is [8] in which matrices are used to record system performance metrics at every interval. The algorithm afterwards detects outliers by identifying the nodes that are far away from the majority. Another example is [49], where the authors implement their own data collection module that gathers relevant data across the system and assembles them into a uniform format. In the second step they apply two feature extraction techniques, principal component analysis (PCA) and independent component analysis (ICA) to generate matrices with lower dimensionality; and in the last step the nodes that are far away from the majority are determined and considered potential anomalies. Their data mining algorithm is specifically designed for HPC systems and the results are different than previous studies.

The second level is represented by system level failure prediction, in which monitoring daemons observe different system parameters (system log, scheduler logs, performance metrics, etc) and investigate the existence of correlations between different components. In the last couple of years, a significant number of papers focus on providing predictions by analyzing different HPC systems. However, most predictors are able to use the information extracted in the training phase for only short prediction span after which a new train-

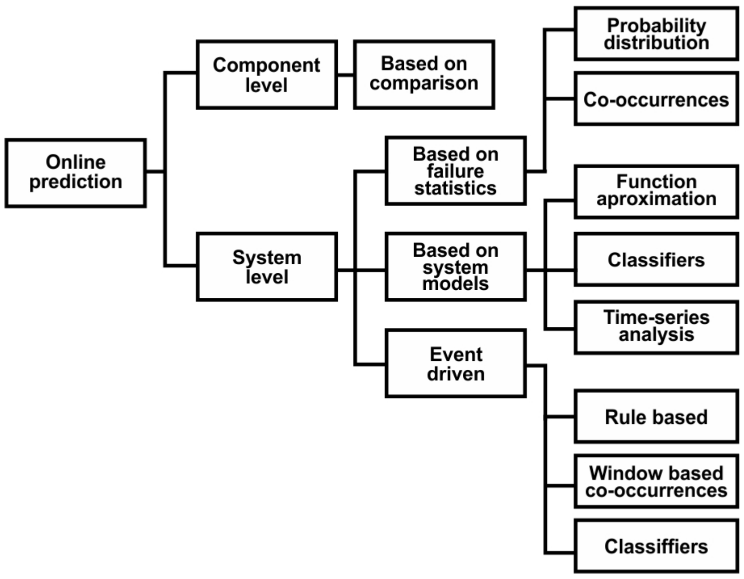


Figure 2.2: Online failure prediction taxonomy

ing phase is required. For example, [50] is using almost 3 months of training for predicting only half of month of execution. When dealing with real long time execution of a HPC system, the results of this type of prediction are unknown and can become unusable for real large-scale applications.

The taxonomy we will be using is presented in Figure 2.2. System level failure prediction has several categories:

1. Based on Failure Statistics. The basic idea of failure prediction based on failure statistics is to draw conclusions about upcoming failures from the occurrence of previous failures. This may include the time of occurrence as well as the types of failures that have occurred. The two sub-categores includes: Probability Distribution Estimation and Co-Occurrence. Prediction methods belonging to the first category try to estimate the probability distribution of the time to the next failure from the previous occurrence of failures. For the second type of failure predictors uses the fact that system failures can occur close together either in time or in space (e.g., at proximate nodes in a cluster environment). This can be exploited to make an inference about failures that might come up in the near future.

2. Based on System Models. The motivation for analyzing periodically

measured system variables such as the amount of free memory or CPU usage in order to identify an imminent failure is the fact that some types of errors affect the system even before they are detected. The key notion of failure prediction based on monitoring data is that some errors can be grasped by their side-effects on the system such as exceptional memory usage, CPU load, disk I/O, or unusual function calls in the system. These side-effects are called symptoms. Symptom-based online failure prediction methods frequently address non-failstop failures, which are usually more difficult to grasp. The following subcategories are included in this category: function approximation that refers to mimicking a target value, which is supposed to be the output of an unknown function of measured system variables as input data (this includes stochastic models, regression and machine learning); classifiers where failure prediction is achieved by classifying whether the current situation is failure-prone or not (this includes for example Bayesian networks); and time-series analysis where sequences of monitored system variables are treated as time series and time-series analysis is used in order to predict outlier moments in the series.

3. Event Driven Failure Prediction. Failure prediction approaches that use error reports as input data have to deal with event-driven input data. This is one of the major differences to system model failure prediction that uses symptom monitoring-based approaches, which in most cases operate on periodic system observations. Furthermore, symptoms are in most cases real-valued while error events mostly are discrete, categorical data such as event IDs, component IDs, log messages, etc.

2.3.1 Prediction based on failure statistics

In order to estimate the probability distribution of the time to the next failure, non-parametric methods as well as Bayesian predictors have been applied. In [51], the authors investigate reliability prediction by analyzing a decade of field data made available by Los Alamos National Lab. They focus on investigating the impact of factors, such as the power quality, temperature, fan and chiller reliability, system usage and utilization, and external factors, such as cosmic radiation, on system reliability. They observed that some types of failures increase the likelihood of follow-up failures more than others and that this information can be used for creating effective failure prediction models based on root cause distribution.

Bayesian failure prediction has the goal of estimating the probability distribution of the next time of failure by benefiting from the knowledge obtained from previous failure occurrences in a Bayesian framework [52, 53]. In [53], the authors use a mixture model of naive Bayes clusters trained by the using expectation-maximization algorithm in order to predict disk failures.

Another paper [54] uses Bayesian statistics to develop an anomaly detection/prediction system that employed naive Bayesian networks to perform intrusion detection on traffic bursts. Their model has the capability to potentially detect distributed attacks in which each individual attack session is not suspicious enough to generate an alert.

Due to sharing of resources, system failures can occur close together either in time or in space (at a closely coupled set of components or computers).

It has been observed several times that failures occur in clusters in a temporal as well as in a spatial sense. Liang et. al. [55] choose such an approach to predict failures of IBMs BlueGene/L from event logs containing reliability, availability and serviceability data. The key to their approach is data preprocessing employing first a categorization and then temporal and spatial compression. Temporal compression combines all events at a single location occurring with inter-event times lower than some threshold, and spatial compression combines all messages that refer to the different locations within a certain time window. Prediction methods are rather straight-forward: Using data from temporal compression, if a failure of type application I/O or network appears, it is very likely that a next failure will follow shortly. If spatial compression suggests that some components have reported more events than others, it is very likely that additional failures will occur at that location.

Fu and Xu [56] further elaborate on temporal and spatial compression and introduce a measure of temporal and spatial correlation of failure events in distributed systems.

Another example of current fault predictor that is using a co-occurrence method is [57] where the authors compare two failure prediction approaches and study the influence that the observation window has on the results. In [9], the authors use a meta-learning predictor to chose between a rule-based method and a statistical method depending on which one gives better predictions for a corresponding state of the system. Another approach for analyzing the logs is given in [58], where the authors investigate both usage and failure logs.

A different approach is given in [59] and [60], where the authors investigate parameter co-occurrences between different application log messages for extracting dependencies among system components. The authors mine dependencies from the tuple-form representations of the log messages looking for patterns that could indicate a failure in the system that prevented tasks from completing.

2.3.2 Prediction based on system models

One frequently used method is represented by regression techniques where parameters of a function are adapted such that the curve best fits the measurement data, e.g., by minimizing mean square error. The simplest form of regression is curve fitting of a linear function.

In [38], the authors apply deterministic function approximation techniques such as splines to characterize the functional relationships between the target function and input data. Deterministic modeling offers a simple and concise description of system behavior with few parameters.

Pattern recognition techniques operate on sequences of error events trying to identify patterns that indicate a failure-prone system state. The most used method for pattern recognition is by far Markov chain models. The approach is based on the assumption that failure-prone system behavior can be identified by characteristic patterns of errors.

In [61] the authors propose to use hidden semi-Markov models (HSMM) in order to add one additional level of flexibility to the theoretical method proposed in [35]. Two HSMMs are trained from previously recorded log data: One for failure and one for non-failure sequences. Online failure prediction is then accomplished by computing likelihood of the observed error sequence for both models and by applying Bayes decision theory to classify the sequence (and hence the current system status) as failure-prone or not.

The second step implemented by [62] uses two semi-Markov models that quantify the reliability of a node in the overall system. In the process the method identifies nodes that tend to be the source of a large number of failures and predicts the reliability of these nodes. The first discrete-time semi-Markov model is built for each system where state transitions are driven by functions derived from the distributions fitted to the result of the neural-gas filtering analysis. The second semi-Markov process computes transaction probabilities and event arrival rates directly from event observations.

[56] built a neural network to approximate the number of failures in a given time interval. The set of input variables consists of a temporal and spatial failure correlation factor together with variables, such as CPU utilization or the number of packets transmitted by a computing node.

Murray et. al. [63] have applied the Support Vector Machines (SVM) method in order to predict failures of hard disk drives. In the case of hard disk failure prediction, five successive samples of each selected SMART attribute set up the input data vector.

Several time series models are employed to model stationary as well as non-stationary effects. One example in this category is Sahoo et. al. [9] where the authors applied various time series models to data of a 350-node cluster system to predict parameters like percentage of system utilization, idle time and network IO.

2.3.3 Event driven prediction

Failure prediction methods in this category analyze the events generated by the system and derive a set of rules/patterns/correlations between different events. In general, the rules express temporal ordering of events in the form "if errors A and B occur within x seconds, then error C occurs within y seconds with probability of P%". Several parameters such as the maximum length of the data window, types of error messages, and ordering requirements had to be per-specified.

The event-set method has been applied in [57], by using a three-phase failure predictor for the Blue Gene/L systems: event preprocessing where the raw RAS log is cleaned and categorized; the base prediction phase where different base learning methods are applied on the preprocessed log to identify fault patterns and correlations; and the meta-learning phase where meta-learning is explored to adaptively integrate multiple base predictors to boost prediction accuracy. Similarly, [9] uses a filtering system close to [57] in that it uses a fixed time window for event grouping. The method consists of 2 steps: a preprocessing step that converts syslogs into a data set that is appropriate for running classification techniques by extracting a set of features. These features can accurately capture the characteristics of failures. In the next step it applies different classifiers on the data (a rule-based classifier, Support Vector Machines and a customized Nearest Neighbor method).

In [31] the authors propose to cluster failure events based on their correlations using an in-depth understanding of the cause of failures and their empirical and statistical properties. The authors use a failure signature that captures the system performance metrics associated with a failure event.

The papers presented in this category have one major characteristic in common: they all cluster the events in classes through different methods by incorporating a training phase. Most of the papers use a predefined number of classes; usually that number is defined by the system administrator. Since log files can change during the course of a system's lifetime and novel errors may appear, the number of classes may need to change in time. In [62] the authors propose a more realistic clustering method that groups events automatically. After the clustering phase, many papers use different filtering techniques. All the presented methods obtain good result for their own validation set.

Extensive research has been focused on using system logs, scheduling logs, performance metrics or disk usage logs in order to extract a correlation between events generated by any component of a system. There are numerous methods, starting with simple brute force extraction of rules between non-fatal events and failures [6] with more sophisticated techniques. In [9], the authors are using a meta-learning predictor to chose between a rule-based method and a statistical method depending on which one gives better predictions for a corresponding state of the system. Other research include SVM [10], hidden-Markov chains [11] or Bayesian networks [12]. A slightly different approach is given in [59] where the authors investigate parameter correspondence between different application log messages for extracting dependencies among components.

In general, current state-of-the-art research is using some kind of data mining algorithms extracting patterns that might lead to failures [57, 50, 9, 58, 55]. Most of these algorithms are using the same workflow: they group the messages in the log file into categories, filter redundant events both in time and space, extract correlations between events based on the small filtered set

System	Method	Precision	Recall	Lead time (s)	Ref
BGL	Rule-based	0.7	0.3-0.4	5 min	[55]
BGL	Statistical	0.5	0.48	-	[9]
BGL	Multiple	0.9	0.7	-	[9]
BGP	Rule-based different lead time	0.4/0.4/0.35	0.8/0.7/0.6	0/300/600	[50]
BGP	Rule-based	0.5	0.5	30 min	[57]

Table 2.3: Prediction results for different state-of-the-art related work

of log messages and in the end use the correlations to predict future events or failures. Each workflow step introduces imprecision or noise that influences the accuracy of the prediction.

Table 2.3 presents the prediction results for the most successful studies in literature up to date. At a first glance, the results presented in this table seem good enough to be used for the construction of failure avoidance techniques. However, these results are obtained either by using long training phases for only a couple of days of prediction, or not considering the lead time between when the prediction is done and when the failure occurs. Considering some preventive actions could take several minutes, the execution time of acting upon a prediction could sometimes exceed the lead time provided. Moreover, all the presented methods do not provide any location information. This makes it impossible for proactive methods to know which application processes should be migrated. Predictions with location information will enable checkpointing data only on those failure-prone components, thereby avoiding application-wide checkpointing which is significantly time consuming.

2.4 Checkpointing challenges

Most predictive methods offer small lead time windows for proactive actions to be taken. To be compatible with short term prediction, checkpointing has to be significantly improved. One promising direction is multi-level checkpointing. There are currently two environments providing multi-level Checkpoint/Restart: SCR (Scalable Checkpoint/Restart) [64] and FTI (Fault Tolerance Interface) [65]. Recent results show that a process context of 1GB can be saved in 2-3 seconds in local SSD (i.e. 2 SSD mounted in RAID0). Such checkpoint speed is orders of magnitude faster than checkpointing on a remote file system which requires tens of minutes in current petascale systems and may require several hours in projected exascale systems. An experiment with FTI on a large scale execution (1/2 million GPU cores) of an earthquake simulation on a hybrid system composed of CPU and GPUs demonstrated very low overhead on the execution time (i.e. less than 10%) when using such checkpoint strategy, compared to no fault tolerance. Other research results demonstrate that checkpointing on remote node memory is even faster than on local HDD or SSD [66]. These results demonstrate that proactive checkpoints can be taken even with a few seconds before the predicted failure happens. However, proactive checkpointing introduces a whole new dimension with several challenges:

- To decrease the checkpoint size and maximize efficiency many applications rely on user-guided checkpointing, in which domain experts specify points in the code where to checkpoint, so that the amount of data that need to be saved is minimal. However, upon a failure prediction, the checkpoint is triggered by the prediction runtime and the application may be in the middle of a complex kernel execution that requires a high memory footprint. Thus, how to combine user-guided checkpointing with proactive checkpointing?

- Furthermore, it is important to remember that the application still needs to restart after the failure and produce correct results. This is the classic checkpointing coordination problem that may imply the use of a fault tolerant protocol. In application level checkpointing, the coordination is implicit, while in system level checkpointing capturing the state of the execution is explicit and relies on a fault tolerant protocols. If the approach relies on coordinated checkpointing or on hierarchical fault tolerant protocols [67], the coordination (global or partial) should be fast enough to store the state of the application before the failure occurs.

Any proactive checkpointing implementation that does not provide high performance solutions for of these two problems will not work efficiently for large HPC systems. In chapter 6 we will present the solutions we propose and how we implement them in order to create a runtime framework that efficiently couples online failure prediction, ultra-fast proactive checkpointing and periodic multi-level checkpointing. As far as we know, this is the first implementation of such a hybrid protocol to date. On the other hand, there are several theoretical studies that propose to combine classic periodic checkpointing with proactive fault tolerance actions in order to study the theoretical benefit of such approaches.

One such example is presented in Aupy et. al. [68], where the authors propose a fault tolerance strategy that uses the prediction alerts to compute an optimal checkpointing interval. In their follow-up work [69], the authors assume that the fault-prediction systems that do not provide exact prediction dates, but instead time intervals during which faults are predicted to strike, with different probabilities at each moment of time.

Li et. al. [70] consider a different prediction model that provides a probability of failure when the application ask for a prediction. Moreover, They consider a specific application model where proactive checkpoints or migration can be performed at a predefined location during the execution.

Cappello et. al. [71] proposed two proactive fault tolerance strategies, both relying on a perfect prediction mechanism. The perfect prediction mechanism is supposed to have a 100% recall, 100% precision and enough lead time to perform either checkpointing or migration. Even though the scenario is not realistic since there is no prediction method that can offer these results, it shows the trade-off of combining prediction either with checkpointing or with migration.

In chapter 6, we will use the model in [72], as well as create our own based on [71] with the actual data from our hybrid protocol implementation in order to study the benefit of our approach.

Chapter 3

HPC systems description

The main focus of our analysis are logs generated by previous and current large scale systems. In chapter 7 we also incorporate performance metrics into our analysis. We analyzed several HPC systems: Mercury, a previous-generation cluster at the National Center for Supercomputing Applications (NCSA) [73], several systems from the Los Alamos National Laboratory (LANL) [24], a Blue Gene/L machine [74], the Blue Gene/P system [50] deployed at the Argonne National Laboratory and Blue Waters [75], currently the biggest supercomputer at NCSA. Mercury and Blue Waters logs are owned by the NCSA and are not available to the public because of privacy issues. The LANL and Blue Gene traces are open and are downloaded from the USENIX Computer Failure Data Repository [76]. Details regarding the system's characteristics are shown in Table 3.1. Additionally, details about the sources that generate notifications for the Blue Water system can be found in Table 3.2.

Log messages generated by all systems contain two separate parts, a message header and body. The header contains information on the component in the system that generated the notification, the timestamp and the affected location in the system. Different systems have different header information. For example, Blue Gene systems contain a severity field that is not present in the any of the Cray systems. We manually created an entry pattern for

System	Events/Day	Total Event Types
Blue Waters	15GB (88mil events)	10,499
BlueGene/L	5.76MB (25,000 events)	186
BlueGene/P	8.12MB (120,000 events)	252
Mercury	152.4MB (1.5mil events)	563
LANL systems	433,490 in 5 years	53

Table 3.1: Log file statistics

Source	Events/Day	Total Event Types
Syslog	8GB (50mil events)	3,852
HPSS	1MB (900,000 events)	358
Sonexion	3.5GB (10mil events)	3,112
Moab	500 MB (15mil events)	725
ESMS	3GB (12mil events)	2,452

Table 3.2: Log file statistics for different sources for the Blue Waters system

Resource	Phase 1	Phase 2
Production	Jan 2004	2Q 2004
Number of Nodes	256	635
Processors	2x Itanium II @ 1.3 GHz	2x Itanium II @ 1.5 GHz
Memory	4 or 12 GB DDR1600 ECC RAM	4GB DDR2100 ECC RAM
Filesystem	GPFS (60TB)	GPFS (170TB)
Storage	1x18GB, 1x73 GB UltraSCSI drives	2x73 GB UltraSCSI drives
Network	Gigabit Ethernet, Myrinet, Management Network (Ethernet)	

Table 3.3: Characteristics of the Mercury System

each system stating the type of data and where in the message the body begins. We modify all the logs for each system into a common format. This new structure log contains the message timestamp, location, facility for the header information and the message description in the body.

3.1 Mercury

The Mercury cluster was a production high-performance computing system at the National Center for Supercomputing Applications used for scientific applications as part of TeraGrid over a 5-year period, from January 2004 to March 2010, with roughly 98% uptime over its lifetime. During its operation, it ran millions of parallel computing jobs for hundreds of researchers in fields ranging from molecular and fluid dynamics simulation to DNA and gene expression analysis.

Detailed technical information regarding the cluster is shown in Table 3.3.

The cluster started with 256 compute nodes, half having large amounts of memory (12GB). In the second quarter of 2004, an additional 635 compute nodes were added with faster processors. Over time, system components were replaced due to failure and nodes repurposed to/from computing or storage purposes. We tracked between 936 to 1050 nodes actively reporting log messages over the lifetime of the cluster - this includes compute, login and storage nodes.

We analyzed logs generated by 10 months of production in the second Phase of the machine from February 2006 to December 2006. Each compute node consisted of two Itanium processors running at 1.3 or 1.5 GHz with 4 or 12 GB ECC protected memory. Login and storage nodes had roughly similar specifications. Storage was a combination of a network file system and local hard disks serving as mount and scratch devices, as well as a wide-area file system using AFS. The AFS system was generally not used directly by applications, so we omit it from our analysis. High speed I/O was handled by a GPFS file system connected by fiber channel. The Mercury system contained three separate networks - a Gigabit Ethernet network for computation, a high speed Myrinet network for latency sensitive parallel applications, and a management network for node maintenance and software updates. By default there was no checkpointing or fail-over mechanism for applications running on the cluster. Each application was responsible for managing its own fault tolerance.

Logs from each node were collected centrally with each log message being sent as a single packet to avoid truncation. Some events generated multiple messages which could be interleaved in the logs with messages from other machines. These logs contain the time of the message, node on which it occurred and possibly details regarding the application which generated it.

3.2 Systems at LANL

Los Alamos National Laboratory has collected data for 22 of their supercomputing clusters, for the duration of 5 years. They publish data from system logs and also information regarding failures that occurred in the system's lifetime. This data has been intensively analyzed in different papers in order to extract data statistics for failure distribution and root cause analysis

[25, 24].

Most of these systems are large clusters of either NUMA (Non-Uniform Memory Access) nodes, or 2-way and 4-way SMP (Symmetric Multi Processing) nodes. In total all systems together include 4,750 nodes and 24,101 processors. The data provided does not include vendor specific hardware information. Instead it uses capital letters (A-H) to denote a systems processor/memory chip model.

In general, systems vary widely in size, with the number of nodes ranging from 1 to 1,024 and the number of processors from 4 to 6,152. Different systems presents different hardware architecture and might not contain identical nodes. While all nodes in a system have the same hardware type, they might differ in the number of processors and network interfaces (NICs), the amount of main memory, and the time they were in production use.

A failure record contains the time when the failure started, the time when it was resolved, the system and node affected, the type of workload running on the node and the root cause identified manually by the system administrators.

Most workloads are large-scale long-running 3D scientific simulations, having long periods of CPU computation, interrupted every few hours by a few minutes of I/O for checkpointing and for visualization. A complete description of the system can be found in [24]. The clusters represent the oldest machines analyzed in this thesis and are used in comparison to the other systems to understand how HPC machines evolved over time.

3.3 Blue Gene systems

In this thesis, we looked at the fastest deployment of a Blue Gene/L machine and a Blue Gene/P system.

The Blue Gene/L system located at Lawrence Livermore National Laboratory (LLNL) boasts a peak speed of over 596 teraFLOPS and a total memory of 69 tebibytes. It is composed of 106,496 dual-processor compute nodes, produced in 130-nm copper IBM CMOS 8SFG technology. Each node is very simple, consisting of a single ASIC containing two processors and nine double-data-rate (DDR) synchronous dynamic random access memory (SDRAM) chips.

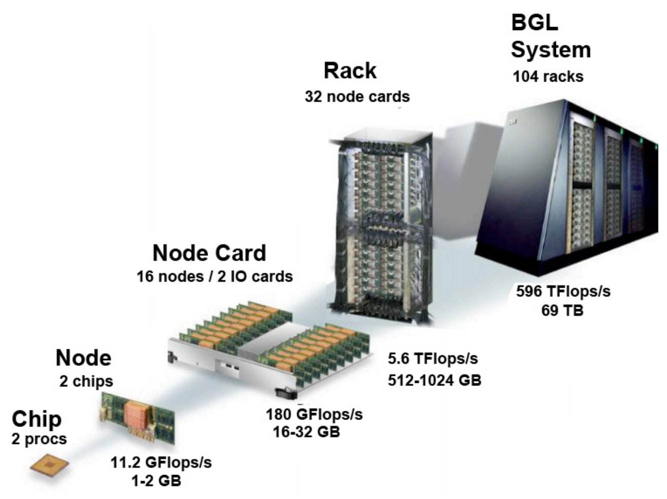
The nodes are interconnected through five networks, the most significant being configured as a 32x32x64 3D torus where each node is connected in six different directions for nearest-neighbor communications. This network handles the bulk of all communication. There are virtually no asymmetries in this interconnect; the nodes communicate with neighboring nodes that are physically close on the same board and with nodes that are physically far removed on a neighboring rack, with the same bandwidth and nearly the same latency. This allows for a simple programming model because there are no edges in a torus configuration. In addition, a global reduction tree supports fast global operations such as global max/sum, and multiple global barrier and interrupt networks allow fast task synchronization.

Each node contains two processors, which allows the system to have several running modes. For example, each processor can handle its own communication (which called virtual node mode), or one processor can be dedicated to communication and one to computation (communication co-processor mode).

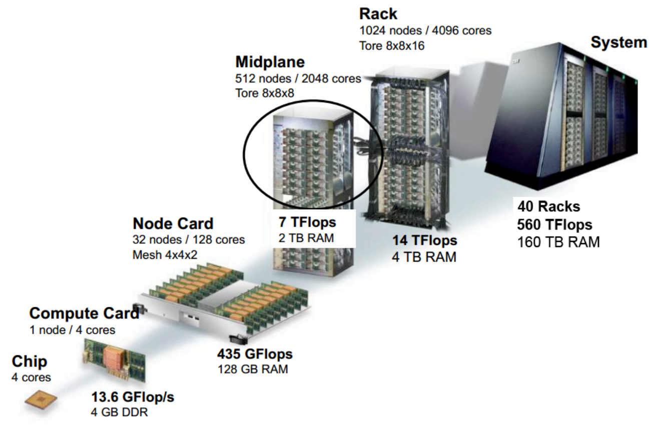
Out of the total 212,992 PowerPC cores, 67% contain 512 MB RAM and 33% contain 1 GB. The first-level caches (L1) are contained within the PPC440 core macro. The second-level caches (L2R and L2W) are very small and basically serve as prefetch and write-back buffers for L1 data. The third-level cache (L3) is large and is expected to provide high- bandwidth, low-latency access. It is shared by instructions and data.

All Blue Gene machines present the same architecture design, as shown in Figure 3.1 for LLNL's Blue Gene/L and ANL's Blue Gene/P. In the case of Blue Gene/L, each node contains 2 chips of 2 processors each, 16 nodes and up to 2 IO cards create a node card and a rack is composed of 32 node cards. Blue Gene/L consists of a total of 104 racks.

RAS (Reliability, Availability, and Serviceability) events are logged through the Central Monitoring and Control System (CMCS), and finally stored in a DB2 database. The logging granularity is less than 1 millisecond. If an individual node on the system fails, the primary RAS strategy is to isolate and replace the failing node while restarting the application from a checkpoint on a set of racks that does not contain the faulty node. Specifically, each 512-node rack is on a separate power boundary, with a separate power domain for the link chips, enabling it to be powered down without affecting any other racks. Once powered down, the card containing the faulty node can be replaced, and the rack can be restarted and brought online for the job scheduling software. Thus, a node failure can temporarily bring down a



(a) Blue Gene/L at LLNL



(b) Blue Gene/P at ANL

Figure 3.1: Blue Gene architecture

512-node rack.

More information about the system can be found in [74]. In this thesis, we analyzed a six month period from the system's production time, from June 3rd, 2005 to January 4th, 2006. During this time, there were 4,747,963 messages sent to the log for a total of 207 different event types. The system at LLNL went into production mode starting with June 2004, so our analysis is done outside the infant mortality phase.

The Blue Gene/P system that we analyzed, called Intrepid, is a 40-rack machine (40,960 nodes, 163,840 processor cores) deployed at the Argonne National Laboratory. The design of the Blue Gene/P represents an evolution from the Blue Gene/L system. Each Blue Gene/P Compute chip contains four PowerPC 450 processor cores, running at 850 MHz. The cores are cache coherent and the chip can operate as a 4-way symmetric multiprocessor (SMP). As for the Blue Gene/L system, the memory subsystem on the chip consists of small private L2 caches, a central shared 8 MB L3 cache, and dual DDR2 memory controllers. The chip also integrates the logic for node-to-node communication, using the same network topologies as Blue Gene/L, but at more than twice the bandwidth.

In the Blue Gene/P case, a compute card contains 4 cores and 32 compute cards form a node card. A midplane contains 16 node cards, 4 I/O cards (containing the I/O chips), and 24 midplane switches (through which different midplanes connect). Each rack consists of 2 midplanes, and a midplane is the granularity of job allocation. Intrepid has a total of 40 racks, each with a total of 1024 node cards. The system utilizes a total of 640 I/O nodes, contributing 7.6 PB of total disk space.

A compute card contains a Blue Gene/P chip with 2 GB DRAM. A single compute node has a peak performance of 13.6 GFLOPS. 32 Compute cards are plugged into an air-cooled node board. By using many small, low-power, densely packaged chips, Blue Gene/P exceeded the power efficiency of other supercomputers of its generation, and at 371 MFLOPS/W Blue Gene/P installations ranked at or near the top of the Green500 lists in 2007-2008.

Blue Gene/P integrates some degree of fault tolerance in the torus network by ensuring that packets are injected in a manner that forces them to avoid failed nodes; this requires non-minimal routing and can handle up to three concurrent failures in a partition provided they are not collinear. A bad node that still has a viable torus network interface can be left in the network. This provides a good solution for a system that has a high node-failure rate.

Additional error-detecting mechanisms that allow monitoring and isolating faults include the power-supply monitor module and additional link cyclic redundancy checks. A extended approach for fault isolation has been developed that allows the ability to detect and isolate failed components. More details about the Blue Gene/P system can be found in [77].

Intrepid was in production mode starting with June 2008 until it was decommissioned in December 2013. We analyzed nine months of activity, from Jan 2009 to Sept 2009, for a total of 1.9GB of generated events and 252 event types.

3.4 Blue Waters

The Blue Waters system is a Cray XE/XK hybrid machine at the National Center for Supercomputing Applications composed by a combination of XE6 nodes (two AMD Interlagos CPU modules - four processor chips) and XK7 nodes (one AMD Interlagos CPU module and one NVIDIA Kepler K20X GPU) connected by the Cray Gemini torus interconnect. Divided into 237 Cray XE6 and 44 Cray XK7 cabinets, Blue Waters contains over 25 thousand computing nodes, reaching a peak performance of 13.2 Petaflops and offering a total system memory of over 1.66 PB. The online storage gives 26.4 PB of total usable storage with an aggregate I/O bandwidth of over 1 TB/s. We used 5 major sources of logs: syslogs that contain usual RAS information, HPSS (High Performance Storage System) which is the near-line storage designed for moving large files and large amounts of data, Sonexion storage system used for storing the Lustre filesystem, Moab job scheduler and ESMS, the data system manager. Table 3.2 presents these sources and their characteristics.

Blue Waters high-speed network consists of a Cray Gemini System Interconnect. Each blade includes a network mezzanine card that houses 2 network chips, each one attached on the HyperTransport AMD bus shared by 2 CPUs and powered by 2 mezzanine dual-redundant voltage regulator modules (VRM). The topology is a three-dimensional (3D) 24x24x24 reentrant torus: each node has 6 possible links towards other nodes. Mapping location ids on the torus of a Cray system follows the algorithm described in figure 3.2. Every cube in the torus represents a Gemini hub (consecutive nodes). Two neighbor Gemini hubs on the OY axes create a slot, multiple consecutive slots on the OZ axes form a cage and multiple consecutive cages on the OZ axes create a cabinet. The cabinets are divided into two dimensions, first on the YOZ plane and then on the OX. For the Blue Waters system, there are 2 Gemini hubs in each slot, 8 slots form a cage and 3 cages create a cabinet. Each cabinet is as wide as the OZ portion of the torus, so the entire Gemini hub set is divided into 24 cabinets on the OX axes and 12 cabinets on the OY for a total of 288 cabinets.

Every node in the system is checked and managed by the Hardware Supervisor System (HSS). This system contains the HSS network, blade and cabinet controllers in charge of monitoring the nodes, replying to heartbeat

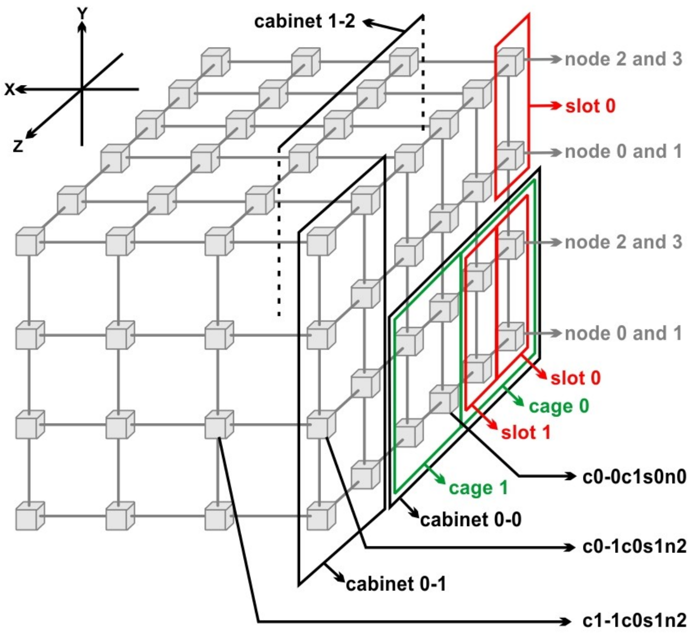


Figure 3.2: Cray system 3D torus network

signal requests and collecting data on temperature, voltage, power, network performance counters, runtime software exceptions. The system also contains a HSS manager in charge of collecting node health data and executing the management software. Upon detection of a failure, represented in most cases by a missing heartbeat, the HSS manager triggers failure mitigation operations. These mitigations include: i) warm swap of a compute/GPU blade to allow the system operator to remove and repair system blades with temporary performance differences of the workload; ii) service node and Lustre node failover mechanisms; and iii) link degradation and route reconfiguration to enable routing around failed nodes in the topology.

Compute and GPU nodes execute the lightweight kernel Compute Node Linux (CNL) developed by Cray. The operating system is reduced to minimize the overhead on the nodes and includes only essential components, such as a process loader, a Virtual Memory Manager, and a set of Cray ALPS agents for loading and controlling jobs. Service nodes execute a full-featured version of Linux, the Cray Linux Environment (CLE), which is based on the Suse Linux Enterprise Server 11 kernel 3.0.42.

All blades are diskless and use the shared parallel file system for IO op-

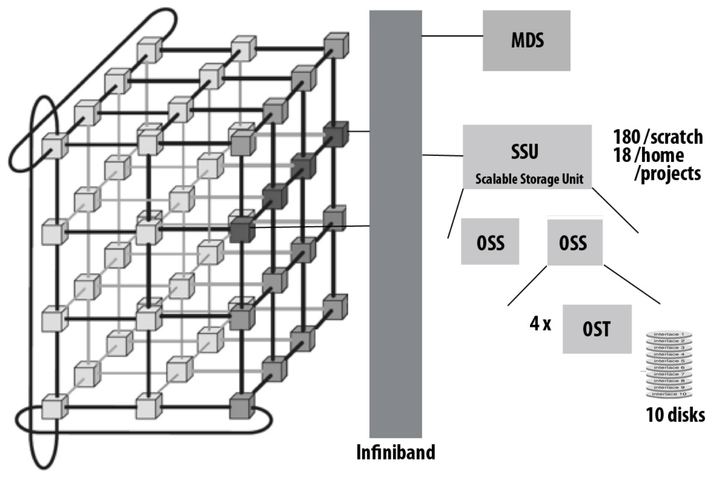


Figure 3.3: Lustre filesystem architecture

erations. Blue Waters hosts the largest Lustre installation to date. The parallel file system consists of Cray Sonexion 1600 storage modules. Lustre is designed as a client-server architecture, with many clients communicating with multiple I/O servers and one or more metadata servers. Figure 3.3 shows the architecture of the filesystem used by the Blue Waters system. Compute nodes are stored in a 3D torus with IO nodes being distributed among them. Applications use the closest IO node in order to access the data on different disks. The communication between the IO nodes and the disk is done through an Infiniband network. Blue Waters provides three different file systems: (i) /u storage for home directories; (ii) /projects storage for project home directories; (iii) /scratch high performance, high capacity transient storage for applications.

All three filesystems on Blue Waters are built using Cray Sonexion 1600 Lustre appliances that provide the basic storage building block for the Blue Waters I/O architecture and are referred to as a ”Scalable Storage Unit” (SSU). Each SSU is RAID protected and is capable of providing up to 5.35 GB/s of IO performance and around 120 TB of usable disk space. The /scratch file system uses 180 SSUs, each SSU containing 4 OSTs (Object Storage Target), each with 10 RAID disks. Blue Waters is using one MDS (Meta-Data Server) for each filesystem that is interrogated each time a file is created, opened or closed.

A Sonexion module has 2 SSD of 2 TB in a RAID 1 configuration for journaling and logging, 22 disks of 2 TB for metadata storage, and 80 disks of 2

TB for data storage, organized in units of 8 disks in RAID 6. All disks are connected to two redundant RAID controllers. In each unit, two additional disks serve as hot spares, which automatically provide failover for a failed drive. Each Lustre service node and Sonexion module is configured as an active-passive pair connected to a shared storage device. In this configuration, the passive replica node becomes active when the HSS detects a failure of a Lustre node; the shared storage device is then mounted in a read-only mode to avoid data inconsistency until the failed node has been replaced with the standby replica. After failure recovery, clients reconnect and replay their requests serially in order to reconstruct the state on the replaced node. Until a client has received a confirmation that a given transaction has been written to stable storage, the client holds on to the transaction (waiting on a timeout), in case it needs to be replayed. If the timeout is reached, all the jobs waiting to reconnect fail. The recovery process can take up to 5-30 minutes (60 for MDS), depending on the number of clients using the file system at the moment of the failure. More information about the system's architecture can be found in [78] as well a study about we the impact of this architecture on application I/O performance. Moreover, [79] presents the experiences observed during the deployment of Blue Waters, involving several steps of preparation, delivery, installation, testing and acceptance.

For the Blue Waters system, we had access to the job logs as well as system and failure logs so we were able to follow the propagation of failures from the hardware to the application level. There are numerous applications that are running on Blue Waters. Examples include earthquake engineering for which simulations want to capture seismic waves in 1Hz range, which is 256 times more computationally demanding than current simulations; cosmology applications that desire to model the first billion years after the Big Bang; epidemiology applications that model local and global disease outbreaks; tornado simulations where forecasters can identify conditions that make a tornado likely, they can pinpoint when and where they start, their path, and strength.

Information about failures is kept into a distinct failure log where Cray system administrators document the approximate timestamp for each failure and the possible cause. At the same time, system administrators from NCSA inspect the logs generated by the system and decide which messages represent failures. We correlated the two sources and only kept entries that appear in both. Since the Cray failure files are manually written, we correlated these failures with events from the logs by using a rather large window of 6 hours, starting 3 hours before the manually written timestamp occurrence. All messages from the log that were marked as suspicious in this timeframe were gathered in a file, after which, together with system administrators from NCSA, we filter out those that were not referring to the same problem. We mention that the Cray failure file has been already analyzed and validated by NCSA staff and their observations were published in [33]. For each failure entry, a failure type has been identified by using one of the following exclusive categories: hardware, software, missing heartbeat, network, environment, and unknown (failures for which the cause could not be determined). We use these categories when analyzing the results of our prediction. Overall, we analyzed 7 months of activity since September 2013, to February 2014.

Chapter 4

Analyzing the system behavior

Event logs are a rich source of information for analyzing the cause of failures in cluster systems. Together with performance metrics, they represent the main method used by system managers to understand the behavior of HPC systems. However, the size of these files has continued to increase with the ever growing size of supercomputers, making the task of analyzing log files a hard and error prone process when handled manually. The most common method used by system managers for searching through the log data is pattern matching, by comparing numerical thresholds or doing regular expression matching on vast numbers of log entries looking for each pattern of interest. By using this method, only those faults that are already previously known to the domain expert can be detected. Moreover, there is not a consistent normal behavior of system state data or how different components react to failures, within the system. For example, once a fault is triggered in the system, it can generate multiple events, that propagate within the system and that, consequently, generate multiple notifications in the log [32, 24]. Conversely, some components have the opposite behavior and stop generating events when a failure occurs. It is important to be able to capture the behavior of each event type and understand what is the normal behavior and how each failure type affects it. Systems experience software upgrades, configuration changes and even installation of new components during the course of their lifetime [80, 24]. This makes it difficult for the algorithms to learn patterns since the system will experience phase shifts in behavior.

There is considerable research on analyzing log files, for different purposes, from detecting outliers in the system [8] and being able to filter events referring to the same problem [81] to offering prediction for the state of the system [57, 9] or analyzing the root cause of a failure [82]. A widely used strategy for extracting information for all these algorithms is to correlate events. However, the correlation is most of the time a statistical observation about the occurrence of different messages. They do not take into consideration the diversity of events' behavior, and treat all notifications in the same way. Furthermore, most of the studies build their analysis on a preprocessing step where events are filtered to decrease the total size of the log. As shown in [83] these methods have limitations and can affect the overall performance of the analysis module.

We proposed, implemented and verified a novel way of analyzing the characteristics of each event described in the log file by considering its own trend and behavior. For this purpose we introduce signal analysis concepts in the context of HPC system. We present novel methods for shaping the normal and faulty behavior of each event and of the whole system in this chapter and used them to propose appropriate models for descriptive and forecasting purposes in the following chapters. After having an accurate model for each event we create a global view of the whole system by merging the information and correlating events. We will show that events are different one from another, and faults affect them in a different way.

4.1 Preprocessing

An event's notification message is constructed by using variables and constant words in order to describe a specific event. Constants are words that carry crucial information since they are in charge of describing the event type, while message variables identify manipulated objects or states for the event. For example, the notification "Connection from 192.168.10.6 port 25" has 3 constant words: "Connection", "from", "port"; and 2 variables: the ip address and the port number.

Extracting the message types from log files makes it possible to abstract the contents of event logs and facilitates further analysis and construction of computational and correlation models. Message type descriptions are the templates that preserve the constants in a message and replaces the variables with wildcards. For example, the line of C code that generates a network notification: sprintf(message, "Connection from %s port %d", ipaddress, portnumber); produces the following entries in the log file:

```
Connection from 192.168.08.1 port 25
Connection from 192.168.08.2 port 25
```

Connection from 192.168.08.1 port 80
We want to be able to retrieve the template "Connection from %s port %d" by just inspecting the log lines.

For this purpose, we developed HELO (Hierarchical Event Log Organiser), a tool for preprocessing log files. HELO parses event logs and identifies frequently occurring messages with similar syntactic patterns that represent different message types. These patterns are called templates and are basically regular expressions that try to mimic the source code that generated each notification. These templates can be used by system managers to set alerts for the occurrence of different message types. At the same time they can be used by analysis methods to find anomalies and detect failures. Table 4.3 presents several templates from different systems and the event they represent. The two template examples in the table that refer to the Blue Water system are used by the alert mechanism at NCSA to trigger notifications to system managers and others whenever these failures occur.

The tool has two different components: an offline classification part where message lines found in historic log files are used to create the initial template set by dividing them according to their description patterns; and an online clustering part that classifies each new event and dynamically reshapes the previous found templates accordingly.

As mentioned, a template represents a line of text where variables are represented by different wildcards. HELO uses three types of wildcards: d+ represents numeric values, * represents any other single words, and n+ represents strings of words that have a value for some of the messages and do not exist for others.

The current set of templates is kept in a radix tree [84] which makes the classification process efficient and capable of dealing with the messages generated by supercomputers even during storms of events. A radix tree, which represents a compact prefix tree, is a space-optimized structure in which each node with only one child is merged with its parent. The classical radix tree implementation was modified to deal with wildcards and partial matches.

Figure 4.1 presents a radix tree for 6 templates from the Blue Waters system. Special nodes have been introduced to accommodate the 3 wildcards HELO is using. The tree is constructed after the HELO's online phase and it is searched for a match each time there is a new message generated in the system. The matching phase uses the classical radix tree search algorithms for nodes that do not contain wildcards. For the * and d+ wildcards, the algorithm jumps the parsing string to the next space character and continues the matching from there (with an extra step for d+ to check if the skipped string represents a numerical value). The n+ wildcard is always a leaf in the tree, so the search can be stopped and a match returned. All leaves have an additional information that represents the ID number given to each template. The IDs are used to describe the signal and the correlations in an easier format.

Figure 4.2 shows an example of how HELO offline finds structure in a subset of logs and how the online phase updates the radix tree for every template that is not a match.

HELO considers that different type of words have different priorities dependent on their semantics. There are three types of considered words: English words, numeric values and hybrid tokens (words that are composed of letters, numbers and symbols of any kind). Numeric values have the lowest priority since the algorithm considers that these words have the most chances of becoming variables in the clusters. Hybrid values are represented by tokens like check..0. The algorithm extracts and considers only the English words incorporated in the hybrid token. For our example both check..0 and check..1 are considered as the word check.

HELO starts with the whole unclustered log file as the first group and recursively partitions it until all groups have cluster goodness over a specified threshold. The cluster goodness characterizes how similar all messages in

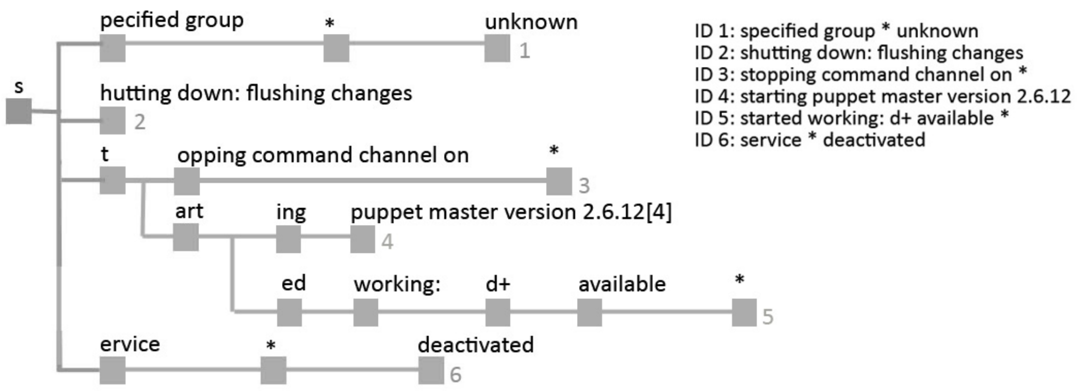

s
pecified group
*
unknown
1
hutting down: flushing changes
2
t
opping command channel on
*
3
art
ing
puppet master version 2.6.12[4]
4
ed
working:
d+
available
*
5
ervice
*
deactivated
6
ID 1: specified group * unknown
ID 2: shutting down: flushing changes
ID 3: stopping command channel on *
ID 4: starting puppet master version 2.6.12
ID 5: started working: d+ available *
ID 6: service * deactivated

Figure 4.1: HELO radix tree implementation

one group are and is defined as the percentage of common words in all the messages over the average message length. We will analyze in the following paragraph the influence of this parameter over how general and specific the templates become and how this affects the prediction results.

In each partition iteration, HELO chooses the best splitting column by identifying the word position that contains the minimum number of constants when looking at all entries in the log. For example, in figure 4.2, the splitting column in the first iteration represents all words that occur at the beginning of each log entry, and in the second iteration all words on the 3rd position. We consider that words with a high number of appearances on one position has more chances of being a constant in the final template, so HELO searches for the column where most unique words have a high appearance rate.

More details about HELO can be found in the 2011 paper [85]. The tool was integrated in NCSA's Blue Waters monitoring software and is the foundation of the failure alert system used by Blue Waters managers to quickly handle system failures. The HELO version used by NCSA has a few of optimizations that the initial version did not contain, a couple of which are described in [86].

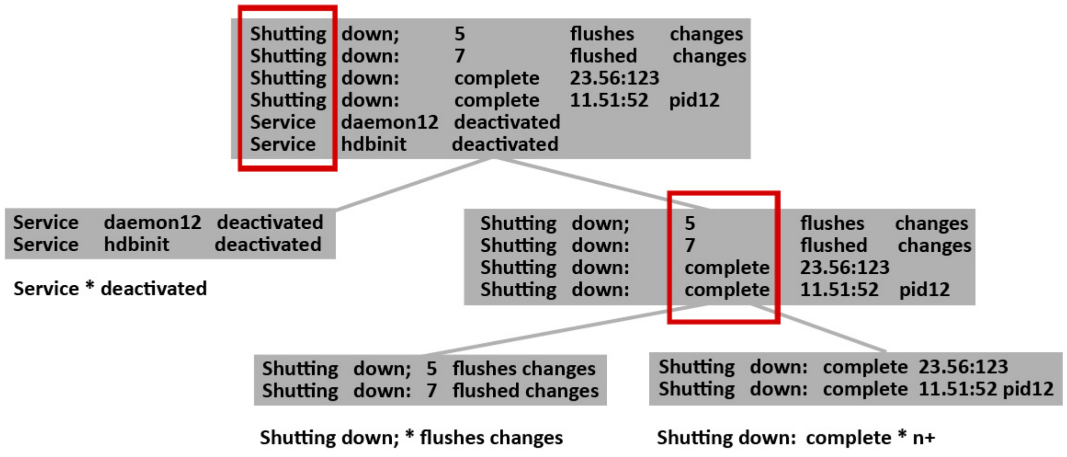

Shutting down; 5 flushes changes
Shutting down: 7 flushed changes
Shutting down: complete 23.56:123
Shutting down: complete 11.51:52 pid12
Service daemon12 deactivated
Service hdbinit deactivated
Service daemon12 deactivated
Service hdbinit deactivated
Service * deactivated
Shutting down; 5 flushes changes
Shutting down: 7 flushed changes
Shutting down: complete 23.56:123
Shutting down: complete 11.51:52 pid12
Shutting down; 5 flushes changes
Shutting down: 7 flushed changes
Shutting down: complete 23.56:123
Shutting down: complete 11.51:52 pid12
Shutting down; * flushes changes
Shutting down: complete * n+

Figure 4.2: HELO methodology

The number of events generated by the Blue Waters system is two orders of magnitude larger than previous generation systems (Table 3.1). The increased number of event types creates complex patterns that need longer training phases to be discovered. HELO did not require any modification, but the correlation and prediction modules needed to become lighter in order to be applied on the Blue Waters system. We present the changes in the following chapters.

Monitoring each event type separately is important since information regarding the events of interest might be hidden when the analysis is made at a lower granularity. For example, when looking at all types of failures at once, the logs show close to no spatial propagation. However, when analyzing only a certain type of filesystem errors only around 20% of failures affect only one node, the rest propagating on a variable number of locations [32]. In the following paragraph we will show such an example for the Mercury system.

4.1.1 Propagation analysis on the Mercury system

We analyzed all failures and then separately 6 individual failures given after inspecting the templates generated by HELO. The individual failures are described in table 4.1. We assume that failure events are instantaneous, and focus on modeling the time between two consecutive failure events.

We derive statistical distributions of failure rates over the whole system. Table 4.2 shows the mean and median rate of each failure over the entire system across all epochs. Among all types of failures, there was an average of between 1.8 and 3.6 failures per day. Assuming an equal probability of failure over all nodes, this is a per-node mean rate of between 248 to 484 days to failure. This table also shows there is a wide range in inter-event times for different types of failures. For example, failure type F1 has a mean inter-event time of 77 hours, while failure type F3 has a mean of 20 hours. Inter-event times can also vary significantly between different epochs in the system for the same failure. For example, mean inter-event time between F1 failures increases from roughly 35 hours in Mercury's second year of production to 78 hours in the third.

We also examine the number of nodes affected by a failure. We observed that F1, F3, F5 and F6 rarely occur at close intervals on separate nodes. Therefore we consider each of these as affecting only a single node. However, F2 and F4 can occur simultaneously on multiple nodes. A plot of the CDF for the number of nodes affected by a failure over all epochs is shown in Figure 4.3 with the black line indicating the actual data and the red line showing the fitted distribution. For example, with the combined failure model, 91% of

Code	Message	Error type
F1	scsi error: * status=02h key=4h (hardware error); fru=02h asc/ascq=11h/00h ””	Hardware reported error in a device on the SCSI bus
F2	rpc: bad tcp reclen * (non-terminal)	NFS related error indicating unavail- ability of the network file system for a machine
F3	pbs_mom: sister could not communicate * in xxxxxx, job_start_error from node xxxxx in job_start_error	Failure of a PBS (Portable Batch Sys- tem) daemon to communicate
F4	ifup: could not get a valid interface name: -> skipped	Node is restarted but could not connect to either the Gigabit or management networks
F5	+ mem error detail: physical address: * address mask: * node: d+ card: d+ module: d+ n+	Error in the memory
F6	processor error map: * processor state param: xxx processor lid: *	Error in the processor cache

Table 4.1: Mercury error templates

failures affect just one node, 3% affect two nodes, and 6% affect more than two nodes. We model the number of nodes affected by the combined failure as a Weibull distribution, by an F2 failure as a log normal distribution, and by an F4 failure as an exponential distribution.

These properties can be used to optimize fault tolerance protocols depending on what components the applications are using. The failure distribution used by Daly's formula depends on two factors, namely the specific components and number of nodes used by the application. Analyzing the specific failures extracted by HELO allowed us to notice that this distribution is different depending on the failure type. We will further analyze how this observations can influence fault tolerance protocols in chapter 6.

		2004	2005	2006	2007	2008
All	Mean	0.39	0.54	0.37	0.27	0.28
	Median	0.21	0.23	0.15	0.08	0.1
F1	Mean	1.45	2.72	3.27	12.7	14.9
	Median	0.98	1.01	2.23	3.46	9.08
F2	Mean	31.2	7.9	6.9	12.2	12.4
	Median	34.6	7.9	0.77	14	14
F3	Mean	-	1.1	0.82	0.42	0.4
	Median	-	0.07	0.31	0.09	0.11
F4	Mean	1.45	1.45	1.5	1.21	1.8
	Median	0.12	0.06	0.12	0.07	0.09
F5	Mean	1.18	5.11	9.88	7.95	3.39
	Median	0.84	2.89	5.42	4.41	1.51
F6	Mean	1.52	2.65	2.68	4.27	4.09
	Median	0.9	1.74	1.82	3.25	2.7

Table 4.2: Failure inter-event statistics (in days)

4.2 Extracting the normal and faulty event behavior

Large scale systems experience a large variety of events during their lifetime and they output notifications for each of them. Once an error is triggered for one component, either software or hardware, there is not a consistent way of recording how the system will behave. For example, in case a node experience a network failure and is incapable of generating log messages, the failure is announced in the log files by a lack of generated messages. Conversely, some component failures may cause logging a large numbers of notifications. For example, memory failures can result in a single faulty component generating hundreds or thousands of messages in less than a day.

At the same time, some errors are notified by a single message. For example on NCSA's Mercury system, NFS related errors that indicate unavailability of the network file system for a machine, need a single instance of the generated message to notify a potentially fatal failure to an application using this resource. However, this is not always the case. Memory errors, for example, are often correctable by the ECC capabilities, so only when the system generates a large numbers of these errors in a short time span, it is likely to have a permanent failure of a component.

Each failure type behaves differently and affects the systems differently. It is important to be able to model the normal behavior of the system for

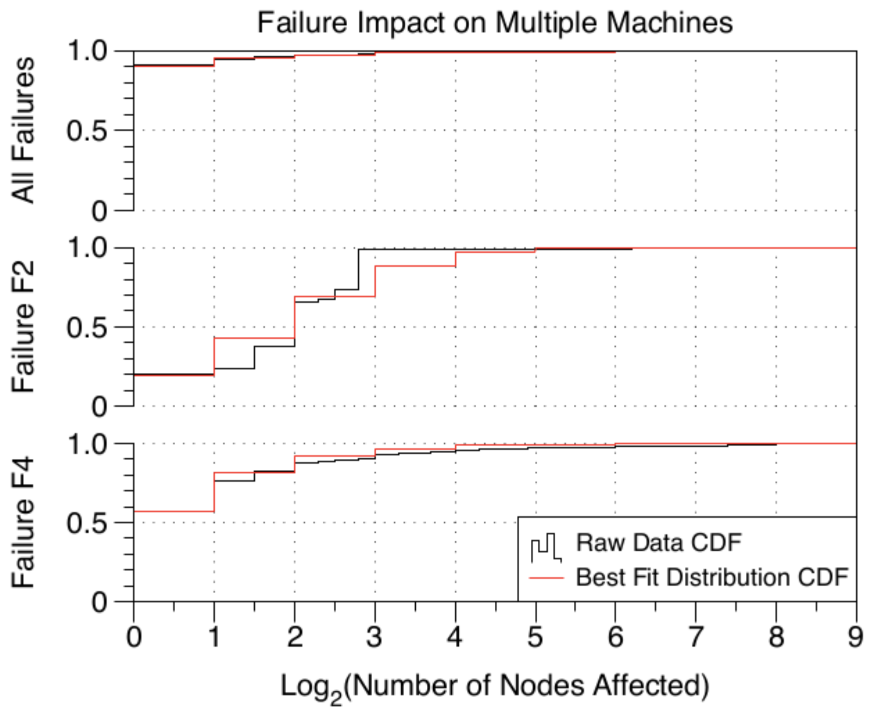


Figure 4.3: Distribution of nodes affected by failures

each of the events that might be generated and characterize the way a failure affects these models. We use HELO to extract all the event types and then treat the number of occurrences per time period for each event type as a separate signal. Each event type has occurrences at different times in a system lifespan. By choosing a sampling rate and mapping the number of messages generated by the system in each time period and for each event type, we extract a time series for each template. The obtained time series are regarded as signals and can be analyzed with signal processing methods. The sampling rate is chosen differently depending on the characteristics of each signal type and we will discuss the implications of the choice in the next sub-paragraphs.

After extracting all the signals for all the analyzed systems, we observed that there are three types of time series: periodic, silent and noise. An example of each of the three types can be seen in figure 4.4. Usually, periodic signals are generated by daemons or by events that deal with monitoring information. Examples of these signals are presented in Figure 4.4c. We call the second type silent signals because most of the signal is a flat line around the zero value, and only from time to time there is a burst of messages.

System	Template	Event type
BGL	failed to configure resourcemgmt subsystem err = d+	Processor cache error
Blue Waters	* panic - * syncing: *	LBUG
Blue Waters	Lustre: * @@@ Request sent has failed due to network error: n+	MDT Failure
BGQ	component state change: component * is in the * state *	Info notification
BGQ	ECC-correctable single symbol error: DDR Controller d+, failing SDRAM address *, BPC pin *, n+	DDR single symbol error

Table 4.3: Examples of templates and their event types

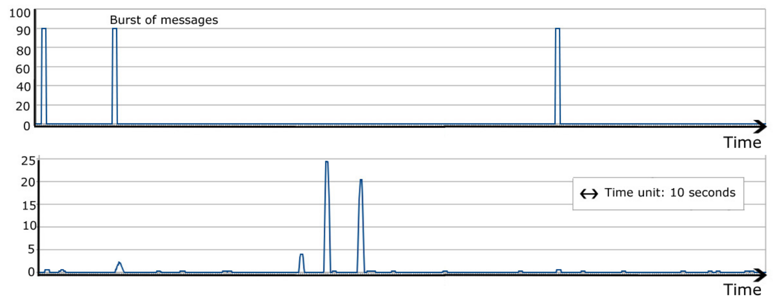
This type is presented in Figure 4.4a and is usually characteristic for error messages, for example in case of PBS errors. Noise are verbose signals that send notifications very often. Two examples of such events are presented in Figure 4.4b. This type of signal are usually warning messages that are generated both in case of normal behavior and failures, usually preceding error messages or when a problem is corrected. We observed that even some failure events can experience this behavior, for example in the case of memory errors that could be corrected by ECC.

In the next paragraphs we will look at the three types of signals that we identified previously.

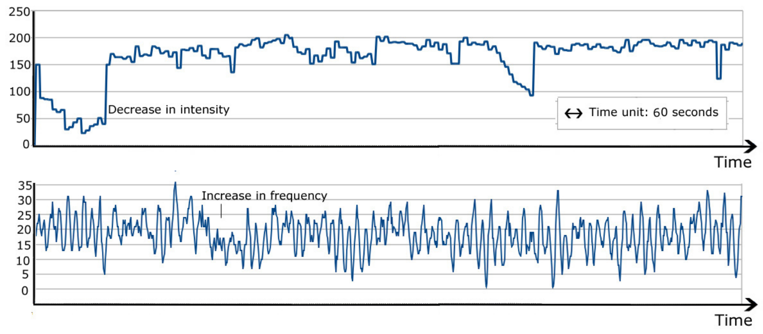
4.2.1 Periodic events

Periodic events generate messages regularly having, in general, a fixed occurrence frequency (for example in the case of daemons) but they could also have multiple frequencies. We extract periodic events in two distinct steps: first we parse the data to find the best sampling rate, and then use the frequency spectrum to find all frequencies that represent the signal. The second step is computational expensive, so after the first step we create a list with only the periodic signals, and apply the expensive second process only to those. Also, the second step gives good results for periodic signals, but is not very accurate when dealing with non-periodic events.

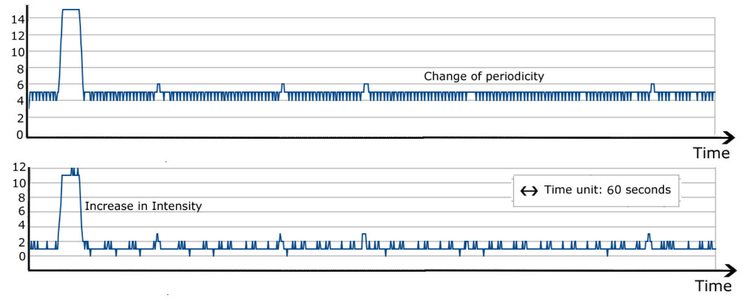
For extracting the correct sampling rate we use the Nyquist theorem. The



(a) Silent signals



(b) Noise signals



(c) Periodic signals

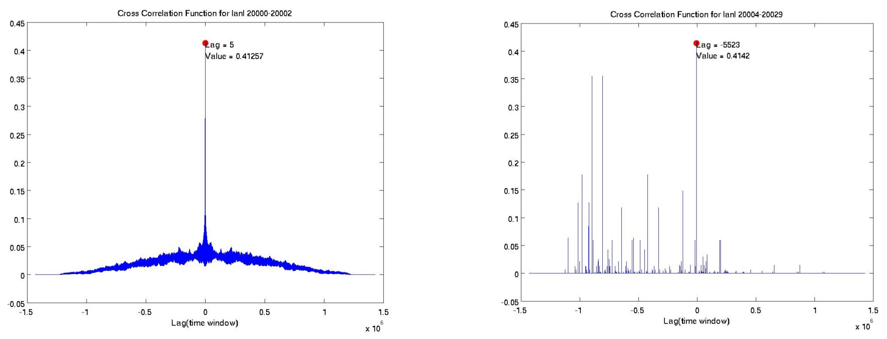
Figure 4.4: Different signals generated by HPC systems

theorem states that, if a function x(t) contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced 1/(2B) seconds apart. In the first step we test if the signal has a periodic behavior while still respecting the Nyquist theorem. We want to choose the smallest possible time sample rate, however, as stated previously, faults triggered in the system might create more messages or might make notifications disappear, so choosing the minimum lag between adjacent events is not realistic and inefficient.

We implemented a recursive process, where we start with an initial low sample rate and keep increasing the rate until either it exceeds the maximum time lag between two adjacent occurrences or the extracted signal is periodic. We stop the process if the signal is periodic only if the ratio between the sampling rate and the period of the signal respects the Nyquist theorem [87]. The initial value used for this process is the mean time delay between two event appearances of the same type. The value increases exponential, in each step being doubled. After extracting the signal, we use the auto-correlation function for determining if a signal is periodic or not. The auto-correlation function is used to compute the similarity between a signal with itself, for different time lags. When applied to signals, if the similarity value is over a threshold than the signal can be considered periodic.

Examples of the auto-correlation function, for a periodic and random signal, can be seen in Figure 4.5. The random signals have only one peak for lag 0, which means that the signal has a high similarity only with itself. Periodic signals have multiple peaks, visible in Figure 4.5b.

In the second step, only for signals that show a periodic behavior, we extract the frequencies that make up the event. A clean view of the signal's fundamental frequencies is given by looking at the signal in the power spectrum. The periodogram is computed from the FFT (Fast Fourier Transform) and it is used to highlight the periodic behavior of a signal. By transforming the signal into the power spectrum, periodic signals of low frequency have a smooth appearance whereas those of high frequency have a irregular behavior. If a time series has a very smooth appearance, then the values of the periodogram for low frequencies will be large relative to its other values. For a purely random series, all of the sinusoids should be of equal importance and thus the periodogram will vary randomly around a constant.



(a) Random signals

(b) Periodic signals

Figure 4.5: Auto-correlation plots for different signals

If a time series has a strong periodic signal for some frequency, then there will be a peak in the periodogram at that frequency. In case of faults that modify the signal, the periodogram might present some picks of lower values that must be filtered out. The idea behind the filtering method is that only the highest peaks are the periodic frequencies and for all cases these peaks represent a very small number from the total frequencies in the periodogram. We are performing a recursive filtering method, in each step leaving only 5% of the values from the periodogram, until we are not eliminating any of the peaks in two adjacent steps. The process is illustrated in Figure 4.6. The signal in the figure has a period of 10 time units, but due to failures affecting this event, the initial image presents a various number of peaks. After filtering them, the last image shows only the peak corresponding to a period of 10.

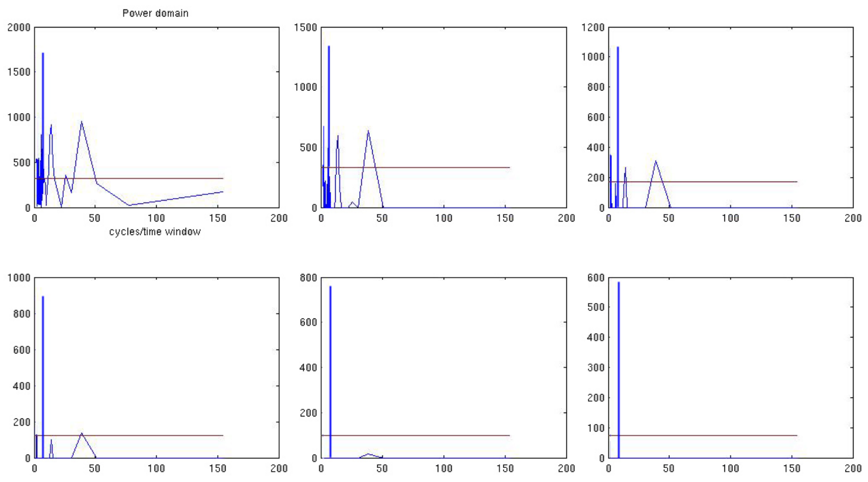


Figure 4.6: Filtering the power spectrum

4.2.2 Noise and silent signals

All the signals that are not periodic are analyzed separately. When we extract the time series, we use a fix sample rate of 10 seconds. We extract the timestamp of the first and last occurrence of any of the log messages and create a signal for each template, for this interval. This makes the size of the signal to be the same for all the event types, and makes it easier to correlate them in the next step. We classify the signal as noise or silent by looking at its behavior in the majority of its lifespan. Silent signals have a high number

System	**Mercury**	**LANL**
Periodic signals		
Number	11	2
Percentage	2.7%	3.8%
Silent signals		
Number	338	39
Percentage	82.6%	73.6%
Noise signals		
Number	60	12
Percentage	14.7%	22.6%

Table 4.4: Statistics for different signal types

of sampling intervals with zero or a low number of messages compared to the noise signals that are formed in general by high number of notifications for the whole length of the signal. Table 4.4 presents the percentage of each type of signals found in the Mercury and LANL system's logs.

In [88], the authors propose a methodology for measuring different noise parameters, by using time-series methods, like extracting the trend, curve-fitting or interpolation. Their research is focused for chip level analysis tools, by looking at the noise induced by coupling capacitance. However, the overall idea can be used for log file analysis. Specifically we use the same main idea of glitch modeling by examining both glitch hight and width to accurately analyze the noise in the signals.

We characterize what is the normal shape of the noise, the behavior that the signal has most of the time, and when the signal frequency or intensity changes. For this reason, we will transform our signal to a mixture of time and frequency domain and apply different signal processing methods to extract anomalies. The algorithm decomposes our data set in chunks of overlapping time intervals, then apply FFTs on each interval in order to create the frequencies of the signal over time. Figure 4.7 presents the spectrogram for one event type occurring on the Blue Waters system in one month. Frequencies are represented on the vertical axes, time on the horizontal one and the intensity of each frequency is given by the color of each point. We also use a couple of filters that act as either an averaging filter or one that produces details depending on the characteristics of the signal that highlight the normal behavior [89].

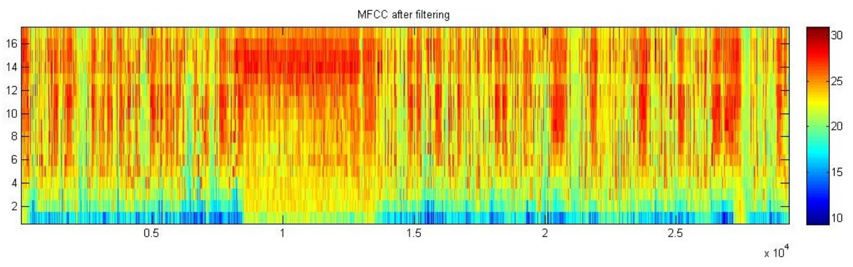


Figure 4.7: The spectrogram of ECC warnings on Blue Waters

4.2.3 Anomaly detection

In this section we focus on the faulty behavior for each of the signals. This step is done offline as a prerequisite for both the proposed filtering method and short-term prediction. Knowing what is the normal shape of the signals, we will now investigate changes in the frequency and intensity. For some signals an increase in the frequency of the messages could indicate a problem, while for others a lack of notifications should be taken into consideration. Intensity is of equal importance, so we are also investigating the total number of messages generated in a single time unit.

A time series spectrogram can be considered as a combination of two components: the set of frequencies and their intensity. The components influence the trend and seasonality of signals. It is important to determine whether trend and/or periodicity exist in a series in order to choose appropriate models and methods for descriptive or forecasting purposes. Exploring the characteristics of a signal is enhanced by suppressing one type of pattern for better visualizing the other patterns. For example, suppressing the changes in intensity can make a modification in the normal frequency rate more visible.

Feature extraction

Transforming the input data into the set of features is called feature extraction which involves simplifying the amount of elements required to describe a large set of data accurately. In general, the more features we have, the better we will be able to distinguish different spike shapes. The spectrogram represents a M $\times$ N matrix where M is the number of features that might appear in a signal and N represents time. The result of the feature extraction step is a K $\times$ N matrix, where K is the number of extracted features. We use the classical technique of Principal Component Analysis (PCA) to reduce dimensionality from M to K. Figure 4.8 shows the amount of information given by each frequency. In this example, we could reduce the dimensionality to 2 components since these first two components are giving over 90% of the information needed to characterize the signal. All the analysis is done of the reduced matrix.

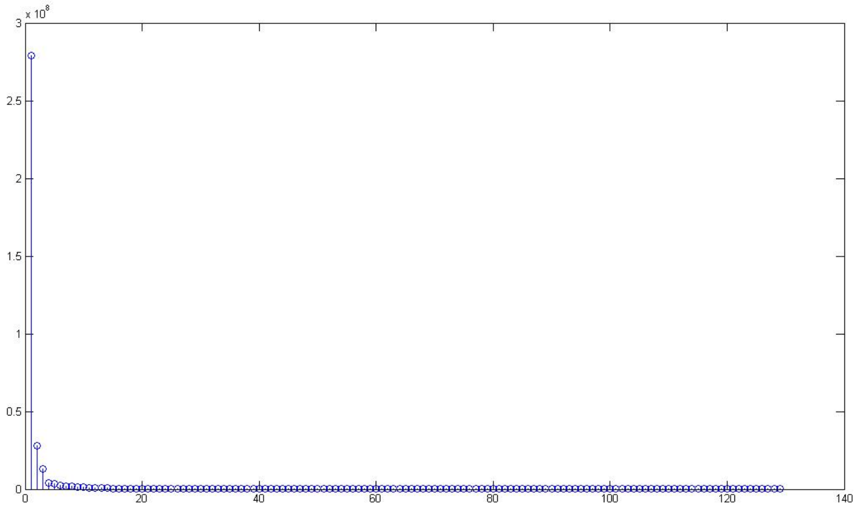


Figure 4.8: Applying PCA on a noise signal

Changes in the frequency

First we identify shifts by looking at the frequency with which the system generates messages. After inspecting all the signals from both systems, we observed that the only concern we need to analyze is an increase in the frequency. For events that stop generating notifications in case of errors, the frequency does not progressively decrease, but rather drops to 0 in a short amount of time. This case is analyzed in the next paragraph and is considered a decrease in the intensity of the signal. Here, we will only focus on increases in the signal frequency. We can use the same method for all three signal types, after we apply a filter to modify the original event by enhancing the difference between the normal frequency and the moments with an increase rate and after we transform the signal into the frequency domain and reduce its dimensionality. For the first step, we use the moving average technique. In signal analysis, a moving average is a type of finite impulse response filter used for analyzing a signal by creating a series of averages of different subsets of the full data set. Basically, this method smooths the signal by simply replacing each data value with the average of neighboring values. After applying the filter, the new signal presents unknown patterns on areas with a high frequency of messages and known values otherwise.

Changes in the intensity

In the second case we focus on identifying changes in the total number of messages generated by the system per time unit. For silent signals, the changes in the normal intensity are seen as peaks at different points in time. However for random and periodic signals we investigate either bursts of messages or a decrease in the normal intensity of the signal.

The signal is modified so that the intensity anomalies are more visible for the extraction algorithm. We are applying two filters, one for putting emphasis on large values in order to enhance the difference between peaks and the normal behavior, and the other on abnormal small values for enhancing the deviation of time units with decreasing intensities from normal. In both cases, we apply a filter for skewing the signal after which we applied the same methods for transforming the signal to frequency domain and for reducing the dimensionality. Anomaly extraction algorithms are then applied in order to find moments of time where there are unknown patterns in the reduced spectrogram.

Figure 4.9 presents the methodology for identifying the anomalies in the signal.

4.3 Filtering methods based on signal analysis

Even though none of our analysis requires a filtering step, other studies on failure analysis [9, 62, 83] use it extensively in their analysis and this step influences the final results considerably. Since log files have a large dimension, most data mining algorithms require a step that compresses the logs, while still keeping intact information about all failures and events generated. For

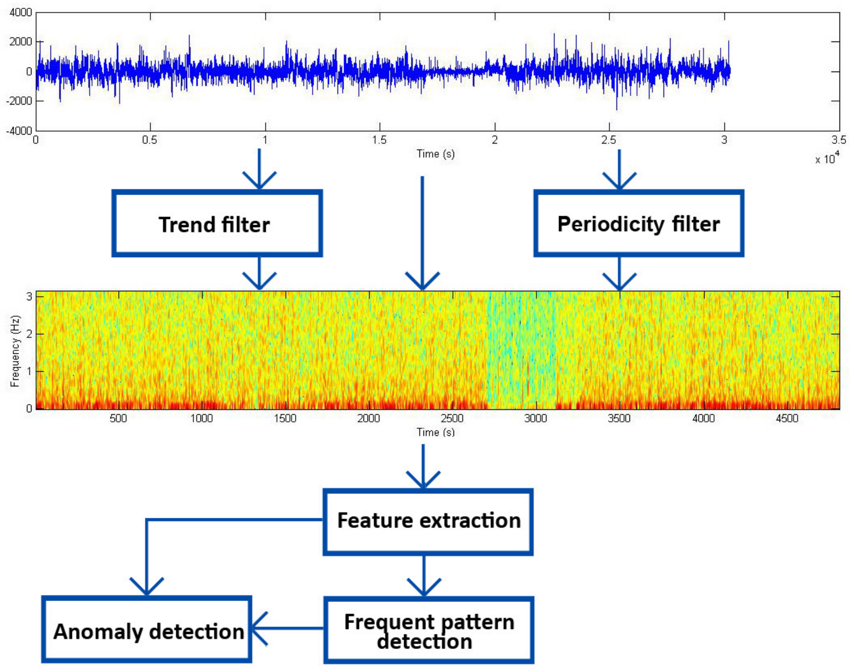


Figure 4.9: Anomaly extraction methodology

this purpose, event occurrences need to be identified and only redundant information needs to be filtered.

We show in this section how analyzing log files as a collection of signals is a flexible and useful way for widely used event log analysis methods. We use the modules described previously in a pipeline manner by applying successive filters to the initial signals, modifying them so that they enhance the information we need to extract. Specifically, we implement a filtering algorithm that can be used as a preprocessing step for a multitude of tasks. We show the use of signal processing concepts allows us to automate this step completely and gives better or similar results compared to recent filtering techniques.

Most research in the area of failure data analysis is using a step for filtering out entries that are not useful or that are redundant error entries from any log file. As stated previously, a fault, once triggered, can generate multiple errors that propagate within the system, so most current analyzing algorithms require a preprocessing step where these multiple entries for an event are filtered. A widely used strategy for filtering the entries in the log is by using a time window.

Current research is analyzing the entries produced by the system belonging to a specific error category, for example memory or network, separately. In [83], the authors show that simplistic filtering methods that just use a fixed time window loose around 10% of the messages that should be analyzed independently. However, when the logs are analyzed at a finer grain, by looking at each type of events and not only at general error categories, the difference is much lower [58]. In our previous work, we found filtering each event type results in the previous algorithm still loosing around 5% of the events.

Usually the entries that need to be analyzed are triggered by either a fault in the system, whose effects propagate between different locations or by two independent faults on different locations that occur coincidentally and generate notifications at the same time. The messages triggered in both cases could involve multiple event types. The filtering method we propose follows this idea and uses the modules described in the previous section.

Firstly, we only group error events that correlated. This means that if two messages of separate types occur frequently together, in a small time window, there are high chances that we are dealing with one cause. Otherwise, we consider the two notifications separately.

In the next step, we investigate each event type, by extracting the signals for each possible location and correlating these signals between each other. We found that signals on multiple locations for the same event type, share the same characteristics in terms of normal and faulty behavior with their base signal. For example, a silent signal analyzed per location will generate silent signals for each of the analyzed nodes. Correlating these signals will give a good statistical information about the propagation behavior of the event. If there are location signals that are generally correlated one to another, we consider that the event type has a propagation behavior.

We are grouping the event types related to the same fault manifestation so that our results are significant to the needs of the analysis methods described in the chapter 2. Consequently, we group events that need to be analyzed together and statistical information about the propagation behavior of each of them. The entire methodology of our filtering technique is presented in Figure 4.10. Previous algorithms have a parameter to decide on the length of the filtering window. For our analysis, this parameter is implemented into the sampling rate used to extract the signals. After identifying the groups of related events and the statistic information about the propagation

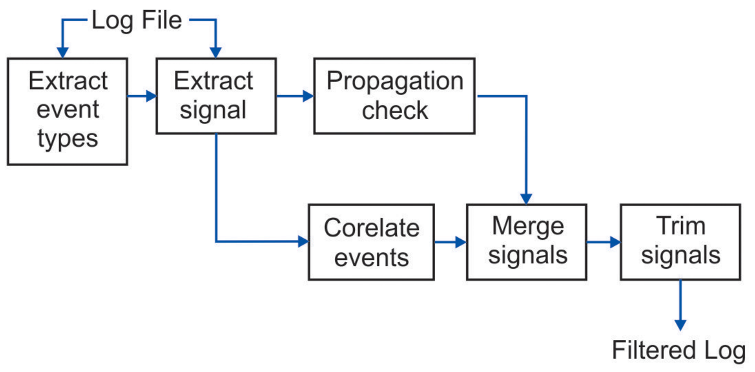


Figure 4.10: The filtering methodology

behavior, the filtering process is straightforward. For all events that do not have a propagation behavior, our method merges the signals from different locations as one. Also, all event types that are in one group will be merged as one. At the end a filter is applied by trimming all signals to a value of 1 in all the sampling units that had at least one occurrence.

This is, basically, equivalent to merging messages that occur together in a filtering window, given by our sampling window. Events merged are always statistically related to the same error cause and, either happen in the same location or have a statistical propagation behavior, in case the messages appear on different nodes. In all other cases, the two messages are analyzed independently.

For comparing the filtering approaches, we use the same measuring unit as in [83]. This paper has the objective of quantifying the extent of the distortion introduced by filtering out events that should have been analyzed independently. According to previous studies [50, 25, 55], choosing a time window of 240s is a reasonable choice for most of large-scale systems, so we will use the same value for the sampling rate of our systems. We then test the influence of different sampling rates around this value on the final results.

The results are summarized in Table 4.5, where T represents the number of messages left in the log after filtering with the previous classic method, and T+ the number obtained with our method. If we analyze the percentage of how many independent events were filtered with the classic method (column 5 in the table), it is visible that for a time window of 240 seconds, used widely in current research, there is a 12% difference for the LANL systems

Time window	T+	T	Diff	Percentage
LANL system				
160	1,450	1,344	106	7.3%
200	1,323	1,194	128	9.7%
240	1,212	1,068	143	11.8%
280	1,171	1,028	143	12.2%
Mercury				
160	12,872	12,189	682	5.3%
200	12,732	12,006	726	5.7%
240	12,644	11,720	923	7.3%
280	12,591	11,596	994	7.9%

Table 4.5: Filtering results

and over 7% for Mercury. Also, it can be observed that the number of filtered independent events increases as the time window increases.

Grouping the error entries related to the same fault manifestation is crucial to obtain realistic measurements. By analyzing events separate for each specific type we were able to extract the set of events that frequently occur together and filter them together. However, since other methods are doing this filtering for all failures at once, they filter failures that occur close in time even though they might refer to two/multiple separate problems. The "Diff" column in Table 4.5 show the number of such cases not identified by simple filtering methods and that are identified by ours. Without manual inspection, the exact number of independent events in a system is not known we cannot show how close to the optimal filtering our method is. We believe that even our method filters some failures that might describe two independent problems. However, it preserves 7-13% more failures that other used techniques.

Our results are similar to the ones presented in [83], however we do not require any preprocessing step where a system managers identify the groups of messages relevant to different error types. Our process is completely automatic. Furthermore, once the initial signals are extracted, our algorithm is applying recursive filters to transform the signals in the final form, making the process efficient and flexible. If we need an additional step we can just add one more filter in the pipeline. Moreover, our approach allows replacing or removing one or multiple steps for an easy change of the scope and results of any process.

4.4 Failure analysis

In this section we analyze the faults and failures of the HPC systems presented in chapter 3 and we highlight the similarities and differences between each. We are looking at root causes as identified manually by system managers and statistical information as well as inter-failure correlations and propagation characteristics. On average, in the analyzed timeframe, for the biggest system, there was a failure of any type every 6.7 hours, while the system suffered system-wide outages approximately every 160 hours. The MTBF has decreased from once every couple of months for LANL systems to every several hours for the Blue Waters system. Moreover, the time to restart the machine and the applications after a system wide outages is taking longer times for larger machines. The frequency of failures and the system complexity is making the task of failure detection and prediction much harder. In this section we open the hood, analyze the behavior of failures, and highlight the properties that can be used by a failure predictor.

4.4.1 Location propagation

In general, 12-25% of failures affect more than one node (without considering system wide outages). For prediction purposes each node affected by a failure is a potential false positive or negative. For example, on the Blue Waters system, failures in the voltage converter module (VRM) of the mezzanine, or problems with the cabinet controller, affect a whole blade consisting of four nodes. From a prediction perspective, these are 4 failures. All nodes failing as part of a multi-node failure represent more than a quarter of total failures affecting the system.

Large-scale systems contain nodes that are organized in an hierarchy. For example for the BlueGene systems, nodes are gathered into midplanes and multiple midplanes form a rack. After a closer analysis, we observed that the propagation path for different error types follows closely the way components are connected in the system. For example, if a fan breaks, all nodes sharing the same rack will be affected. In general, sequences of events following a failure do not propagate on different locations and if they propagate they affect a small number of nodes, only around 22% of sequences extracted for Mercury and 25% for BlueGene/L show any kind of propagation.

Between 80% and 85% of the sequences that show a propagation behavior affect less than 10 nodes. The rest, which represents less than 2% from the total number of correlations, influence a large number of nodes. An example of such a failure can be seen on the Mercury machine, when we investigated NFS (network file system) problems. The event "*rpc: bad tcp reclen d+ (non-terminal)*" indicates the network file system unavailability to any requests for a node. In applications using the network file system this could cause file operations to fail and the application to quit. Also all nodes from which the application tries to access the network file system will be affected by this problem. This failure usually occurs nearly simultaneously on a large number of nodes.

To get a more realistic view of the behavior of sequences, we analyzed the initial pair of correlated events for the BlueGene/L machine and broke down the propagation on racks, midplanes and nodes. Figure 4.11 shows that around 75% of correlations show no propagation at all and only around 2.16% propagates outside of the same midplane.

From our observation, in the Mercury system network failures can occur nearly simultaneously on multiple nodes. For example, the event "*ifup: could not get a valid interface name: -> skipped*" represents an unexpected node restart caused by unexpected hardware failure, and propagates across different nodes. In general, errors in memory or processor caches do not show the same behavior. On the other hand, in the BlueGene/L system we observed that some memory errors propagate to different node cards in a midplane. For example, the sequence

*d+ ddr errors(s) detected and corrected on rank 0, symbol * bit **

*total of * ddr error(s) detected and corrected*

occurs frequently together and refers to a ddr memory error that was detected and corrected in a certain locations and in most of the cases affects multiple nodes in the same midplane in a short period of time.

On the other hand, errors related to node cards do not propagate on multiple locations. For example the sequence:

can not get assembly information for node card

*linkCard power module * is not accessible*

*no power module * found found on link card*

gives information about a node card problem that is not fully functional. Events marked as "severe" and "failure" occur about one hour after the first

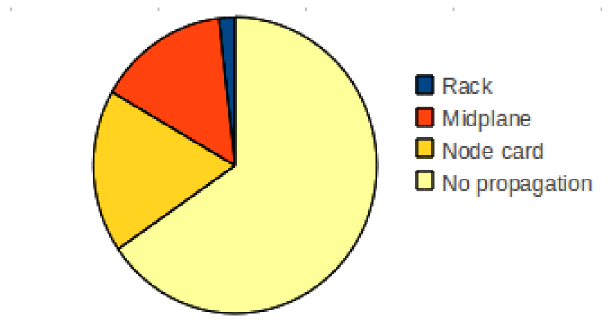


Figure 4.11: Percentage of sequences propagating on different racks, midplanes and nodes on Blue Gene/L

Category	Blue Waters	Blue Gene/P	LANL systems
Hardware	43.12%	52.38%	61.58%
Software	26.67%	30.66%	23.02%
Interconnect	11.84%	14.28%	1.8%
Facility/Environment	3.34%	2.66%	1.55%
Unknown	2.98%	-	11.38%
Heartbeat	12.02%	-	-

Table 4.6: Percentage of different failure types

message and report that the link card module is not accessible from the same midplane and that the link card is not found. The sequence is generated by the same node for all its occurrences in the log.

For 75% of correlations containing messages that do not appear on multiple nodes, the analysis and prediction system does not need to worry about finding the right location that is affected by a failure. However, for the other 25% that propagate, a wrong prediction will lead to a decrease in both precision and recall. For the Blue Waters system, the complexity of the propagation behavior increases since we have more sources of notification, thus more failure data than for other systems. We will further analyze this behavior in Chapter 5

4.4.2 Failure statistics

We divided all failures in 5 main categories that can be encountered in all systems: hardware, software, network or interconnect, facility and unknown. Figure 4.12 presents the percentage of failures of each type and table 4.7 shows the average number of failures for one month of production. Both

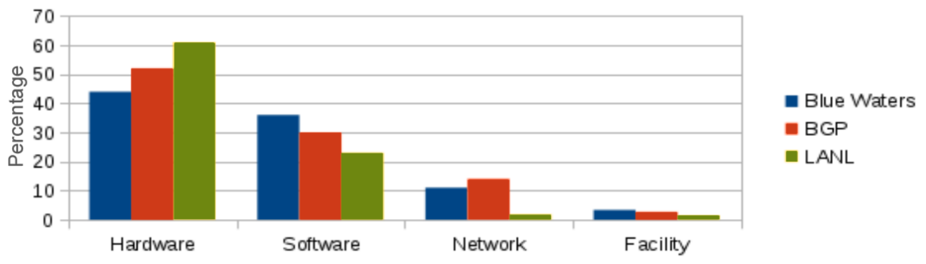


Figure 4.12: Percentage of different failures

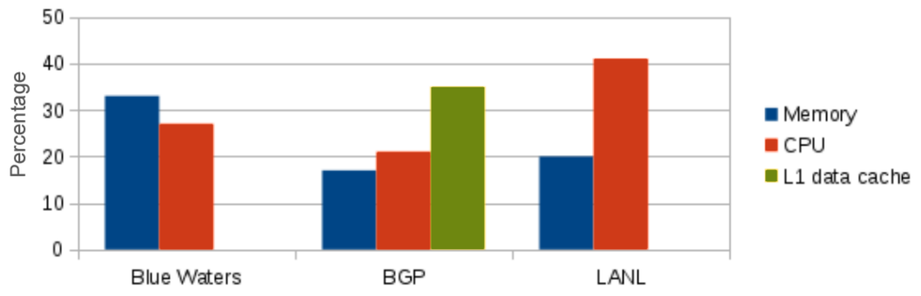


Figure 4.13: Percentage of main hardware failures

tables consider the categories that were identified by system managers for each system.

Hardware represents the majority of failures for all systems, with the lowest percentage of 43.12% for the Blue Waters system and 61.58% for the LANL systems. As shown in figure 4.13, the majority of hardware failures were memory and processors errors. Moreover, failures with hardware root causes were limited to a single node in 96% of the cases, or a single blade consisting of 4 nodes in 99.3% of the cases. In over 90% of the correlation chains (46 out of 51 total correlation chains), precursor events appear on the same node where the failure occurs. For this reason, the mis-predictions caused by not offering the correct location are rare for hardware failures.

On the Blue Waters system, each node receives a periodical heartbeat request that triggers a number of specific tests. If the tests fail or the heartbeat is not received by the system management console, the node is marked as down. The test is automatically repeated, by default, every 60 seconds for 35 minutes following a failure. We use this information to filter out messages that refer to the same problem and what is left is reported in Figure 4.12. In general, failures labeled as heartbeat failures have a separate root cause, either hardware, software or network.

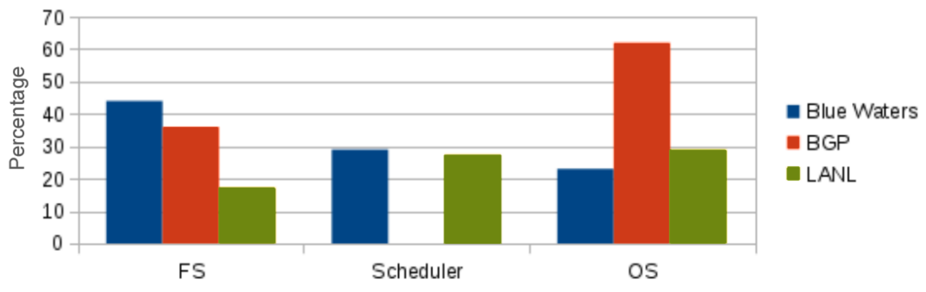


Figure 4.14: Percentage of main software failures

Category	Blue Waters	Blue Gene/P	LANL systems
Total	118	56.5	40.9
Hardware	55.4 (std 6.3)	27.5 (std 4.1)	26.3 (std 3.2)
Software	27.1 (std 5.82)	17.4 (std 3.7)	7.2 (std 2.4)
Interconnect	14.1 (std 3.1)	8.9 (std 1.3)	0.5 (std 0.4)
Facility/Environment	2.9 (std 1.2)	4.2 (std 2.9)	0.4 (std 0.3)
Unknown	2.7 (std 1.7)	-	6 (std 2.3)
Heartbeat	15.3 (std 2.43)	-	-

Table 4.7: Average number of failures per month for each type (with the standard deviation)

Software errors represent over 30% of total failures for the Blue Waters system, while for the LANL system they represent only 23%. For the systems presented in this study we can observe that current generation suprecomputers present a higher percentage of software failures while the hardware failures represent a smaller percentage compared to older systems.

Figure 4.14 presents the main causes of software failures. The main ones are filesystem problems (Lustre for the Blue Waters system, GPFS for BGL and several for LANL: Cluster File System, Parallel File System, NFS, Scratch FS and Vizscratch FS); failures of the job scheduler and operating system problems. On the Blue Waters system, 12% of the software failures caused system wide outages (SWO) and represent over 75% of all causes that triggered SWO. Moreover, software failures, when not causing a system-wide outage, propagate to more than one node in 15% of the non SWO failures. The correlation chains are also more complex than for hardware failures, over 67% (49 out of 73) of chains having at least one precursor on a different node than the predicted failure, and almost 37% having all precursors on different nodes.

Blue Waters	Blue Gene/P	LANL systems
Hardware		
RAM 33.12% CPU 27.04%	L1 data cache parity error 35.27% CPU 21.81% Memory 16.72%	CPU 41.35% DIMM 20.08%
Software		
FS 27.2% Scheduler 18.9%	OS 62.11% FS 36.02%	Other software 21.89% OS 20.99% DST (Distributed storage) 21.02% FS 12.33%

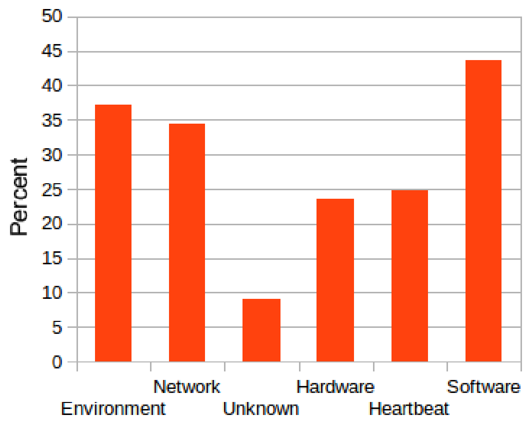
Table 4.8: Main specific failure types

Environmental failures include power-outages, failures related to temperature, cooling hardware problems and others. Most of the failures in this category are predictable when performance metrics are added in the prediction methodology so we will take a closer look at this type of failures in the following chapters.

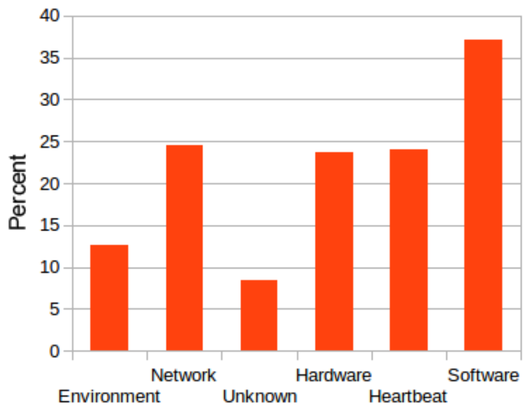
Table 4.8 presents the primary failure types for each main category for each system. The table presents the terms used by system managers from each data center when annotating the failure logs. The same term might refer to slightly different errors depending on the system. For example, CPU for Blue Waters and the LANL systems include L1 cache errors while for Blue Gene/P the two are separate types.

4.4.3 Failure correlation

In the second part of our study we focus on correlations between failures. This study focuses on the results for the Blue Waters system and only briefly discusses the differences for all other systems. As a starting point, we calculate the daily probability of a node failure. For this we compute the number of locations that fail in one random day (as the mean of all days in our timeframe). We then compare this result against the probability of a node failing during a day following another failure (in a 24h time window). For example, if we consider two days, and one of them had a SWO where all nodes failed, the results would suggest that, on average, almost 13,000 nodes fail per day



(a) Probability of a node having a failure of any type after it had failure of type X



(b) Probability of failure of type X following another failure of any type

Figure 4.15: Correlations between failures on the Blue Waters system within a time window of one hour

(and a probability of node failure of 50%). Since SWOs skew the results, we compute the same probabilities after filtering out SWOs.

We found that the unconditional probability of a node failure within one day is 0.63% when filtering out SWOs and the daily failure probability is higher during the day following a previous failure, namely 1.19%. This corresponds to roughly a 2X increase, which suggests that failures are correlated in the system. We also extracted the same data per failure type. We look at the probability that a node will fail within 24 hours following a failure of a particular type. At the same time we are looking at the percentage of cases when a node failure of any type follows a particular type of failure within an hour window. The percentage of cases when a failure of a particular type follows any failure within a one hour time window is also investigated. The results are presented in Figures 4.15.

Figure 4.15a shows what types of failures are good precursors for other failures and Figure 4.15b shows the types of failures that have precursors. Many failures seem to follow environmental and network failures. Also, by looking at Figure 4.16, we observe that these failures in general affect a large number of nodes which suggest they propagate not only in time but also space. Software errors have many precursors in other failures (37% of failures have a previous failures within an hour time window), more than hardware

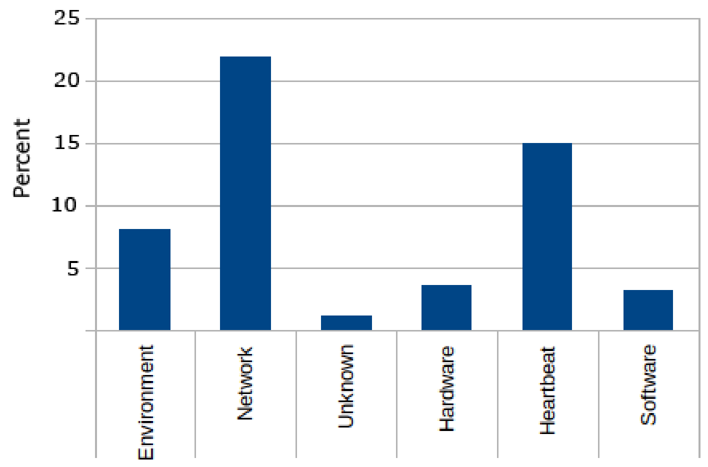


Figure 4.16: The probability that any node-failure follows a failure of type X on the Blue Waters system

failures. One explanation is that failed hardware is often shut down, while a software error does not shut off failed components, which means that our correlations reflect not only intrinsic properties of failures, but also recovery actions performed. This result might also indicate that a failure that was assigned to a software cause had, in reality, a different root cause. The prediction for hardware failures, however, has the potential of having better results because hardware failures have more non-failure precursors in the log files that could make the prediction more successful.

Chapter 5

Failure prediction

Over the years, different methods have been developed that deal with failure prediction in the HPC community [6], methods that have been used extensively on different HPC systems and that present a variety of results. A widely used strategy for extracting information for all these algorithms is to correlate events. However, the correlation is most of the time a statistical observation and does not take into consideration the diversity of events behavior. Furthermore, most of the studies build their analysis on a pre-process step where events are filtered to decrease the total size of the log. As shown in the previous chapter these methods have limitations that affect the overall accuracy of the analysis module.

In the previous chapter, we introduced the concept of signal analysis in the context of event analysis, which allowed us to characterize the behavior of different events and to identify anomalies. Our own analysis of inter-failure correlations and propagation behavior, as well as current related work have shown that failure prediction is a theoretical viable solution for future fault tolerance techniques.

A large fraction of experiments and results for failure prediction methods used in the literature have been the result of the analysis of different HPC systems in simulated online environments. We call simulated online predictions the predictions obtained by methods that manually tune the parameters used in the offline phase in order to achieve the best possible results in the studied online phase. While these methods show prediction results that could theoretically be achieved in real scenarios, they do not reflect the reality of running in realtime and predicting failures using best local parameters.

In this chapter, we present an analysis of failure prediction on different HPC system and we introduce a new method for predicting failures based on signal analysis concepts. Firstly, we analyze this method in simulated online environments in order to compare the results of our implementation with other state of the art methods. Second, we will investigate the feasibility of online failure prediction methods on a petascale machine by looking at a online approach on the Blue Waters system. Our method does not use ”simulate online” approaches by automatically choosing parameters for the online phase. With a sustained performance of 1 Petaflop on a range of real-world science and engineering applications, the Blue Waters supercomputer is representative of todays large scale systems and provides new insights into the performance of current fault predictors.

5.1 Failure prediction based on signal analysis

In the previous chapter, we introduced signal analysis concepts in the context of log file analysis. We observed that a fault does not have a consistent representation in the logs. For example, a memory failure will cause the faulty module to generate a large number of messages. Conversely, in case of a node crash the error will be characterized by a lack of notifications. Data mining algorithms in general assume that faults manifest themselves in the same way and in consequence fail to handle more than one type of behavior.

For example, even though silent signals represent the majority of event types, data mining algorithms fail to extract the correlation between them and other types of signals. This affects fault prediction in both the total number of faults seen by the method and in the time delay offered between the prediction and the actual occurrence of the fault.

Signal analysis methods can handle all three signal types, and thus provide a larger set of correlations that can be used for prediction. However, data mining algorithms are more suited in characterizing correlations between different high dimensionality sets than the cross correlation function offered by signal analysis. Data mining is a powerful technology that converts raw data into an understandable and actionable form, which can then be used to predict future trends or provide meaning to historical events.

Additionally, outlier detection has a rich research history in incorporating both statistical and data mining methods for different types of datasets. Moreover, it is able to implicitly adapt to changes in the datasets and to apply threshold based distance measures separating outliers from the bulk of good observations. In this chapter, we combine the advantages of both

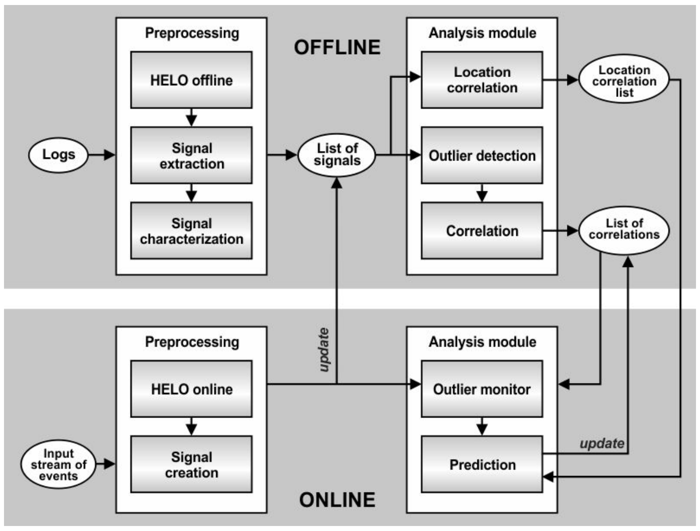


Figure 5.1: Methodology overview of the hybrid ELSA approach

methods in order to offer a hybrid approach capable of characterizing different behaviors resulting from events generated by a HPC system and provide an adaptive forecasting method by using latest data mining techniques.

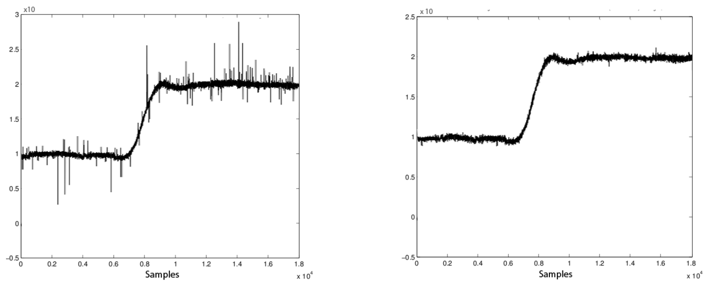
In the following sections we present the methodology used for preprocessing the log files and extracting the signals and then we introduce the novel hybrid method that combines signal analysis concepts with data mining techniques for outlier detection and correlation extraction. An overview of the methodology is presented in figure 5.1.

In the next sections we will build on the signal analysis methods in ELSA, described in section 4 and add data mining prediction functionalities. We will refer this combination of signal analysis with data mining as the hybrid version of ELSA.

5.1.1 Analysis modules

Outlier detection

All analysis modules are novel hybrid modules that apply data mining techniques on the previously extracted set of signals and their characterization. Since the offline phase is not run in real-time and the execution time is not constrained, we did not optimize this step. For outlier detection in the online phase, we use as input the adapted set of signals and apply a simple data



(a) Original data (b) Signal after filtering

Figure 5.2: Online outlier detection

cleaning method for identifying the erroneous data points.

We implement this step as a filtering signal analysis module so that is can be easily inserted between signal analysis modules. The transformation was intuitive since the data mining algorithm is based on a causal moving data window that is appropriate to realtime applications: the observed data point y_k is compared to the median y_k^m of past data points, both the erroneous and the replaced ones. If the distance between these points is large relative to a threshold based on the normal behavior of the system, y_k is declared an outlier and a replacement with a more reasonable value y_k^c is proposed. Figure 5.2 presents a synthetic noise signal in its original form and after applying the online outlier detection with replacement for the erroneous data points.

For a window of N points, the analyzed list of points for the current y_k is:

$$V_k = \{y_{k-N}^c, ..., y_{k-1}^c, y_{k-N}, y_{k-N+1}, ..., y_k\}$$

out of which the median is extracted y_k^m. For our experiments we use an N value of two months.

We use predefined thresholds for each signal, specified automatically in the preprocessing step based on knowledge about the normal behavior of the event type and how this was affected by outlier in the offline phase.

The replacement strategy decreases the influence of severe outliers on signals by saving both the initial value and one that is more consistent with the rest of the dataset. At the same time it minimizes the effects of a large number of faults hitting the same signal for a larger period of time.

Having a low execution time is a requirement for the online modules. The on-the-fly filter makes the process faster than what is proposed in the pre-

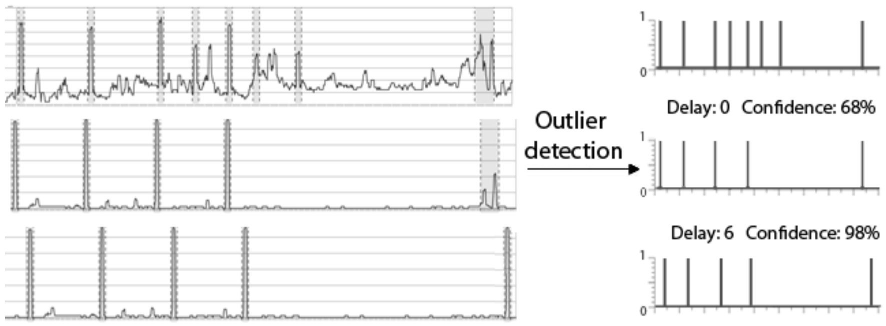


Figure 5.3: Correlation example between three signals

vious chapter. We will show in the experiment section the number of faults missed because the outlier detection and prediction took too long to notify the applications.

Signal correlation

Gradual itemset mining [90] is used in the data mining community for extracting patterns of the form "the more/less X_1, .. , the more/less X_n". The goal of the algorithm is to discover frequent co-variations between different attributes. This method has the advantage of extracting multiple event correlations instead of only pairs like the output of the signal cross-correlation function. A large number of data mining algorithms divide the logs into chunks and extract sets of failures that frequently occur together in these chunks. This method is called itemset mining. Gradual itemset is a slightly more advanced form of itemset mining in the sense partitioning the logs is no longer necessary. The algorithm searches for shifts in the data and creates automatic snapshots with the state of all events at that particular time. Afterwards, frequently occurring patterns inside these snapshots are extracted.

We use the sequential GRITE algorithm presented in [91] by adapting it to work with our signals. Since the purpose of our method is to predict faults, we are only correlating signals depending on the occurrences of outliers in each of the signals. For this, we filter out the normal behavior and leave only the outliers. In order to simplify the correlation process, we replace each point in the signal with 0 in case of normal behavior and 1 for outliers, no matter on the real representation in the log. For example, if a failure is manifested as a lack of notifications, then the portion of the signal with value 0 will become a 1. At the same time some failures are represented by bursts of messages, like the second and third signal in Figure 5.3). In this case, the highlighted moments in the figure become 1s in the final signal and the rest will be set at 0. This representations makes all signals uniform and thus, it allows to represent signals as attributes that can be handled by the gradual itemset mining algorithms.

The sequential algorithm relies on a tree-based exploration, where each level is built by using information from the previous level. In its original form, the first level of the tree is initialized with all attributes. However in our case, the initial level is composed of the 2-pair correlations obtained with the signal cross-correlation function. Gradual itemset mining is a very complex and computationally expensive data mining algorithm so sequential methods cannot yet scale to large datasets. By merging it with a fast signal analysis module we were able to guide the extraction process toward the final result and so reducing the complexity of the original data mining algorithm. Recently, research on gradual itemset mining has focused on proposing parallel methods that are able to use multi-core architecture for the extraction of itemsets [90]. We plan to investigate the use of such methods online in order to adapt correlations to changes in the system.

Itemsets from the L level are computed by combining frequent itemsets siblings from the L-1 level by using a procedure for joining two itemsets into a larger one. Candidates which are more frequent than a pre-defined threshold are retained in level L and are further used in the next level.

In its usual form, gradual itemset mining algorithms look for patterns that take place at the same time in a subset of attributes. For example, the pattern (0$\rightarrow$1, -, 0$\rightarrow$1) related to Figure 5.3 describes the fact that frequently in the analyzed time frame, the first signal experiences an anomaly at the same time as the third, while the second signal can have any state. For our purposes, we are interested in associating signals that have a fixed delay one from another. For example, if one event type usually occurs T time units after another event type, these two signals will be shifted with T time units one from the other (as an example, in figure 5.3 the last two signals have a time delay of one minute, with the third signal following the second one). The correlation module must be able to capture this scenario.

We modify the initial algorithm to check different delays between signals by shifting one of the signals with the corresponding delay and applying the gradual itemset mining algorithm. To optimize the process we choose a small time window for the delay values based on the results given by the initial cross-correlation function.

The general gradual mining algorithm uses a comparison operator in $\{\geq, \leq\}$, meaning that it identifies when a signal changes its state either from normal to anomaly or the opposite. However in our case we only care about the decreasing patterns (if an outlier occurs in S_1, we want to find all other S_i signals where an outlier occurs with a fixed delay). We change the algorithm to only search for the $\geq$ operator. This means we are only looking for 0->1 moments when a signal goes from normal behavior to anomaly. For a better understanding of our hybrid approach we will present an example in the next paragraph that goes through all the steps used by the method.

Given a table set of signals S, a gradual item is a pair (S_i, θ_i) where S_i is an attribute in S and θ_i represents a delay in the signal. A gradual itemset $G = \{(S_1, \theta_1), ..., (S_k, \theta_k)\}$ is a set of gradual items of cardinality greater than or equal to 2.

In the example illustrated in figure 5.3, the initial set of gradual itemsets which is given by the cross-correlation function between all combinations of signals, is $S_{G_{init}} = \{\{(S_1, 0), (S_2, \theta_{12})\}, \{(S_1, 0), (S_3, \theta_{13})\}, \{(S_2, 0), (S_3, \theta_{23})\}\}$. This means that signal S_1 is correlated with S_2 and S_2 occurs θ_{12} time units after S_1.

The join function used in GRITE will return the merge between the sets in $S_{G_{init}}$ and create:

$$S_{G_1} = \{\{(S_1, 0), (S_2, \theta_{12}), (S_3, \theta_{13})\}, \{(S_1, 0), (S_2, \theta_{12}), (S_3, \theta_{12} + \theta_{23})\}, ...\},$$

with different delays. In case all delays are consistent, for example if $\theta_{13} = \theta_{12} + \theta_{23}$, S_{G_1} will have only one element. The testing part is left almost unchanged from the gradual itemset mining with the difference that we only use one operator.

We use two different thresholds in order to decide what patterns occur frequently and what patterns are sub-frequent, meaning they do not have enough appearances in the analyzed time frame, but have the potential of becoming frequent in the future. We keep both sets of patterns, but only use the frequent ones for prediction. The sub-frequent patterns are monitored as time goes by and their count is updated. If it exceeds the "frequent pattern" threshold they are upgraded and moved in the frequent set.

Location correlation

Large-scale systems contain a large number of nodes that are organized in an hierarchy. For example for the Blue Gene systems, nodes are gathered into midplanes and multiple midplanes form a rack. When analyzing different errors that might affect a HPC system we investigated the propagation behavior of each of them. Our observation show that some errors influence multiple nodes, depending on their location in the machine.

After a closer analysis, we observed that the propagation path for different error types follows closely the way components are connected in the system. For example, if a fan breaks, all nodes sharing the same rack will be affected. For a better understanding of the behavior of different event types, we analyzed the logs generated by Blue Gene/L and Mercury systems, and later by the Blue Waters system. We show that it is important to consider the topology of the network when modeling the propagation behavior of failures.

The heuristic used to extract location correlations is based on the offline correlation chains extracted in a previous step. We parse the logs and monitor each occurrence of a correlation $G_i = \{(S_1, \theta_1), ..., (S_k, \theta_k)\}$. Based on it we extract the list of possible locations for each chain $Loc_i = \{(L_{11}, .., L_{1k_1}), ..., (L_{m1}, .., L_{mk_m})\}$, where $(L_{11}, .., L_{1k_1})$ is a list of unique locations where events in the chain have occurred and m is the number of occurrences for the corresponding sequence of events. In case of a correlation that does not propagate events from one node to another, the list of locations will be composed of only one element for each occurrence: $Loc_i = \{(L_1), (L_2), ..., (L_m)\}$.

5.1.2 Dissecting event correlation

Our first set of experiments are made mostly on the BlueGene/L machine. Most modules from our framework are platform independent and so are easy to adapt to run on different machines. To demonstrate this and to compare the results from different systems, we made additional experiments on the Mercury system.

In this section we focus on analyzing the correlations we were able to extract with our method. First, we were interested to understand what type of patterns our method is able to extract in general. Table 5.1 presents

Memory error
correctable error detected in directory *
after 6 time units (one minute)
uncorrectable error detected in directory * capture first directory correctable error address..0
after 1 time unit
DDR failing data registers: * * number of correctable errors detected in L3 EDRAMs.* parity error in read queue PLB.*
Node card failure
midplaneswitchcontroller performing bit sparing on * bit *
after 44 time units
linkcard power module * is not accessible
after 4 time unit
problem communicating with service card, ido chip: * java.io.ioexception: could not find ethernetswitch on port:address 1:136
after 6 time unit
prepareforservice is being done on this part * mcardsernum(*) * * mtype(*) by *
Multi-line messages
general purpose registers: lr:* cr:* xer:* ctr:*
Component restart sequence
idoproxydb has been started: $name: d+ $ input parameters: -enableflush -loguserinfo db.properties bluegene1 ciodb has been restarted. bglmaster has been started: ./bglmaster –consoleip 127.0.0.1 –consoleport 32035 –configfile bglmaster.init –autorestart y mmcs_db_server has been started: ./mmcs_db_server –usedatabase bgl –dbproperties * –iolog /bgl/bluelight/logs/bgl –reconnect-blocks all n+

Table 5.1: Sequence of correlated events

several examples returned by our method. At a first look, we observed that the method was capable of detecting sequences of events that lead to a failure but was also able to capture the relationship between informational messages. For example, multi-line messages are identified by HELO as multiple event types. However, they have the exact same behavior so our correlation was able to cluster them together.

Messages generated during the installation of a component or during a restart are another example of informational messages. Our tool characterizes their behavior as silent signals since most of the time they do not appear in the log. Every time there is a restart, these event types' occurrences are regarded as outliers. This allows our system to correlate these signals with every other event type and extract the complete restart sequence. At the same time, these sequences do not give any benefit to our prediction since they do not affect an application's execution in any way. As a natural consequence, we investigated what is the percentage of correlations that are not useful in the prediction phase. This turned out to be a complex task since some messages might indicate harmless events in some contexts and indicate failures in others. At this point we only eliminated the obvious non-error sequences and analyzed the rest separately, as described in the prediction section of this paper.

We observed that only around 23% of sequences do not have any potential of predicting a problem in the system. We did this only for the Blue Gene/L machine because it offers a severity field that helped us in determining if a event type could be a failure in at least one context. We eliminated these sequences for the rest of the analysis. For the Blue Gene/L system this was done automatically by eliminating all sequences that contain only event types with INFO severity messages. For other systems, a system manager identifies which templates represent failures. This information can be used to filter correlations that do not contain at least one failure.

In-depth Analysis

First, we investigated how many events are in average part of a correlation chain. For this, we plotted the distribution of the event types that compose a sequence in figure 5.4. The figure shows that in general the sequences contain a small number of event types, the average length of the chain being 4 for both systems. However, there are some correlations containing more event types, 20% of them containing more that 8 events.

Next, we analyzed the time delay between correlations offered by our system. First we analyzed simply the pair of initial correlations and then the complete sequences. We observed that 33.7% of the correlations have less than 10 seconds delay between events, the majority (56%) having delays between 10 seconds and one minute and the rest having time delays of more than one minute. For both systems, about 2.5% of the sequences have more than 10 min between events. For a better understanding of how this large percentage of correlations affect the final prediction, we analyzed the complete sequences as well.

We plotted the time delay distribution between the first message indicating the beginning of a sequence and the last visible symptom. Figure 5.5 on page 80 presents the results only for Blue Gene/L but Mercury has a very similar distribution. Only 12.8% of the sequences do not offer any prediction window larger than 10s, 48.4% correlations offer between 10 seconds and one minute and there is a significant percentage for which the delay is larger than one minute. Moreover, the correlation system is able to extract some sequences with hours time delay between the first symptom and the final failure message.

We also observed that there is a relatively simple pattern between the confidence of a sequence and the delay between the first and last event in the sequence. The confidence of a correlation represents how frequent the sequence has been seen occurring before a failure type over all occurrences of this failure type. In general, for delays larger than 5 minutes, the larger the delay the lower the similarity degree between the signal and so the lower the confidence. Sequences with a confidence of over 95% usually contain the correlations between events that are generated close in time, in the order of seconds. However, there are some node card failure sequences that have high confidence and offer more than one hour prediction window.

In the following section, we look into closer detail for failures that have extreme time delays. In general, we observed that node card failures offer sequences with longer time delays. This should be reflected in a larger prediction window for these kind of errors, and as a consequence more time for fault avoidance strategies. For example, the node card failure presented in table 5.1 offers around 9 minutes (the equivalent of 54 time units) between the first and the last event in the sequence. Other node card examples show even one hour after the first symptoms occurs. The memory errors detected with our system, like the one presented in table 5.1, usually offer in average a one minute prediction window.

The Blue Gene/L system has a separate process, CIODB, that runs on service nodes and handles the job loading and starting. This process starts and monitors jobs, and updates the job table as the job goes through the states of being loaded, started and terminated. We observed that sequences or events related to CIODB usually have a very short time delay between

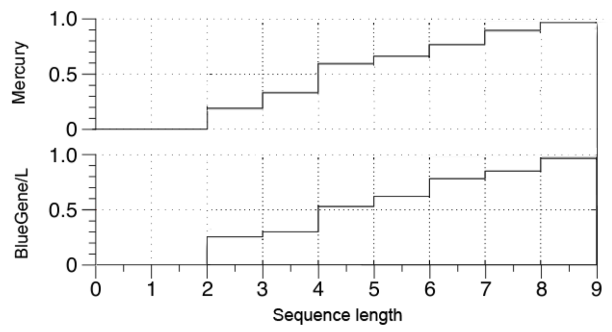


Figure 5.4: Sequence size distribution

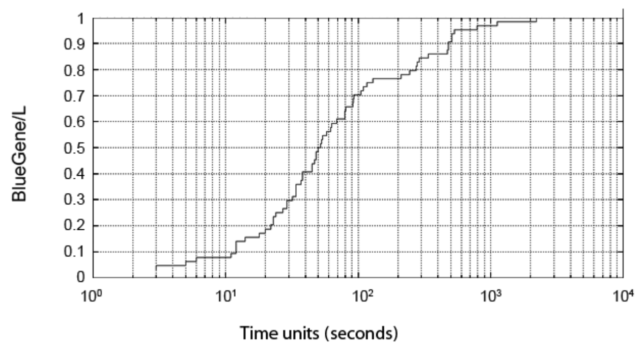


Figure 5.5: Time delay distribution between events in sequences

them, the majority happening almost at the same time.

In all our experiments, we used logs that offer less than 10 months of activity. Thus, there was no reason to implement any correlation updating modules since the changes in such a short time are not relevant to the whole lifetime of a system.

5.1.3 Dissecting prediction

Figure 5.6 shows an overview of the prediction process. The observation window is used for the outlier detection. The analysis time represent the overhead of our method in making a prediction: the execution time for detecting the outlier, triggering a correlation sequence and finding the corresponding locations. The prediction window is the time delay until the predicted event will occur in the system. The prediction window starts right after the observation point but is visible only at the end of the analysis time.

In the next section we analyze the prediction based on the visible prediction window.

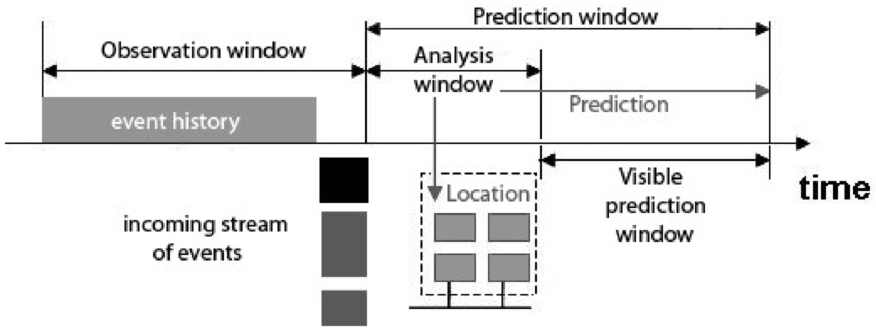


Figure 5.6: Prediction time window

Analysis

In the online phase the analysis is composed of the outlier detection and the module that triggers the predictions after inspecting the correlation chains. We computed the execution time for different regimes: during the normal execution of the system and during the periods that put the most stress on the analysis, specifically periods with bursts of messages. If the incoming event type is already in an active correlation list, we do not investigate it further since it will not give us additional information.

The systems we analyzed generate in average 5 messages per second and during bursts of messages the logs present around 100 messages per second. The analysis window is negligible in the first case and around 2.5 second in the second. The worst case seen for these systems was 8.43 seconds during an NFS failure on Mercury. By taking this analysis window into consideration we examined how many correlation chains are actually used for predicting failures and which failures are we able to detect before they occur.

The results on LANL systems showed 43% recall and 93% precision by using a purely signal analysis approach. However, at that time, we did not attempt to predict the location where the fault would occur. In this chapter, we focus on both location and the prediction window. We compute the results for the Blue Gene/L system.

We analyzed the number of sequences found with our initial signal analysis approach, the data mining algorithm described in [50] and the present hybrid method. Signal analysis gives a larger number of sequences, in general having a small length, making the analysis window higher. Also, the online outlier detection puts extra stress on the analysis making the analysis window exceed 30 seconds when the system experiences bursts. Due to our data mining

Prediction method	Precision	Recall	Seq used	Pred failures
ELSA hybrid	91.2%	45.8%	62 (96.8%)	603
ELSA signal	88.1%	40.5%	117 (92.8%)	534
Data mining	91.9%	15.7%	39 (95.1%)	207

Table 5.2: Precision and recall for different methods

extraction of multi-event correlation we were able to keep only the most frequent subset making the online analysis work on a much lighter correlation set. On the other extreme, the data mining approach looses correlations between signals of different types, so even if the correlation set is much smaller than our hybrid method, the false negative count is higher.

Table 5.2 shows the precision and recall obtained with three methods: i) ELSA signal, a purely signal analysis method by using cross-correlation to extract the failure patterns; ii) ELSA hybrid, the method described in this chapter and iii) Data mining, the GRITE based method described in this chapter applied on the raw log file before applying signal analysis. The recall value for the signal analysis method is lower than in our previous findings. This can be explained by the location prediction since now there is room for errors in this part as well. What is interesting is that the precision value for the data mining approach is higher than the other methods. This can be explained by the fact that the low number of sequences found by the data mining method are mostly the ones that do not show a propagation behavior. When running ELSA hybrid without checking the location we obtain a precision of around 94%. The results show that the hybrid method combines the precision given by the data mining approach with the recall of the signal analysis method.

We analyze in detail the results by breaking down the predicted events on different categories. The results are presented in figure 5.7, where each bar represents how often a certain type of error appears in the log as a percentage to all errors reported in the system. The dark portion of every bar represents the correctly predicted cases out of the total occurrences. We observed that the node card errors were the type that our system detected with a high rate, more than 80% of the occurrences were predicted. This is explained in the high confidence sequences obtained for this type in the offline section. We plan to analyze in the future the reason why there is such a low percentage in detecting network and cache failures.

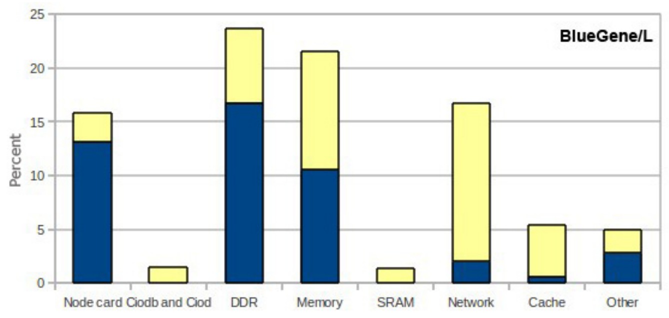


Figure 5.7: Recall breakdown on different categories

The total number of error messages in the log represent 18% of everything that is recorded in the log. An interesting thing we observed after this analysis is that even though the large majority of correlations are used at least once, there is a small set that is used frequently. More exactly, 3.12% of sequences are never used for prediction (the events occur only in the training set) and 23.4% are used in the majority of the cases.

We also analyzed the visible prediction window offered by the sequences used in this process and observed that around 85% of the prediction offer more than 10 seconds after the analysis window ended, out of which more than 50% offer one minute or more and around 6% more than 10 minutes.

This means that fault avoidance techniques that take a checkpoint or migrate a process in less than one minute could be applied on 42% of the total predicted failures on Blue Gene/L, respectively 20% of total failures. When using a fast checkpointing strategy, like the one from [65] the total number of failures for which avoidance techniques could be applied increases to 40%.

5.2 Parameter influence on the results

5.2.1 Preprocessing parameters

HELO generates templates that consist of constant words and variables. Variables identify manipulated objects or states for the program and are replaced by wildcards. In case constants are mistakenly replaced by wildcards the template becomes too general and, when variables are identified as constants by HELO, we call the corresponding templates too specific. In order to compute HELO's accuracy, we analyzed the Blue Gene/P system

MTBF	Failure distribution	Nodes	System lifespan	Propagation	Lead time
1day	Weibull scale=8116.7 shape=0.387,187	40960	one year	Yes 20% of failures	Weibull mean=50s

Table 5.3: System parameters

which offers error_codes for every message line in the log file. The following line:

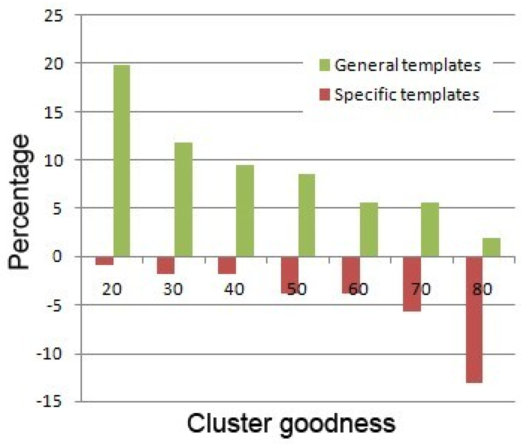
26124022 KERN_080A KERNEL _bgp_unit_ddr _bgp_err_ddr_SSE_count WARN 2009-01-05-00.41.28.758978 - 0 - ANL-R45-M0-512 R45-M0-N11-J17 DDR controller 1, chipselect 0 single symbol error count 19747

is an example message that appears in the log generated by this machine. The string _bgp_unit_ddr represents the error code and "DDR controller 1, chipselect 0 single symbol error count 19747" represents the message description that will be used by HELO for extracting the templates.

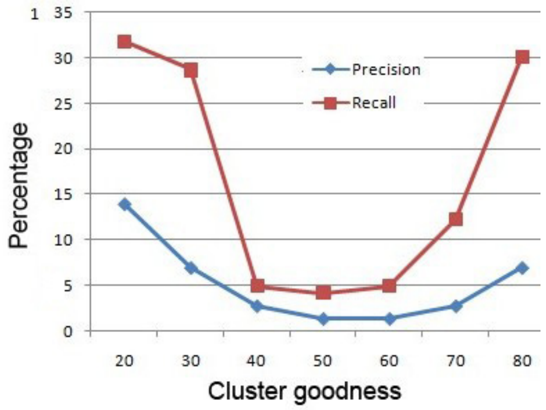
We plotted the ratio of general and specific templates generated by HELO compared to the error codes of Blue Gene/P in Figure 5.8a and how these differences affect the final prediction in Figure 5.8b. Cluster goodness represents the similarity threshold that defines when two messages are part of one single events. Depending on the cluster goodness, there is a 2 to 30% difference between the template set generated by HELO and the error codes of Blue Gene/P. However this is translated into a much smaller difference when looking at the impact on prediction, the highest impact showing a median difference of only 5% for recall and 3% for precision. Only for extreme values the impact is higher. We argue that this step affects in a small way the final prediction so automatic processes provide great benefits compared with human interaction without significantly affecting the final results.

5.2.2 Prediction parameters

All modules implemented in ELSA have online phases where they update the data generated in the training phase. In general, current research is not updating the correlations found offline and thus have limitations when they are using a short training set. We believe this limitation makes the prediction unrealistic when used on real production systems. To study the impact of not adapting the correlation set on the prediction's result, we used the data



(a) Percentage of incorrect templates



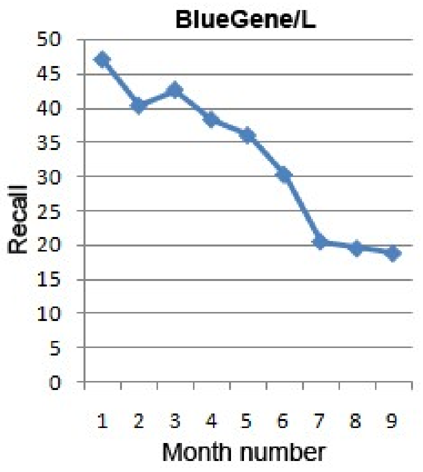
(b) Precision/Recall decrease

Figure 5.8: The influence of the HELO "cluster goodness" parameter on the template list and on the prediction results

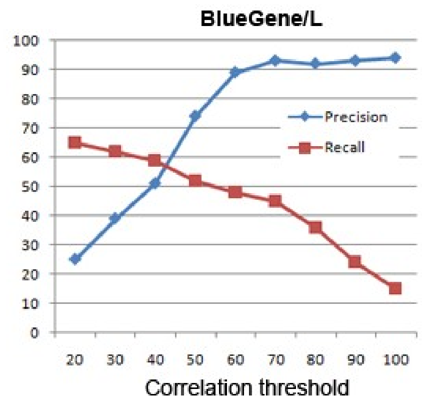
collected by ELSA during the training phase to predict the next 9 months of Blue Gene/L and plotted the recall for each month in Figure 5.9a. We have similar results when using the synthetic logs, however due to space limitation we did not present this figures.

It is clear that the prediction keeps a high recall value only for the first couple of months and then decreases dramatically. We argue that by adapting the correlations and signal characterization over time we were able to keep the recall value almost constant throughout the entire studied life cycle of the system.

For some fault avoidance techniques the cost of predicting a failure that does not appear in the system is low compared with experiencing an unpredicted failure. This is, for example, the case of object migration with



(a) Recall on different months without updating the correlation list



(b) Recall/Precision when varying the "correlation threshold" parameter

Figure 5.9: The influence of ELSA's parameters on the prediction result

Charm++ [66]. Therefore, we did a study of the recall/precision trade-off in Figure 5.9b. These values represent the precision/recall from analyzing all 9 months of log data and using updates every 3 months. In general, the recall increases when using low threshold values for deciding when a correlation is strong. Interestingly, the maximum recall value reached by ELSA is 63% which is close to what we observed in the previous section as being the amount of predictable failures. However, the cost in precision is really high, making more than 70% of ELSA's predictions wrong. It is also noticeable that the precision decreases at a higher rate than the increase in recall. Depending on the fault avoidance technique different values might be the best option.

5.2.3 Discussion

Accurate predictions are necessary for proactive fault tolerance solutions. These solutions have the benefit of reducing the overhead due to fault tolerance actions and the amount of lost work due to predicted failures. However, an extra overhead is added due to wrong predictions. The trade-off between this overhead and the benefit is highly influenced by the predictors recall and precision as well as by the cost of the fault tolerance action. We believe that understanding current prediction methods and their limitations is crucial in designing failure avoidance techniques for exascale systems.

The correlation extraction method has the highest impact on prediction results. Therefore the choice of the methodology is the most important part of the prediction. We plan in the future to analyze various algorithms and study their results on large-scale production systems to get a better understanding of their limitations. Data mining algorithms have particularly poor results on noise and periodic signals. Our observation that failures do not affect in the same way the system and are represented in different ways in system logs allowed us to analyzed failures differently and, in the end, offer more accurate predictions.

Adapting the set of correlations and behavior characterization is a necessity when working on real systems. Correlations using the last couple of months become unusable after less than one month of predictions. Since patterns are no longer useful after only a couple of months and since new patterns are added throughout the lifetime of a systems due to upgrades or configuration changes, it is no longer viable for system managers to manually tune the patterns extracted.

With the implementation of more accurate failures predictors there have been developed a number of mathematical models [60, 92] that deal with characterizing the benefit of merging predictors with current checkpointing protocols. We further look at analyzing the benefit of a hybrid preventive and proactive checkpointing strategy in the following chapter.

5.3 Online failure prediction

The prediction methods used in literature, as well as the one presented in the previous sections, follow the same general design. The historic log files are divided in two parts: training and testing. Predictors use a variety of methods in the testing phase in order to learn patterns and correlations between different events in the system. This phase usually uses 10% of the whole log and can take between a couple of weeks [9] to months [57]. These patterns are then used in the second phase of the analysis by applying them on the second part of the log in order to predict failures. Based on the actual failures from this part, recall and precision values can be computed for a given method.

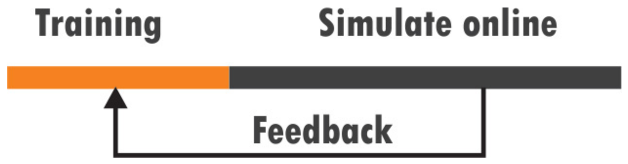


Figure 5.10: Failure prediction: simulate online

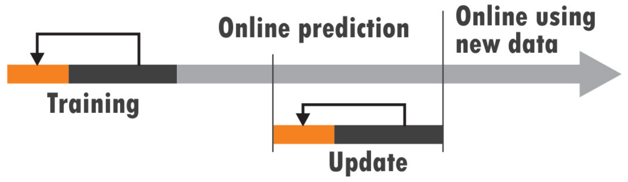


Figure 5.11: Online failure prediction

Every step in the training method contains parameters used to decide when a pattern is reliable enough to be used in the second phase. As shown in the previous section, these parameters have a high influence on the final results of a predictor. Depending on the results obtained on the testing part of the log, the parameters can be tuned and the training analysis can be redone in order to increase the accuracy of the final results.

Choosing and tuning the parameters is usually a manual process and can be done using domain knowledge about the system or using previous experiences with other similar systems. Repeating the prediction for the same testing piece of log until obtaining the best possible results is a good way of analyzing the limitations of a predictor. However, it is not a valid methodology when deploying a predictor to work online on a real system since it does not provide a way of preparing for unknown events. Moreover, any manual process is unrealistic when dealing with supercomputers at petascale size.

The solution we propose is presented in figure 5.11 and is currently implemented in the ELSA toolkit. For online prediction there is a historic log file and a stream of events that are generated as the system continues to run. The historic log file is divided in two parts as before, one for training and one for testing. The training is done automatically in order to achieve the best results on the second part of the log, either by implementing rules as to how parameters modify the precision and recall ratio or by a brute force strategy. By best results, we mean tuning the parameters in order to obtain either the best possible precision, recall or a ratio between the two. We call this process simulate online, because the methodology takes advantage by the fact that the testing phase can be redone multiple times.

After the best results have been reached, the process stops and the parameters are used online on the incoming stream of events. The methodology can be tested on a historic log, by dividing the log into 3 parts, one for training, one for simulate online and the rest for online predictions. The best parameters are chosen based only on the training and simulate online parts either manual or automatic, then online predictions are made on the third part of the log only once and the results of this one time execution defines the recall and prediction values.

In time the learned patterns used for prediction become less and less accurate because of new updates in the system and so both precision and recall decrease over time. Values for this degradation can be found in the previous section. Our online methodology (figure 5.11) deals with this problem by triggering a new session of training and simulate online in parallel with the online prediction each time the precision or/and recall decrease below a threshold.

We replace the manual process with an automatic algorithm. This online methodology can be applied to any failure predictor that works with a predefined set of parameters. The historic log file is divided in three parts, one for training, one for validation, and one for testing. The training and validation phases are repeated by an automatic supervisor that is using a different parameter mix each time. The testing phase is done only once after the automatic algorithm has finished finding the best set of parameters for a given goal.

The automatic algorithm works on a couple of input files that need to be provided by the user. These files contain information specific to the failure prediction method that will be used. In this chapter, we use the ELSA predictor [92] to demonstrate how online failure prediction can be achieved, but any other predictor can be used as long as the files needed by our algorithm are provided. In the current format of our online methodology there are three files needed. The first one is an xml file that describes the parameters used by the predictor with a range of values that need to be inspected. The second file contains information about how the executables are using the parameters that are described in the first file. The last one describes the objectives of the validation process. A snapshot of these files is presented in Table 5.4.

```
Example parameter description
< parameters >
< param name='correlation_threshold' >
    < range >
        < min > 50 < \min >
        < max > 100 < \max >
        < step > 5 < \step >
    < \range >
< \param >
...
< \parameters >
```

```
Example file describing executables
< rule >
    < executable > elsa_preprocessing < \executable >
    < param name='correlation_threshold' > $1 < \param >
    < param name='input_log' > bgl_log < \param >
...
< \rule >
```

```
Example objective file
< objective >
    < recall > MAX < \recall >
    < precision > GT60 < \precision >
< \objective >
```

Table 5.4: Input file entries required by the online methodology

In our example, the correlation_threshold parameter is used by the preprocessing modules of ELSA (specifically the elsa_preprocessing executable is using it as its first input parameter). This specific parameter describes the minimum threshold required by the pattern detection module in order to consider two separate events as being correlated. The values that need to be inspected for this parameter, begin from 50 (events that occur together 50% of the time) and end with 100 in 5 unit increments. The objective of the online methodology described in the last file is to find the maximum value for the recall, while preserving a precision of more than 60%. At the time of this paper, we implemented the basic objective functions: MAX, GT (greater than) , GE (greater or equal). In the future it might be desirable to be able to specify a ration between the precision and recall or any cost function as an objective function.

The online methodology uses the provided files to search the space of results offered by different parameter mixes, in order to chose the best one for a given objective. Due to failure predictors generally having a very large search space, we implemented a data mining algorithm used to find pattern and trends between the parameter values and the precision and recall values obtained for them. We use the GRITE algorithm [91], a specialized data mining algorithm that extracts complex order patterns in the form *"The more/less* X_1,X_2,*... the more/less* Y_1, Y_2*"*, where X_i represent parameters and Y_i represents recall and precision. We store every entry containing the set of values used as input parameters and their corresponding precision and recall into a data structure. We apply the data mining algorithm on this data structure and save the generated patterns into a separate structure. These patterns are used to decide what mix of parameters to use in the next iteration. We apply the classical sequential GRITE algorithm (see [91] for the detailed algorithm) every time there is an entry that does not correspond to the patterns already extracted. We call this method a guided space search, since not all parameter combinations are tested. The experiments done with the ELSA predictor over the Blue Waters system have shown that less than 15% of parameter combinations were necessary to reach a solution. One example of such a pattern is the ratio between the correlation parameter and recall seen in Figure 5.9b on page 5.9b. Specifically, each time the correlation_threshold is increased, the recall decreases. This pattern guided the online methodology to start with low values for this parameter and go up until the precision reached the 60% limit.

After the results best fitting the given objective have been reached, the process stops and the parameters are used online on the testing phase or on an incoming stream of events if the predictor is running as a daemon in the system.

5.3.1 Results

We tested our method, first onto the same Blue Gene/L log used in our previous studies and on the Blue Waters system. For the training phase, we chose to tune the parameters in an automatic way with a hybrid version of domain knowledge and brute force. There are three categories of parameters in ELSA's testing phase, one in the classification phase, two in the outlier

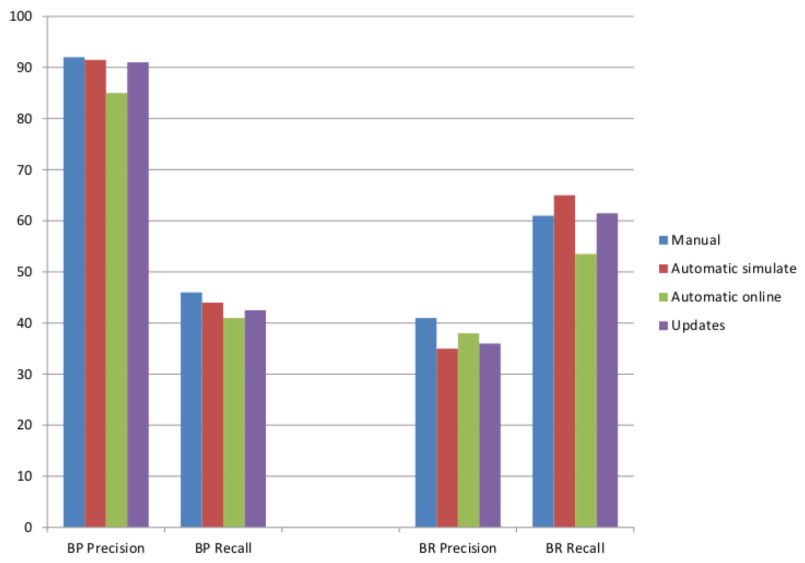


Figure 5.12: Precision and recall for simulate online and online

identification phase and two in the correlation extraction phase. The previous section presented a few of the most influential parameters. Some parameters have a direct relation with both precision and recall (for example, it is clear from Figure 5.9b on page 86 that an increase in the threshold parameter decreases the recall and increases the precision). For the second type of parameters there is no straightforward relation. Our method starts with all parameters in brute force mode by trying all possible values. After computing values for the first few tries, our method uses a regression algorithm in order to find relations between thresholds and recall/precision values. If a relation is found, a guided search is used for the rest of the threshold values.

The first experiment was done on the Blue Gene/L log which was downloaded from the computer failure data repository [93]. In the previous section we focused on offering the best precision possible because we desired to have as few wrong predictions as possible. There are fault avoidance techniques for which the overhead of a misprediction is higher than the benefit of covering a larger set of failures in which case highest precision is the best solution. In the next chapter 6, we investigate the impact of combining our prediction method with a multi-level checkpointing strategy and we observed that a decrease in recall has a higher impact on the benefit of this hybrid fault tolerance method than precision. For this reason, in this chapter we remade our previous experiments on Blue Gene/L focusing on both best precision and best recall. Figure 5.12 shows the precision and recall for different scenarios. The left part of the figure shows the results when focusing on obtaining the best precision possible and the right side focuses on best recall.

The first column in each bar set (the blue column) shows the results for the manual simulate online method. In this case, we divide the log in two parts (training and testing) and manually change the parameters used in the testing phase in order to get the best results on the testing phase. The second column from the left (the red one) in each bar set represents the results when using the automatic script for finding the best parameters. For computing the first two bars, the first part of the log is used for extracting patterns and the last part is used for computing the recall and precision value. The other two experiments compute these values also using the last part of the log but using the first two parts for training as described in figure 5.11 on page 88. This way all four experiments compute the recall and precision by predicting failures only on the online part of the logs and by using only data from the training phase. The online methodology uses the simulate online part of the log only to tune the parameters and to optimize the correlations found in the training part. If a correlation cannot be found in the training part and its present in the simulate online part our predictor will not see it.

The results using the automatic script are very similar to the ones obtained by manual tuning. For the best recall values, the automatic method gives a better recall, however, paying the price of having a lower precision. The variations happens because the automatic algorithm goes through a larger set of parameter combinations. Overall, the difference is less than 1% which shows that the manual process can be completley replaced by our automaic method.

The third column (green column) presents recall and precision values for the online methodology when no updates are being made and the last column shows the results when there is one update after one month of execution. When no update is made, the recall and precision values both are much smaller than in the simulate online case, having a difference of almost 5% compared to the manual process. However, after one update the values become again very similar.

The results on Blue Gene/L shows that the online methodology gets similar results by choosing automatically the parameters that otherwise should be manually tuned and updated every couple of months. However, when we

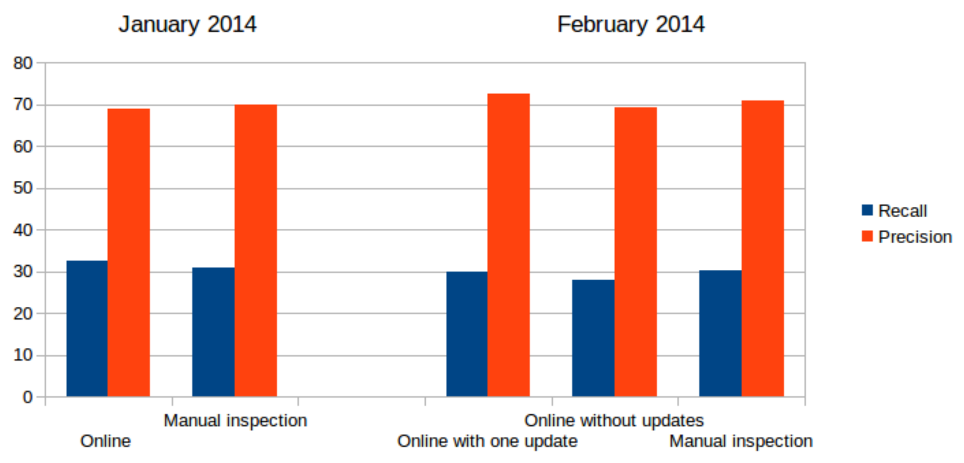


Figure 5.13: Precision and recall for the Blue Waters

applied the same strategy on the Blue Waters systems we observed a couple of limitations. We will focus on analyzing these limitations and proposing solutions in the next section.

5.4 Results on the Blue Waters system

Figure 5.13 presents the precision and recall of applying prediction on the Blue Waters system. We focused on achieving the best possible recall, at the same time trying to keep the precision at a reasonable value. Based on the study from the next chapter, we chose this value to be 60% since it offers the best results for balancing the overhead of a misprediction with the benefit of covering a large set of failures for checkpointing strategies.

Figure 5.13 shows the precision and recall obtained for the Blue Waters system for both manual tuning of the parameters (manual inspection bars) for each individual month, and by using the online methodology with and without updates (online bars). The manual and automatic methods give similar results, which shows that the online methodology is a valid strategy on larger systems. The automatic method gives a better recall value for January 2014, however, paying the price of having a lower precision. The variation happens because the automatic algorithm goes through a larger set of parameter combinations. Overall, the difference is less than 1%, which shows that the manual process can be completely replaced by our automatic method.

The last two sets of columns present the recall and precision values for the online methodology, when no updates are being made, and when there is one update after the first month of execution. When no updates are made, the recall and precision values are slightly smaller showing a difference of around 8% compared to the manual process (which was optimized for each month). This difference corresponds to a power failure that propagated on multiple nodes inside one of the cabinets. This failure was not predicted so all node failures became false negatives. After an update, by including January in the validation process, this failure was correctly predicted, which corresponds to the 8% increase. We will analyze into detail all these correlations in the next sections. In general, from our previous observations, an update is required every few months, depending on the system.

We further investigate the reason for the low recall value obtained for the Blue Waters system when compared to previous generation HPC machines, like Blue Gene/L. For prediction purposes each node affected by a failure is a potential false positive or negative. For example a fan malfunction can bring a increase the temperature on a cabinet consisting of 96 four nodes. Depending on how many nodes start presenting failures, a prediction will have to identify multiple failures.

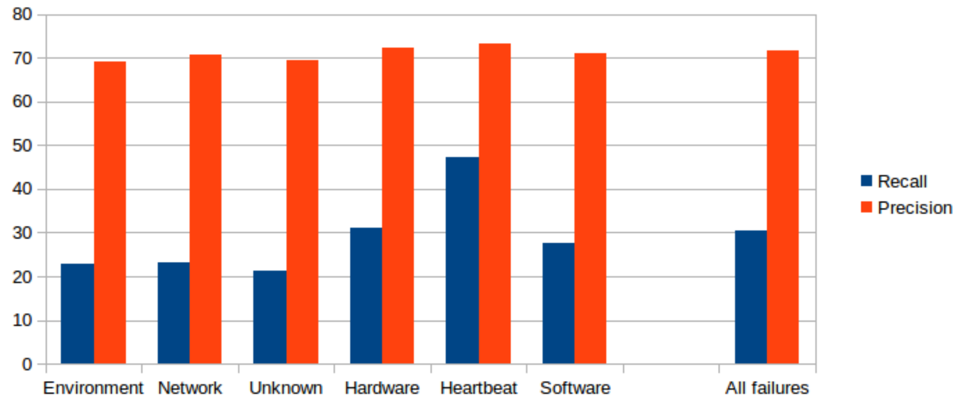


Figure 5.14: Breakdown precision and recall for different root causes

In general our method has good results for hardware and heartbeat failures and less than desirable recall values for the software, network and environmental category. We will analyze into more detail each category in the next

Hardware	Software	Environmental	Network
Memory 33.12%	FS 27.2%	Fan Tray Assy 27.02%	Gemini Lane 66.41%
CPU 27.04%	Scheduler 18.9%	XDP Valve 21.62%	Cabling 8.39%

Table 5.5: Most frequent types of failures

section.

5.4.1 Detail breakdown of prediction results

In order to better understand the prediction results, we further break them down on more specific failure types. Table 5.5 presents the most frequent failure types for each category: hardware, software, environmental, network. We correlate these specific types with log messages and analyzed their behavior.

An analysis similar to the one in the previous subsection shows that memory failures are preceded by other failures in 21.9% of the cases and act as precursors for other failures in 20.8% of the cases. This happens because we discovered that memory failures are self-correlated. When looking at all events and not only failures, memory failures have precursors in 72.4% of the cases. We investigated the message in the log file describing the Memory Error Check and observed that the message is self-correlated in 87.31% and is preceded by Correctable Memory Errors in 69.5% of the cases. However, from the event's perspective, Correctable Memory Errors lead to a Memory Error Check in less than 10% of the cases. The signal analysis outlier detection modules implemented in ELSA caught the fact that, even though Correctable Memory Errors happen frequently and without causing any failure, when their frequency increase in a short time interval it leads to Memory Error Check in over 75% of the cases. When our predictor is only considering cabinets and not node predictions, memory errors can be predicted with a 53% recall. The complexity of error propagation (85% of following memory failures are on different cabinets than the precursor) is making the prediction reach only 31% recall when node prediction is required. We will analyze this further in the next section.

Node Heartbeat Faults messages are self correlated in 98.66% and their location propagation is constrained within the same cabinet. Moreover, in

98% of the cases they are preceded by a Blade Power Failure message 400 to 800 seconds in advanced. With a recall of 62% and over 72% precision, this type of failure has the best results. Moreover, 7.1% job crashes in the system follow a node heartbeat failure so these predictions could be also useful at the application level.

Lustre failures are very complex and in general have many, but low confidence, correlations. Most of Lustre failures are not predicted, with some exceptions. OST Write Operation Failed is self correlated in 91.1% cases and has a precursor event in 89.2% of cases. This precursor refers to a Write Operation Timeout. In over 75% of the cases the failure occurs on the same cabinet as the precursors. While this type of failure does not cause application crashes it does lead to a MDT Failure within the next 45 to 950 seconds, failure that causes job crashes in 23% of the cases. Both these failures were correctly predicted by our method.

In general, network failures have few precursor events in the log. The Gemini Routing Table corruption for example has no precursors in other failures or in other events. System administrators are still working to understand the root cause of this failure. However, it is visible from our correlations that it leads in 72.41% of the cases to a Network Quiesce Error. For this reason this second failure can be predicted with lead times of over one minute. In general, we observed that network failures need different types of precursors since logs files are not sufficient for predicting it. Our preliminary work indicates that performance metrics could predict network failures, improving the prediction. In the future we will focus more on network failures.

Environmental failures are in general in strong relation to basic performance metrics, like temperature, power or fan speeds. We created log messages each time we see an anomaly in the following metrics: PCB temperature, INLET temperature, XDP air temperature and fan speeds. We use the ELSA signal analysis module to detect outliers. The new introduced messages seemed good precursors for a large percentage of the environmental failures. After this step, the recall value for the environmental failures has increased to almost 40% having the same precision as before (when predicting cabinets and not node failures).

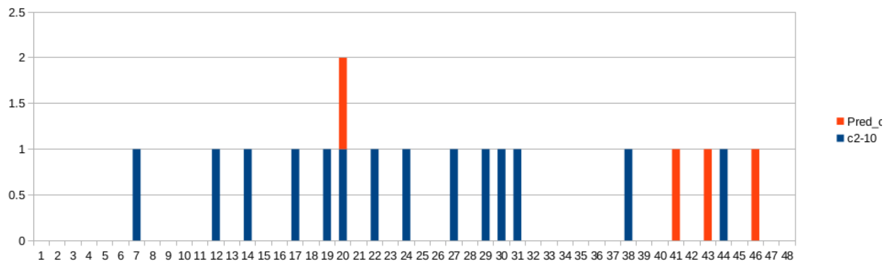


Figure 5.15: Errors propagating in the same cabinet. Predicted failures are marked with orange and unpredicted failures with blue

5.4.2 Location propagation

One of the main problems brought by the complexity of Blue Waters is the fact that the patterns between precursor events and failures are now more complicated. Often, precursors from one node location indicate a failure on a completely different node. Moreover, some problems, like the majority of Lustre failures, although they have precursors on the same set of Sonexion nodes, they crash applications running on sets of compute nodes. In this section we will analyze the relation between the location of precursors and their corresponding failures.

On Blue Waters, the locations for compute nodes have the following format: c0-0c0s0n0 which represents the cabinet id, cage id, slot id and node id. The location prediction used by ELSA keeps a structure with learned correlations between locations based on the correlation chains between events and failures. Our first optimization to this method uses a simple algorithm that finds patterns between different locations using the location ids. We investigate, for each of the correlation chains that contain events appearing on multiple locations, if they are propagating beyond the same node/slot etc. If, for example a precursor and its error are always on the same slot but not necessary on the same node, the prediction engine could use this information and predict that the whole slot will fail.

This new methodology uses a hierarchical algorithm for locations prediction. It basically over-predicts the number of nodes that fail in order to increase the true positive base. Figure 5.15 presents an example of a heartbeat failure caused by a power failure that affected 17 nodes in cabinet c2-10 (out of the 96 total nodes). Only 25% of locations are predicted correctly. If the prediction could predict the whole cabinet then all failures will be captured. We implemented this methodology in ELSA and after running it on Blue Waters we obtained the results from figure 5.16. We observed that 87% of the correlation chains contain events that appear in the same cabinet, out of which 43% appear in the same cage (from the same cabinet), 48% in the same slot and 71% on the same node. In Blue Waters, a slot has 4 nodes, a cage has 8 slots (32 nodes) and a cabinet has 3 cages (96 nodes). This number gives the upper bound on the recall improvement when considering the new method (when all failures propagating are corrected predicted) and a worst case over node estimation (for example when a cabinet is predicted but only one node failed, the method has a mis-prediction of 95 nodes). The prediction module is using the hierarchical location method by predicting different location classes (node level, slot level, etc.) depending on the chains used to trigger the future failure event. A true positive in this scenario represents a prediction for which, firstly, the failure occurred, and, also, the set of locations affected by the failures is a subset of the predicted locations.

In order to make the location propagation method even more general, we mapped the location ids of each compute node on the 24x24x24 3D torus network used by the Blue Waters system. We looked at patterns at this level rather than just in the location ids. In chapter 3, we presented the topology used by the Blue Waters system in figure 3.2 on page 36.

The algorithm used for extracting patterns in the torus looks at relations between precursors in a chain and the predicted failure, and also at location propagation patterns for the same failure. For this paper, we implemented a simple neighbor regression algorithm that looks for mathematical relationships between nodes on each of the three axes. We use the method from [94] that extracts repeatedly occurring shape structures from a set of solid models for design-rule mining, and model data compression. Basically after eliminating outlier nodes, the algorithm extracts a 3D shape of minimal dimension that captures all the active nodes (nodes where failures are occurred).

We recomputed the precision and recall values considering the new hierarchical prediction method with its corresponding definitions for true positives. Figure 5.17 presents these results. The heartbeat, hardware, and network failure types present very little improvement compared to the initial results, each for a different reason. Network failures did not have many high accuracy correlations to begin with, so the main reason for the low results is

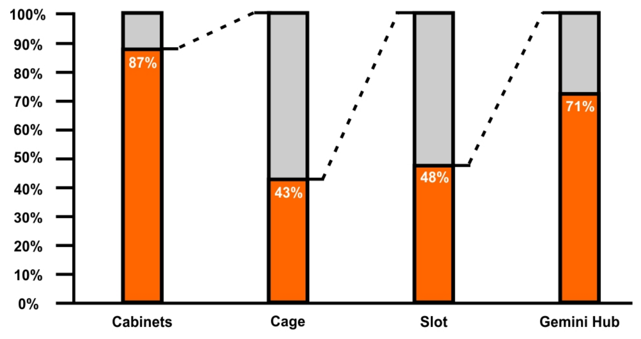


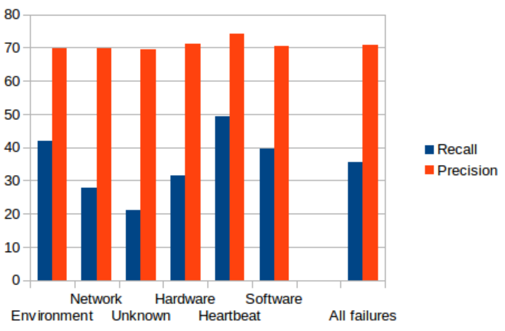
Figure 5.16: Location propagation results

not location mis-prediction, but a lack of precursors events in the logs. 85% of network failures that were mis-predicted because of wrong location were transformed into true positives with the new method. This accounts for the 1.2X improvement factor from figure 5.17b. The heartbeat failures, on the other hand, had a high recall value from the beginning, because most of the failures were correctly predicted even without using the new methodology. Cases like the one presented in figure 5.15 are rare and account for the 1.04X improvement factor. Hardware failures do not propagate on more than one node in over 90% of the cases, so the lack of improvement was to be expected.

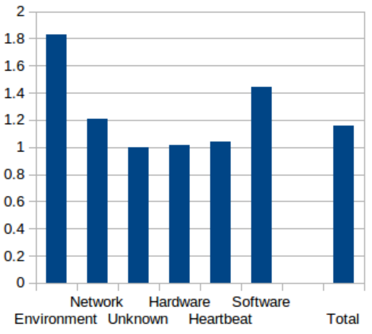
Software and environmental failures had the most improvement since they are the top failure types that propagate in the system. With the new location propagation method, the predictor was able to forecast 41.8% of all environmental failures and 39.7% of software failures. Moreover, almost half of failures reported as heartbeat errors (49.24%) were correctly predicted, which corresponds to a total recall of 35.5%. This means that more than a third of the failures on Blue Waters were correctly predicted both in time and space. Moreover, our analysis indicates a couple of potential future directions for improving even further this result.

5.5 Comparison Blue Waters results with smaller systems

Similar to section 4.4.2 in chapter 4, we are comparing the results on different systems side by side in order to better understand the differences that cause



(a) Precision and recall



(b) Improvement factor

Figure 5.17: Prediction results when using the location propagation method for 70% precision

Category	Blue Waters	Blue Gene/P	LANL systems
Hardware	40.9%	45.7%	49.1%
Software	28.2%	49.2%	53.1%
Network	22.3%	41%	43.1%
Facility/Environment	26.1%	31.4%	32.7%
Total	32.9%	47.3%	49.8%

Table 5.6: Recall of different failure types for different systems

the gap in recall values. We are using the automatic online method in ELSA to find the best recall while keeping the precision value as close to 60% as possible. Following the structure given in table 4.6, we show in table 5.6 the recall obtained for different general categories for each system.

Hardware recalls are higher for smaller systems compared to Blue Waters. However, the recall values for this category is somewhat similar to all three systems. When looking into detail at the results obtained for the main hardware failures (Memory, CPU and L1 data cache), the recall for the Blue Waters system is around 15% less for each of the failure types. Since the failures are more complex and the propagation behavior harder to predict, we were expecting smaller overall recall values. We believe that having longer training phases will decrease the gap between the systems.

Blue Waters and Blue Gene/L have similar percentage of network failures and much smaller than software or hardware failures (Table 5.6). The recall obtained for Blue Waters is half as much as for the other two systems and it cannot be explained by the small training phase or by the size of the correlation chains. After manually analyzing the logs, we observed that there are not a lot of precursors in the logs for this type of failures. Since the total percentage of network failures is around 10-15%, by increasing the recall value for these failures to what was obtained for the LANL systems, we would get the overall recall to 35% (from the initial 33%). Therefore there is not much to be gained from improving recall for network failures.

Facility failures have the lowest recall value for all systems. We believe that including environmental and performance metrics into the analysis will help prediction for this category of failures. However, they represent less than 3% of all failures so improving the results will not influence greatly the overall recall.

Software failures have increased in percentage as clusters increased in size. Specifically, there are more and more complex filesystem and scheduler failures that could not be predicted. Figure 5.18 shows the recall value obtained for the main software failures. We did not have information about scheduler failures for the Blue Gene/L systems so they were not included in the study. Considering the other two analyzed systems had low recall values for scheduler failures, we believe that including scheduler failures into the results for the Blue Gene/L would negatively influence the overall recall.

Operating system failures represent a large percentage (over 60%) of software failures affecting Blue Gene/L. The recall results for this type of failures are the best for all systems. This explains the good results obtained for the LANL and Blue Gene machines. For the Blue Waters system, the OS failures represent only 22% of software failures while file system represent over 40%. The low recall values for file system failures influence the total recall for Blue Waters and we believe this is the area that needs more investigation. If we could increase the recall for the file system failures to what is obtained for the LANL system, the resulted total recall becomes 48% and if the file system recall can be increased to 100% we would be able to predict 53% of all failures that affected Blue Waters. We investigate creating specific detectors for the types of failures that do not present precursors in the log files in chapter 7.

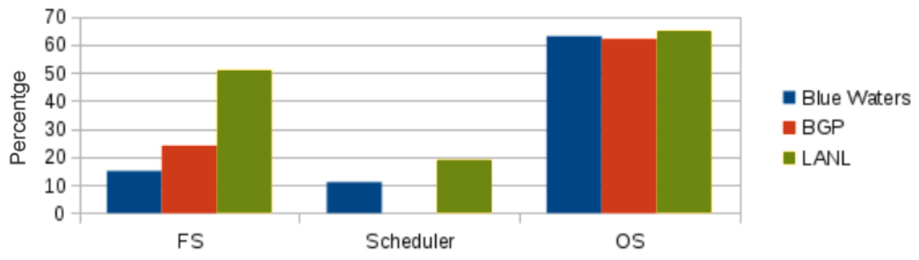


Figure 5.18: Recall for the main software failures

5.6 Prediction from the application's perspective

Figure 5.20a shows the usage of the system for the analyzed period and the number of failures in the analyzed time frame summarized per cabinet. In general, we would expect higher failure rates with heavier workload. However, our results show no correlation between number of failures and the average load of the corresponding cabinet. Moreover, a failure in the system most of the time does not seem to impact a large numbers of jobs. At a closer analysis, we observed that only around 44% of the failures led to at least one application crash. The same analysis shows that 62% of the failure types predicted by ELSA refer to failures that lead to application crashes. Filtering out from the analysis all failures that have no effect on any of the running applications corresponds to an increase in the recall value of 5%-15% compared to failure prediction at the system level.

Location prediction gets a slightly new meaning when application crashes need to be predicted instead of system failures. If our method predicts a failure correctly in time, but the failure occurs on a different node, our previous method will give a false negative and a false positive in the final results. However, if an application was running on multiple nodes, one of which corresponds to the predicted node, and the application takes global preventive actions, the mis-predicted failure could be masked. Depending on the fault avoidance strategy, a predictor that only looks at applications as a whole and not as a set of running nodes could increase the recall significantly. By taking the lead time and the new definition of location prediction into consideration we recomputed the results and obtained 35% recall and 70% precision for predicting application failures.

5.6.1 Details statistics

We extracted information about applications running on Blue Waters for 6 months of production activity from May 2014 to October 2014. Our dataset includes 1,051,353 user application runs of more than 1,500 code bases, 396,178 jobs and 201,502,257 error events stored in over 2TB of logs. During this time over 60% of the total user runs are XE applications and the rest XK applications using CPU and GPU accelerators.

Many applications do not execute for a long time. During the analyzed time frame around one third of the full-scale XE applications ran for less than 5 h, with a median of 1.2 h. In general, over 50% of the jobs used less than one cabinet, or 96 nodes and over 80% use less than 10 nodes. Large-scale applications that occupy close to 25% of the machines represent less than 5% of all applications running on Blue Waters. Applications that take more than 50% of the machine are rare and represent less than 0.5% of all applications. These percentages are obtained by looking at the count of the jobs. When the same analysis is done from the node hours used on the machine, over 50% of the total time is allocated to jobs that run on over 2,000 nodes.

We analyzed all exit codes of applications and determined the cause of their crash. Overall, in the analyzed time frame, three quarters of the applications are successful, 5% not completing within the allocated wall clock time, 15% terminating abnormal caused by user-related problems. The left of 5% represent application instances that are terminated due to system-related issues caused by any of the considered system errors.

Failures do not necessary crash applications. We observed that 38% of failures that experience at least one system error during their lifetime terminate in a crash, and over 50% complete successfully.

We computed the ratio between the number of applications completing successfully during a system error of different category to the total number of applications experiencing this type of failure. We notice that smaller applications present higher numbers for all types of failures than large ones. A more visible decrease is presented for interconnect and system problems. The ratio for small applications (<96 nodes) is relative constant for all failure types, between 35-40%. Applications that take 25% of tha machine present an uniform decrease of 15% for all failure types, while applications running on more than 50% of Blue Waters nodes show values of 10-15% for lustre and node failures and 30-35% for Gemini and LNET errors.

The ratio for applications running at full scale are not uniform and depend heavily on the failure type. This is because at full scale application, users generally adopt many resiliency mechanisms to protect application against variety of errors. Also, these applications usually run for only a few hours which means that they will be limited in how many types of failures they will encounter.

Operating system (OS) failures, like kernel panics, are very critical at any scale. Although, they are not frequent they crash applications almost in all cases. File system failures, on the other hand, are more likely to degrade applications' performance rather than crash them.

5.6.2 Details prediction

Our study in chapter 5 has shown that only around 44% of the failures on the Blue Waters system lead to at least one application crash. The same analysis shows that 62% of the failure types predicted by ELSA refer to failures that lead to application crashes. We analyze, in this section, the error/failure sensitivity and their influence on prediction of a variety of more than 1 million of user applications launched during the analyzed data.

In average, there were 1,093 jobs per day on the Blue Waters system, with a gross load (utilization) computed over 24h of around 70%. Overall, an average job lasts for less than 10h (in 95% of the cases the average is 1.5h). We consider that an application crashed if its exit code is different than 0. Table 5.7 shows some examples of exit codes, their meaning and the number of occurrences in 6 months of production in 2014.

Depending on the month, between 3-10% of all application crashes can be correlated to different types of system failures. When looking at all applications that experienced an error during their lifetime, depending on the month, between 35-40% of applications terminated abnormally.

Table 5.8 presents the correlation statistics between each type of failure and application crashes on the Blue Waters system. For each failure type, we show the ratio between the number of failures of the given type that leads to application crashes over the total number of failures of the given type, as well as the ratio of how many application crashes are correlated to this

Failure reason	No.	app MTBF (h)
Assertion/Error Meassage (runtime bug)	235	18.4
Hangup/death controlling process	1093	2.1
Illigeal Instruction	35	110.1
job exec failed after files	99	30.6
job exec failed before files; no retry	25	185.6
Machine Problem	194	23.4
Possible Memory Leak	3	1658
Segmentation violation	118	38.6
Total	1.802	97

Table 5.7: Blue Waters application exit codes, their meaning and the number of occurrences

Failure type	% lead to app crashes	% from total app crashes
Hardware	62	38
Software	18.3	32
Network	34.1	12
Facility/Environment	45.5	18
Total	44	100

Table 5.8: Correlation between different system failure types and application crashes

type of failure to the total number of application crashes. It is clear that hardware, network and facility failures have the most influence on applications while software seems not to lead to application interruptions. Note that system wide outages were not included into this table. Out of the software failures, operating system problems represent the large majority of application crashes. When looking in detail for file system failures (which have the worst prediction result), we notice that less than 10% of file system failures actually crash applications.

The method used for finding correlations between failures and application crashes is based on creating a window of time following each failure and checking if during this time there are any application crashes with exit code different than 0 and that does not represent a well known code for application bugs. The window size was chosen empirically to be one hour.

File system failures, in general, degrade the performance of applications, rather than crash them. For example, during an OST failure, when applications attempt to read or write to/from a failed Lustre target, they are

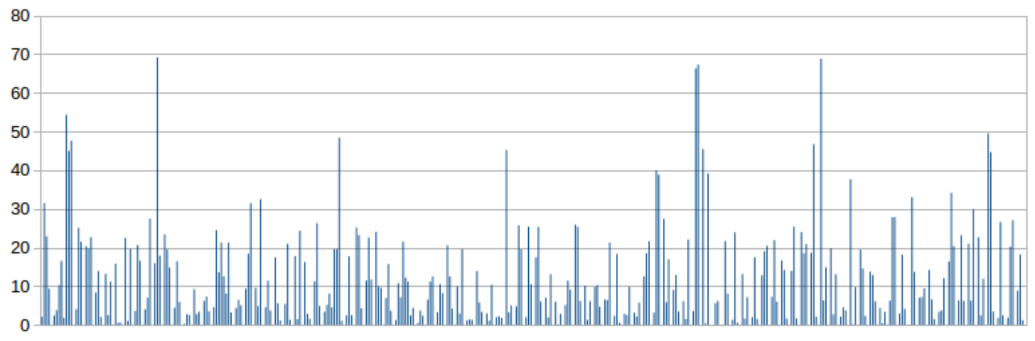


Figure 5.19: I/O throughput decrease (percentage per application instance, over 400 applications) on Intrepid.

blocked waiting for the OST recovery. An application does not detect anything unusual, except that the I/O may take longer to complete. An analysis of Intrepid, the Blue Gene/P system at Argonne, shows that congestion can cause up to a 70% decrease in the I/O efficiency seen by an application (Figure 5.19).

Rarely, when an OST is marked as inactive, the file operations that involve the failed OST will return an IO error and the application might be terminated. However, this process can take a long time, so our 60 minute window might not be long enough to capture this behavior. Moreover, in most cases, this degradation seen at the application level can cause the application to exceed its maximum allocated time, which will make this scenario seem as an unintentional termination and not a crash.

We use the Darshan tool [95], an application level I/O characterization tool developed at Argonne, to capture the behavior of applications running on Intrepid in order to extract their instances of degraded performance. There are numerous reasons for application performance degradation, starting with resource exhaustion to network and file system problems. The Darshan tool has just started working on the Blue Waters system. We plan to fully analyze the causes of performance degradation on this system. We also plan to investigate the correlations between performance degradation as well as crashes at the application level and file system failures.

Table 5.9 presents the recall for application crashes due to different failure types. Since file system, scheduler and some network failures do not crash applications, overall the recall value for these types of failures is considerable better than for predicting system failures. Another interesting obser-

Failure type	Recall August	Recall September	Recall October
Hardware	52	67	58
Software	54	63	61
Network	50	60	55
Facility/Environment	43	53	49
Total	52	64	59

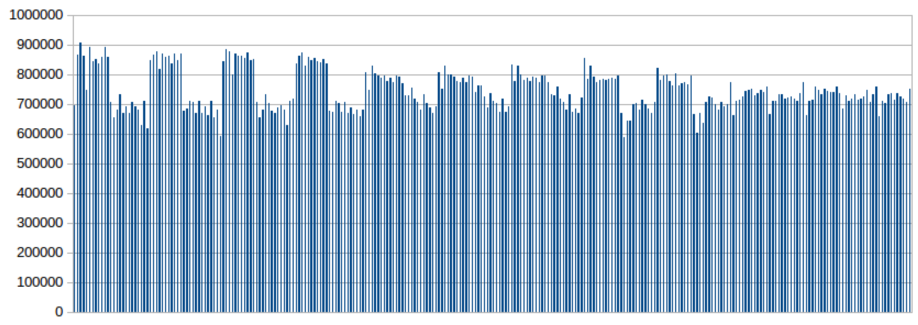
Table 5.9: Recall for predicting application crashes (Precision between 70-75%) for the Blue Waters system

vation from the table is that the prediction result is highly dependable on the workload for the analyzed framework, recall values ranging from 52-64% depending on the analyzed time frame.

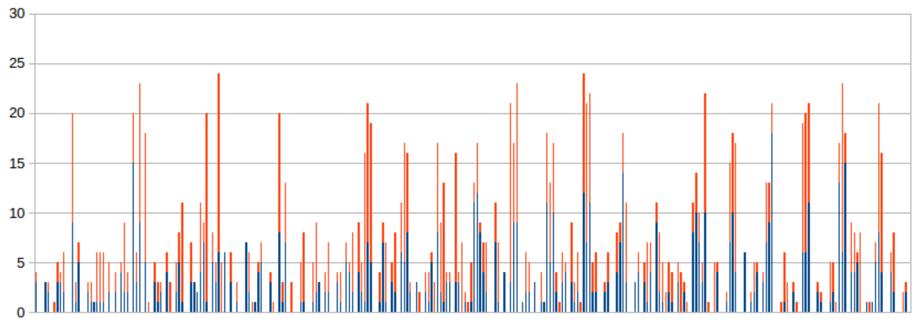
5.7 Discussion

In general, the results show that the recall and precision depend greatly on the failure type. Figure 5.20b shows the number of failures that occurred in around 3 months of production, on each cabinet of the Blue Waters system. Cabinets are order by the way they are arranged in the torus network. Consecutive points in the figure represent adjacent cabinets on the OY axes and points 12 spaces apart are adjacent cabinets in the OX axes. Blue parts represent failures that were correctly predicted using the location prediction method and orange parts are failures that were not seen by our method. Since this figure shows only the compute node cabinets, every failure that does not occur on these nodes will not be visible. This means that scheduler and filesystem failures have been filtered. We observed that by removing these failures, the overall recall value increased to 40.08% with a precision of approximately 70%. When looking only at software failures, the new predictor can correctly forecast over half of the software failures after file system and scheduler failures have been filtered out. One visible example is the LNET failure that refers to moments of low memory conditions when the out-of-memory (OOM) killer kicks in and picks a process to kill using a set of heuristics. This result shows that prediction is possible on the Blue Waters system and can offer good results for some types of failures.

Another observation after examining figure 5.20b is that some cabinets are more error prone than others, but this does not seem to influence the recall



(a) Usage (node hours) per cabinet



(b) Number of failures per cabinet

Figure 5.20: Summary of per cabinet behavior

value (some cabinets with a higher number of failures have low recall values while others behave the opposite). Recall is also not influenced by usage. We also examined the types of failures occurring on each cabinet. The analysis showed the higher failure rate for some cabinets cannot be attributed to a particular type of failure. However, the recall value per cabinets is strongly correlated to the percentage of each type of failures on the corresponding location. In fact, our observations show that the characteristics of each type of failures is the main factor that explains recall values.

File system failures are one of the main reasons for the low recall obtained for software errors. For example, the Lustre Metadata failures have very few precursors and since most of them occur at the same time with the actual failure, our method disregards the majority. Metadata servers for the Lustre filesystem store namespace metadata, such as filenames, directories, access permissions, and file layout. When applications detect an MDT failure, they connect to the the backup MDT and continue their execution. Just in less than 17% of the cases, applications having trouble connecting to the back-up MDT fail. During an OST failure, when applications attempt to do I/O to a failed Lustre target, these are blocked waiting for OST recovery. An application does not detect anything unusual, except that the I/O may take longer to complete. Rarely, when an OST is marked as inactive, the file operations that involve the failed OST will return an IO error and the application might be terminated. In general prediction from the application's point of view is more complex and differs in results compared to the one for system failures. Similar to system failures, application failures caused by Lustre failovers have low recall values, but for slightly different reasons. Lustre MDTs and OSTs are stored on the Sonexion storage system and so they use different location ids than the compute nodes. The Sonexion nodes are using a fat tree network and comunicate with the compute nodes through Infiniband. Since applications are running on compute nodes, the prediction does not have enough information to predict the exact applications that might suffer from a Lustre failover. Even in the case of Lustre failures that cause system wide outages, we observed that less than half of the applications that were running in the system at the time of the failure are terminated. Information about what files are used by an application is necessary in order to predict application crashes. Better understanding of how and when filesystem and scheduler failures occur and in what cases they cause application crashes would greatly benefit the prediction process.

Overall, the conclusion of our detailed analysis is that prediction is possible; the preliminary results are very promising. As a general observation, it would be good to have more/better precursors in order to increase our results on the Blue Waters, either from the system level or from the application level. For example, we observed that whenever a certain mix of application runs concurrently in the system, in over 50% of the cases the system experiences a MTD failover. Information about what applications are doing, in general, might offer new failure precursors both at the system and application level. Monitoring the I/O patterns of an application could help with location prediction for application crashes caused by Lustre errors. Moreover, we observer that filesystem performance degradation can often cause performance degradation at the application level that can later lead to filesystem failures. We plan to investigate all these directions in the future.

Chapter 6

Combining failure prediction with checkpointing

Future large scale HPC systems are expected to have a higher error rate than current system. Most of the existing projections [19, 96] for exascale systems consider that soft errors and in particular uncorrectable soft errors in memory cells, latches and processor logic will significantly increase, leading to a higher frequency of application interruptions. Several fault tolerance techniques have been proposed and studied in the HPC community. The most popular and widely used one is Checkpoint/Restart. Unfortunately, classic periodic checkpointing, as used today, will be prohibitively expensive at exascale because of the large amount of time that is required to dump the checkpoint data into the remote Parallel File System (PFS) [97]. To decrease this overhead, multi-level checkpointing [65, 98, 99, 100] has been proposed with a large arsenal of techniques combined with new hardware devices and it has successfully decreased the *checkpoint-storing* time drastically. However, multi-level checkpointing remains a preventive technique in which the application is restarted from the last saved snapshot, after each application failure.

The prediction performance presented in the previous chapter is good enough to envision using failure prediction to reduce application execution failures. For this purpose, failure prediction is useful only when coupled with a proactive failure management that tries to apply countermeasures. The decision to actually trigger a countermeasure may follow a complex process involving (i) cost of the action, (ii) the confidence in the prediction and (iii) the effectiveness and complexity of the actions [6]. The advances in failure prediction precision and recall open the possibility to reduce drastically the rework time by actually checkpoint right before the failure; a technique know as proactive checkpointing.

However, proactive checkpointing alone, cannot systematically avoid re-executing the application from scratch if failures are not perfectly predicted.

Since executions on large scale HPC systems are very expensive (in time and energy), taking the risk of long (potentially near to full) re-executions is unacceptable. Therefore, failure prediction and proactive checkpointing should be combined with periodic checkpointing. Nevertheless, little is known about the benefits of failure prediction and proactive checkpointing when combined with periodic checkpointing. The objective of this chapter is to provide a better understanding of this combination on execution performance, using state of the art failure prediction and checkpointing techniques.

6.1 Analysis of prediction methods for failure avoidance

6.1.1 Real Time Failure Prediction Challenges

Failure prediction methods have been exhaustedly analyzed in the previous chapter. However, in order to create a failure avoidance technique, systems need to couple failure prediction with some fault proactive tolerance technique; this involves three important challenges:

- The failure prediction framework needs to perform online detection of propagation chains on the same compute nodes where the application is running without imposing a too expensive overhead on the application execution.
- In addition, it is critical to study the behavior of systems in the presence of frequent false positives and verify whether the overhead involved by the proactive schemes is acceptable or not.
- Moreover, the lead time observed for the most efficient prediction approaches is short (e.g. seconds), hence proactive techniques need to be capable of reacting extremely fast to successfully finish before the failure strikes.

To successfully couple failure prediction, proactive fault tolerance techniques and preventive checkpointing, these three aspects need to be studied and any implementation should demonstrate low overhead (even in the presence of false positives) and high reactivity.

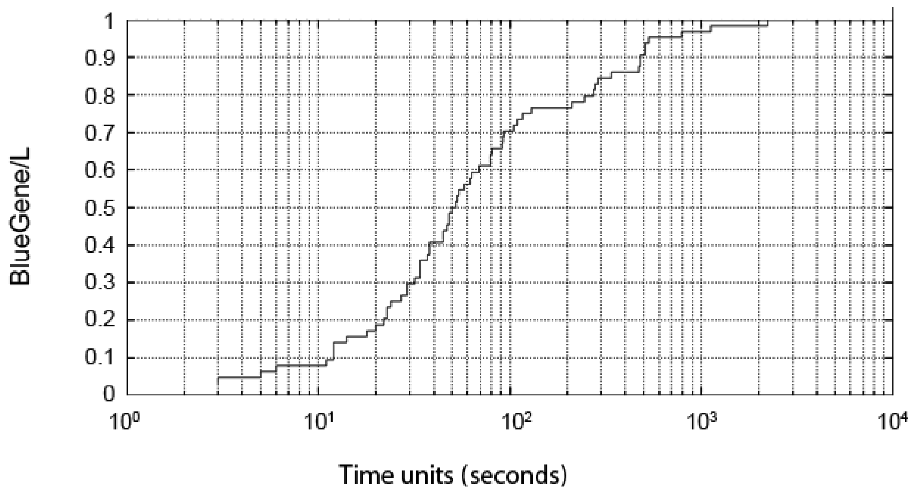


Figure 6.1: Lead time distribution between events in sequences for the BGL predictions

6.1.2 Lead time distribution

We plotted the time delay distribution between the first message indicating the beginning of a sequence and the last visible symptom. Figure 6.1 presents the results for BlueGene/L.

In general, we observed that node card failures offer sequences with longer time delays. This is reflected in a larger prediction window for these kind of errors, and as a consequence more time for fault avoidance strategies. For example, the node card failure presented in table 5.1 offers around 9 minutes (the equivalent of 54 time units) between the first and the last event in the sequence. Other node cards examples show even one hour after the first symptoms occurs. The memory errors detected with our system, like the one presented in table 5.1, usually offer in average a one minute prediction window.

The Blue Gene/L system has a separate process, CIODB, that runs on service nodes and handles the job loading and starting. This process starts and monitors jobs, and updates the job table as the job goes through the states of being loaded, started and terminated. We observed that sequences or events related to CIODB usually have a very short time delay between them, the majority happening almost at the same time.

For the purpose of combining prediction with checkpointing strategies, we are including the lead time when triggering a prediction. Depending on the preventive action that the fault tolerance strategy might take, we only keep the predictions that offer a large enough lead time. In case the prediction lead time represents a window, with different probabilities on each moment of time, we apply the same strategy. However, in this case we check the entire window before triggering the preventive action or ignoring the prediction.

We analyzed Blue Waters as well and noticed that the lead time for the extracted correlations are larger than for the Blue Gene/L system. For computing the benefit of a hybrid checkpointing/prediction strategy we will use the worst case scenario, which is given by the lead time offered for the Blue Gene machine.

6.1.3 False negative distribution

In this section, we are investigating the time-varying behavior of failures in large-scale distributed systems in the presence of a prediction module. Specifically we are interested to see how the failure distribution changes when filtering out all failures that are predicted. False positives, namely the predictions that are not actually failures, pose additional overheads on fault tolerance techniques. Even worse, failures that are not visible to the prediction module (false negatives), are the failures that crash applications and require the restart step in the classical checkpointing strategy. Thus, the mean time between two consecutive false negatives, as well as their distribution influence the choice of the optimal checkpoint interval.

We analyzed all the failures that affected the Blue Gene/L as well as LANL systems. We use the annotated failure information for the LANL systems, provided in the additional failure file, and a filtered set of failures for the Blue Gene machine.

For the LANL systems, the system managers divided the failure types into the 6 categories studied in the previous chapters. For the Blue Gene/L, we use the type of failure given in the header of each message log. Table 6.2 and 6.1 show the percentage of each type of failure for each system. Human errors has no representation for the Blue Gene/L system because traces do not give context information about the failures and so the actual root cause is unknown.

In general, there is a large difference of coverage between different types

LANL (MTBF 125h)		
Category	Percentage	Recall/Precision
Facilities	2%	38% / 89.2%
Hardware	62%	45.1% / 93.8%
Human Error	<1%	9.2% / 80.8%
Network	2%	42.8% / 91.2%
Software	23%	41.1% / 93.7%

Table 6.1: Precision results for LANL systems

Blue Gene/L (MTBF 24.4h)		
Category	Percentage	Recall
Node Cards	16%	61%
Midplane switch	4%	45%
Memory	22%	15%
Network	17%	62%
APP_IO	25%	41%

Table 6.2: Precision results for the Blue Gene/L system

of failures which indicates that certain failure types appear in patterns and correlations more than others. Depending on the resources an application might use, and so on which parts of the system are more stressed and prone to failures, the overheads and benefits of preventive checkpointing techniques might vary.

We investigate both the classical stochastic model that describes the inter-arrival time between failures as well as the influence of failure prediction on this model. The interval of time that separates two false negatives can be later used either to compute the optimal interval between checkpoints [101, 102, 103] (following sections) or to schedule jobs in order to maximize the reliability [104].

Fitting methodology

Different methods are available to fit the empirical data to probability distribution functions. The most common methodology is to first select a set of candidate distributions, after which to estimate the values of distribution parameters based on the empirical distribution and keep the best one. We conduct the fitting process using the commonly used distribution functions to model failures in HPC systems [105, 32, 24], namely exponential, Weibull, log-normal, normal and gamma.

As a second step, we look for the maximum likelihood estimates (MLE) [106] that will show what is the distribution that is more likely fit to the empirical data. Technically, MLE aims to maximize the logarithm of the likelihood function that corresponds to the closest distance between the empirical distribution and samples from distributions with certain parameters. We then use the Negative log likelihood value produced by the MLE to rank the different distributions.

This still does not means that this given distribution is a good model for the empirical data. Thus we check also the goodness of fit between the data sample and synthetic sample. Literature describes dozens of goodness-of-fit tests, but only a handful are used in practice. We use the Kolmogorov-Smirnov [107] test and the standard probability-probability PP plot as a visual method.

Fitting results

Table 6.3 reports the fitting results with the best probability distribution for each system. We note that the parameter μ is the mean in seconds for the exponential distribution. For the weibull parameter a denotes the scale and b denotes the shape parameter.

System name	Failures				False negative			
	Mean	CV	Best Fit	KS	Mean	CV	Best Fit	KS
Blue Gene/L	1040.5	0.92	exponential $\mu = 62431.3$	0.10	1888.1	1.10	exponential $\mu = 113289$	0.79
LANL Sys 3	3595.1	1.1	exponential $\mu = 215705$	0.98	6559.0	1.1	exponential $\mu = 393538$	0.70
LANL Sys 4	3409.1	1.1	exponential $\mu = 204544$	0.77	6187.0	1.1	exponential $\mu = 371218$	0.99
LANL Sys 5	3294.5	1.1	exponential $\mu = 197671$	0.95	6377.9	1.2	exponential $\mu = 382671$	0.35
LANL Sys 6	16796.7	0.9	exponential $\mu = 1007800$	0.81	31878.2	1.1	exponential $\mu = 1912690$	0.99
LANL Sys 23	9288.2	1.3	weibull a = 509380 b = 0.846905	0.97	16272.3	1.2	weibull a = 895274 b = 0.851258	0.98

Table 6.3: Best fitting distributions (fitting parameters scale are in seconds)

It is visible that there is a relationship between the initial failure distribution and the false negative distribution. In fact, as it can been seen in table 6.3, the best fitted distribution for the data concerning the false negative alerts is exactly the same distribution for the failures intervals, but having different parameters. Hence, intuitively we can say that the failure prediction process does not change the initial distribution and it affects only the scale parameters of the initial distribution.

Also, we can notice, for the exponential distribution, the ratio between the initial parameter μ_u and the false negative parameter μ_y is given by $\mu_y/\mu_u \approx 1 - r$ where r is the recall. For Weibull as well, we have approximately the same shape parameter for both distributions and the scale parameter of the false negative a_y is approximately equal to $a_u/(1 - r)$. This means that the failure prediction mechanism act as a scaling filter that affects only the scale in terms of time. Therefore we can estimate the distribution of the false negative alerts using just the recall and the initial failure distribution.

We note that the failure prediction process does not have an impact on the variability of the data. As we can see in table 6.3 the coefficient of variation (CV) is almost the same for both data. The Kolmogorov-Smirnov test values (denoted by KS) in table 6.3 indicate that all the found distributions pass successfully the test of goodness.

In order to asses the fitting results visually we report in figures 6.2a and 6.2b the probability plots. Figure 6.2a reports the pp-plot for the exponential type distributions, while Figure 6.2b looks at the Weibull distribution. As it can be seen the figures confirm that the exponential/Weibull distributions present a good visual fitting.

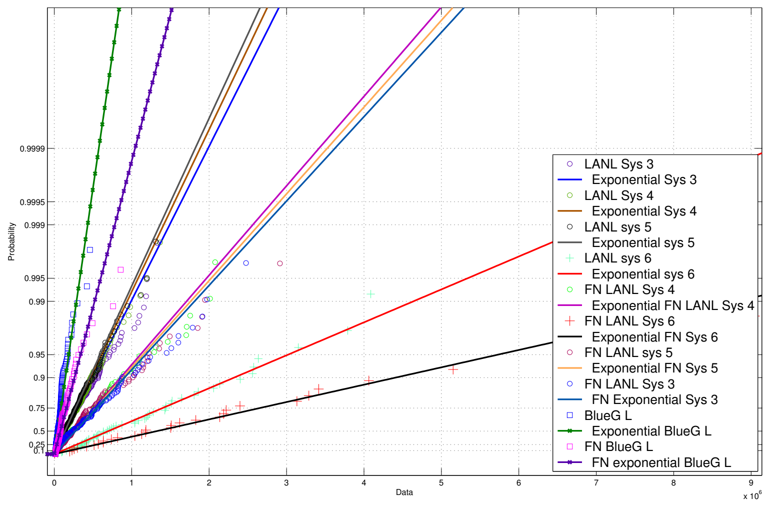
Since the false negative distribution can be computed from the original failure distribution and the recall values, we can simplify the model for combining checkpointing with prediction in the next sections.

6.2 Implementation of the hybrid approach

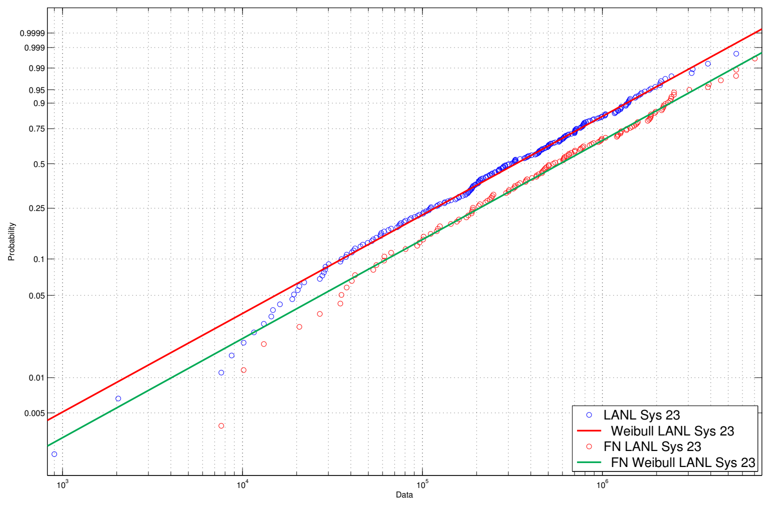
In this section, we are coupling failure prediction with a fast proactive checkpointing implementation. For this purpose, we have merged the ELSA tool presented in the previous chapter, with the Fault Tolerance Interface (FTI [65]) that provides fast multi-level checkpointing. This merging has introduced multiple technical challenges that we have addressed in order to achieve high efficiency and low overhead.

6.2.1 Adapting FTI to handle checkpoint requests

FTI is a multi-level checkpointing library that works in a distributed fashion and applies erasure codes on the checkpoint data stored locally to guaran-



(a) PP-plot for BlueG/L and LANL systems: 3, 4, 5 and 6



(b) PP-plot for LANL system 23

Figure 6.2: PP-plot and CDF for exponential type distribution (a) and Weibull type (b)

tee checkpoint availability. FTI requires to spawn an extra thread per node to encode the checkpoint files concurrently with the application execution. This fault tolerance dedicated thread is called the Head of the node. In the multi-level scheme implemented in FTI, L1 only checkpoint the data in local storage, L2 stores in local and encodes the checkpoints using Reed-Solomon (RS) and L3 stores the checkpoint data in the PFS. Between two consecutive L2 checkpoints (and therefore encoding), one or multiple L1 checkpoints might take place. Between two consecutive encodings the Head of the node is *idle* and therefore, can execute extra actions. Hence, we have embedded ELSA in the Head of the node, which introduced some other technical challenges that we have addressed. However, in this section we will see ELSA as a black box that is interrogated frequently to know if there are predicted failures.

FTI implements application-level checkpoint where the user calls the checkpoint function every x iterations and then contact the Head of the node trough the FTI API to start the encoding. In other words, the application processes act as clients and the Head as a server. This communication pattern in which the clients trigger the encoding does not match the proactive checkpointing technique in which upon a failure prediction, the Head needs to quickly order a checkpoint to all the application processes. However, interrupting the application processes at any arbitrary point may not guarantee a consistent state and may lead to save execution states of huge sizes. Therefore, it is necessary to develop a scheme in which application processes can be quickly checkpointed in a coordinated fashion, at execution points where the state is small, upon a failure prediction communicated by the Head of the node.

Our solution is to extend the FTI API to check at high frequency if the Head has predicted a failure. This function calls should be placed in a inner loop of the application that is executed at high frequency (1 second or less). FTI will measure the duration of the inner loop and will decide how many iterations (same for all processes) should pass before interrogating the Head for predictions. For instance, if the optimal interrogation frequency is every 10 seconds and each iteration lasts 0.1 seconds, the application processes will communicate with the Head every 100 iterations. It is important to notice that the communications between the Head and the application processes are intra-node communications that are usually optimized in most of MPI

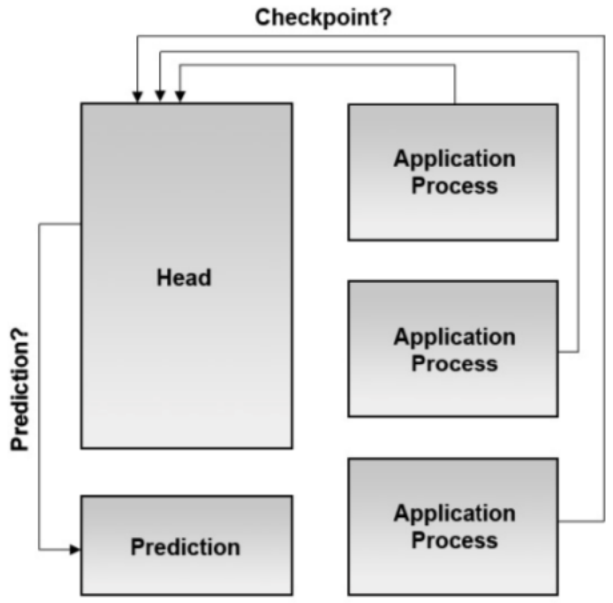


Figure 6.3: The hybrid implementation architecture

implementations. Furthermore, FTI can dynamically adapt the checking window (i.e. number of iterations between two checkings) if the duration of the inner loop iterations changes trough the execution.

6.2.2 Embedding ELSA into FTI

Figure 6.3 presents the merged architecture of ELSA with FTI. In the current implementation, ELSA is embedded into each FTI Head process, which forces the prediction to be called by FTI at a fixed interval instead of running continuously. This poses a number of challenges that will be discussed in this section. First, by distributing the prediction process on each node where FTI is running, the access to the generated log events is limited to the context of each node, forcing ELSA to analyze only per-node predictions and so to loose failures that affect multiple-nodes. Depending on the system, the new methodology might have a higher number of undetected failures [92], thus in the future we plan to focus on improving this limitation. However, Tsubame 2.0, the system we used in our experiments, does not present many correlated failures across multiple nodes [64, 65] so the results are not influenced by this limitation.

Another challenge is represented by disrupting the continuous execution of ELSA. FTI interrogates the prediction engine at a fixed interval and feeds the entire set of events generated in the corresponding time interval. This gives a more general view of the state of the node and increases the accuracy of the prediction. However, this reduces the prediction's lead time. We will show in the experiment section that by tuning the time interval at which FTI pulls predictions, we can keep the impact of the prediction window on the application wasted time due to failures and the proposed protocol negligible.

However, discontinuity in the prediction process makes it harder to keep the correlation set updates. Past suspicious events and predictions are saved and verified each time ELSA is triggered and in case a prediction is wrong it is straightforward to adapt the correlations accordingly. However, when the prediction is correct and the corresponding node experience a failure, the information about past predictions is reset with the application's restart, making it more complicated to have positive updates on the correct correlations. One possible positive way of updating the correlation set that is incorporated in ELSA is described in the following paragraphs.

Based on the predictions made by ELSA, the FTI Head requests proactive checkpoints to the application processes running on the same node. However, in practice, not all predictions are beneficial and it is ELSAs responsibility to filter the predictions forwarded to the FTI Head. There are two categories of filtering rules, the first one for prediction that cannot be used by the fault tolerance protocol and the second for predictions that cannot be used in the current moment but that have the potential of being useful in the near future.

Predictions that do not leave enough time to take a checkpoint or that happen just after a checkpoint has been taken are both represented by the first category. In the second case, we use a simple mathematical model to decide when the overhead of taking a checkpoint is greater than the overhead of loosing the work made since the last checkpoint.

Predictions with low confidence values and with high lead time are part of the second type of filtering rules. In this case the predictions are added in the suspicious list and are monitored and triggered when they give a higher benefit. In the first case, for low confidence values, the log is monitored for further symptoms that might increase the confidence. A confirmed suspicious prediction gives ELSA a way of positively updating the correlation set. For predictions with high lead time, it is more beneficial to take the checkpoint as close to the predicted moment as possible. For this reason, these predictions are added in another list and the prediction is triggered later when the waste due to an application crash is minimal.

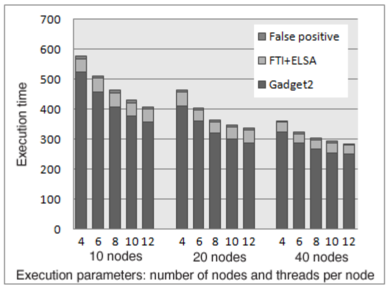
6.3 Hybrid implementation overhead

Our first set of experiments shows the overhead of our preventive checkpoint implementation on a real large-scale system and computes its overhead compared to the time to execute an application without any fault tolerance protocol for failure free scenarios. Tsubame 2.0 is a supercomputer deployed at the Tokyo Institute of Technology. Details about Tsubame 2.0's configuration can be found in [108].

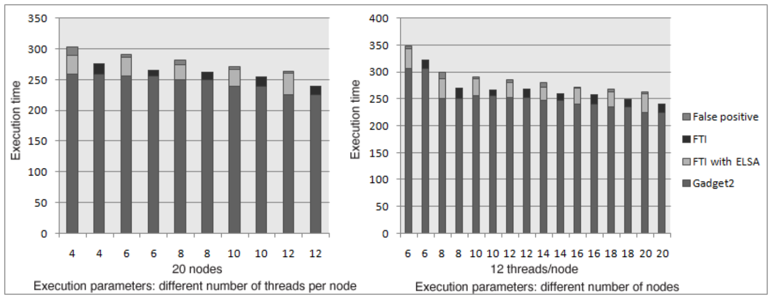
The applications we considered are part of the Gadget2 Code [109] for cosmological N-body/SPH simulations on massively parallel computers. The same code can be used for studies of isolated systems, for simulations of the cosmological expansion of space or for other fluid dynamics simulations. Gadget2 was used for the Millennium Run, one of the largest N-body simulation ran to investigate how matter in the universe evolved over time. We used two different applications, the Blob test [110] and the Kelvin-Helmholtz test [111], that are both used for testing the evolution of multiphase flows in smoothed particle hydrodynamics. The Blob test simulates a spherical cloud of gas that is placed in a wind-tunnel with periodic boundary conditions and the Kelvin-Helmholtz test records the evolution of mixing two fluids in pressure equilibrium with opposing velocities when the interface between them is perturbed. Both applications have around 100 MB checkpoint size per process, are using MPI and were modified in order to incorporate the FTI library for fault tolerance.

Overheads

We computed the overheads for FTI before the modifications and for FTI with ELSA for failure free executions considering different scenarios and different checkpointing intervals. The extracted overheads include the preventive and proactive checkpoint waste and also protocol specific overheads for example due to the communication between FTI and the application processes. Moreover, the measured overhead includes the overhead of dedicating



(a) KT12 Overhead



(b) Blob Overhead

Figure 6.4: Overhead of the proactive checkpoint implementation

1 extra thread per node for FTI. Figure 6.4 presents the results for the Blob test and the Kelvin-Helmholtz test (KH12) when tuning the number of nodes the application executes on or the number of threads in each node.

The first analysis examines the overhead when increasing the number of threads per node. In this scenario the total number of nodes does not change, so FTI will start the same number of Head processes, keeping the same inter-node communications constant. Also, the checkpoint size per node will not change from one case to the next. However, there is more intra-node communications that could increase the overhead when the number of threads per node increases.

In general, for KH12 the difference between the execution time with our current FTI and ELSA implementation and the version without checkpoint stays almost the same, so it is normal to see a slight increase in the overhead with more number of threads per node. However, for the Blob test this is not true, the results show no pattern in the overhead. One cause for this is that Blob is an irregular application in the sense that iterations have different execution times. This forces FTI to adapt and change the interval for checking the predictions. This overhead will affect more the cases with a higher execution time since FTI will need to adapt multiple times. This is the reason for the high overhead for small number of threads and why the overhead decreases until 6 threads when the irregularity does not change significantly. From this point Blob behaves as KH12. Overall the overhead for FTI in its original form is around 6%, our current implementation of FTI and ELSA has min 10% and max 15% of overhead.

For all the experiments we also tested the case when ELSA has a false positive. For the extreme case, ELSA will generate a prediction on one node

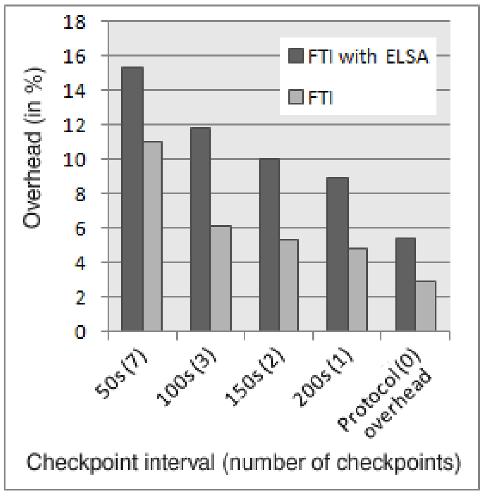


Figure 6.5: Overhead for different checkpoint intervals

each time FTI checks. We used an interval of 10 seconds and observed an additional overhead of around 2% in this case.

Figure 6.4 presents also the overhead results when the number of threads per node are kept the same and the total number of nodes are changed. Since there is one FTI Head per node, increasing the number of nodes also increases the inter node communication. However, since we made experiments for the same number of total particles in all applications, the checkpoint size per node decreases for more nodes. Figure 6.4 also shows that the false positives affect slightly less the overhead when the number of nodes is increased. The node that predicts a false positive every 10s and that saves its checkpoint on another nodes has a smaller amount of memory to save, accounting for around 1.5% overhead.

Blob shows a similar pattern as in the case of modifying the number of threads per node. For KH12 the difference between the execution time for FTI with ELSA and the execution of the application without any checkpoint slightly increases when the number of nodes increases. This is probably because even though the per-node checkpoint size decreases, overall the whole application needs to save the same amount of data or event more due to local variables in each thread. Thus, the overall overhead increases.

Figure 6.4 also shows that the false positives affect slightly less the overhead when the number of nodes is increased. The node that predicts a false positive every 10s and that saves its checkpoint on another nodes has a smaller amount of memory to save, accounting for around 1.5% overhead.

Overall, both the figures presented above show that, in general, the overhead differs depending on how the processes are divided on the processing units. The same number of total processes divided in more nodes with less number of threads per node will induce a lower overhead.

Figure 6.5 plots the overhead for the same number of total processes but for different checkpoint intervals. The lower the checkpoint interval, the higher number of checkpoints the application will take during its execution which as a consequence increases the total overhead. Interestingly, when the application is executed with FTI and ELSA, but no checkpoints are taken, the overhead is around 5.44%. This represents the lower bound overhead of our implementation and reflects its internal communications, metadata management and the fact that one thread per node has been dedicated for fault tolerance. The overhead for FTI without prediction is with 2 to 6 percent lower in all cases, the protocol overhead alone representing 2.87%. The difference of 3% represents the overhead of the multi-level checkpoint when combined with prediction, compared with the classical strategy.

In order to understand the impact of the prediction parameters on the overhead, we analyzed the degree at which the correlation and template set used by ELSA influence the results. In general, we observed that if the analysis of the correlation is shorter than the interval at which FTI is checking the predictions, there is no extra overhead due to ELSA. As an example, for the Tsubame 2.0 system, the analysis takes in average 2 seconds and FTI checks prediction every 10 seconds, thus for our experiments there is no visible impact.

Checking interval

The observations made in the previous subsection show the overheads for FTI with ELSA and allow us to predict the overhead of a large-scale application when running longer periods of time. We investigate what is the impact of the checking interval on the number of usable predictions made by ELSA (which influences the recall value and so the benefit of the protocol). We first allowed ELSA to monitor the activity of the system in real time without being interrupted by FTI and use the results as a baseline. Afterwards we executed FTI with ELSA with different checking intervals and compared the results with the base line. For Tsubame 2.0, we observed that the benefit for

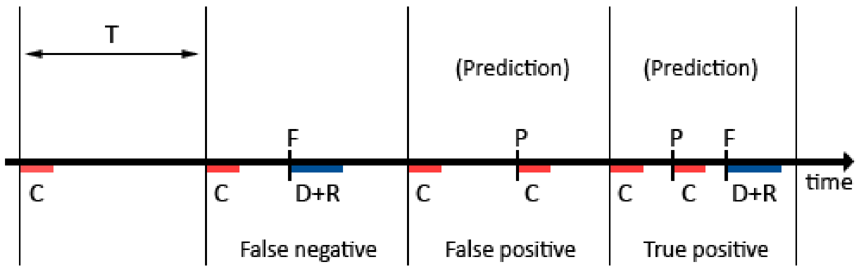


Figure 6.6: Checkpoint Restart mechanism

our chosen time interval of 10 seconds shows a similar result as the baseline, having recall differences of less than 1%. However, for larger checking intervals we observed significant differences in the recall and the benefit value, since some of the predictions will require shorter lead time and will be filtered out. It is important to tune this parameter for the configuration of each system.

6.4 Simplistic model to compute the protocol's benefit

In this section we derive an analytical model for the impact of prediction on adaptive checkpointing strategies in order to highlight the benefit brought by our method. We will modify the formula used to compute the optimal checkpointing interval [71] in order to consider ELSA's prediction.

If no failure prediction is available, then fault tolerance mechanisms must use periodic checkpointing and rollback recovery. We start from the model from [71] that computes the waste of a coordinated checkpointing strategy when no prediction is offered and then integrate the impact of precision and recall on this model. Figure 6.6 presents the variables used in creating the model. With T, we represent the checkpoint interval, W represents the percentage of wasted time and MTTF the mean time to failure for each node. We also assume a task of a job running on a node can be checkpointed locally on that node in C seconds, and the checkpoint can be loaded back into memory in R seconds. The downtime of a node and the time to restart the application on a different node or the same node in case of rejuvenation is D seconds.

We assume that we are able to predict a fraction N of the failures with a precision of P, with N,P$\epsilon[0, 1]$. We assume the failure distribution for the non-predicted failures remains exponential and that preventive actions are taken before the failure occurs for all predicted failures. This is a realistic assumption since the empirical data from the previous section has shown that the false negativ distribution follows the initial failure distribution, with different parameters.

In case of no prediction, we start with the following model:

$$W = \frac{C}{T} + \frac{T}{2mttf} + \frac{R + D}{mttf} \tag{6.1}$$

with the assumption that T$\ll mttf$.

The formula accounts for the lost of C seconds every T seconds for taking checkpoints, the lost due to faults that occur every mttf seconds and lose an average of $\frac{T}{2}$ time-steps each time and in the last term for the lost due to the recovery time that is taken for every failure.

The optimal checkpointing interval can be used to compute the minimum waste and it is given by Young's formula.

$$T_{optimum} = \sqrt{2Cmttf} \tag{6.2}$$

We now introduce the prediction model. First we assume having a recall of N and perfect precision. In this case the mttf of the unpredicted events will become $mttf_{new} = \frac{mttf}{1-N}$. For example if 25% of errors are predicted, the new mttf is $\frac{4mttf}{3}$. The rest of the failures are predicted events and have a mean time between them of $\frac{mttf}{N}$ seconds. We showed in the previous section that the exponential distribution of failures is preserved for unpredicted failures.

By applying the new mttf for the unpredicted failures to equation 6.2, the new optimal checkpoint interval becomes

$$T_{optimum} = \sqrt{2C\frac{mttf}{1 - N}} \tag{6.3}$$

The first two terms from equation 6.1 need to change to consider only the unpredicted failures since for all the others preventive actions will be taken. By adding the first two terms and incorporating the value for the checkpoint interval from equation 6.3, the minimum waste becomes:

$$W_{min}^{recall} = \sqrt{\frac{2C(1-N)}{mttf}} + \frac{(R+D)}{mttf} \tag{6.4}$$

The last term from equation 6.1 will not change since for all failures, both predicted and unpredicted, the application needs to be restarted. Additional to the waste from 6.4, each time an error is predicted, the application will take a checkpoint and it will waste the time execution between this checkpoint is taken to the occurrence of the failure. This value depends on the system the application is running on and can range between a few seconds to even one hour. However, for the systems we analyzed, in general, the time delay is very low and for our model we consider that is negligible compared to the checkpointing time. We add the waste of C seconds for each predicted failure, which happens every $\frac{mttf}{N}$ seconds. After adding this waste equation 6.4 becomes:

$$W_{min}^{recall} = \sqrt{\frac{2C(1-N)}{mttf}} + \frac{(R+D)}{mttf} + \frac{CN}{mttf} \tag{6.5}$$

In the ideal case, when N=1, the minimum waste is equal to the time to checkpoint right before every failure and the time to restart after every failure. The formula assumes a perfect precision. In case the precision is P, the waste value must also take into consideration the cases when the prediction is wrong. The predicted faults happen every $\frac{mttf}{N}$ seconds and they represent P of total predictions. This means that the rest of (1-P) false positives predictions will happen every $\frac{P}{1-P}\frac{mttf}{N}$ seconds. Each time a false positive is predicted, a checkpointing is taken that must be added to the total waste from equation 6.5:

$$W_{min}^{recall} = \sqrt{\frac{2C(1-N)}{mttf}} + \frac{(R+D)}{mttf} + \frac{CN}{mttf} + \frac{CN(1-P)}{Pmttf} \tag{6.6}$$

As an example, we consider the values used by [71] to characterize current systems: R = 5, D = 1 in minutes and study two values for the time to checkpoint: C=1 minute and from [65] C=10 seconds. We computed the gain from using the prediction offered by our hybrid method with different precision and recall values and for different MTTFs. Table 6.4 presents the

C	Precision	Recall	MTTF for the whole system	Waste reduction
1min	92	20	one day	9.13%
1min	92	36	one day	17.33%
10s	92	36	one day	12.09%
10s	92	45	one day	15.63%
1min	92	50	5h	21.74%
10s	92	65	5h	24.78%

Table 6.4: Percentage waste improvement in checkpointing strategies

results. The first 4 cases present numbers from real systems and checkpointing strategies. Interestingly, for future systems with a MFFT of 5h if the prediction can provide a recall over 50% then the waste time decreases by more than 20%.

We use the same model for the Blue Waters system, with the prediction results presented in the previous chapter, and considering FTI as the chekpointing strategy used. This scenario assumes a MTBF of 8 hours, 10 seconds checkpoiting time, around 70% precision and 40% recall. The gain of such a scenario gives 14.7% gain compared to having the checkpointing strategy alone. Even in the worst case month, where we obtained only 30% recall, with 70% precision, our implementation still gives us a benefit of 10% compared to checkpointing alone. The overhead of our method when the optimal checkpoint interval is used for the Blue Waters scenario is between 3-4% over the overhead given by FTI running alone. According to our model, on average, we optimize the fault tolerance protocol with 8-10% even on current petascale systems and when using a state of the art checkpointing strategy. The predictions for future systems shows an even larger improvement.

Moreover, when looking at prediction from the application's perspective, we have shown in chapter 5 that we can predict up to 60% of the failures. When considering applications crashes the results are showing around 25% waste reduction on petascale systems.

Since the model used is simplistic, it does not consider the optimizations done in our implementation. For example, not taking checkpoints when the last one was done close in the past to where the prediction was triggered, or restarting the chekpointing interval after each triggered checkpoint. In order to better study the impact of this hybrid approach on future exascale systems, we look, in the next section, to more complex models.

6.5 Evaluation of the hybrid strategy benefit

We will also present, in this section, a set of simulations to investigate the impact of precision and recall values on the benefits that we can obtain by using the proposed combination of failure prediction, proactive checkpointing and preventive checkpointing in actual and future HPC systems. To provide the simulation results, we developed a discrete event simulator. We consider the following inputs to feed the simulator with failures and prediction alerts:

- Randomly generated failure times using an exponential distribution. Numbers are generated using the GSL[112] random number generator library.
- As failures are randomly generated, true positive alerts are raised by generating uniformly a number between [0,1] each time a failure is happening. Then, if the generated number is lower than r (i.e. recall), this failure is considered as correctly predicted since this event is supposed to be a true positive alert. Otherwise this failure hits the application. Similarly false positive alerts are generated during the execution such that the ratio between the number of false positive alerts ($f_{positive}$) and true positive alerts ($t_{positive}$) verifies the following inequality $f_{positive}/t_{positive} \leq \overline{p}/p$.
- To feed the simulator with lead time intervals, we use actual data collected from BlueGene/L system located at Los Alamos National Center. In this failure log, we have 235 failure occurrences and it covers a period of 6 months, from June 3rd to December 8th 2005. Using ELSA we extract 113 failure alerts with the corresponding lead time. Then to estimate the parameter s in our model that depends on the distribution of the lead time the lead time distribution is used. This distribution is obtained by fitting the current date to probability distribution functions. In our case, the best fit distribution is the Burr Distribution with the following parameters $k = 0.5189$, $a = 2.3798$ as a continuous shape and $b = 10.873$ as a continuous scale. We point out that the obtained P-values are 0.1968 using the Kolmogorov-Smirnov test and 0.12602 using Chi-Squared test. One outcome of this analysis is that the lead time distribution should be modeled carefully and the exponential distribution can not be chosen arbitrarily to represent it.

Before investigating any model, we set the parameters that characterize current systems as well as future exascale systems. The used parameters are presented in table 6.5. For petascale systems, we take as references the Jaguar system installed at ORNL and other systems of the 10 Petaflops class based on commodity microprocessors, including the Blue Waters system. These systems have typically a MTBF between 24h to 6h. Checkpointing the full memory to the PFS takes about half an hour. There are some variations: some systems may take less time (15 minutes), while others, may need several hours for this operation. We consider intermediate system the machines that have a sustained performance of 100 Petaflops. We consider a reduction of the MTBF to 6h or 4h for the full system. This is consistent with a significant increase of the number of nodes compared to 10 Petaflops systems and an increase of the error detection and correction mechanisms at hardware level. We assume that in such systems, the checkpoint size per node is between 100GBs and 200GBs and the writing speed is about 350MB/s. The preventive checkpoint cost in such systems is in the range of several minutes. For optimistic exascale system, we take a value discussed during a DoE ICIS workshop on resilience in summer 2012 [113], namely 2h. In [96], the expected MTBF is more in the range of 30 minutes. We take this last value for the pessimistic exascale system. exascale systems are expected to have between 32 and 64 petabytes of memory, divided by 100k nodes makes several hundreds of GBs per node, thus we can assume a checkpoint size per node between 200GBs and 500GBs. Also by 2018, we should have technologies, like non volatile RAM (NVRAM), Phase Change Memories (PCM) and 3-D circuit staking that are supposed to improve drastically the writing bandwidth to GB/s (for the optimistic scenario we consider a writing bandwidth of 3GB/s and 1GB/s for the pessimistic scenario). Hence, the checkpointing times could remain in the order of several minutes. The proactive checkpoint time is estimated from the amount of memory per node to move over a 10-40GB network interface. We estimate the proactive checkpoint time in these conditions between 1 to 5 seconds. For recovery from checkpointing, we consider a down time of 1 minute after a failure happens. This time corresponds to the killing of all remaining processes after the detection of a failure, the allocation of enough resources to restart and the re-scheduling of the application processes on the allocated nodes (we assume that the job is not re-queued but immediately relaunched after the failure). We also consider that the time to retrieve a checkpoint image from storage is the same as the checkpoint time. This implies that the restart cost is equal to the checkpoint cost.

Paramters	petascale Jaguar, 10PF	Intermediate 100PF	exascale Optimistic	exascale Pessimistic
MTTF	24h to 6h	6h to 4h	2h to 1h	30 min
Preventive Checkpoint time	30 min	10 min	2.5 min	10 min
Proactive Checkpoint time	10 to 5 sec	5 to 1 sec	5 to 1 sec	5 to 1 sec

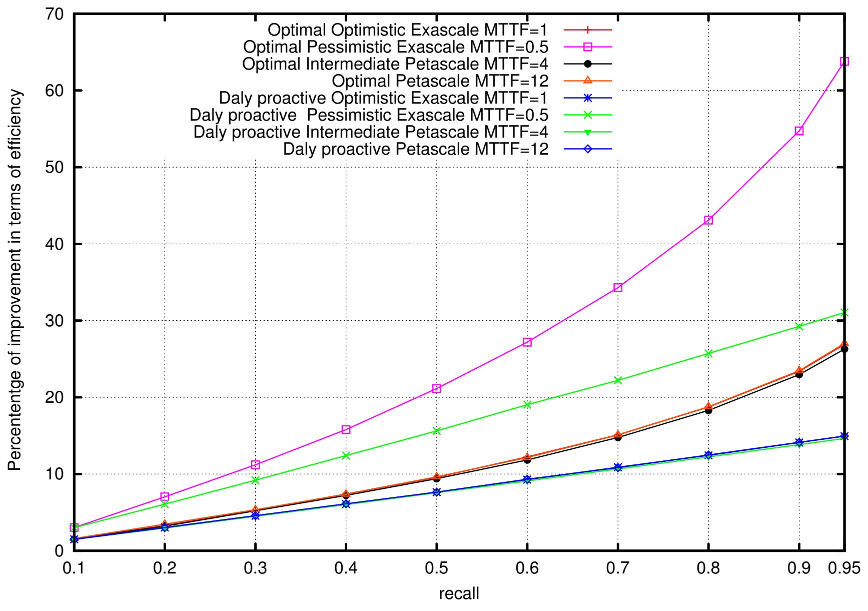
Table 6.5: Computing platform configuration

We constructed a mathematical model for our hybrid implementation. Details about this model can be found in [72]. In our simulations, we compare the performance of the failure prediction associated to an optimal proactive checkpointing and preventive checkpointing protocol with two strategies. The first is the classic periodic checkpointing strategy based on Young's interval [101] without proactive actions. The second is the simplistic model, presented in the previous section, that uses the classic periodic checkpointing strategy for the unpredicted failures and the online method to perform proactive checkpoints for each prediction triggered. The second method is called "Optimal" in the following figures.

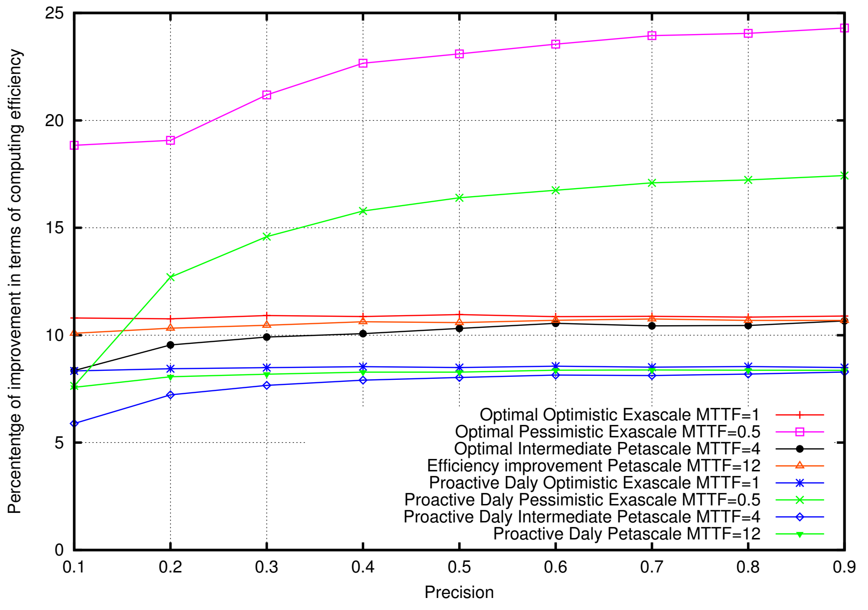
In the first set of simulations, we investigate the impact of the recall r on the application performance. We plot in figure 6.7a the improvement in terms of computing efficiency that we can obtain using the optimal or the sub-optimal comparing to the classic periodic checkpointing alone.

These results show clearly that the proposed strategy outperforms the classical periodic strategy and the sub-optimal as well. Moreover, we can see that the recall parameter has an important impact on the improvement. It is important to notice that figure 6.7a demonstrates that with a prediction recall value corresponding to the best known prediction approach (i.e. 50%), the proposed fault tolerance strategy improves the computing efficiency. This improvement ranges between 10% and 20%. Moreover, this figures shows that we can improve the computing efficiency of the application up to 30% with a prediction recall of 90%.

The results of these simulation, when varying the precision value, are depicted in figure 6.7b for a prediction recall of 50%. This figure shows that the precision has minor impact on the percentage of improvement that we



(a) Recall variation when precision is set to 70%



(b) Precision variation when recall is set to 40%

Figure 6.7: Variation of the improvement percentage versus the variation of the prediction recall (a) and the prediction precision (b).

can obtain using the an optimal strategy. This is explained by the fact that the cost of the of the proactive checkpointing is bounded. However, in the pessimistic exascale case, a precision value less than 40% can have an impact of more than 10% for the hybrid strategy. These two figures, each focusing on one of the recall and the precision values, suggest that our future research on failure prediction will focus on improving the recall.

The figures show that the benefit of our predictor in its present form gives a decrease in the waste of the checkpointing strategy of around 10% for the Blue Waters system and over 20% for Blue Gene/L. Considering the extra 3-4% overhead induced by the initial hybrid approach, we expect to get over 15% benefit for the Blue Gene/L system, around 7% for Blue Waters, and 20% for future exascale systems compared to classical checkpointing.

Since checkpointing is applied at the application level, not all failures are important for this study. Only those that lead to application crashes should be analyzed since the rest are tolerated withing the application. The recall for application crash prediction is around 52-64% while keeping a precision of around 70-75%. The benefit for our hybrid approach in this case is over 15% for the Blue Water system and up to 30% for future exascale systems.

Chapter 7

Future work

7.1 Application level

7.2 Specific predictors: File systems

In this section, we are investigating the benefit of adding performance and environmental metrics into the analysis in order to better understand and predict file system failures. This is an on-going work, in collaboration with the system administrators at NCSA. Thus, in this section, we will present the preliminary results and the future work in this direction.

7.2.1 Metrics

Blue Waters system administrators gather 256 metrics across all levels of the system. For our problem we only focused on file system and network metrics, since these components give the most frequent problems for Lustre (as shown in chapter 5). We analyze OS-level and network-level performance metrics, without requiring any modifications to the file system, the applications or the OS. In Blue Waters these performance metrics are made available in a database in time units of 20 to 60 seconds depending on the metric.

We have collected data at the OSS, OST, MDS and network levels (as we are primarily concerned with performance problems due to storage and network resources, although other kinds of metrics are available) as shown in table 7.1.

We only had access to performance metrics for 3 weeks in September 2014. Blue Waters experienced a total of 14 failures attributed to Lustre in the first 2 weeks that we used for training and 6 more in the last week. We

MDS_cpu	CPU utilization (in percentage) for the MDS
MDS_mem	Memory utilization (in bytes) fot the MDS
OSS_cpu	CPU utilization (in percentage) for the OSS
OSS_mem	Memory utilization (in bytes) for the OSS
QOS_write	Quality of Service - The amount of time (in ms) it takes for one write to each OST
QOS_read	Quality of Service - The amount of time (in ms) it takes for one read to each OST
OST_write	The amount of data (in bytes) written to each OST
OST_read	The amount of data (in bytes) read to each OST
OST_cpu	CPU utilization (in percentage) for the OST
OST_mem	Memory utilization (in bytes) for the OST
Net_prec/ptrans	Packets received/transmitted per second
Net_brec/btrans	Bytes received/transmitted per second

Table 7.1: Metrics used in the analysis

analyzed LBUG problems (a panic-style assertion in the storage node kernel), the unavailability of the Object Storage Target (OST) or of the metadata, configuration problems. About 14% of Lustre failures in 2014 were due to hardware problems, but these problems did not occur during the analyzed time frame.

7.2.2 Anomaly detection

In general, we are interested in correlating anomalies moments on different metrics with failures since we observed that failures change the behavior of metrics. We are making the assumption that metrics present a fault-free behavior most of the time. For OST, we also compare the behavior of multiple OSTs at the same moments in time. Since clocks are not synchronized, we shift the timestamps up to one minute before and after a failure in order to find the most relevant correlations.

There is one MDS, 180 OSS and 1440 OSTs in the Blue Waters on-line storage. There are moments in time when only one or a few OST have degraded performance and there are moments when multiple OST perform degraded simultaneously. For example, figure 7.1 shows several QOS per OST for different interesting time units. It is visible that there are moments of time when up to 160 OSTs degrade at the same time, but most of the time all OSTs have relatively small, similar values.

The maximum number of OSTs that an application can access is 160, so whenever an application takes a checkpoint that is stretched on the maximum number of OSTs it is normal to see a spike in the QOS for these 160 OSTs. Since information about what applications are doing at all moments in time is not available to us, it is clear that looking at anomalies in one metric is not enough.

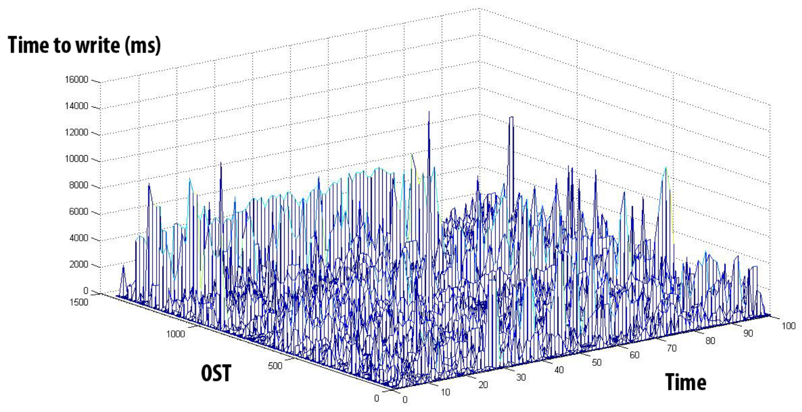


Figure 7.1: Quality of service for 100 OST for 100 time units

We gathered all the metrics relevant to each OST into one big matrix (OST_* metrics, the metrics related to the OSS that contains the analyzed OST and the MDS metrics). Because each metric has different values and anomalies have a different behavior, we first extract anomalies for each line of the matrix. We use the same methodology as described in chapter 5 by using the signal analysis anomaly detection algorithm. Figure 7.2 presents the original signal for the QOS write for a random OST, its spectrogram and the anomalies plotted against the signal.

Next, we looked at the behavior of all OSTs at each moment of time. We used the exact same method as before to detect anomalies, but this time on the columns of the matrix, keeping the metric and timestamp constant and analyzing the differences between OSTs.

Our first observation, and most trivial, is that whenever a failure occurs in the system at least one metric in one OST presents an anomaly. We couldn't find any failure that occurred when all metrics where in their normal state. However, the inverse does not hold. There are many moments when there

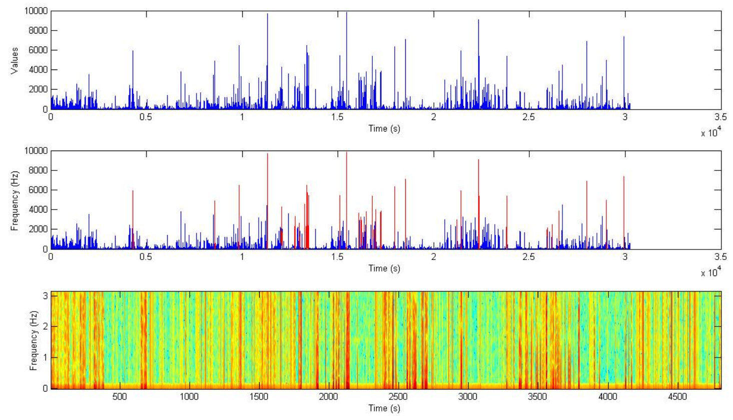


Figure 7.2: Signal and spectrogram for OST ID 101

are anomalies in the metrics of one or several OSTs and/or MDS when there is no failure in the system.

Another observations is that QOS anomalies can be caused by anomalies in the OST_write/read metric or by MDS anomalies. When an application is trying to write a large amount of data (which is seen as an anomaly in OST_write) the amount of time it takes to make one write increases. We filter out the moments when there are correlated anomalies in these two metrics. We afterward use the correlation method described previously in chapter 5. There are several patterns extracted that we are in the process of analyzing and understanding. For example, we notice a correlation between a decrease in the IO throughput in for large number of OSTs without having any anomalies on the metadata server or in the OST_Write with a Luster problem caused by additional IO traffic that creates contention for disk access. In most cases, the problem was given by an accumulation of hung I/O threads on the file server from disconnected clients, which caused the file server to eventually crash.

Even though the patterns are seen each time a particular failure occurs, the reciprocal is not true. The same pattern could occur multiple times when the failure does not occur. Thus, we are able to use these observations post-failure to decide what the problem was, but not for prediction. We believe we need to include more metrics and to combine them with log notifications in order to find patterns that can be used for prediction. We plan to focus on this topic in the future.

Chapter 8

Conclusion

8.1 Overview of the thesis

According to recent studies, current fault tolerance mechanisms will not be able to cope with the increasing rate of failure with the future exascale systems. This thesis has focused on offering ways of reducing the overhead induced by fault tolerance strategies, by combining them with failure avoidance methods. Failure avoidance deals with predicting the occurrence of a fault and triggering preventive measures. In order to offer a realistic alternative to current fault tolerance techniques, fault predictors must be able to accurately detect faults' precursor effects in the system. This thesis aims to propose an accurate and novel failure prediction method that can be combined with several failure tolerance protocols.

The research done in my first year resulted in a clustering engine that identifies frequently occurring messages with similar syntactic patterns from log files. These message templates are essentially regular expressions that describe a set of syntactically-related messages that refer to the same system event. By taking advantage of the characteristics of log files, our algorithms are computational efficient, accurate and are able to keep up with rapidly changing environments.

This thesis shows that different system components exhibit different types of syndromes, both during normal operation and as they approach failure. The key observation is that errors are often predicted by changes in the frequency or regularity of various events. For this purpose, the thesis investigates the linkage between signal processing concepts and data mining techniques in the context of failure analysis for large-scale systems. By shaping the normal and faulty behaviour of each event, and of the whole system, we were able to propose appropriate models and methods for descriptive and forecasting purposes. Multiple experiments on different production HPC systems, from Argonne's Blue Gene systems, to NCSA's Mercury and Blue Waters and Tokyo Institute of Technology's Tsubame2, have been made. The results show that conventional signal processing techniques can create clear markers for changes in events behavior. Moreover, machine learning techniques become much more efficient when applied to the derived markers, rather than to the original signal. Consequently, the thesis is proposing the first hybrid fault tolerance implementation that combines proactive with preventive checkpointing methods based on the signal analysis predictor. Our method improves the performance of classical fault tolerance techniques when dealing with failures in petascale systems and the results show the potential of using such an approach on future exascale systems.

8.2 Summary of contribution

We present, in this section, the list of contributions of this thesis. Each was covered in one of the chapters and future work has been identified:

1. Characterizing the behavior of events generated by HPC systems

    - We characterize the normal behaviour of HPC systems and the effects of failures. For this purpose, we demonstrated the value of combining signal processing concepts and data mining techniques in the context of failure analysis for large-scale systems.
    - We made experiments on different production HPC systems, including Argonne's Blue Gene systems, NCSA's Mercury and Blue Waters systems. Our results show conventional signal processing techniques can create clear markers for changes in events behavior.
    - We developed a fingerprinting algorithm that characterizes set of events that frequently occur together. This set of events can be used to give a summary report of events that happen during an application run or to improve the prediction algorithm by monitoring outliers in events' fingerprints.

2. Failure prediction methods for HPC systems

- We developed a hybrid methodology by combining data mining and signal analysis for online failure prediction based on a pattern extraction algorithm specifically designed for streams of data with multiple dimensions. We showed machine learning techniques, in general, become much more efficient when applied to derived markers given by outliers, rather than to the original signal.
- We did experiments on small and large production system from the Blue Gene/L and systems at LANL to NCSA's Blue Waters. The experiments focus on the difference between the two types of systems and detailed observations are provided.
- In order to improve the results on the Blue Waters system, we developed a location propagation algorithm specifically designed for the 3D torus architecture used by the Blue Waters system.

3. Proactive checkpointing coupled with periodic multi-level checkpointing

   - We developed a prototype implementation of proactive checkpointing coupled with periodic multi-level checkpointing by combining our failure predictor with FTI [65].
   - We experimented using Tsubame 2.0 logs and we show that failure prediction, proactive checkpointing and periodic multi-level checkpointing can be coupled successfully, imposing only 2% to 6% of overhead in comparison with a preventive checkpoint execution only, giving a total of 10-12% total overhead.
   - In order to compute the benefit of our method, we use different mathematical models of failure prediction combined with proactive and preventive checkpointing. These models capture the checkpoint cost, the failure distribution, the precision and recall of the failure prediction and the probability of success of the proactive action. We add to these models the overhead of the prediction method and study the theoretical benefit of the hybrid fault tolerance method depending on the prediction results.

8.3 Future work

There are several topic directions that can be investigated in the future related to fault tolerance and performance degradation for HPC systems. The question that first needs an answer is ”How frequent do we need to snapshot the state of different components in order to extract meaningful precursor to different failures?” The preliminary results after analyzing file system failures and performance degradation show there are clear markers in several metrics (for example network and disk throughput) that indicate problems at the file system level. However, since metrics are usually gathered in an aggregated form, every minute (or several minutes), these markers cannot be used online, but rather in a post-mortem manner. We noticed that increasing the frequency of taking snapshots for different metrics could identify anomaly moments that lead to failures with enough time in advance for applications to take checkpoints (or other action). However, the space needed to store all metrics at a frequent snapshot rate becomes unrealistic for exascale systems (even for current petascale systems). Dynamically choosing the rate to gather each metric could solve this problem. This is a short term research that has the potential to shape the way we store and deal with performance and system metrics for future systems.

As a long term research direction, one topic focuses on the analysis of the effects of different type of failures on applications and the design of novel fault tolerance protocols that consider the type of failures that might affect the application. The effects of failures on applications can also be included into the predictor which later can be combined with different fault tolerance protocols. For example, for failures that affect the memory used by the processes of one node, saving the state of that particular node in another nodes memory might be enough; for failures that degrade the performance of one node, migration of the entire node might be a better solution; and so on. Some of these solutions could be implemented at the programming language level (for example, in MPI). Moving to more specific observations, file systems can experience performance problems that can be hard to diagnose and isolate. Often, the most interesting and trickiest problems to diagnose are not the outright failures, but rather those that result in a degraded but not failed system, where the system continues to operate, but with degraded performance. Understanding file system performance degradation and how it propagates to the application level and modeling this propagation could give useful guidelines to be included in future monitoring systems and have the potential to bring advances in handling and diagnosing storage systems.

Another topic of interest comes from the positive preliminary results obtained for the specific precursor for Lustre. We believe we can improve the results for each failure type as long as we create a specific predictor for each. For example, [114] proposes a new OS-based approach that proactively avoids memory errors using prediction by focusing the analysis just on memory failures.

References

[1] T. Thanakornworakij, R. Nassar, C. B. Leangsuksun, and M. Paun, "Reliability model of a system of k nodes with simultaneous failures for high-performance computing applications," *International Journal of High Performance Computing Applications*, vol. 27, no. 4, pp. 474–482, November 2013.

[2] "Inter-Agency Workshop on HPC Resilience at Extreme Scale," http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf, 2012, [Accessed on July 2013].

[3] "U.S. Department of Energy Fault Management Workshop," http://shadow.dyndns.info/publications/geist12department.pdf, 2012, [Accessed on July 2013].

[4] M. Snir, W. Gropp, and P. Kogge, "Exascale Research: Preparing for the Post-Moore Era," *Computer Science Whitepapers*, 2012.

[5] W. Jones, J. Daly, and N. DeBardeleben, "Application monitoring and checkpointing in HPC: looking towards exascale systems," *Proceedings of the 50th Annual Southeast Regional Conference*, pp. 262–267, 2012.

[6] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *Computing Surveys*, vol. 42, pp. 1–42, 2010.

[7] N. Bolander, H. Qiu, N. Eklund, E. Hindle, and T. Rosenfeld, "Physics-based Remaining Useful Life Predictions for Aircraft Engine Bearing Prognosis," *Conference of the Prognostics and Health Management Society*, 2009.

[8] Z. Zheng, Y. Li, and Z. Lan, "Anomaly Localization in Large-Scale Clusters," *IEEE International Conference on Cluster Computing*, pp. 322–330, 2007.

[9] 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*, ser. KDD '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/956750.956799 pp. 426–435.

[10] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines," *WASL'08 Proceedings of the First USENIX conference on Analysis of system logs*, 2008.

[11] N. Muthumani, D. Thanamani, and A. Selvadass, "Optimizing Hidden Markov Model for Failure Prediction - Comparison of Gaines optimization and Minimum message length Estimator," *International Journal on Computer Science and Engineering*, vol. 3, 2011.

[12] Q. Guan, Z. Zhang, and S. Fu, "Ensemble of bayesian predictors for autonomic failure management in cloud computing," *20th International Conference on Computer Communications and Networks*, pp. 1–6, 2011.

[13] J. Stearley, "Defining and measuring supercomputer reliability, availability and serviceability (ras)," *Proceedings of the Linux Cluster Institute Conference*, 2005.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Journal on Dependable and Secure Computing*, vol. 1, January 2004.

[15] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, and A. A. C. et al, "Addressing failures in exascale computing," *Argonne Report ANL/MCS-TM-332*, April 2013.

[16] E. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. Plank, and P. R. et al, "System resilience at extreme scale," *Technical report for the Defence Advanced Research Project Agency*, 2008.

[17] N. DeBardeleben, J. Daly, S. Scott, and W. Harrod, "High-end computing resilience: Analysis of issues facing the hec community and path forward for research and development," *National HPC workshop on Resilience*, 2009.

[18] F. Cappello, A. Geist, B. Gropp, L. Kale, W. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, pp. 374–388, November 2009.

[19] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, and X. Chi, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[20] C.-D. Lu and D. A. Reed, "Scalable diskless checkpointing for large parallel systems," Ph.D. Dissertation, Univ. of Illinois at Urbana-Champain, Tech. Rep., 2005.

[21] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series 78:012022*, 2007.

[22] Leangsuksun, G. Ostrouchov, and S. L. Scott, "Using log information to perform. statistical analysis on failures encountered by large-scale hpc deployment," *Proceedings of the 2008 High Availability and Performance Computing Workshop*, 2008.

[23] Oliner and J. Stearley, "What supercomputers say: A study of five system logs," *IEEE International Conference on Dependable Systems and Networks*, 2007.

[24] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[25] Z. Zheng and L. Yu, "Co-analysis of RAS Log and Job Log on Blue Gene/P," *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, pp. 840–851, 2011.

[26] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 111–122, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2150989

[27] C.-D. Lu, "Failure data analysis of hpc systems," *Technical Report CoRR abs/1302.4779*, 2013.

[28] T. Tsai, N. Theera-Ampornpunt, and S. Bagchi, "A study of soft error consequences in hard disk drives," *IEEE International Conference on Dependable Systems and Networks*, pp. 1–8, June 2012.

[29] T. J. Hacker, F. Romero, and C. D. Carothers, "An analysis of clustered failures on large supercomputing systems," *J. Parallel Distrib. Comput.*, vol. 69, no. 7, pp. 652–665, July 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.03.007

[30] D. Nurmi, J. Brevik, and R. Wolski, “Modeling machine availability in enterprise and wide-area distributed computing environments,” in *Euro-Par05*, 2003, pp. 432–441.

[31] S. Fu and C.-Z. Xu, “Exploring event correlation for failure prediction in coalitions of clusters,” in *Supercomputing, 2007. SC ’07. Proceedings of the 2007 ACM/IEEE Conference on*, Nov 2007, pp. 1–12.

[32] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello, “Modeling and tolerating heterogeneous failures in large parallel systems,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, 2011, p. 45.

[33] C. D. Martino, F. Baccanico, J. Fullop, W. Kramer, Z. Kalbarczyk, and R. K. Iyer, “Lessons Learned From the Analysis of System Failures at Petascale: The Case of Blue Waters,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

[34] J. Stearley and A. J. Oliner, “Bad words: Finding faults in spirits syslogs,” *The Eighth IEEE International Symposium on Cluster Computing and the Grid*, pp. 765–770, 2008.

[35] F. Salfner, “Modeling event-driven time series with generalized hidden semi-markov models,” *Technical Report 208, Department of Computer Science, Humboldt University*, 2006.

[36] K. Yamanishi, “Dynamic syslog mining for network failure monitoring,” in *In Proceedings of the 11th ACM SIGKDD, International Conference on Knowledge Discovery and Data Mining.* ACM Press, 2005, pp. 499–508.

[37] M. Y. Chen, A. Accardi, E. Kcman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, “Path-based failure and evolution management,” in *Proceedings of the International Symposium on Networked System Design and Implementation (NSDI04*, 2004, pp. 309–322.

[38] A. Andrzejak and L. M. Silva, “Deterministic models of software aging and optimal rejuvenation schedules.” in *Integrated Network Management.* IEEE, 2007, pp. 159–168.

[39] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, “Online detection of utility cloud anomalies using metric distributions.” in *NOMS.* IEEE, 2010, pp. 96–103.

[40] S. Rani, C. Leangsuksun, A. Tikotekar, V. Rampure, and S. Scott., “Toward efficient failure detection and recovery in hpc,” *In Proceedings of High Availability and Performance Workshop*, October 2006.

[41] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Scalable, fault-tolerant membership for mpi tasks on hpc systems," in *ICS06*, 2006.

[42] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," *Neural Networks, IEEE Transactions on*, vol. 16, no. 5, pp. 1027–1041, Sept 2005.

[43] K. Kharbas, D. Kim, T. Hoefler, and F. Mueller, "Assessing hpc failure detectors for mpi jobs," *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 81–88, 2012.

[44] I. Gertsbakh, "Reliability theory: with applications to pre- ventive maintenance," *Springer-Verlag*, 2000.

[45] F. A. Nassar and D. M. Andrews, "A methodology for analysis of failure prediction data," *IEEE Real-Time Systems Symposium*, pp. 160–166, 1985.

[46] R. Vilalta, C. Apte, J. Hellerstein, S. Ma, and S. Weiss, "Predictive algorithms in the management of computer systems," *IBM Systems Journal*, vol. 41, pp. 461–474, 2002.

[47] W. Farr, "Software reliability modeling survey," *Handbook of software reliability engineering*, pp. 71–117, 1996.

[48] A. Patra, S. Bidhar, and U. Kumar, "Failure Prediction of Rail Considering Rolling Contact Fatigue," *International Journal of Reliability, Quality and Safety Engineering*, vol. 3, 2010.

[49] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, pp. 147–187, 2010.

[50] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A Practical Failure Prediction with Location and Lead Time for Blue Gene/P," *IEEE Conference on Dependable Systems and Networks Workshops*, pp. 15–22, 2010.

[51] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[52] A. Csenki, "Bayes predictive analysis of a fundamental software reliability model," *IEEE Transactions on Reliability*, vol. 39, pp. 177–183, 1990.

[53] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," *In Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 202–209, 2001.

[54] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Improving the accuracy of network intrusion detection systems under load using selective packet discarding," in *Proceedings of the Third European Workshop on System Security*, ser. EUROSEC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1752046.1752049 pp. 15–21.

[55] Y. Liang, "Bluegene/l failure analysis and prediction models," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 425–434, 2006.

[56] S. Fu and C. Xu, "Quantifying temporal and spatial fault event correlation for proactive failure management," *IEEE Proceedings of Symposium on Reliable and Distributed Systems*, 2007.

[57] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, "Practical Online Failure Prediction for Blue Gene/P: Period-based vs Event-driven," *IEEE Conference on Dependable Systems and Networks Workshops*, pp. 259–264, 2011.

[58] N. Nakka, A. Agrawal, and A. Choudhary, "Predicting node failure in high performance computing systems from failure and usage logs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 1557–1566.

[59] J. Lou, "Mining dependency in distributed systems through unstructured logs analysis," *ACM The Special Interest Group on Operating Systems (SIGOPS)*, vol. 44, 2010.

[60] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.19 pp. 588–597.

[61] F. Salfner and M. Malek, "Using hidden semi-Markov models for effective online failure prediction," in *Symposium on Reliable Distributed Systems*, 2007, pp. 161 – 174.

[62] T. Hacker and F. Romero, "An analysis of clustered failures on supercomputing systems," *Journal of Parallel and Distributed Computing*, vol. 69, pp. 652–665, 2009.

[63] J. Murray, G. Hughes, and K. Kreutz-Delgado, "Hard drive failure prediction using non-parametric statistical methods," *Proceedings of ICANN/ICONIP*, 2003.

[64] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18 pp. 1–11.

[65] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–32.

[66] G. Zheng, L. Shi, and L. Kale, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing, 2004 IEEE International Conference on*, Sept 2004, pp. 93–103.

[67] A. Guermouche, T. Ropars, M. Snir, and F. Cappello., "Hydee: Failure containment without event logging for large scale send-deterministic mpi applications," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 1216–1227.

[68] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Impact of fault prediction on checkpointing strategies," INRIA, Rapport de recherche RR-8023, July 2012.

[69] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing strategies with prediction windows," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, Dec 2013, pp. 1–10.

[70] Y. Li and Z. Lan., "Exploit failure prediction for adaptive fault-tolerance in cluster computing," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, 2006.

[71] F. Cappello, H. Casanova, and Y. Robert, "Checkpointing vs. migration for post-petascale supercomputers," in *Parallel Processing (ICPP), 2010 39th International Conference on*, Sept 2010, pp. 168–177.

[72] M. S. Bouguerra, A. Gainaru, F. Cappello, L. B. Gomez, N. Maruyama, and S. Matsuoka, “Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing,” in *Proceedings of IEEE IPDPS 2013*. IEEE press, 2013.

[73] “National Center for Supercomputing Applications at the University of Illinois,” *www.ncsa.illinois.edu*, Accessed on 2010.

[74] B. Steinmacher-Burow, “Blue Gene/L Architecture,” *IBM Watson Journal*, 2004.

[75] W. Kramer, “Introduction to the blue waters project,” *National Center for Supercomputing Applications*, 2014.

[76] Z. Xue, X. Dong, S. Ma, and W. Dong, “A survey on failure prediction of large-scale server clusters,” in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, vol. 2, July 2007, pp. 733–738.

[77] S. Alam, “Early evaluation of ibm bluegene/p,” in *Proceedings of 2008 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008, p. 23.

[78] K. Chadalavada and R. Sisneros, “Analysis of the blue waters file system architecture for application i/o performance,” in *Cray User Group Meeting (CUG2013)*, 2013.

[79] C. Mendes, B. Bode, G. H. Bauer, J. Enos, C. Beldica, and W. Kramer, “Deploying a large petascale system: the blue waters experience,” in *Proc. 14th Int. Conf. Comput. Sci. (ICCS 2014)*, 2014.

[80] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema, “Analysis and modeling of time-correlated failures in large-scale distributed systems,” in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, Oct 2010, pp. 65–72.

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

[82] E. Kiciman and L. Subramanian, “A root cause localization model for large scale systems,” in *Proceedings of the First Conference on Hot Topics in System Dependability*, ser. HotDep’05. Berkeley, CA, USA: USENIX Association, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973400.1973402 pp. 2–2.

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

[84] K. Knizhnik, “Patricia tries: A better index for prefix searches,” in *Dr. Dobb’s Journal*, 2008.

[85] A. Gainaru, F. Cappello, S. Trausan-Matu, and W. Kramer, “Event log mining tool for large scale hpc systems,” in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, ser. Euro-Par’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–64.

[86] J. Fullop, A. Gainaru, and J. Plutchak, “Real time analysis and event prediction engine,” *Cray User Group (CUG)*, 2012.

[87] W. Waters and B. Jarrett, “Bandpass signal sampling and coherent detection,” *Aerospace and Electronic Systems, IEEE Transactions on*, vol. AES-18, no. 6, pp. 731–736, Nov 1982.

[88] G. K. Varshney, “An efficient methodology for noise characterization,” *International Conference on VLSI Design*, pp. 330–335, 2005.

[89] T. Lane and C. E. Brodley, “Temporal sequence learning and data reduction for anomaly detection,” *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 3, pp. 295–331, Aug. 1999. [Online]. Available: http://doi.acm.org/10.1145/322510.322526

[90] A. Laurent, B. Negrevergne, N. Sicard, and A. Termier, “Pgp-mc: Towards a multicore parallel approach for mining gradual patterns,” *Database Systems for Advanced Applications*, vol. 5981, pp. 78–84, 2010.

[91] D. Jorio, A. Laurent, and M. Teisseire, “Mining frequent gradual itemsets from large databases,” in *Int. Conf. on Intelligent Data Analysis*, 2009.

[92] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, “Fault prediction under the microscope: A closer look into hpc systems,” in *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE press, 2012.

[93] “The computer failure data repository,” http://cfdr.usenix.org, 2011, [Accessed on July 2011].

[94] L. Ma, Z. Huang, and Q. Wu, “Extracting common design patterns from a set of solid models.” *Computer-Aided Design*, vol. 41, no. 12, pp. 952–970, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/cad/cad41.html

[95] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Proceedings of CLUSTER09*. IEEE, 2009, pp. 1–10.

[96] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, R. Harrison, W. Harrod, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling, "Exascale software study: Software challenges in extreme scale systems," 2009.

[97] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth, "Modeling the impact of checkpoints on next-generation systems," in *24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 30–46.

[98] L. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 63–72.

[99] L. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka, "Low-overhead diskless checkpoint for hybrid computing systems," in *High Performance Computing (HiPC), 2010 International Conference on*, 2010, pp. 1–10.

[100] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[101] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[102] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[103] M. Bouguerra, D. Kondo, and D. Trystram, "On the scheduling of checkpoints in Desktop grids," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*, ser. CCGRID '11. NewPort Beach, CA, USA: IEEE Computer Society, May 2011, pp. 305–313.

[104] E. Jeannot, E. Saule, and D. Trystram, "Optimizing performance and reliability on heterogeneous parallel systems: Approximation algorithms and heuristics," *Journal of Parallel and Distributed Computing*, 2012.

[105] B. Javadi, D. Kondo, J. Vincent, and D. Anderson, "Discovering statistical models of availability in large distributed systems: An empirical study of seti@home," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 11, pp. 1896 –1903, 2010.

[106] E. Lehmann and G. Casella, *Theory of Point Estimation.* Springer Verlag, 1998.

[107] J. Massey and J. Frank, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[108] S. Matsuoka, "Making tsubame2.0, the world's greenest production supercomputer, even greener: Challenges to the architects," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, Aug 2011, pp. 367–368.

[109] V. Springel, "The cosmological simulation code gadget-2," in *Monthly Notices of the Royal Astronomical Society*, vol. 364. Blackwell Science Ltd, 2005, pp. 1105–1134.

[110] O. Agertz, B. Moore, J. Stadel, D. Potter, F. Miniati, J. Read, and L. Mayer, "Fundamental differences between sph and grid methods," *Monthly Notices of the Royal Astronomical Society.*

[111] D. J. Price, "Modelling discontinuities and kelvinhelmholtz instabilities in sph," *Journal of Computational Physics*, pp. 10 040 –10 057, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999108004270

[112] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*, 3rd ed. Network Theory Ltd., 2009.

[113] "Addressing failures in exascale computing," in *Institute of Computing in Science (ICiS) workshop*, 2012.

[114] C. Costa, Y. Park, B. Rosenburg, C.-Y. Cher, and K. D. Ryu, "A system software approach to proactive memory-error avoidance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.63 pp. 707–718.