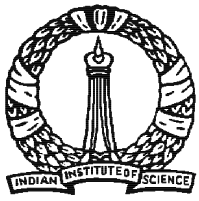
Adaptive Fault Tolerance Strategies for Large Scale Systems

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Cijo George



Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

August 2012

Acknowledgments

First of all I would like to extend my sincere thanks to my research supervisor Dr. Sathish Vadhiyar for his constant guidance and support during the entire period of my post-graduation at IISc. He was always approachable, supportive and ready to help in any sort of problem. Under his guidance I learned to approach problems in an organized manner and set realistic goals for my research. I thank him for his extreme patience and excellent technical guidance in writing and presenting research. He has always been a source of inspiration with his hard work and passion for research.

I would like to thank Prof. K. Gopinath, Dr. Varsha Apte and Dr. Satish Govindarajan for making the courses that I had taken in IISc a good learning experience. I am especially grateful to Dr. Varsha, who is a faculty of IIT Bombay, for the guidance and support during her time in IISc as a visiting professor. I really enjoyed the brain storming sessions in her classes and she has been a source of inspiration even outside academics. I am also thankful to Prof. N. Balakrishnan, Prof. R. Govindarajan and Prof. Mathew Jacob for always inspiring us with their motivational talks.

I would like to thank Preeti, Rajath and Hari for providing me with the much needed technical and emotional support at all times. I am also thankful to my other lab mates Akshay, Yogesh, Santanu, Sharat, Manogna and Sameer, who made working in the lab a pleasant and memorable experience.

Thanks are also due to my friends Arun, Scaria, Harisankar, Vivek, Nitin, Anas, Manu, Ravi and Jayadevan for their constant support and encouragement and for all the adventures we had, which provided me with good refreshing breaks in between my research work. I also thank my friends Syama, Parvathi, Ashwin, Arvind, Asha, Sindhu, Anierutha and Divya for being there with me through thick and thin.

Back home, I would like to thank my parents for being the best parents and my pillars of strength. I would also like to thank Cino, my sister, for always supporting me and being my advocate at home. I am grateful to Varghese, my uncle for always being there for me with guidance and inspiration. Last but not the least, I would like to thank Aparna, for being my best friend and companion in moments of joy and despair.

Abstract

Exascale systems of the future are predicted to have mean time between node failures (MTBF) of less than one hour. At such low MTBF, the number of processors available for execution of a long running application can widely vary throughout the execution of the application. Employing traditional fault tolerance strategies like periodic checkpointing in these highly dynamic environments may not be effective because of the high number of application failures, resulting in large amount of work lost due to rollbacks apart from the increased recovery overheads. In this context, it is highly necessary to have fault tolerance strategies that can adapt to the changing node availability and also help avoid significant number of application failures. In this thesis, we present two adaptive fault tolerance strategies that make use of node failure prediction mechanisms to provide proactive fault tolerance for long running parallel applications on large scale systems.

The first part of the thesis deals with an adaptive fault tolerance strategy for *malleable* applications. We present ADFT, an adaptive fault tolerance framework for long running malleable applications to maximize application performance in the presence of failures. We first develop cost models that consider different factors like accuracy of node failure predictions and application scalability, for evaluating the benefits of various fault tolerance actions including checkpointing, live-migration and rescheduling. Our adaptive framework then uses the cost models to make runtime decisions for dynamically selecting the fault tolerance actions at different points of application execution to minimize application failures and maximize performance. Simulations with real and synthetic failure traces show that our approach outperforms existing fault tolerance mechanisms for malleable applications yielding up to 23% improvement in work done by the application in the presence of failures, and is effective even for petascale and exascale systems.

In the second part of the thesis, we present a fault tolerance strategy using *adaptive process replication* that can provide fault tolerance for applications using partial replication of a set of application processes. This fault tolerance framework adaptively changes the set of replicated processes (replicated set) periodically based on node failure predictions to avoid application failures. We have developed an MPI prototype implementation, PAREP-MPI that allows dynamically changing the replicated set of processes for MPI applications. Experiments with real scientific applications on real systems have shown that the overhead of PAREP-MPI is minimal. We have shown using simulations with real and synthetic failure traces that our strategy involving adaptive process replication significantly outperforms existing mechanisms providing up to 20% improvement in application efficiency even for exascale systems. Significant observations are also made which can drive future research efforts in fault tolerance for large and very large scale systems.

Publications

- Cijo George, Sathish Vadhiyar. ADFT**: An Adaptive Framework for Fault Tolerance on Large Scale Systems using Application Malleability**. In the Proceedings of *International Conference on Computational Science (ICCS)*, June 2012, Omaha, USA.
- Cijo George, Sathish Vadhiyar. **Fault Tolerance on Large Scale Systems using Adaptive Process Replication**. (Under Preparation).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Parallel Systems and Applications

Advanced scientific research today in domains such as engineering, quantum physics, climate modeling, oil and gas exploration, molecular dynamics, drug discovery and physical simulations deals with very complex tasks and problems which are highly computation intensive. The introduction of parallel systems and the constant evolution of High Performance Computing (HPC) has provided massive computational power to solve some of the most complex research problems in these domains. Many of the present day scientific discoveries and engineering innovations are results of the existence of highly parallel large scale systems, also referred to as *supercomputers*. Today, HPC systems have massive number of processors, with the current fastest supercomputer IBM Sequoia (BlueGene/Q) consisting of $98,304$ compute nodes comprising of a total of approximately 1.6 million processor cores [53]. Sequoia has a theoretical peak performance of more than 20 petaflops.

Parallel scientific applications involve large communications. Some of these applications such as climate and whether modeling applications [13, 15] and molecular dynamics applications [5] are long running and their execution can go on for several weeks, if not months, even in very large scale systems. Molecular dynamics applications like NAMD [5] have to simulate billions of individual time steps for biomolecular simulations to study biological phenomena. This is because with atomic-level time and length scales being modeled, each time step can be in the order of only a femtosecond while most biological phenomena of interest require simulations in

the order of microseconds or even more. Hence, such molecular dynamics simulations are long running. For example, with each individual time step of size 2 femtoseconds carried out in just 10 milliseconds, it will take 2 months to complete a microsecond simulation. Similarly, in the domain of climate science, long running simulations for weeks or even months have to be done using climate modeling applications like CCSM [13] to study the effects of climate change far into the future. Maximizing performance of these applications on large scale parallel systems is a constant challenge for researchers both in academia and industry.

1.2 Fault Tolerance on Large Scale Systems

Node failures can happen in a parallel system due to several reasons including human error, software errors, environment problems like power outages, network issues and hardware failures. Among these, hardware failures of components including disk, memory and network components like Ethernet card are responsible for majority of the node failures in large scale systems [52]. These failures can result in the abnormal termination of processes executing on the those nodes. Process failures can result in failure of an entire application executing on the system due to dependencies between terminated processes and other processes of the parallel application. Such an abnormal termination of the application may result in the loss of work already completed by the application. If the time between occurrences of such node failures is less than the execution time of an application, which is mostly the case for long running applications, the application will never complete its execution. Also, as the number of nodes in the system increase, the mean time between node failures (MTBF) decreases and hence the frequency of node failures increases. Current systems with massive number of processors have an MTBF in the order of just a few hours. Hence, fault tolerance mechanisms that can avoid significant number of failures and minimize the work lost due to failures is an essential requirement for parallel applications on current and future large scale systems.

Fault tolerance mechanisms for parallel systems can be classified into two categories, namely, *reactive* mechanisms and *proactive* mechanisms.

Reactive mechanisms can be considered as “damage control mechanisms”, whose aim is to minimize the work lost due to a failure. All checkpointing mechanisms [48, 26, 2] come

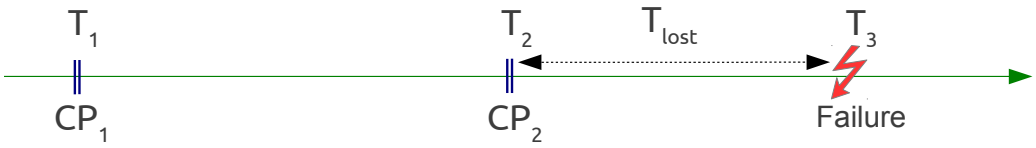


Figure 1.1: Illustration of checkpointing mechanism

under this category. Checkpointing mechanisms save the state of the application at different times during application execution, so that when the application fails, it can restart from the latest checkpoint. Hence, strategically taking checkpoints at different points during application execution can minimize the work lost due to failures. Figure 1.1 gives an illustration of checkpointing mechanism. Let the green arrow in the figure represent the application execution time line and CP_1 and CP_2 represent the two checkpoints taken at times T_1 and T_2, respectively. When the application fails at time T_3, it has to restart from the latest checkpoint, i.e., CP_2. The work done during time interval T_{lost}, between T_2 and T_3 is lost. Hence, apart from the overhead of resuming execution from a checkpoint, the application also incurs an extra overhead to redo the work done during the time T_{lost}. A good checkpoint strategy will try to minimize T_{lost}, thereby reducing the work lost due to failures.

Proactive mechanisms, on the other hand, can be considered as "damage prevention mechanisms", whose aim is to avoid as many application failures as possible. These mechanisms rely on the ability to foresee node failures in the system, so that proactive actions can be taken to avoid impending application failures. One such mechanism is proactive process migration [57, 9, 46], in which application processes from failure prone nodes are migrated to healthy spare nodes at regular intervals of time during application execution, thereby avoiding application failures.

Proactive fault tolerance mechanisms are complemented by node *failure prediction* mechanisms [27, 42] that help foresee impending failures. A *failure predictor* estimates the node failures in the system for a given time window. Strategies used for making predictions vary across different prediction mechanisms, with some using data mining techniques on failure logs to predict future failures and some others using hardware sensors to detect an impending failure. Failure predictors are not 100% accurate. Irrespective of their implementation, all failure predictors can be evaluated using two accuracy metrics, namely, *precision* and *recall*. Precision

of a predictor is defined as the ratio of the number of correct predictions to the total number of predictions made. Recall of the predictor is defined as the ratio of the number of predicted failures to the total number of failures in the system. The higher the precision of a predictor, higher will be the probability that a given prediction is true. The higher the recall of a predictor, higher will be the probability that an actual failure is predicted. Hence, the higher the values of precision and recall, better the predictor. Even with proactive fault tolerance mechanisms in place, unforeseen failures can happen in the system depending on the recall of the predictor, which may result in application failures. Hence, it is necessary that even with proactive fault tolerance, there should be some reactive mechanism like checkpointing to tolerate unexpected failures, albeit at a much lower frequency.

1.3 Malleability and Rescheduling

Some checkpointing systems support the development of parallel applications that can change the number of processors during execution [54, 62]. For example, this is achieved in SRS [54] by instrumenting the application with SRS calls for specifying data for checkpointing, along with the distribution to processors. These applications are referred to as *malleable* applications, and the action of changing the number of processors of a malleable application during execution is referred to as *rescheduling*.

Malleable applications can resume instantly after an application failure caused by a node failure in the system, even if there are no spare nodes in the system. The application can be *shrunk* to execute on smaller number of processors in healthy nodes on failure of some executing nodes rather than waiting for the failed nodes to be repaired. A non-malleable application, after such a failure, has to wait for the failed nodes to recover in order to resume execution in case there are no spare nodes in the system. Malleable applications can also be *expanded* to execute on larger number of processors when failed nodes are repaired and become available. The ability of malleable applications to adapt to varying node availability makes them highly useful in systems with large number of nodes where the node availability can vary frequently.

1.4 Process Replication

Process replication or *state machine replication* [51] is a popular technique for fault tolerance for high-availability systems. In process replication, the state of a process is replicated in another process called *replica* or *shadow* process, such that even if one of them fails, the application can continue execution without interruption. An application failure can happen only if both a process and its replica fail simultaneously. Since the probability of this is very less, process replication significantly increases the mean time between interrupt (MTTI) of an application with all processes replicated. Process replication is costly in terms of space since dedicated resources are allocated for the replica processes. If all processes are replicated, providing *dual redundancy*, then the maximum efficiency that an application can achieve on a given set of nodes will be only 50% since the number of nodes contributing to the computation is only half of the total number of nodes.

Due to the high costs involved, process replication has not been considered for fault tolerance on HPC systems in the past. But, recent research efforts [21, 7, 20] have shown that this technique is a promising fault tolerance mechanism for applications on current and future large scale systems, since the cost due to overheads caused by traditional fault tolerance mechanisms in such large scale systems is higher than the costs incurred by replication. It is a highly favorable option for systems with large number of failures like petascale and exascale systems, since node failures will result in application failures only if a replica is not present or if the replica also fails simultaneously, thus resulting in reduced number of application failures. Replication also leads to increased intervals between reactive mechanisms like checkpointing, thereby reducing the overheads caused by such mechanisms.

1.5 Motivation

With the development of high performance systems with massive number of processors [53] and long running scalable scientific applications that can use large number of processors for executions [44, 4], the mean time between failures (MTBF) of the processors used for a single application execution has tremendously decreased [47]. The current petascale systems are reported to have MTBFs of less than 10 hours [1, 34], and future exascale systems are anticipated

to have MTBFs of less than an hour [34]. However, long running scientific simulations including climate modeling and molecular dynamics [13, 15, 5] typically have execution times of the order of weeks to even months. Hence, it is highly imperative to develop efficient fault tolerance strategies to sustain executions of real scientific applications on future large scale systems like exascale systems.

Increased number of node failures in large scale systems due to low MTBF leads to the following two challenges for application execution in these systems.

1. The number of processors available for execution at a given point of time widely varies throughout application execution due to large number of node failures. In such cases, choosing a good static number of processors for execution is difficult, since a low number leads to sub-optimal application performance while a high number can result in large down times due to failures.

2. Relying on traditional fault tolerance techniques like periodic checkpointing on systems with low MTBF (hence, large number of node failures) results in a significant increase in the amount of work lost due to rollbacks since each node failure will result in an application failure. Reduced checkpoint intervals on these systems will also lead to more number of checkpoints contributing to the overall reduction in application efficiency.

In such a scenario, it is essential to have fault tolerance strategies that can adapt to the changing node availability in the system and also take proactive actions in anticipation of node failures to avoid significant number of application failures. In this work, we have focused on developing fault tolerance strategies for large scale systems that addresses the above challenges on these systems.

1.6 Problem Statement

In this thesis, we present two adaptive fault tolerance strategies that make use of node failure prediction mechanisms to provide proactive fault tolerance on large scale systems.

The first strategy, ADFT, focuses on fault tolerance for malleable applications by leveraging the benefits of rescheduling. We use cost models to dynamically select the best fault

tolerance action at different points of application execution, based on failure predictions. We have shown that our strategy involving malleability outperforms the popular but static periodic checkpointing approach by at least 21%, and also yields up to 23% higher amount of work than the dynamic *FT-Pro* strategy that does not involve malleability. Our results also show that our adaptive strategy yields high performance even for petascale and exascale systems.

The second strategy makes use of partial process replication to provide fault tolerance for parallel applications. Our adaptive process replication framework changes the replica set dynamically based on node failure predictions to avoid application failures. We have also developed an MPI prototype implementation PAREP-MPI that enables partial process replication and dynamic changing of the replica set for MPI applications. Overheads due to replication and adaptive replica change in PAREP-MPI is shown to be be only in the order of seconds even with multiple replica changes in parallel in a step. Our fault tolerance framework involving adaptive partial process replication is shown to significantly outperform existing mechanisms for large and very large scale systems providing up to 20% improvement in application efficiency even for exascale systems.

1.7 Thesis Organization

The rest of the thesis is organized as follows.

In Chapter 2, we discuss some of the related research works in the domain including that in fault tolerance mechanisms, failure prediction mechanisms and adaptive fault management. We give an overview of some of the important works in these areas and briefly discuss their advantages and disadvantages.

In Chapter 3 we present ADFT, an adaptive fault tolerance framework for malleable applications. We first develop cost models that evaluate the benefits of various fault tolerance actions including checkpointing, migration and rescheduling. Our adaptive framework then uses the cost models and failure predictions to make runtime decisions for dynamically selecting fault tolerance actions at different points of application execution.

In Chapter 4, we present a strategy for proactive fault tolerance using adaptive process replication which involves partial replication of a set of application processes. Our framework

adaptively changes the set of replicated processes based on failure predictions, to avoid application failures. We have also developed an MPI prototype implementation, PAREP-MPI that allows dynamically changing the processes in the replicated set with minimal overhead.

In Chapter 5, we first give an overview of the failure simulator which we have developed for the evaluation of our adaptive fault tolerance strategies. Subsequently we have presented our evaluation experiments and their results for both the strategies presented in this work. We have also given significant observations based on the experimental results.

In Chapter 6, we give the conclusions of our work. We also discuss some future directions for our research in this chapter.

Chapter 2

Related Research

Research in fault tolerance is primarily in three categories, namely, fault tolerance mechanisms, failure predictions and fault management. In this chapter, we look at some of the existing research works in the domain and analyze the pros and cons of these efforts.

2.1 Fault Tolerance Mechanisms

2.1.1 Periodic Checkpointing

Most of the fault tolerance mechanisms in literature are based on checkpointing [48, 26, 2]. It is the most popular and widely used technique to tolerate failures in parallel applications. In *periodic checkpointing*, the application state is saved or *checkpointed* at regular intervals of time. Whenever the application fails, it *rolls back* to the latest checkpoint and resumes execution from there. For each failure, the work done by the application between the latest checkpoint and the time of the failure is lost.

Research efforts have been made to study the impact of sub-optimal checkpoint intervals [33] and to develop analytical models for finding optimal checkpoint interval to maximize application performance [61, 16, 37]. Daly's higher order checkpoint/restart model [16] is a popular model that estimates the optimal checkpoint interval based on platform MTBF and overhead of checkpointing.

Periodic checkpointing does not avoid application failures. It is designed to only tolerate

application failures by taking regular checkpoints to minimize the work lost by the application on occurrence of failures. With periodic checkpointing, every node failure in the system will result in an application failure and the corresponding rollback-recovery overheads and lost work. Hence, periodic checkpointing gives low application efficiency for very large scale systems with low MTBF and high number of node failures.

2.1.2 Malleability and Rescheduling

Some checkpointing systems like SRS and programming languages like Charm++ support the development of parallel applications that can change the number of processors during execution [54, 62]. This is achieved in SRS [54] by instrumenting the application with SRS calls for specifying data for checkpointing, along with the distribution to processors. Charm++ is an object-oriented programming language based on C++, which enables a programmer to develop applications that can be decomposed into several cooperating objects called *chares*, which communicate with each other through messages. The Charm++ runtime system can dynamically change the assignment of chares to processors and the number of processors used by the application.

The applications which can change the number of processors during execution are known as *malleable* applications, and the action of changing the number of processors of a malleable application during execution is known as *rescheduling*. Malleable applications can resume instantly from an application failure caused by a node failure, by rescheduling to the remaining healthy nodes in the system and hence it can adapt to varying node availability in the system.

2.1.3 Proactive Process Migration

Proactive process migration [57, 9, 46] is a technique in which processes in failure prone nodes are migrated to healthy spare nodes periodically to avoid application failures. In live process migration, the process images are transferred from failure prone nodes to healthy spare nodes transparent to the application. Overhead of live process migration is much lesser compared to that of checkpointing.

Proactive migration relies on the presence of node failure prediction mechanisms that can

periodically estimate the failure-prone nodes in the system. Any of the failure prediction mechanisms in literature can be used for the purpose. Section 2.2 gives an overview of the existing failure prediction mechanisms.

Proactive migration can avoid application failures caused by node failures which are predicted by the failure predictors. But, there can be unpredicted node failures in the system. Proactive migration relies on checkpointing to tolerate such unforeseen failures, but with a much higher checkpoint interval.

Cappello et al. [12] have analyzed proactive migration as a fault tolerance option for petascale and exascale systems. The work compares proactive migration with proactive checkpointing using analytical performance models, based on the assumption of having a perfect failure predictor that can predict all failures in the system just before they happen with 100% accuracy. They have reported that proactive migration can provide better fault tolerance compared to periodic checkpointing for future exascale systems.

2.1.4 Process Replication

Process replication or *state machine replication* [51] is being increasingly considered for fault tolerance for current and future large scale systems [21, 7, 20]. In process replication, the state of a process is replicated in another process called *replica* or *shadow* process, such that even if one of them fails, the application can continue execution without interruption. Process replication results in reduced number of application failures since node failures will result in application failures only if a replica is not present or if the replica also fails simultaneously. Replication also leads to increased intervals between checkpointing, thereby reducing overheads due to checkpointing.

Ferreira et al. [21] have shown that a process replication strategy with dual hardware redundancy, in which all processes in a system are replicated, can significantly increase the mean time to interrupt (MTTI) of an application. This is because in such a system the probability of both a processes and its replica failing at the same time is very low, as modeled by the authors using the *birthday problem* [39]. They have shown using simulations with an exponential failure distribution that replication outperforms traditional periodic checkpointing for socket counts greater than $20,000$ and is a viable fault tolerance technique for socket counts and I/O band-

widths anticipated for future exascale systems. But, Weibull failure distribution is shown to be a more challenging scenario where both periodic checkpointing and dual redundancy gives low application efficiency. They have also developed *r*MPI, which is a user-level MPI library that enables replication for MPI applications. Using real MPI applications, they have shown that the worst case overhead due to replication using *r*MPI is about 4.9%.

In spite of these benefits, dual redundancy can achieve only less than 50% application efficiency, since only half the total number of nodes is used for actual application execution. This can result in huge amount of resource wastage on large scale systems. Elliott et al. [20] have developed a fault tolerance strategy that combines periodic checkpointing with partial redundancy. They have shown that partial redundancy can be beneficial even for medium scale systems with $4,000$ to $25,000$ processors with a degree of redundancy between 1.5 and 2, where a degree of 2 corresponds to dual redundancy. Even a degree of redundancy of 2.5 is shown to give maximum efficiency for some MTBF values. They have also developed an analytical model to determine the optimal degree of redundancy and checkpoint frequency for maximum performance. While this work deals with finding an optimal degree of replication that gives maximum performance, the nodes that are replicated are arbitrarily chosen. Our adaptive replication strategy dynamically changes the set of replicated processes based on failure predictions, to avoid the nodes that are predicted to fail. They have also developed RedMPI, which is a user-level MPI library similar to *r*MPI that enables redundant computing for MPI applications, with support for both partial and dual redundancy. RedMPI provides wrappers around MPI calls, which are implemented in the PMPI layer to enable redundancy.

The work by Yang et al. [60] has developed FTPA, which is an application level fault tolerance mechanism in which the surviving processes recompute the workload of the failed processes in parallel. It is a reactive mechanism to survive after a failure happens. Our work on the other hand, provides comprehensive frameworks that can perform proactive actions to avoid predicted failures as well as reactive actions to survive after failures.

2.2 Failure Prediction Mechanisms

To help a runtime system use proactive fault tolerance mechanisms like process migration, techniques have been developed to predict node failures in the system [42, 27, 56, 50, 36, 55, 28, 29, 32, 38, 10]. These predictions can either be *categorical* where the predictor estimates whether a node will fail or not, or *probabilistic* where the predictor gives probabilities of node failures for a given time window.

Failure prediction mechanisms are evaluated using two metrics, namely, *precision* and *recall*. Precision of a predictor is the ratio of the number of correct predictions to the total number of predictions made by the predictor. Recall of the predictor is the ratio of the number of predicted failures to the total number of failures in the system. The higher the precision of a predictor, the higher will be the probability that a given prediction is true. The higher the recall of a predictor, the higher will be the probability that an actual failure is predicted. Hence, the higher the values of precision and recall, better the predictor.

Failure prediction mechanisms use different techniques to predict node failures. These techniques can be broadly classified into hardware based techniques and software based techniques, which are explained below.

2.2.1 Hardware Based Prediction Techniques

Computer systems today are designed with health monitoring mechanisms that use hardware sensors that can continuously monitor the health of various hardware components in the system. This helps in early detection of a significant number of the hardware errors that causes system failure.

The Intelligent Platform Management Interface (IPMI) [32] is a popular message-based hardware-level computer system interface that helps system administrators monitor the platform parameters including system temperatures, voltages, fans, power supplies and chassis intrusion. The IPMI standard also defines an alerting mechanism that uses Simple Network Management Protocol (SNMP) Platform Event Trap (PET) which can help in hardware failure predictions. Linux-Monitoring Sensors (lm sensors) [38], is a free open source software-tool for Linux, which applications can use to connect to IPMI to obtain various system parameters for

health monitoring using an API library. Self-Monitoring, Analysis and Reporting Technology (SMART) [10] is a system that helps in detecting the possibility of hard disk failure in a system by monitoring the disk activity for various indicators of disk reliability.

2.2.2 Software Based Prediction Techniques

Many software based techniques have been developed to predict failures based on failure patterns developed by analyzing the failure history of the system. These techniques can be classified into *model-based* techniques and *data-driven* techniques. In model-based techniques [55, 28, 29], an analytical or probabilistic model of the system is developed and any deviation from the model trigger a warning. Data-driven techniques [56, 50, 36, 27, 42] make use of data-mining to dynamically learn the failure patterns in the system.

Gujrati et al. [27] has developed a meta-learning failure predictor for BlueGene/L that integrates the predictions of two base predictors - one that uses a statistical based method and another that uses an association rule based method. The statistical based method uses statistical characteristics of fatal events to predict failures, while the rule based method identifies casual correlations between fatal and non-fatal events to predict failures. The meta-learning predictor uses these two base methods to provide better prediction accuracy. Evaluation based on real logs of clusters with up to 2048 processors from Argonne National Laboratory and San Diego Supercomputer Center show that the predictor gives a precision of up to 0.88 and a recall of up to 0.78. Though this work is targeted towards BlueGene/L, it is claimed that a similar technique can be used for other large scale systems too.

Recent work by Nakka et al. [42] has applied data mining classification schemes for failure prediction in large scale systems. For each failure instance from a failure log, both past and future failure and usage information is collected including time of usage, system idle time, time of unavailability time since last failure, time to next failure. Decision tree classifiers are applied on these data to predict future failures with a time window of 1 hour. Results based on real logs from Los Alamos National Laboratory have shown that their prediction mechanism achieves a precision of about 0.8 and a recall of about 0.73.

One way to use existing techniques for prediction in large scale systems in a scalable fashion will be to divide the system into different set of nodes, with separate failure prediction systems

for each set. Predictions from each such system can then be combined to generate predictions for the entire system. Our work highlights the significance of proactive fault tolerance for very large scale systems like future exascale systems and hence, will also serve as a motivation to develop failure prediction mechanisms for such systems.

2.3 Adaptive Fault Management

Recently, there have been efforts to develop fault management frameworks with runtime policies for fault tolerance. Lan and Li have developed an adaptive fault management framework [35] similar to the focus of one of the fault tolerance strategies, ADFT, presented in our work. Their *FT-Pro* framework provides fault tolerance for non-malleable applications by performing proactive migration or checkpointing based on a cost model. They have evaluated their work using stochastic modeling for a maximum of 192 nodes and trace based simulations for a maximum of 64 nodes using short running applications, based on the time taken by the applications to finish execution, and have compared their results with periodic checkpointing. Our work focuses on developing a fault tolerance strategy for long running applications on very large scale systems, using application malleability. ADFT takes runtime proactive fault tolerance actions including proactive rescheduling, to avoid failures. Our cost model takes into consideration the malleability of the application and the option to change the number of processors and recover immediately after a failure. Application scalability is also taken into account in making fault tolerance decisions. We evaluate ADFT using real failure traces from LANL for 512 and 1024 nodes and also using synthetic traces for very large scale systems. We use synthetic scalability curves as well as the scalability curves of real long running applications for the simulations. Simulations are also done for petascale and exascale systems.

Chapter 3

Adaptive Fault Tolerance for Malleable Applications

3.1 Introduction and Motivation

Most of the traditional fault tolerance techniques including checkpointing [48, 2], and live process migration [57, 9, 46] resume the application on the same number of processors after failures. Choosing a good static number of processors for execution is difficult in large scale systems like petascale and exascale systems where the number of processors available at a given point of time widely varies throughout application execution due to the very low MTBFs on these systems. In such cases, choosing a good static number of processors for execution is difficult, since a low number leads to sub-optimal application performance while a high number can result in large down times due to failures. Checkpointing systems like SRS [54] and programming languages like Charm++ [62] enables the development of malleable applications which can change the number of processors during execution. Such applications can adapt to the varying node availability in the system by rescheduling to the available healthy nodes in the event of a failure. In these systems, the application can be *shrunk* to execute on smaller number of processors on failure of some executing processors rather than waiting for the failed processors to be repaired, or *expanded* to execute on larger number of processors when failed processors are repaired and become available.

With the development of these different fault tolerance strategies, namely, checkpointing

and continuing on the same number of processors (referred to as simply checkpointing in the rest of the thesis), live process migration, and checkpointing and rescheduling to different number of processors (referred to as simply rescheduling in the rest of the thesis), the selection of a strategy for application execution has to be carefully made to maximize the application performance in the presence of failures. Adopting a single fault tolerance strategy may not yield high performance of the application on large scale systems in the presence of failures. Depending on failure predictions and cost of the different strategies, a runtime system may have to dynamically select the most cost-effective fault tolerant strategy (including not taking any action at all) at a given instance of application execution. Lan and Li have developed an adaptive fault management system, called *FT-Pro* [35], that makes dynamic decisions for fault tolerance. However their work confines to applications that execute on a fixed number of processors throughout application execution. Also, they have performed their analyses on small-scale systems.

In this work, we have developed ADFT, an adaptive fault tolerance framework for long running malleable applications to maximize application performance in the presence of failures. We first develop cost models that consider different factors like accuracy of failure predictions and application scalability, for evaluating the benefits of various fault tolerance actions including checkpointing, live-migration and rescheduling. Our adaptive framework then uses the cost models to make runtime decisions for dynamically selecting the fault tolerance actions at different points of application execution to maximize performance. Our work is applicable for mostly regular applications following regular domain decomposition. We have also extended *FT-Pro* to consider application scalability and live-migration and made other adaptations to make it suitable for long running applications and large number of nodes.

Using simulations with synthetic and real failure traces, and synthetic and real application scenarios, we evaluate our dynamic ADFT strategy in terms of work done per unit time by the application in the presence of failures on large and very large scale systems. Our results show that our strategy involving malleability outperforms the popular but static periodic checkpointing approach by at least 21%, and also yields up to 23% higher amount of work than the dynamic *FT-Pro* strategy that does not involve malleability. Our results also show that our adaptive strategy yields high performance even for petascale systems and beyond, and that application malleability will be highly essential for future exascale systems.

3.2 ADFT Framework

We assume the presence of a *failure predictor* [27, 42] that can estimate the node failures in the system for a given time window. Specifically, the predictor is expected to periodically estimate the list of nodes that are expected to fail in the next time interval. *Precision* of such a predictor is defined as the ratio of the number of correct predictions to the total number of predictions made. *Recall* of the predictor is defined as the ratio of the number of correct predictions to the total number of failures. Higher the values of precision and recall, the better the predictor.

Work done by an application is considered as being comprised of logical *units of work* (e.g. for an iterative application, a unit of work can be a single iteration). ADFT takes runtime fault tolerance actions upon completion of every x units of work, where x is a constant. These fault tolerance action points or decision making points are denoted as adaptation points (AP) and the constant work to be completed between each AP is denoted as W. Ideally, the interval between adaptation points should be equal to the failure prediction interval for which a predictor gives maximum accuracy. In our evaluations, we set W as the work done by the application in 30 minutes in a failure free environment on the given number of nodes. This is based on the results in previous efforts on failure predictors [27, 42] that report the best accuracy metrics for a time window between 15 minutes to 1 hour depending on the system.

An *adaptation manager* is responsible for determining the most suitable action upon each AP. Following are the possible actions that can be taken by the adaptation manager.

- *SKIP*, where no action is taken.
- *CHECKPOINT*, where the application takes a proactive checkpoint. We assume coordinated checkpointing, where the processes synchronize to perform checkpointing.
- *MIGRATE*, where the processes on failure-prone nodes (i.e the nodes predicted to be failure-prone in the near future) are migrated to healthy nodes. We assume that live-migration method [57, 9, 46], which does not involve checkpointing, is used for the purpose.
- *RESCHEDULE*, where the application is rescheduled to a different set of nodes, which does not include any failure prone node. This also involves redistribution of data to the

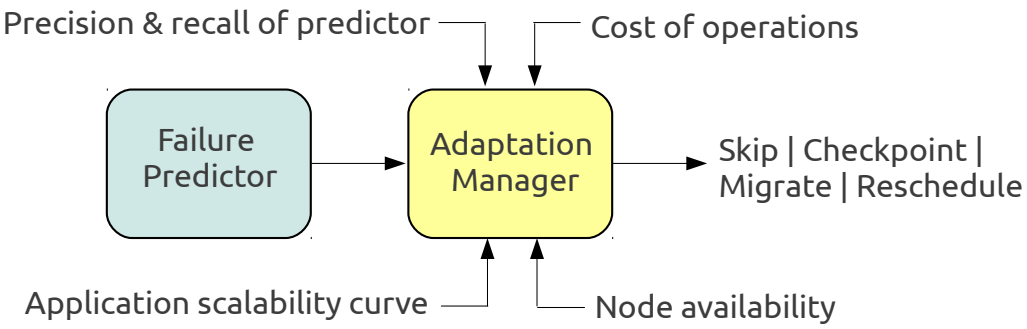


Figure 3.1: ADFT **framework**

new set of nodes. The adaptation manager performs two kinds of rescheduling: *proactive* and *reactive*. Proactive rescheduling is performed based on our cost model and using failure predictions in anticipation of failures. Reactive rescheduling is performed on occurrence of failure of the executing application due to failure of some node(s) on which the application is executed. The application is recovered from the failure by rescheduling to a different number of available/active nodes.

The overall working of our ADFT framework is illustrated in Figure 3.1.

3.3 A Cost Model for Application Execution between Adaptation Points

The adaptation manager takes into account three sets of parameters for decision making, namely, prediction accuracy, operation costs of different actions and the number of available resources for proactive actions. It estimates the cost of each of the possible actions based on the given parameters using a cost model and takes the action with minimal estimated cost. Specifically, the action that minimizes the time required to complete W amount of work and reach the next AP is selected.

The adaptation manager also requires the specification of scalability of the application in the form of work done per unit time on various number of processors on the system. For example, in the case of an iterative application, the scalability data can specify the number of iterations done per unit time on different number of processors. Our work is primarily intended for long-running large scientific applications. Such applications are typically benchmarked by

the users for different problem sizes and number of processors for application development and performance optimization. The application scalability data is used to compute the following two variables in our cost model.

- $N(n)$: The number of nodes, $p \leq n$, corresponding to maximum work done by the application per unit time.
- $T(w, n)$: Time taken by the application to perform w units of work on n nodes. This is obtained by dividing w with the work done per unit time for n nodes obtained from the application scalability data.

At each AP, the failure predictor forecasts the expected node failures for the next time interval I, where I is the estimated time to complete W amount of work using the current working set of nodes in a failure-free environment (given by $I = T(W, N_w)$). The adaptation manager uses the cost model to compute E_{next}, the expected time to complete the next W amount of work for each possible fault tolerance action that can be taken.

We list a set of nomenclatures that will be frequently used in the rest of this thesis in Table 3.1.

3.3.1 Illustration: Cost Model for 3 Nodes

We assume that malleable applications can be recovered instantly from node failures by rescheduling to a different set of nodes (*reactive rescheduling*). Suppose at AP_i, the predictor predicts that nodes A, B and C are prone to failures in the next time interval I. The worst case in which these nodes can fail is as follows.

- Node A fails when the application is about to reach AP_{i+1}.
- Node B fails after the application recovers from failure of node A and is about to reach AP_{i+1} again.
- Node C fails after the application recovers from failure of node B and is about to reach AP_{i+1} again.

Table 3.1: Nomenclature

Symbol	Description
AP	Adaptation point
N_w	No. of working nodes (nodes on which the application is currently running)
N_f	No. of nodes among the working nodes which are predicted to fail in the next interval
N_s	No. of unused nodes in the system which are not predicted to fail in the next interval
T_{resch}	Time overhead to reschedule an application to a different no. of nodes (not including checkpointing overhead)
$T_{recover}$	Time overhead for startup of the application on a set of nodes
T_{ckp}	Time overhead for checkpointing the application
T_{mig}	Time overhead for migration of an application process from one node to another
P	Precision of the predictor, which is the ratio of the no. of correct predictions to the total number of predictions
R	Recall of the predictor, which is the ratio of the no. of correct predictions to the total number of actual failures
W	A constant amount of work. APs are set to where the application completes each work segment of size W
$AP_{current}$	Index of the current AP
AP_{ckp}	Index of the latest adaptation point where a checkpoint was taken
W_{lost}	Total work done by the application from the time the latest checkpoint was taken to the latest AP ($W_{lost} = (AP_{current} - AP_{ckp}) * W$)
E_{next}	Expected time to reach the next AP (i.e expected time to complete the next segment of work of size W)
$T(w, n)$	T is a function that returns the time required to complete w amount of work if n nodes are used
$N(n)$	N is a function that returns the optimal number of nodes to be used for best performance if a total of n nodes are available
I	Time interval for which the predictor predicts failures. I is calculated at each adaptation point as $I = T(W, N_w)$

Table 3.2: Failure scenarios and their probability (example)

Scenario	*Probability*
All three of the nodes A, B and C fail	P^3
Any two of A, B and C nodes fail	${}^3C_2 * P^2 * (1-P)$
Any one of A, B and C nodes fail	${}^3C_1 * P^1 * (1-P)^2$
None of the nodes fails	$(1-P)^3$

If a ***SKIP*** action was taken at AP_i, the total time taken to reach AP_{i+1} in this example can be expressed as follows.

$$\begin{aligned} E_{next} = {} & T(W, N_w) \\ & + [T_{resch} + T_{recover} + T((W_{lost} + W), N(N_w - 1 + N_s))] \\ & + [T_{resch} + T_{recover} + T((W_{lost} + W), N(N_w - 2 + N_s))] \\ & + [T_{resch} + T_{recover} + T((W_{lost} + W), N(N_w - 3 + N_s))] \end{aligned} \tag{3.1}$$

Equation (3.1) can be explained as follows. The application takes $T(W, N_w)$ time to perform W amount of work using the current set of nodes, N_w, to reach AP_{i+1}. At this point one of the three nodes fails and the application spends T_{resch} time for reactive rescheduling and $T_{recover}$ time for recovering the application on the new set of nodes, $(N_w - 1 + N_s)$. The new set of nodes is obtained by excluding the node that has failed and including the spare nodes, N_s. The application then spends $T((W_{lost} + W), N(N_w - 1 + N_s))$ time to reach AP_{i+1} using the new set of nodes. Here, W_{lost} is the work done between the last checkpoint and AP_i. When the application almost reaches AP_{i+1}, the second node fails and similar costs are involved to reach AP_{i+1} again. Then the third node fails and the process repeats. Hence the total cost is as given in Equation (3.1).

Equation (3.1) can be simplified and expressed as follows.

$$\begin{aligned} E_{next} = {} & T(W, N_w) + 3 * (T_{resch} + T_{recover}) \\ & + \sum_{j=1}^{3} T((W_{lost} + W), N(N_w - j + N_s)) \end{aligned} \tag{3.2}$$

Equation (3.2) gives the time taken to reach AP_{i+1} if a SKIP decision was taken at AP_i and all 3 nodes predicted to fail actually fails and in the worst possible way, which is just one possible scenario. Table 3.2 shows all the possible scenarios and their corresponding probabilities if 3 nodes are predicted to fail. Note that the probability that a given node which is predicted to fail will actually fail is equal to precision, P, of the predictor.

In general, the probability that i nodes out of the 3 nodes which are predicted to fail will actually fail is given by ${}^3C_i * P^i * (1-P)^{3-i}$. Now the estimated cost of the *SKIP* decision in the given example considering all scenarios can be expressed as given below.

$$E_{next} = \sum_{i=1}^{3} {}^3C_i * P^i * (1-P)^{3-i} * [T(W, N_w) + i * (T_{resch} + T_{recover}) + \sum_{j=1}^{i} T((W_{lost} + W), N(N_w - j + N_s))] + (1-P)^3 * [T(W, N_w)] \quad (3.3)$$

Similarly, the cost model is developed for different actions for N_f number of predicted node failures.

3.3.2 A General Cost Model

At each AP, ADFT computes the estimated cost of each of the possible actions using the cost model given below and takes the action that has the least estimated value of E_{next}.

- *SKIP*: Depending on the number of nodes that fail, the application may have to perform rollback recovery several times. If none of the nodes fail, no extra costs are incurred and the time taken to reach the next AP will be the time taken to complete W amount of work using N_w nodes. Hence, E_{next} corresponding to *SKIP* decision by application manager, is computed as follows.

$$E_{next} = \sum_{i=1}^{N_f} {}^{N_f}C_i * P^i * (1-P)^{N_f - i} * [T(W, N_w) + i * (T_{resch} + T_{recover}) + \sum_{j=1}^{i} T((W_{lost} + W), N(N_w - j + N_s))] + (1-P)^{N_f} * [T(W, N_w)] \quad (3.4)$$

Equation (3.4) is a simple adaptation of Equation (3.3) for N_f number of nodes. In the equation, $\sum_{i=1}^{N_f} {}^{N_f}C_i * P^i * (1-P)^{N_f-i}$ is the probability that the application will fail and $(1-P)^{N_f}$ is the probability that the application will not fail.

- *CHECKPOINT*: When the adaptation manager decides to checkpoint, the application spends some time for checkpointing at the beginning of the next interval. The expected time to reach the next adaptation point, E_{next}, if the application is checkpointed at the current AP, is computed as follows.

$$E_{next} = \sum_{i=1}^{N_f} {}^{N_f}C_i * P^i * (1-P)^{N_f-i} * [T_{ckp} + T(W, N_w) + i * (T_{resch} + T_{recover}) + \sum_{j=1}^{i} T(W, N(N_w - j + N_s))] + (1-P)^{N_f} * [T_{ckp} + T(W, N_w)] \quad (3.5)$$

  If the application fails, the cost involved will be the sum of the time for checkpointing, T_{ckp}, the time to reach the next adaptation point, $T(W, N_w)$, the cost of rescheduling (T_{resch}) and recovery ($T_{recover}$) for each of the node failures and the time taken to redo the work to reach the next AP for each of the node failures, $T(W, N(N_w - j + N_s))$. If the application does not fail, the cost will be the sum of T_{ckp} and the time to reach the next AP, $T(W, N_w)$.

- *MIGRATE*: In this case, the adaptation manager decides to perform *live-migration* at the beginning of the next interval. In live-migration, the images of the processes executing on a set of nodes predicted to fail are simply transferred to a set of spare nodes while the application continues execution [57]. There are two possible scenarios.

  1. if $N_f <= N_s$, i.e. the number of nodes predicted to fail is less than the number of spare nodes, all failure prone nodes can be migrated to healthy spare nodes and hence failure probability will be *ZERO*.
  2. if $N_f > N_s$, only N_s number of failure prone nodes can be migrated to healthy nodes. Hence there is still a failure probability involving $N_f - N_s$ nodes.

The above two conditions are taken care of by defining a variable, N_{fm}, as follows.

$$E_{next} = \sum_{i=1}^{N_{fm}} {}^{N_{fm}}C_i * P^i * (1-P)^{N_{fm}-i} * [T_{mig} + T(W, N_w) + i * (T_{resch} + T_{recover}) + \sum_{j=1}^{i} T((W_{lost} + W), N(N_w - j + N_s))] + (1-P)^{N_f} * [T_{mig} + T(W, N_w)] \tag{3.6}$$

If the application fails, the cost involved will be the sum of the time for migration, T_{mig}, the time to reach the next adaptation point, $T(W, N_w)$, the cost of rescheduling (T_{resch}) and recovery ($T_{recover}$) for each of the node failures and the time taken to redo the work to reach the next AP for each of the node failures, $T((W_{lost} + W), N(N_w - j + N_s))$. The latter time includes W_{lost} since no checkpoint is taken for live-migration at the current AP. If the application does not fail, the cost will be the sum of T_{mig} and the time to reach the next AP.

- *RESCHEDULE*: Here, the probability of application failure is *ZERO* since the application is rescheduled, avoiding all failure prone nodes. Hence, the cost involves only the overhead for rescheduling and the time taken to complete W amount of work using the new set of nodes. E_{next} is computed as follows.

$$E_{next} = T_{ckp} + T_{resch} + T_{recover} + T(W, N(N_w - N_f + N_s)) \tag{3.7}$$

The cost model explained above relies on the precision of the predictor. But, for a predictor with a recall value of less than 1, there can also be failures which are not predicted. To tolerate such unforeseen failures, a *precautionary checkpoint* is taken when the time since last checkpoint reaches a threshold. Given the recall of the predictor, the time interval between such unpredicted failures can be estimated as $\frac{platform\ MTBF}{1 - recall}$. This value is taken as the threshold for precautionary checkpointing.

Chapter 4

Fault Tolerance using Adaptive Process Replication

4.1 Introduction and Motivation

Periodic checkpointing [48, 26, 2] is the most popular and long-established strategy for fault tolerance in such systems and applications. In periodic checkpointing systems, the application is periodically made to store its state in anticipation of failures. Such periodic checkpointing allows the application to rollback to a recently checkpointed state on the occurrence of failures. With lower platform MTBFs on systems with large number of processors, the number of failures increases resulting in a significant increase in the amount of work lost due to rollbacks. Reduced checkpoint intervals on systems with low platform MTBF will also lead to more number of checkpoints contributing to the overall reduction in application efficiency. Recent studies [24, 12, 35] have shown that periodic checkpointing on peta and exa scale systems results in application efficiencies of only 20-30%! Thus, traditional periodic checkpointing is not a viable fault tolerance option for large scale systems. In such scenarios, it is highly necessary to employ proactive fault tolerance mechanisms that can help avoid significant number of failures.

Process replication or *state machine replication* [51] is being increasingly considered for fault tolerance for current and future large scale systems [21, 7, 20]. In process replication, the state of a process is replicated in another process called *replica* or *shadow* process, such that even if one of them fails, the application can continue execution without interruption. Process

replication is a highly favorable option for fault tolerance on large scale systems with large number of failures, since unlike periodic checkpointing, node failures will result in application failures only if a replica is not present or if the replica also fails simultaneously, thus resulting in reduced number of failures. Replication also leads to increased intervals between checkpointing, thereby reducing checkpointing overheads. Ferreira et al. [21] have shown that a process replication strategy with dual hardware redundancy, in which processes in all sockets (or nodes) in a system are replicated [1], can significantly increase the mean time to interrupt (MTTI) of an application. This is because in such a system the probability of both a socket and its replica failing at the same time is very low, as modeled by the authors using the *birthday problem* [39]. As shown in their work, all these advantages of replication result in significantly higher application efficiency than periodic checkpointing for peta and exascale systems.

In spite of these benefits, dual redundancy can achieve only less than 50% application efficiency, since only half the total number of nodes is used for actual application execution. This can result in huge amount of resource wastage on large scale systems. To verify this, we performed experiments involving execution on 200,000 nodes/processors with dual redundancy for one week with a node MTBF of 25 years and Weibull distribution of failures. We found that the total number of node failures before an application failure, due to both a process and its replica failing, is only between 50 to 400. Thus only a maximum of 400 of the 200,000 nodes needed to be replicated. However, in anticipation of this very small number of failures, the dual redundancy scheme replicates $100,000$ nodes, thereby utilizing only $100,000$ nodes for application execution! While about $199,600$ nodes or more than 99% of the total number of nodes could have been used for application execution, the dual redundancy scheme utilizes only 50%, resulting in large resource wastage. The primary reason for this "safe" approach of dual redundancy is due to the lack of knowledge of the specific 400 nodes that may fail during execution. While Elliott et al. [20] have explored partial redundancy with redundancy degrees between 1.5 and 2, where a degree of 2 corresponds to dual redundancy, their results do not show clear advantages of partial redundancy over dual redundancy in all cases. Moreover, the subset of nodes that were replicated in their partial redundancy scheme is arbitrarily chosen (e.g., replication of every other node for 1.5 redundancy).

[1] We use process replication and node replication interchangeably.

In this work, we have developed a mechanism for proactive fault tolerance using partial replication of a set of application processes. Our framework starts an application with a fixed small number of processes replicated. At regular intervals of time, our framework adaptively changes the set of replicated processes based on failure predictions such that all failure prone nodes have healthy replica, thus attempting to avoid failures. Our adaptive strategy leverages on the advantages of process replication while keeping the number of replica nodes to a minimum, thereby minimizing resource wastage.

Our fault tolerance framework relies on a cost efficient mechanism to adaptively change the set of replicated processes without having to checkpoint/restart the application. We have developed an MPI prototype implementation, PAREP-MPI that makes this possible by providing the ability to transparently copy a process state from a process in a local node to a process in a remote node and modify the remote process to act as the replica of the local process. We have shown through experiments on real systems that PAREP-MPI can change the set of replicated processes efficiently by changing individual replicas in parallel. We have also shown that the overheads due to replication in PAREP-MPI is minimal even when all the processes in the set of replicated processes is changed in parallel in a step.

Simulations using synthetic failure traces for exponential and Weibull distributions have shown that our fault tolerance framework involving adaptive partial process replication significantly outperforms existing mechanisms for large and very large scale systems providing up to 20% improvement in application efficiency even for exascale systems. Our evaluations using real traces from LANL [23] shows that our strategy is effective for small and medium scale systems as well.

4.2 Adaptive Process Replication Framework

We assume the presence of a *failure predictor* that can estimate the node failures in a system for a given time window. Specifically, the predictor has to periodically estimate the list of nodes that are expected to fail in the next time interval. Any of the failure prediction mechanisms mentioned in Section 2.2 can be used for the purpose. *Precision* of such a predictor is defined as the ratio of the number of correct predictions to the total number of predictions made. *Recall*

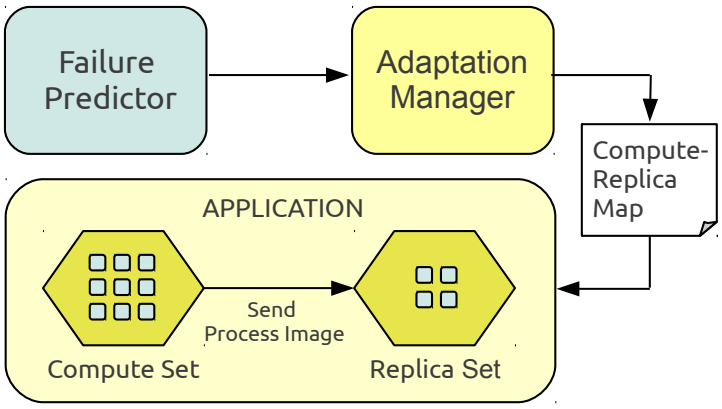


Figure 4.1: Adaptive process replication framework

of the predictor is defined as the ratio of the number of correct predictions to the total number of failures. Higher the values of precision and recall, the better the predictor. A node is considered as *failure prone* if it is predicted to fail, and considered as *healthy* if it is not predicted to fail, in the next interval.

An application, which is alloted a fixed N_{tot} number of nodes, starts its execution on a fixed N_{cmp} number of compute nodes, using the remaining N_{rep} ($= N_{tot} - N_{cmp}$) number of nodes as *replica* nodes. For simplicity of explanation, we assume that a node/processor is a computation unit and that only a single process executes on any given node/processor at a given point of time. Each application process either belongs to the *compute set* or to the *replica set*. The number of replicas nodes, N_{rep}, is chosen based on various factors, as explained in Section 5.3. We assume partial replication ($N_{rep} < N_{cmp}$), i.e., not all application processes will have an associated replica process. A *compute-replica map* is used by the processes to find the mapping between compute nodes and replica nodes. This can be a file or a global data structure which is accessible to all the processors in the system.

The overall working of our adaptive replication framework is illustrated in Figure 4.1. The goal of the framework is to ensure that at any given point of time, all failure prone nodes in the system have healthy replica nodes associated with them. The basic idea behind the fault tolerance strategy is that if all failure prone nodes have a corresponding healthy replica, then even if a failure prone node actually fails, the application is not interrupted due to the presence of a healthy replica.

At regular intervals of time, an *adaptation manager* determines the compute-replica map

for the next interval, based on the node failure predictions given by the failure predictor. Given the list of predicted node failures in the system, the adaptation manager selects a set of nodes for allocation of healthy replica for the next interval. A node that is predicted to fail is selected for this allocation if it is a compute node and it satisfies one of the following conditions.

- The node does not have a replica allotted to it.
- The node has a replica allotted to it, but the replica is also failure prone in the next interval.

For each node in the list of selected nodes for healthy replica node allocation, the adaptation manager then takes the following steps.

- Select a random healthy replica node, which is not currently associated with a failure prone compute node.
- Modify the compute-replica map, such that the selected replica node is now mapped to the given failure prone compute node. This step requires the modification of two entries in the map. Firstly, the current mapping of the replica node is removed, which means that the compute node to which the replica node is currently associated with will not have a replica any more. Secondly, a new mapping is added between the replica node and the given failure prone compute node.

Each of the application processes checks the compute-replica map regularly. If the map is modified the processes take the following steps.

- If the process belongs to the compute set and finds that it has been allotted a new replica node, it sends its process image to the new replica node. Note that a new replica node can be allotted to a process which either does not have a replica or has a different replica allotted to it currently. In both cases, the same action is taken by the process.
- If the process belongs to the replica set and finds that it has been allotted a new compute node, it receives the process image of the corresponding compute node and initiates a process replacement. Once the process replacement is completed, it acts as the replica of the newly alloted compute node and continues execution.

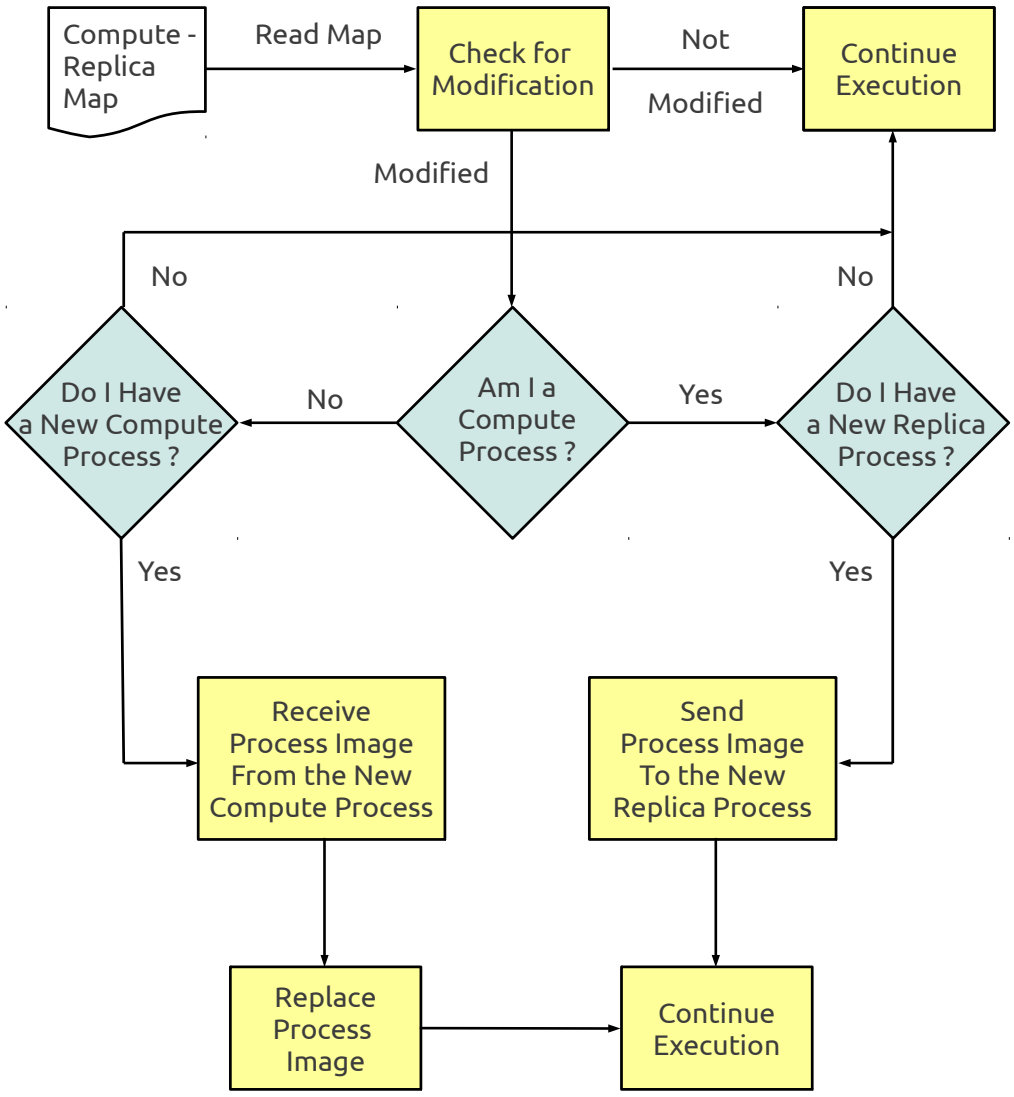


Figure 4.2: Control flow at each process for adaptive replica change

Figure 4.2 shows the flow of actions at each application process starting from the point when the process checks the compute-replica map for modifications and leading to a possible replica change and associated process image replacement.

Our framework can avoid all the node failures which are correctly predicted if the number of healthy nodes in the replica set at any given point of time is at least equal to the number of nodes predicted to fail in a given interval. While our framework can avoid a significant number of predicted node failures by using replication, it tolerates unpredicted node failures in the system by taking *precautionary checkpoints*. The checkpointing interval is calculated based on the interval between unpredicted failures, and using this interval in Daly's higher order checkpoint/restart model [16]. To determine the time interval between unpredicted failures, recall of the failure prediction is used. As mentioned earlier, the recall of the failure predictor gives the ratio of the number of predicted failures to the total number of failures in the system. The lower the recall of the predictor, the higher will be the number of unpredicted failures. Given the recall, the time interval between such unpredicted failures can be estimated as $\frac{platform\ MTBF}{1 - recall}$.

4.3 PAREP-MPI: A Prototype Implementation for Adaptive Process Replication

4.3.1 Basic Design

PAREP-MPI is our MPI implementation that supports partial process replication with adaptive changing of the replica set for MPI applications. It acts as a profiling layer between the application and the MPI library and allows users to specify a replication degree less than or equal to 2, where a degree of 2 corresponds to dual redundancy. The implementation of support for process replication and the MPI communication strategies used to enable redundancy in PAREP-MPI are similar to those of RepMPI [20]. The $Init\ ()$ call from the MPI application is intercepted by the PAREP-MPI layer and the application is started with a fixed number of nodes replicated according to the specified replication degree. MPI_COMM_WORLD is divided into two communicators, one for the processes in the compute set and the other for the processes in the replica set. All subsequent MPI library calls made by the application are intercepted by this

profiling layer, and presence of replica processes is handled, transparent to the application. For example, redundancy is handled by the PAREP-MPI layer for a $Send$ () call from the application by posting a redundant $Send$ () call for the replica process, if the destination has one, thereby sending the message to both the compute and the replica processes. Our current implementation supports the MPI functions $Comm\_rank$ (), $Comm\_size$ (), $Send$ (), $Isend$ (), $Recv$ (), $Irecv$ (), $Reduce$ (), $Allreduce$ (), $Alltoall$ (), $Alltoallv$ () and $Bcast$ ().

4.3.2 Adaptive Replica Change

The fault tolerance framework explained in Section 4.2 relies on the existence of a reliable, cost efficient mechanism for adaptively changing the processes in the replica set. An easy method is to simply checkpoint and restart the application with a different set of processes replicated. But the high cost of checkpoint/restart will overshadow the advantages of adaptive replica change, resulting in reduced application efficiency.

PAREP-MPI uses a strategy which involves replacing process image to change processes in the replica set. A process image is basically the information in the address space of the process that includes the state of the process and its data structures. Process image replacement is used in many process migration systems [14, 41, 57] in which an entire process image of a process is migrated to another process or a machine and continued execution. This complete process image replacement is not applicable in ParRep-MPI since replica processes are MPI processes and are part of the MPI environment (communicator) of the parallel application. Processes in an MPI environment will have their own identities which are defined by the MPI communicator structures. A process replacement strategy to change the replica processes should make sure that these identities are retained by the processes, while the state of the process and its data structures should change to reflect that of its newly allotted compute process. For example, every process in an MPI environment will have its own rank. A full process image replacement will result in the loss of this rank information of the process. This will result in a need to restart the entire MPI application incurring the associated overheads. ParRep-MPI performs partial process image replacement such that processes retain their identity in the MPI environment, while the code executed by the process changes dynamically.

We assume that there exists an adaptation manager and a failure predictor in the system. At

regular intervals of time, the adaptation manager is expected to determine the compute-replica map for the next interval based on failure predictions as described in Section 4.2 and write it into a file accessible to all application processes. The implementation of adaptive replica change in PAREP-MPI is independent of the implementation of the adaptation manager and the failure predictor. The only requirement is that whenever the compute-replica map changes, the new map should be available in a global compute-replica map file accessible to the processes.

Our current prototype implementation assumes that all the application processes execute the same code, except during point-to-point communication, in which case the sender will be executing $Send$ () or $Isend$ () and the receiver will be executing a $Recv$ () or $Irecv$ (). A significant number of parallel codes that perform domain decomposition follow this paradigm. We have considered two such large-scale applications in our experiments, namely, LAMMPS [30] and HPCCG [31]. LAMMPS [30] is a molecular dynamics simulation application used for particle simulations at the atomic, meso or continuum scale. It can run on both single processors and multiple processors in parallel using message-passing techniques and a spatial-decomposition of the simulation domain. For parallel execution, it runs the same code on different processors with each of the processors handling different sub-domains of the decomposed simulation domain. HPCCG [31] is a parallel application used for unstructured implicit finite element or finite volume computations. It is a linear system solver using the Conjugate Gradient (CG) method which can run on an arbitrary number of processors by decomposing its three dimensional problem domain into multiple sub-domains and different processors handling different sub-domains. NAMD [5] is another application developed for the simulation of large biomolecular systems, which follows the same paradigm. This application makes use of hybrid spatial and force decomposition technique and can scale to thousands of processors. The technique used by NAMD is also used in other molecular dynamics simulation applications including Blue Matter [22] and Desmond [8]. AMBER [58] is another similar molecular dynamics application. Cosmological simulation applications like ChaNGa [25], PkdGRAV [18] and FLASH [59] also fall under the same category of applications. ChaNGa performs domain decomposition techniques to solve N-body problems and also scales to thousands of processors. A number of other applications including weather modeling applications like CCSM [17] and computational fluid dynamics (CFD) simulation applications [45] are also suitable candidates

for fault tolerance using PAREP-MPI.

The control flow in PAREP-MPI for adaptive replica change is illustrated in the flowchart shown in Figure 4.2. On completion of every MPI routine call from the application, PAREP-MPI checks the compute-replica map file for modifications before the control is returned to the application routine. This is done in the MPI profiling layer by calling a special routine at the end of every MPI function call from the application. If the file is modified, the application processes first synchronize to determine whether there are pending MPI requests in any of the processes. A pending MPI request refers to non-blocking communication operations that have not completed yet. In such a case, the processes will have to wait for the requests to be completed before proceeding further. This is to ensure that there are no in-flight messages when a replica change is triggered. If there are in-flight messages while a process image replacement happens, this can result in the failure of the message delivery if the message is to be delivered to the process whose image is being replaced. PAREP-MPI keeps track of MPI request handles in an internal data structure in each of the processes. Whenever an asynchronous MPI communication is triggered which involves an MPI request handle, the handle is stored in a data structure. Once the request is completed, the corresponding handle is removed from the data structure. When an process image replacement has to take place and there are pending MPI requests (i.e., the data structure holding MPI request handles is non-empty), a $Wait\ ()$ call is posted on existing handles to complete the requests. After this step, each of the processes whose mapping has changed initiates the send/receive of process image to/from its newly mapped process, and the receiving process performs process image replacement.

Assuming that each process executes the same code, all processes will notice any change in the compute-replica map file almost simultaneously. Thus a source process sends its process image and at the same time the destination process receives this image for replacement of its own image. Process image replacement which is the most significant part of PAREP-MPI is explained in detail in Section 4.4.

4.4 Process Image Replacement in PAREP-MPI

The strategy used in PAREP-MPI for replacing the image of a process with that of another process involves three steps - replacing the initialized and uninitialized data (henceforth referred to as data segment), replacing the user data structures in the heap (henceforth referred to as heap data structures) and replacing the stack segment. It is also important that after the image replacement is complete, the new replica process continues from exactly the same instruction the corresponding compute process executes just after sending its process image. Hence the *stack context* should also be saved.

Saving and restoring the stack context is implemented similar to Condor Checkpoint/Migration [14], using standard C functions $setjmp$ () and $longjmp$ (). The function $setjmp$, when called with a pointer to a variable of jmp_buf data type (henceforth referred to as JMP_BUF), saves the stack context into JMP_BUF and returns 0. Subsequently, when $longjmp$ is called with this same variable and a non-zero number n, the stack context is restored and it returns to the point were $setjmp$ had returned to, but this time the return value will be n instead of 0. Replacing the process image in a different node and resuming application execution relies on the fact that after a $setjmp$ call, JMP_BUF will be in the data segment of the process, and will be accessible at the node where the application process has to be resumed after the data segment replacement is complete. A $longjmp$ can then be called on JMP_BUF to restore the stack context and hence resume the application process.

Subsections 4.4.1, 4.4.2 and 4.4.3 describe how each of the segments in the address space of a destination process are replaced by the corresponding segments from a source process for a replica change. Subsection 4.4.4 gives an illustration of the actions involved in process image replace with a simple example.

4.4.1 Data Segment

Replacing the data segment is trivial if the start and end addresses of this segment are known. In Unix/Linux, the boundary addresses of this segment can be obtained from $/proc/self/maps$ file while the application is executing. Replacing this segment in the destination process involves replacing the entire data between the start and the end addresses.

MPI communicator handles have to be backed up in the stack before initiating a data segment replacement and should be restored back after replacing the data segment. This is done so that the process does not lose access to its communicator structures and hence retain its identity in the MPI environment, such as the process rank and other MPI specific information.

4.4.2 Heap Data Structures

Many of the MPI related data structures are stored in the heap space of a process. A complete heap replacement will result in the replacement of these MPI data structures. Since we require the process to keep its identity, these data structures have to be retained. However, keeping track of the locations of all the MPI data structures would require the modification of the standard MPI source code. We adopt a different strategy where we keep track of only the user data structures in the heap and selectively replace them in the destination process. Our assumption that all the application processes execute the same set of instructions also implies that all the processes will have the same set of data structures. Hence, for each of the heap data structures in the source process, there will be a corresponding heap data structure in the destination process. But since these data structures are in the heap, we cannot assume that they are in the same locations in both the processes. Hence, heap data structures from the source process are copied to the corresponding locations of the same data structures in the destination process.

The above strategy to handle heap data structures gives rise to another problem. The pointers to the heap data structures reside in the data segment. After the data segment is replaced, these pointers would be pointing to addresses corresponding to the locations of these data structures in the source process. Since these locations may have changed in the destination process, the pointers will have to be updated. For this, it is necessary to keep track of the addresses of these pointers in the data segment along with keeping track of the data structures itself. For example, suppose a user process allocates memory in the heap for an array of 10 integers using the following C statement.

$$int\ *p\ =\ (int*)\,malloc\,(10\ *\ sizeof\ (int));$$

To implement the selective heap data structure replacement, for each data structure, we

should have the information about the actual data structure, i.e., its address in the heap and its total size in both the source and destination processes. Apart from this, we should also have the address of the pointer p in the data segment. Data structure information can be obtained by using wrappers for $malloc$ () and $free$ () to keep track of memory allotted and freed from the heap. To enable keeping track of the addresses of the pointers, we use a script that does a text replace of all the $malloc$ statements in the application source code with custom $malloc$ statements that will also pass the address of the pointer variable along with the size parameter of $malloc$. For e.g., the above $malloc$ statement in a C program will be replaced by the script as follows.

$$int\ *p\ =\ (int*)\ myMalloc\ (\&p,\ 10\ *\ sizeof\ (int));$$

Here, $myMalloc$ () is the custom $malloc$ implementation that internally keeps track of the required heap data structure information in a linked list and calls the real $malloc$ function for dynamic memory allocation.

4.4.3 Stack Segment

In Unix/Linux, the starting address of the stack can be obtained from the $/proc/self/maps$ file. The ending address of the stack is basically the stack pointer. This can be obtained by using a $setjmp$ call to save the stack context to JMP_BUF and taking the stack pointer out of it.

Replacing the stack segment cannot be done by simply copying the contents of the stack from the source process to the stack space of the destination process. This would result in the stack which is used by the current function doing the process image replacement itself to get replaced and the process will not be able to continue further. Hence, before copying the stack segment, the current execution stack at the destination process should be shifted to another location and the stack pointer should be made to point to this location. This is done by allocating a fixed amount of space in the data segment to act as the temporary stack while the real stack space is getting replaced. To use the temporary stack in the data segment, first a $setjmp$ call is made to save the stack context to JMP_BUF, then the stack pointer in JMP_BUF is made to point to the temporary stack, and finally a $longjmp$ call is made on JMP_BUF.

4.4.4 Illustration of Process Image Replacement: A Scenario with 3 Nodes and 3 Processes

Consider a system with 3 nodes and each node has an application process running on it. Suppose A, B' and B are the 3 processes in the system, such that B' is the replica process of B and A does not have a replica. Now, suppose that at a given point of time, the compute-replica map of the system is modified and according to the new map, the process B' should be a replica of A and B no longer should have a replica, i.e, B' should be changed to A'. Figure 4.3 shows the execution time line of the 3 processes starting from the point where the processes see the compute-replica map modification, to the point where B' has changed to A' and has resumed execution, i.e., the replica change is in effect. Following are the steps taken by the processes during this time line.

- All processes synchronize to check for pending non-blocking communication requests in any of the processes. If there are pending requests, processes proceed further only after they are completed.
- A calls $setjmp$ (JMP_BUF) in an if condition that checks for a return value of 0. $setjmp$ returns 0 in A and hence executes the instructions in the if block (as follows in the remaining steps until $longjmp$ is called). A saves the start and end addresses of its stack segment in a global data structure, $stackSegAddr$.
- A sends the start address and size of its data segment to B'. B' receives them and takes a backup of MPI communicator handles in the stack.
- A sends its entire data segment. B' receives the data segment at the previously received start address. It restores the backed up data to the new data segment. This completes the data segment replacement.
- A sends the heap data structures one by one to B'. Heap data structures' information is stored in a link list in the heap. Since the data structures are assumed to be same in all the processes, B' uses the information in its own heap data structures' list to receive them one by one, at their corresponding locations in the heap. This completes the replacement of the heap data structures.

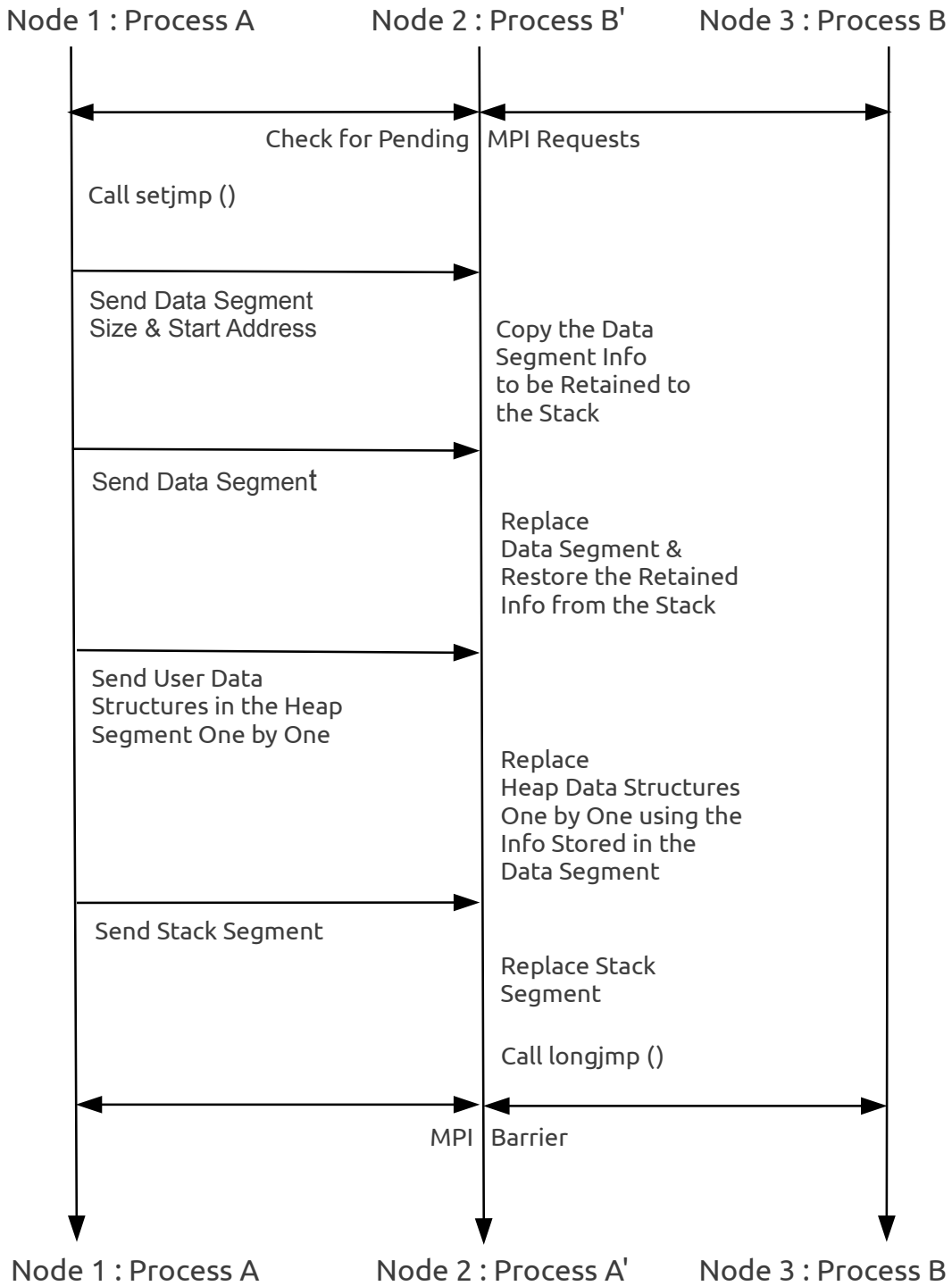


Figure 4.3: Application execution time line during process replacement

- A sends its stack segment to B'. Note that once the data segment is replaced, B' has the global data structure, $stackSegAddr$ containing the start and end addresses of the stack segment of A. B' shifts its execution to a temporary stack in its data segment and receives the stack segment from A at the stack start address in $stackSegInfo$.

- B' calls $longjmp$ (JMP_BUF, n), where n is a non-zero integer. This restores the stack context of A in B'. Now B' has become A', i.e., the replica of A. The process resumes at the point where $setjmp$ had returned when it was first called in A, with the return value n. The process continues execution from the next instruction after the if block. This completes the entire process image replacement.

- All processes synchronize at a barrier and continues execution from there.

Our current implementation of process replacement does not support *Address Space Layout Randomization (ASLR)* [3] feature present in current versions of the Linux kernel. It also does not support the *Pointer Encryption/Pointer Guard* [49] feature present in the current versions of glibc. Both these features should be turned off for the correct working of our current prototype implementation.

Chapter 5

Experiments and Results

5.1 Application Execution and Failure Simulator

For evaluating our fault tolerance strategies, we have developed a robust discrete-event failure simulator that simulates application failures of a failure trace for a set of nodes, with fault tolerance techniques implemented at different adaptation points, and application execution in the presence of failures. Simulation application execution with the traditional periodic checkpointing approach is supported with a fixed checkpoint interval or with an interval calculated using Daly's higher order checkpoint/restart model [16]. It also supports simulations with full/partial replication, with the option to specify the degree of redundancy.

If redundancy is enabled, the application execution simulation is done with a subset of nodes as active nodes and the rest as replica nodes, each mapped to any one of the active nodes. The number of active nodes and replica nodes depend on the degree of redundancy specified by the user. The simulator keeps track of whether a node has a replica or not and the state of the replica node, if it has one. An application failure will occur as a result of a node failure only if either the node does not have a replica or if its replica also fails simultaneously.

The simulator takes as input the failure trace of a system with a given number of nodes, accuracy metrics of the predictor, type of fault tolerance to be adopted at different adaptation points and other data including cost of each of the fault tolerance operations and estimated MTBF of the system. It simulates application execution, tracking the behavior of the application while introducing failures based on the given failure trace and taking proactive/reactive fault tolerance

actions including checkpointing, proactive migration, rescheduling and adaptive replica change based on the type of fault tolerance technique adopted. The scalability data of the application can also be given as an input to the simulator, in which case it simulates the execution based on the given scalability. It also has the option to provide different kinds of scalability curves based on the popular *Downey model* [19] for speed-up of parallel applications. Recovery action taken after a failure will depend on the malleability of the application. If the application is set as malleable, it has the option to reschedule to a different number of processors and recover from a failure instantly. Otherwise, the application can recover only when enough nodes become available.

The simulator gives as output the work done per unit time by the application and its efficiency in the presence of failures, and also other details of the simulation including number of various fault tolerance actions taken and number of times the application failed. Our simulator can consider both real and synthetic failure traces. For synthetic failure traces, the simulator has a trace generation component that can generate failures of different distributions including Weibull and exponential distributions and simulate repairs of log-normal distribution.

Proactive fault tolerance techniques need the presence of a failure predictor for performing proactive actions based on predictions. The failure prediction component in our simulator simulates the behavior of such a predictor. At each adaptation point, it estimates the list of nodes that might fail in the next time interval with the given accuracy metrics (precision and recall). For each failure entry in the given failure trace the prediction component adds the entry to the list of predicted failures with a probability equal to the recall of the predictor. For each interval, the predictor also adds a number of incorrect predictions to the list so as to maintain the precision of the predictor.

5.2 Evaluation of ADFT

We evaluate the adaptive strategies of ADFT using total work done in unit time by the application in the presence of failures. This is determined by simulating the application execution using our failure simulator. Failure trace, application scalability data and accuracy metrics of the failure predictor are given as input to the simulator. ADFT is evaluated against *FT-Pro* and

periodic checkpointing with checkpointing interval that gives maximum performace.

For fair comparison, we have extended *FT-Pro* to consider live-migration and scalability of application. *FT-Pro* was modified to consider the scalability of applications to decide what number of nodes out of the available nodes should be used for execution. This is helpful especially in case of applications whose scalability decreases on increasing the number of processors beyond a threshold. *FT-Pro* takes a precautionary checkpoint when the number of consecutive *SKIP* decisions reaches a threshold. This is based on the assumption that *SKIP* decision is the only one that does not involve checkpointing. Since we also consider live-migration that does not involve checkpointing, we have modified *FT-Pro* such that it takes a precautionary checkpoint whenever the time since last checkpoint reaches a threshold. This makes the precautionary checkpointing strategy of *FT-Pro* similar to that of ADFT.

FT-Pro requires allocation of constant number of spare nodes so that the application can be executed on the constant remaining number of nodes in the system till completion. In our analysis, we have found that the number of spare nodes to be allotted for *FT-Pro* to give maximum performance can vary based on various factors. For the purpose of evaluation, we allocate the number of spare nodes equal to the average of the number of failed nodes (or nodes that were down) at any point of time in the failure history before the time when the simulated application starts execution. This is to make sure that there are enough spare nodes to exercise the option of process migration while avoiding high spare node allocation to reduce the amount of idling in the system.

For each of our experiments, we set W, which is the constant work to be completed between each AP, as the work done by the application in 30 minutes in a failure free environment on the given number of nodes. This is based on the results in previous efforts on failure predictors [27, 42] that report the best accuracy metrics for a time window between 15 minutes to 1 hour depending on the system. MTBF of the system, which is used for precautionary checkpointing is taken as the observed MTBF from the trace history. We also assume the following costs for the various fault tolerance actions: T_{ckp} : 5 minutes, T_{mig} : 0.33 minutes, T_{resch} : 3 minutes, $T_{recover}$: 5 minutes. These are in accordance with the values given for the 2011 cost scenario in [12]. While in reality these overheads vary depending on the number of processors, failures and pattern of failures, we have used an average to worst case overhead values for our experiments.

Most of the literature on failure predictions and proactive fault tolerance based on failure predictions are independent of each other. Hence, little is known about the overheads caused by failure prediction mechanisms on applications running on the system. We assume this overhead to be negligible considering that a failure predictor can run in the background, independent of the application running on the system and without significantly affecting its performance. For example, the failure prediction mechanism developed by Gujrati et al. [27] analyzes failure traces to learn about failure patterns initially and makes predictions dynamically based on events in the system. While the initial processing of failure traces can take some time, this is a one time effort and the overhead of making predictions while the application is executing and dynamically updating its knowledge about failure patterns can happen in the background with minimal overhead on the application. Hence we did not consider these prediction overheads in our experiments.

Real traces from LANL [23] corresponding to system 20 which is a 512-node system and system 18, which is a 1024-node system are used for simulations of small and medium scale systems. Synthetic traces are used for simulations of very large scale systems for which real failure traces are not available. We generate synthetic traces for a given number of nodes based on the observation in [52] that the times to failure of nodes in a system follows a weibull probability distribution and the times to recover follow a log-normal probability distribution. The synthetic trace generation component of our simulator takes as input the number of nodes, required time period for which traces have to be generated, mean node time to failure and mean node time to recover, and generates a failure trace with the above mentioned probability distributions. Most of the experiments are done on real traces for 1024 nodes from LANL. We have used traces that show low MTBF of only a few hours to represent the failure pattern in large-scale systems.

Simulations are done for a period of 30 days and evaluation is based on the work done by the application in unit time. For the LANL traces, a random year is chosen from the trace of a system for simulation. Application execution is simulated for the last month of the one year trace. Observed MTBF of the system and the number of spare nodes to be allotted for *FT-Pro* is obtained using the trace history of the previous eleven months. A similar strategy is also adopted for synthetic traces, where a trace is generated for one year and simulations are

Table 5.1: Comparison of ADFT with other techniques

$No.of Nodes$	$WorkDone/s_{(AdFT)}$	$\%Gain_{FTPro}$	$\%Gain_{per.ckp}$
512 (LANL)	481.96	23.22	156.38
1024 (LANL)	642.33	8.70	77.57
16384 (Syn.)	10038.62	15.16	87.27

performed for the last month of the year.

5.2.1 Performance on Small and Medium Scale Systems

Simulations on 512 and 1024 nodes are done using real failure traces from LANL. While the primary objective of our work is on fault tolerance for large-scale systems, we present these results on small and medium-scale systems primarily to validate our framework on real systems and traces. A synthetic trace is generated for 16384 nodes with an MTBF of approximately 10 hours. We assume a synthetic application with linear and perfect scalability, such that the work done per unit time by the application on N nodes in a failure free environment is N units. Precision and recall of the predictor are assumed to be 0.7.

Table 5.1 shows the work done per second by ADFT ($WD/s_{(AdFT)}$) and the percentage improvement in work done by ADFT compared to *FT-Pro* and periodic checkpointing ($\%Gain_{FTPro}$ and $\%Gain_{per.ckp}$, respectively). The result shows that ADFT gives 8-23% improvement over *FT-Pro* and more than 87% improvement over the traditional periodic checkpointing method. Periodic checkpointing shows very low performance since it is a static strategy that does not consider failure predictions. The number of checkpoints taken will be huge for periodic checkpointing and the high cost of checkpoint operation impacts the application performance negatively. In order to analyze the reasons for the high performance with ADFT, we also show the number of various fault tolerance actions taken by ADFT during the application execution period in Table 5.2. In the table, $N_{p.resch}$, $N_{r.resch}$ and $N_{pre.ckp}$ correspond to proactive rescheduling, reactive rescheduling and precautionary checkpointing respectively.

We find that most of the fault tolerance actions are migrations since unlike other actions, live-migration does not involve checkpointing, and incurs much lesser cost (0.33 minutes in our experiments) than the others. We find that a significant percentage of fault tolerance actions are related to rescheduling ($N_{p.resch} + N_{r.resch}$). This shows that rescheduling is a significant

Table 5.2: Fault tolerance actions by ADFT

$No.of Nodes$	$MTBF(hrs)$	N_{skip}	N_{ckp}	N_{mig}	$N_{p.resch}$	$N_{pre.ckp}$	$N_{r.resch}$
512 (LANL)	23.95	0	0	16	3	5	4
1024 (LANL)	5.31	0	0	101	24	17	24
16384 (Syn.)	10.86	0	0	64	7	7	11

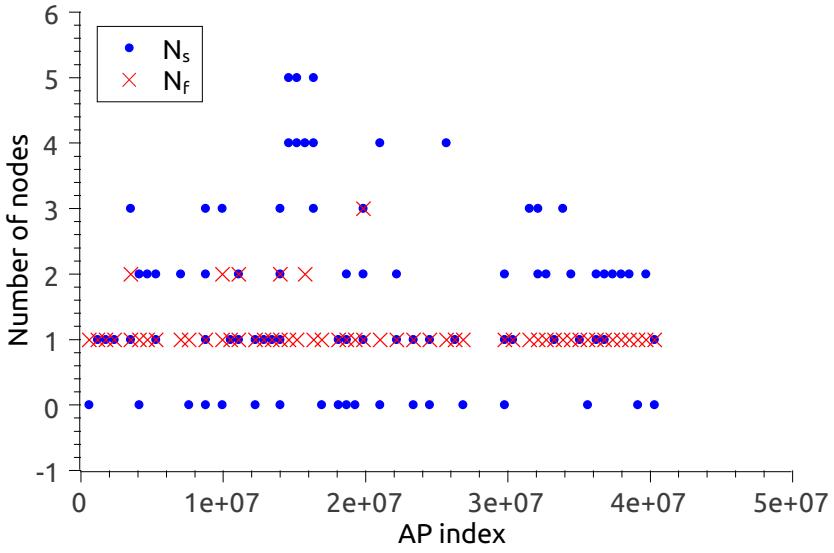


Figure 5.1: Number of spares vs Number of failure predictions for ADFT

contributing factor to performance gains shown by our ADFT method over the other two methods. It can also be observed that the number of rescheduling decisions increases as the MTBF decreases. This is because the number of failures in the system increases as MTBF decreases, resulting in the need for more proactive rescheduling decisions. Number of reactive rescheduling decisions also increase due to the increase in the number of unanticipated failures. Hence rescheduling plays an important role in ADFT in adapting to a large scale failure environment with high failure dynamics or low MTBFs.

5.2.2 Spares vs Failure Predictions

We also found that rescheduling also contributes to increase in application performance in an indirect way. Rescheduling to smaller number of nodes results in the presence of sufficient number of spare nodes most of the time during application execution. Figure 5.1 shows the variations in the number of spare nodes and predicted failures with ADFT for the 1024 nodes

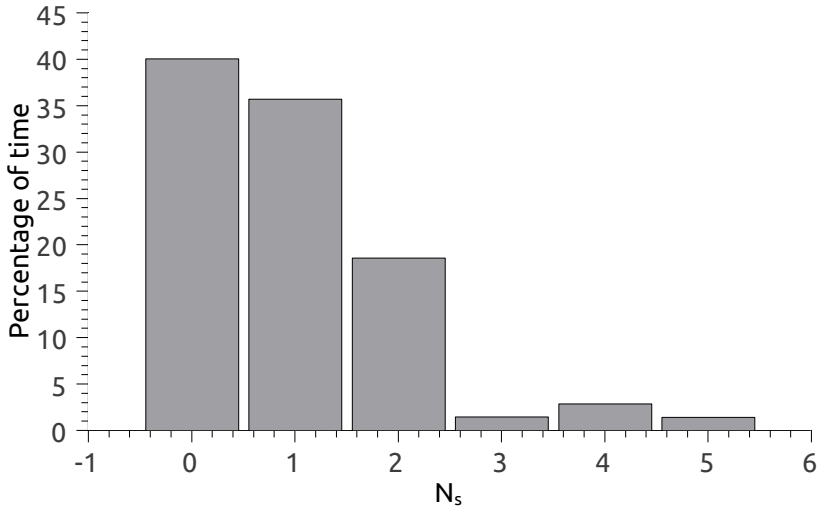


Figure 5.2: Analysis of number of idle nodes for ADFT

(LANL) at each adaptation point where failures are predicted. It can be seen that in majority of the cases, the number of spare nodes in the system is greater than or equal to the number of predicted failures. This helps increase the number of low-cost migration decisions to avoid failures in ADFT. Thus ADFT dynamically maintains sufficient number of spare nodes to favor migration most of the time.

5.2.3 Resource Utilization

In addition to increasing application performance, rescheduling also helps in utilizing spare nodes in the system, that become available after recovering from a failure, for application execution. Whenever a proactive or reactive rescheduling is performed, ADFT tries to accommodate spare nodes in the system which are not predicted to fail in the near future. This shrinking and expanding of the number of nodes by ADFT keeps the number of idle nodes in the system at any given time to a minimum.

Figure 5.2 shows the percentages of time during application execution for different number of idle nodes in the system for 1024 nodes (LANL). We can observe that for up to 40% of the time there are no idle nodes in the system and for about 94% of the time the number of idle nodes is less than or equal to 2. A similar analysis on *FT-Pro* showed that 99.99% of the time,

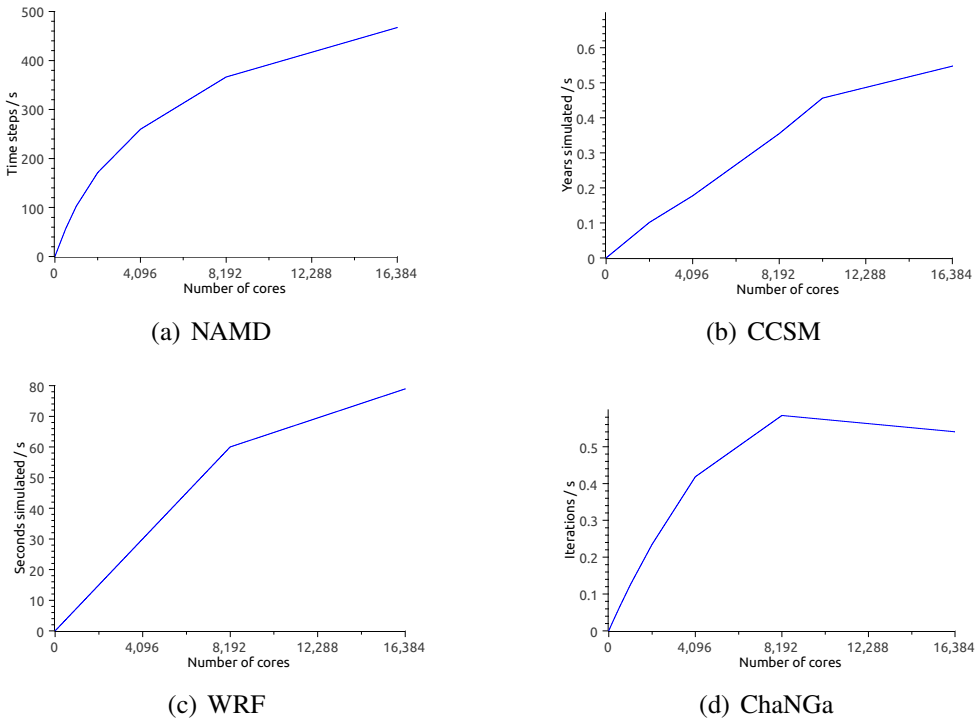
Table 5.3: Varying precision and recall for ADFT

$Precision$	$Recall$	$WorkDone/s$	$\%Gain_{FTPro}$	$\%Gain_{per.ckp}$
1.0	1.0	1017.82	0.31	181.376
0.8	1.0	1017.55	0.32	181.301
0.6	1.0	1016.62	0.35	181.044
0.4	1.0	1015.69	0.31	180.787
0.2	1.0	1012.73	0.65	179.968
1.0	0.8	664.04	0.56	83.5734
0.8	0.8	673.17	6.19	86.0974
0.6	0.8	595.35	17.31	64.5841
0.4	0.8	586.65	4.94	62.179
0.2	0.8	671.20	2.85	85.5528
1.0	0.6	639.00	5.20	76.6511
0.8	0.6	648.26	3.24	79.211
0.6	0.6	652.80	4.27	88.5108
0.4	0.6	651.55	6.60	80.1205
0.2	0.6	667.56	0.16	84.5465
1.0	0.4	701.71	1.21	96.3675
0.8	0.4	680.60	0.11	88.1514
0.6	0.4	681.91	2.06	88.5135
0.4	0.4	633.51	1.56	77.8647
0.2	0.4	741.26	1.48	104.921
1.0	0.2	709.24	-1.68	96.0689
0.8	0.2	686.11	-1.30	89.6746
0.6	0.2	683.03	-3.71	88.8232
0.4	0.2	676.96	-2.94	87.1451
0.2	0.2	712.71	-1.26	97.0282

the number of idle nodes in the system is 3, which is the allotted number of spare nodes. This shows the effectiveness of ADFT in dynamically adapting to failures while keeping the number of idle nodes to a minimum.

5.2.4 Accuracy of Failure Predictions

Table 5.3 shows the performance of ADFT for different precision and recall values of the predictor for 1024 nodes (LANL). It can be observed that in all the cases, ADFT outperforms periodic checkpointing by a huge margin. It also outperforms *FT-Pro* in most of the cases. It is seen that *FT-Pro* performs better when the recall is as low as 0.2 or below. This is due to the



(a) NAMD (b) CCSM

(c) WRF (d) ChaNGa

Figure 5.3: Scalability of different real applications

large number of unforeseen failures, which results in a large number of reactive rescheduling. Since rescheduling involving checkpointing incurs higher cost than only checkpointing, *FT-Pro* that does not perform rescheduling, gives slightly better performance than ADFT for low recall values. It is also observed that ADFT gives the most improvement over *FT-Pro* for *recall* values between 0.6-0.9. Most failure predictors today have a recall value of more than 0.6, and hence ADFT performs better than *FT-Pro* for practical purposes.

5.2.5 Real Applications

Simulations were also done using scalability data of four real life applications as observed in real large scale systems. The simulations correspond to application execution on 16384 nodes using synthetic failure trace. The application scalability curves used are that of NAMD [5], CCSM [17], WRF [40] and ChaNGa [25]. Figure 5.3 shows the scalability curves of each of these applications as observed in BlueGene/L and their corresponding units of work done.

Table 5.4 shows the performance gain due to ADFT. As shown in the table, ADFT performs much better than periodic checkpointing for all the applications and gives about 15% better

Table 5.4: ADFT **vs other techniques for real applications**

$Application$	$WorkDone/s_{(AdFT)}$	$\%Gain_{FTPro}$	$\%Gain_{per.ckp}$
NAMD	286.32	15.15	87.28
CCSM	0.34	15.13	86.37
WRF	48.42	15.26	87.47
ChaNGa	0.461	-0.21	21.22

performance than *FT-Pro* for NAMD, CCSM and WRF. *FT-Pro* shows a small performance increase over ADFT in case of ChaNGa. This can be attributed to the modification done to *FT-Pro* that allows it to start the application on the number of nodes which gives maximum performance. It can be observed from the scalability curve of ChaNGa in Figure 5.3 that the application performance decreases after 8192 nodes. Hence, the modified *FT-Pro* technique will start the application only on 8192 nodes, leaving a large number of spare nodes. This allows it to perform low-cost live migrations at all adaptation points, resulting in performance improvement over ADFT by a small margin.

5.2.6 Petascale and Exascale Systems

We also evaluated our ADFT framework on current large scale and future very-large scale systems. Simulation of NAMD is done on a hypothetical petascale system with 2^{17} nodes and a hypothetical exascale system with 2^{23} nodes. For the petascale system, each node is assumed to have approximately 7.6 GF/s peak performance. For the exascale system, we assume that each node is quad-core, so that the total number of processors is 2^{25}. Each processor is assumed to have a peak performance of approximately 29.8 GF/s. The per processor peak performance is approximated assuming that the approximate ratio of the average processor peak performance of an exascale system to that of a petascale system will be approximately equal to a similar ratio between a petascale system and a terascale system.

The scalability curve for NAMD was obtained from a study on BlueGene/L system [5] with each node having approximately 2.7 GF/s peak performance. Approximate scalability curve for the petascale system was generated based on the assumption that the work done by a node in the hypothetical petascale system will be approximately equal to the work done by 3 nodes of BlueGene/L. Scalability curve for the exascale system is generated in a similar way.

Table 5.5: ADFT **vs other techniques for a petascale system**

$No.of nodes$	$WorkDone/s_{(AdFT)}$	$\%Gain_{FTPro}$	$\%Gain_{per.ckp}$
2^{17}	802.96	11.21	145.79

Table 5.6: ADFT **vs other techniques for an exascale system**

$No.of nodes$	$WorkDone/s_{(AdFT)}$	$\%Gain_{FTPro}$	$\%Gain_{per.ckp}$
2^{23}	3065.11	12.5	21

We generated synthetic traces with MTBF of approximately 4 hours for petascale system as reported in [11, 34] and 35 minutes for exascale system, as reported in [34].

Tables 5.5 and 5.6 show the work done per unit time and the corresponding percentage gains compared to other methods for petascale and exascale respectively. For petascale system, ADFT outperforms *FT-Pro* by about 11% and periodic checkpointing by about 145%. For exascale system, ADFT outperforms *FT-Pro* by about 12.5% and periodic checkpointing by about 21%. For petascale system, ADFT performs 114 migrations, which is the same as that of the number of migrations done by *FT-Pro*. For exascale system, ADFT performed 406 migrations whereas *FT-Pro* performs only 300 migrations. This again shows the effectiveness of ADFT in dynamically maintaining enough spare nodes to maximize migrations, which is not possible using *FT-Pro* with static spare node allocation. The number of rescheduling decisions increased from 51 in petascale to 364 in exascale, showing that rescheduling plays a significant role in the performance of ADFT. It also shows that with increasing size of the systems and decreasing MTBF, application malleability and rescheduling can play a very important role in developing better fault tolerance strategies, and that application malleability will be highly essential for future exascale systems. Hence, we find that our ADFT framework is highly applicable and provides good performance for real scientific applications in the presence of failures for petascale systems and beyond.

5.3 Evaluation of Adaptive Process Replication

We evaluate our adaptive replication framework based on the efficiency achieved by the application in the presence of failures. Efficiency is defined as the percentage of work W_{opt} that the application completes in the presence of failures in a given time duration, where W_{opt} is the

work done by the application in the time duration in a failure free environment. It is assumed that the application has linear scalability. Evaluation is done based on simulations using a failure simulator, which takes as input the failure trace of the system and accuracy metrics of the failure predictor. All experiments were done for an application execution time of one week.

Evaluation is done for both real and synthetic failure traces. For real traces, we use the well-known failure traces from LANL [23] corresponding to system 20 which is a 512-node system and system 18, which is a 1024-node system. Experiments with synthetic failure traces for large scale systems were done for both exponential and Weibull failure distributions. A projection based on failure statistics in the Jaguar supercomputer of Oak Ridge National Laboratory with $45,208$ processors has shown that the per processor MTBF in the platform can be estimated as approximately 125 years [6]. For our experiments, we consider a per processor MTBF of 25 years. The parameter λ for exponential distribution is then calculated as $\lambda = \frac{1}{MTBF}$. For Weibull distribution which is defined by two parameters λ and k, k is set as 0.7 based on the study in [52] and λ is calculated as $\lambda = MTBF/\Gamma(1 + \frac{1}{k})$, where Γ denotes the gamma function. Synthetic failure traces were generated for a period of 1 year and for each of the generated traces, the subset of the trace corresponding to the 1^{st} week of the 6^{th} month was taken for our experiments involving the simulation of application execution for 1 week.

Our framework divides the set of nodes allocated for an application into two mutually exclusive sets, namely, compute set and replica set. In our analysis, we found that the maximum number of node failures experienced by a system with number of nodes as high as $200,000$ is only less than 400 (refer Figure 5.6) over a one week period. Based on this analysis, we fix the number of nodes in the replica set as 1% of the total number of nodes. This assumption would result in 2000 nodes in the replica set for a $200,000$ node system, which is much higher than the observed number of node failures. 400 node failures in a week implies an average of only $1-2$ failures per interval assuming a 30 minute interval between adaptation points. Even if the number of actual failures per interval is as high as 10, the precision of the predictor should be as low as 0.005 for generating 2000 node failure predictions in an interval. Hence, the probability that the number of healthy spare nodes in the system is always more than the number of failure predictions in a given interval is very high. Also, a 1% reduction in the number of nodes used for application execution will only result in a maximum of 1% reduction in efficiency.

We have evaluated our framework against periodic checkpointing and dual redundancy. We have choosen periodic checkpointing for comparison because it is still the most popular and widely used fault tolerance strategy in parallel systems. Recent works have advocated the use of dual redundancy for fault tolerance in very large scale systems with large number of failures [21], and hence we have evaluated our strategy against dual redundancy. We have also compared our adaptive replication framework with proactive live process migration, another approach that uses process image replacement and failure predictions. For periodic checkpointing, the optimal checkpoint interval is calculated using Daly's higher order checkpoint/restart model [16], which takes as input, the platform MTBF and the overhead of checkpointing. Platform MTBF required by this model is determined as the average observed platform MTBF during a 1 month failure trace history before the start of application execution. Dual redundancy is implemented as given in [21]. Our adaptive framework can avoid all the predicted node failures in the system. But there can be unpredicted failures too, depending on the recall of the predictor. We use periodic checkpointing to tolerate such faults as explained in Section 4.2.

Evaluations on large scale systems were done for number of nodes ranging from $10,000$ to $200,000$. A system with $100,000$ nodes can be considered as representative of the existing petascale systems [53]. Similarly, a system with $200,000$ nodes can be considered as representative of a projected exascale system based on exascale computing studies [34].

For the purpose of our evaluation, we assume the following overheads for checkpoint/restart: checkpoint time = 5 minutes, down time = 1 minute, recover time = 5 minutes. These values are in accordance with the values given for the 2011 cost scenario in [12]. We set the time interval between failure predictions as 30 minutes. This selection is based on the results in previous efforts on failure predictors [27] that report the best accuracy metrics for a time window between 15 minutes to 1 hour depending on the system. In our experiments, unless otherwise mentioned, both precision and recall of the failure predictor are assumed as 0.7.

5.3.1 Runtime Overhead of PAREP-MPI

Overhead due to Replication

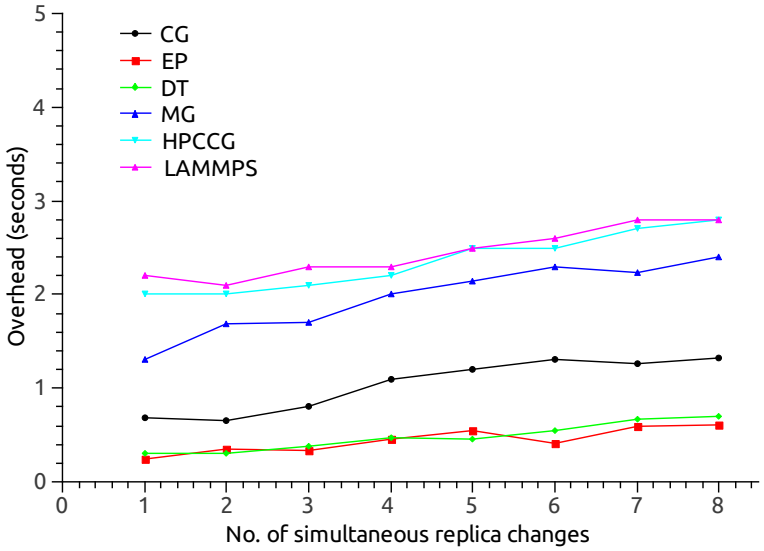
Evaluations using *r*MPI have concluded that with dual redundancy the worst-case overhead due to replication for the application SAGE on a projected exascale system is 4.9% [21]. Based on this, we assume a worst-case scenario with 4.9% of the work done by the replicated nodes in a failure free environment as the overhead due to replication for any number of nodes, in all our experiments.

Overhead of Adaptive Replica Change

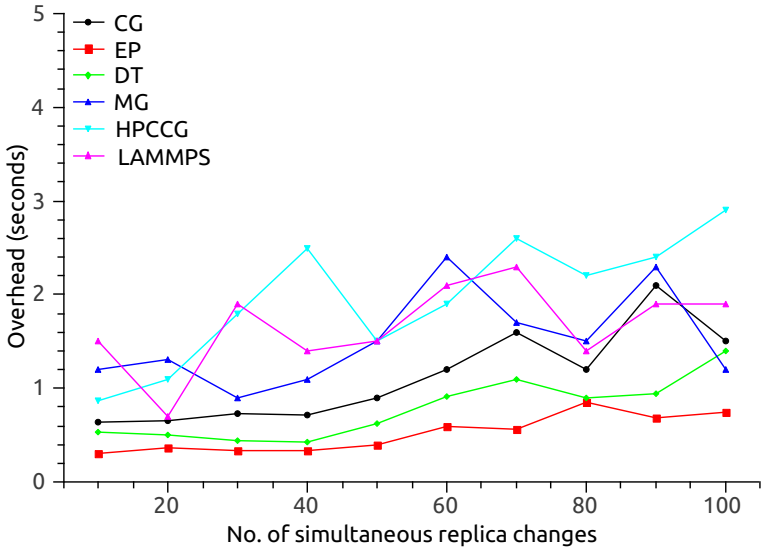
Overhead of adaptive replica change will depend on the memory footprint of the application. The more the memory used by a single process, the higher the process replacement overhead. This is because for a process replacement, most of the data from the process address space has to be transfered from the source process to the destination process. In adaptive replication, modification of the compute-replica map at a given time can result in the need for multiple replica changes. PAREP-MPI can handle these multiple process replacements in parallel since a process replacement involves only the source process and the destination process. Hence the overhead due to single process replacements is the same as the overhead due to multiple process replacements.

For the overhead analysis, we have taken four applications from NAS Parallel Benchmarks (NPB 3.3) [43], namely, CG, EP, DT and MG and two other scientific applications, namely, HPCCG (version 0.5) [31] for unstructured implicit finite element or finite volume computations and LAMMPS [30] for molecular dynamics simulations. All the NPB applications were executed with Class C data. HPCCG was executed with a problem dimension of 64x64x64 grid points and LAMMPS was executed with the Rhodo spin protein benchmark from the LAMMPS website.

The replica change overheads were analyzed on two types of settings involving an IBM Bluegene/L system running on Linux. In the first setting, the applications were executed on 16 processors. An extra 8 processors were used for replication, making the total number of processors used for execution 24. In the second setting, the applications were executed on



(a) Overheads on 16 processors



(b) Overheads on 1024 processors

Figure 5.4: Overhead of dynamic replica changes using PAREP-MPI on IBM Bluegene/L

1024 processors. An extra 100 processors were used for replication, making the total number of processors used for execution 1124. Thus the analysis was conducted on both small and large number of processors. The evaluation was done on 16 processors because for a given data set, a smaller number of processors would result in a higher memory footprint per process, and thereby help indicate the worst-case overhead. The evaluation on 1024 processors was to analyze the possible network contentions under large number of simultaneous process replacements.

Runtime overhead was observed for applications for increasing number of simultaneous replica changes, ranging from 1 to 16 in the first setting and 1 to 100 in the second setting. The 16 and 100 simultaneous replica changes, in the first and second settings, respectively, correspond to the cases where all the processes in the replica set have to be changed. Figure 5.4 shows the results of the experiments. Each value in the graph is the average of values obtained from five runs. It can be observed that for all the applications, the overhead is a few seconds, with a worst-case of only less than 3 seconds, for both 16 processor and 1024 processor experiments. For some applications, the overhead increases marginally as the number of simultaneous replica changes increases. This could be due to network contention while transferring process images by multiple processes at the same time. In our experiments, we have observed that with a time interval between failure predictions of half an hour, the number of predicted failures and hence the number of simultaneous replica changes required is less than 10 even for a projected exascale system with $200,000$ nodes. Moreover, with the emergence of high performance networks with very high bandwidth, the impact of network contention is relatively less. Though our experiments have shown runtime overheads of only a few seconds for a single set of parallel replica changes, for all our experiments, we have assumed this overhead to be 1 minute for worst-case analysis.

5.3.2 Performance on Small and Medium Scale Systems

Even though the main goal of our adaptive replication framework is to provide fault tolerance for very large scale systems like exascale systems, our evaluations on real traces from two LANL systems, system 18 and system 20, with 512 and 1024 nodes respectively, show that our framework can provide effective fault tolerance even for smaller systems.

Table 5.7: Performance of adaptive replication on LANL systems

$No.\ of\ Nodes$	$Periodic\ CP$	$Dual\ Redun.$	$Adapt.\ Rep.$
512	90.5%	47.25%	89.51%
1024	86.81%	46.77%	88.75%

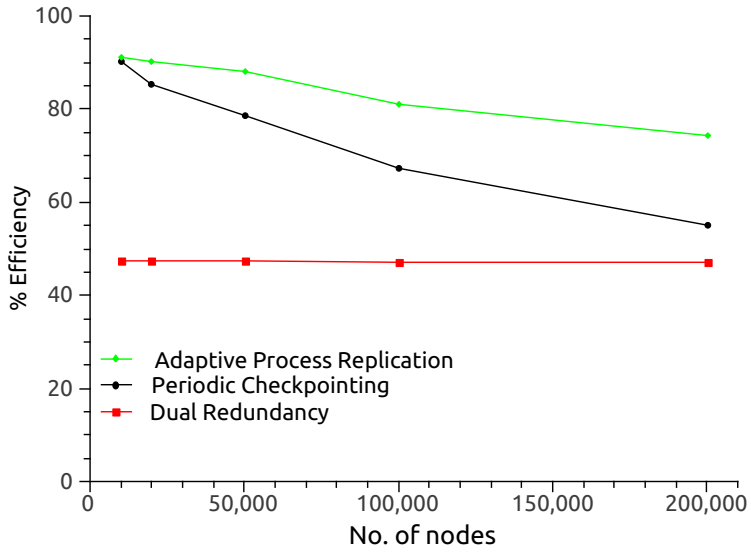
Table 5.7 shows the percentage efficiencies of the three fault tolerance strategies for the given LANL systems. It can be observed that adaptive replication performs better than periodic checkpointing for 1024 nodes, and almost as good as periodic checkpointing for 512 nodes. Dual redundancy is clearly not suitable for small scale systems as it cannot give an efficiency of more than 50%.

5.3.3 Performance on Large Scale Systems

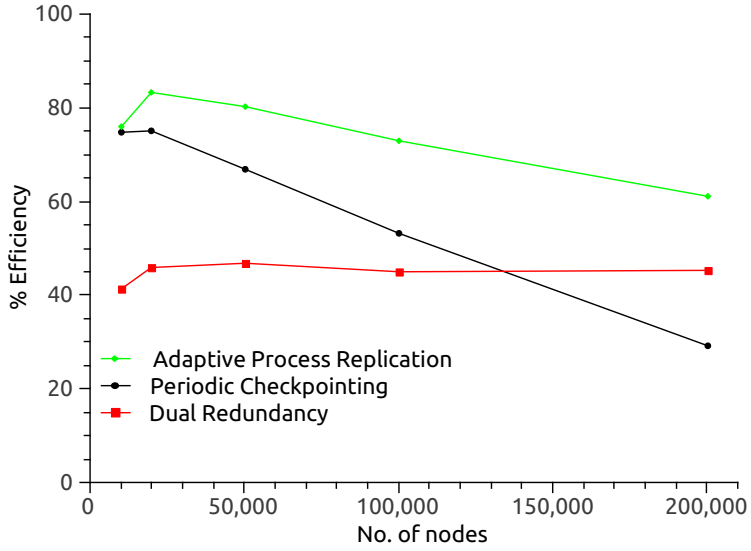
Figure 5.5 shows the performance of our framework against the other two techniques for both exponential and Weibull distributions. The graphs show that adaptive replication significantly outperforms the other techniques for both exponential and Weibull distributions. Our strategy performs 20% better than the best of the other two techniques for exponential distribution and 16% better than the best of the other two for Weibull distributions even for $200,000$ nodes, which is the projected number of nodes for an exascale system.

It can be observed that the performance of periodic checkpointing decreases drastically as the number of nodes increases, which proves again that periodic checkpointing is not an efficient fault tolerance technique for future systems. An interesting observation is that dual redundancy consistently gives around 45% efficiency for any number of nodes. This is because in this technique, all nodes have a replica and the probability of a node and its replica failing at the same time is very less. In fact, in our analysis, we have observed that with dual redundancy, the application experienced at most only a single failure during a 1 week execution. But, though dual redundancy is extremely effective in avoiding failures, it has the downside that the maximum possible efficiency is just 50%, while adaptive replication can provide much better efficiencies as shown by the results.

Though adaptive replication outperforms dual redundancy by a significant margin even for a projected exascale system with $200,000$ nodes, it should also be observed from Figure 5.5 that there is a downward trend in the efficiency achieved by adaptive replication as we increase the



(a) Exponential Distribution



(b) Weibull Distribution

Figure 5.5: Comparison of adaptive replication with other techniques

Table 5.8: Actions by adaptive replication and periodic checkpointing for 1 week

$No.\ of\ Nodes$	$No.\ of\ Rep.\ Changes$	$No.\ of\ Pre.\ CPs$	$No.\ of\ CPs\ (Per.\ CP)$
10,000	15	71	128
20,000	39	101	177
50,000	91	155	265
100,000	179	210	351
200,000	371	288	454

number of nodes. If this trend continues, it is possible that at some higher number of nodes, the efficiency of adaptive replication goes below that of dual redundancy. But, projecting the graphs for higher number of nodes show that this will happen only at around $300,000$ nodes for Weibull distribution and at around 1 million nodes for exponential distribution. Based on current trends and an anticipated node count of only around $200,000$ for exascale [34], it is likely to take at least 10-15 years for the node count in supercomputing systems to reach $300,000$ and a count of 1 million is unlikely to happen in the foreseeable future.

Table 5.8 shows the number of replica changes made and checkpoints taken by our framework for Weibull distribution. Note that in our framework, precautionary checkpoints are taken to tolerate unpredicted failures as explained in Section 4.2. The number of checkpoints taken to tolerate such unpredicted failures (given in Column 3) is significantly lesser than the total number of checkpoints taken by the periodic checkpointing method (given in Column 4). It can be observed from the table that our strategy makes significant number of replica changes for fault tolerance, resulting in improved efficiency. The relevance of adaptive replica change is more pronounced for higher number of nodes, with the number of replica changes increasing at a faster rate than the number of checkpoints. It can be seen that for $200,000$ nodes, our strategy makes more number of replica changes than checkpointing, which asserts that adaptive replication is a promising fault tolerance solution for future systems like exascale systems.

Results in Figure 5.5 show that a Weibull distribution provides a much more challenging failure management scenario than an exponential distribution. Moreover, studies have shown that real life systems experience failures that follow Weibull distribution [52]. Hence, for the rest of our evaluations, we present only the results using Weibull distribution.

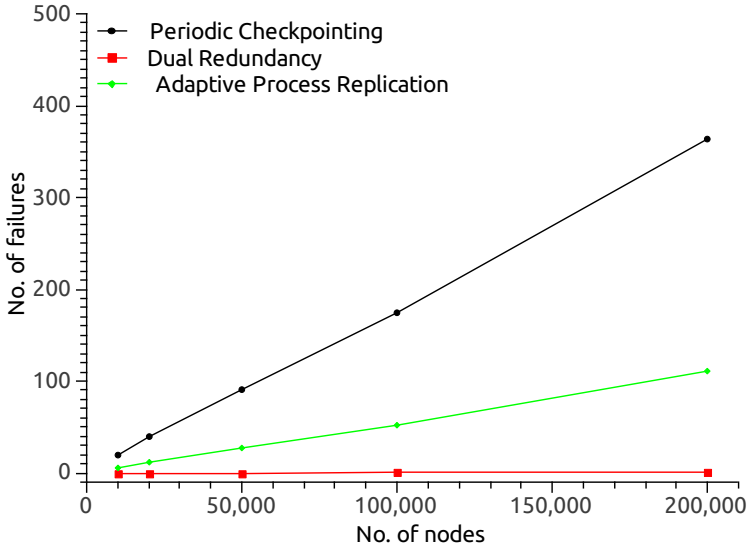


Figure 5.6: Number of application failures for adaptive replication and other techniques

5.3.4 Number of Failures

Figure 5.6 shows the observed number of application failures for different fault tolerance techniques. With dual redundancy, the application experienced only at most a single failure during its execution time. Though dual redundancy achieves an impressive 0 or 1 application failure scenario, it gives less than 50% application efficiency as mentioned earlier. Moreover, dual redundancy achieves this dramatic reduction in application failures at the cost of a huge number of resources used just for replication. The number of node failures in the system is equal to the number of application failures for periodic checkpointing, since it does not avoid any failures. From Figure 5.6, it can be observed that this number is only less than 400 for $200,000$ nodes. In this scenario, dual redundancy uses half of the total number of nodes, i.e., $100,000$ nodes for replication, to avoid only 400 failures!

Our framework aims at providing a balance between utilizing the potential of process replication to reduce number of failures, and at the same time reducing resource wastage and hence improving efficiency. Figure 5.6 shows that the number of application failures for our adaptive replication strategy, though more than that of dual redundancy, is much lesser than that of periodic checkpointing. We use only 1% of the total number of nodes as replica nodes, hence dramatically reducing the nodes used for replication from $100,000$ while using dual redun-

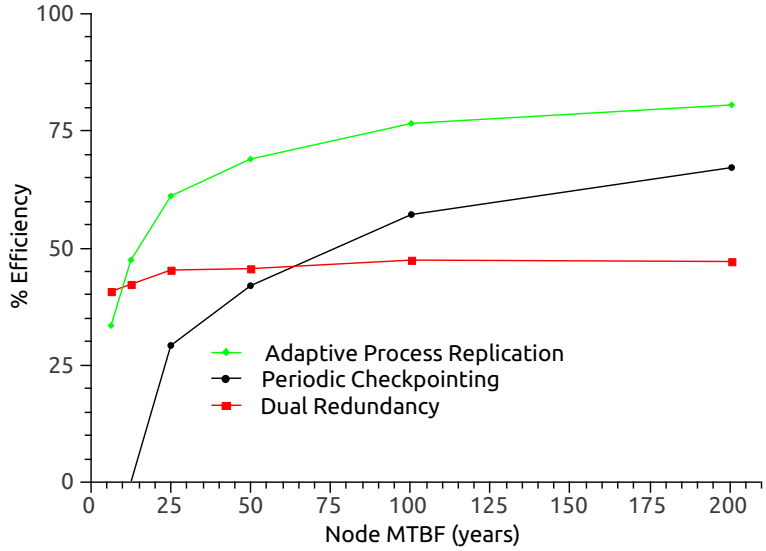


Figure 5.7: Effect of varying node MTBF on adaptive replication

dancy to just 2000 for a $200,000$ node system! Previous results have shown that our strategy also provides significant improvement in application efficiency. This shows the effectiveness of our strategy in achieving the right balance between the various aspects, resulting in significant reduction in the number of application failures and good application efficiency, with minimal number of nodes used for replication.

5.3.5 Performance for different node MTBFs

Figure 5.7 shows the performance of our framework for node MTBFs up to 200 years for $200,000$ nodes. As expected, both adaptive replication and periodic checkpointing shows better performance for higher node MTBFs due to the reduced number of node failures, and dual redundancy remains consistent in performance. It is observed that dual redundancy performs better than adaptive replication for node MTBF below 10 years. This is because of increase in the number of node failures which also results in increase in the number of unpredicted failures. However, studies have estimated an approximate per processor MTBF of 125 years based on analysis of failure statistics in real systems [6]. Hence, adaptive replication will outperform dual redundancy by significant margins in real systems. Periodic checkpointing gives zero efficiency for very low MTBFs because all the useful work done by the application is lost due to

rollbacks. Though adaptive replication outperforms periodic checkpointing for all the cases, it can be observed that the rate of increase of periodic checkpointing is higher than that of adaptive replication. Such a trend could mean that if the reliability of hardware keeps on increasing, then some day in the future, periodic checkpointing can again turn out to be an effective fault tolerance solution.

5.3.6 Comparison with Proactive Process Migration

Proactive process migration [57] uses a similar strategy to that of adaptive process replication, using failure predictions and spare nodes. In proactive migration, the processes in failure prone nodes are migrated to healthy spare nodes at different points of application execution, to avoid failures. Given that the overheads of process migration and adaptive replica change are comparable, both the strategies appear to be similar and hence it becomes essential to evaluate the benefits of adaptive replication over proactive migration. In this section, we compare our adaptive replication strategy with the proactive process migration strategy.

Consider the following four possible failure prediction scenarios and the actions taken by both the strategies for each of them.

1. *True Positive*, where a node is predicted to fail, and it actually fails.
    - Proactive Migration: A process migration is initiated.
    - Adaptive Replication: A process image replacement is initiated only if the node does not already have a replica in the system.
2. *False Positive*, where a node is predicted to fail, but it does not fail.
    - Proactive Migration: A process migration is initiated.
    - Adaptive Replication: A process image replacement is initiated only if the node does not already have a replica in the system.
3. *True Negative*, where a node is not predicted to fail, and it does not fail.
    - Proactive migration: No action is taken.

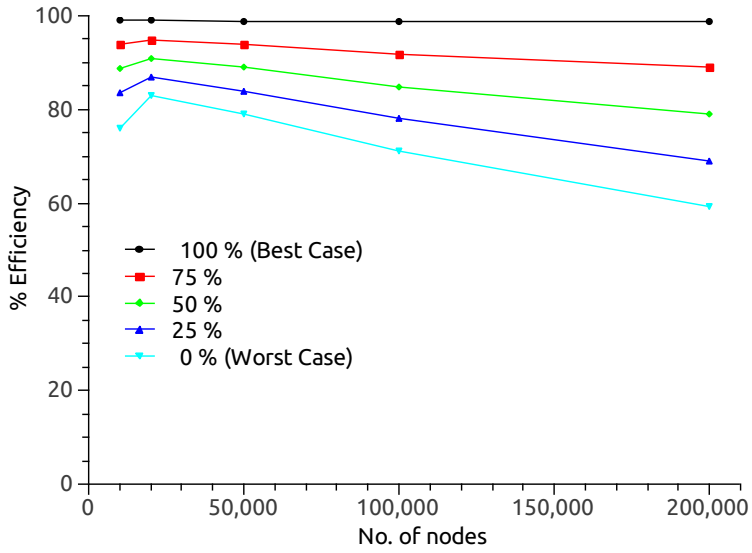


Figure 5.8: Proactive migration vs Adaptive replication

- Adaptive replication: No action is taken.

4. *False Negative*, where a node is not predicted to fail, but it actually fails.

   - Proactive migration: No action is taken and the application will fail.
   - Adaptive replication: No action is taken, but the application will fail only if the node does not already have a replica in the system.

In three out of the four failure prediction scenarios listed above (scenarios 1, 2 and 4), adaptive replication has the potential to win over proactive migration and in the remaining one scenario, both strategies are the same. The essential condition for adaptive replication to win in any of the 3 scenarios is that that the given node should already have a replica in the system, thereby avoiding a process image replacement or an application failure, which cannot be avoided in case of proactive migration. But, the probability of this condition being true at a given time depends on the failure pattern in the system. To be specific, the probability that at any given adaptation point, a node which is predicted to fail in the next interval or is actually going to fail in the next interval irrespective of the prediction, already has a replica in the system depends on the failure pattern. Figure 5.8 shows the impact of varying this probability on application efficiency, for different number of nodes.

In Figure 5.8, a probability of 0 represents the worst case scenario for adaptive replication, where for all the 3 potential winning scenarios, the given node does not have an already existing replica. This scenario gives the same application efficiency as proactive migration, assuming that the overheads of adaptive replica change and process migration are the same. Hence, for the purpose of comparison, the application efficiency corresponding to a probability of 0 can be taken as the efficiency of proactive migration for a given number of nodes. A probability of 100% represents the best case for adaptive replication, where for all the 3 potential winning scenarios, the given node has an already existing replica.

It can be observed in the figure that as the probability increases, adaptive replication outperforms proactive migration by greater margins. It can also be observed that the margins between the efficiencies of adaptive replication and proactive migration increases with node count, with a margin of around 25% for $10,000$ nodes and around 40% for $200,000$ nodes with 100% probability. The increase in margin with increasing node count shows that adaptive replication can provide much better fault tolerance than proactive migration for future large scale systems like exascale systems with large number of nodes.

One possible way to increase the probability that for the 3 potential winning scenarios for adaptive replication, the failure prone node has an already existing replica is to increase the replication degree or the percentage of nodes replicated. The higher the percentage of nodes replicated, higher will be the probability that a given node in the system has a replica. Research efforts such as [20] have explored the possibility of varying replication degree to maximize application performance. Elliott et al. [20] have reported good application efficiencies at replication degrees between 1.5 and 2, i.e., between 25-50% of the nodes replicated. Also, when the recall value of the predictor is low, the number of failures predicted will be very less. In such a scenario where we have limited confidence on the ability of the predictor to predict impending node failures, having a high replication degree will be beneficial because of the high number of replicated nodes. In this context, we also compare our adaptive replication strategy against proactive migration for different degrees of replication.

Figure 5.9 shows the effect of varying the percentage of nodes replicated on application efficiency for both the strategies. It can be observed that as the percentage of nodes replicated increases, though the overall application efficiency decreases for both the strategies due

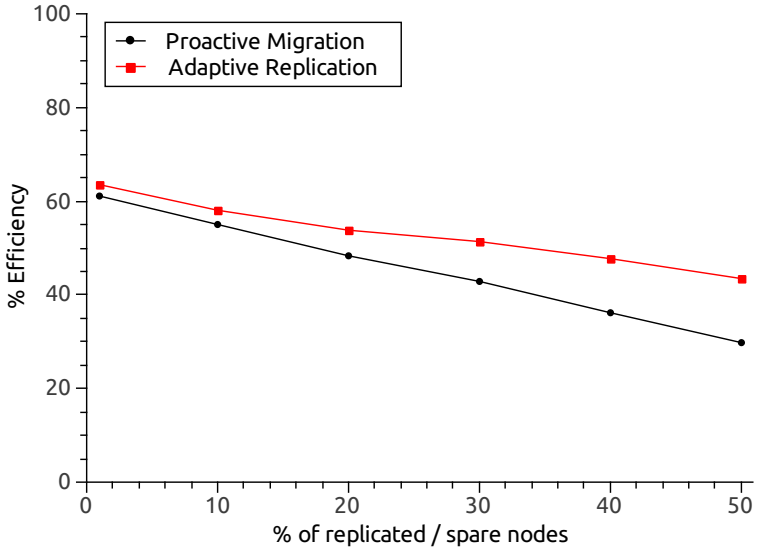


Figure 5.9: Proactive migration vs Adaptive replication (varying degree of replication)

to the decrease in number of compute nodes, the margin between application efficiencies given by adaptive replication and proactive migration increases. Adaptive replication outperforms proactive migration by greater margins as the replication degree increases. This is because, as the number of replicated nodes in the system increases, the probability that a given node has an already existing replica increase, resulting in more number of instances where adaptive replication wins over proactive migration by avoiding process image replacement/application failure.

5.3.7 Accuracy of Failure Predictions

The surface plot in Figure 5.10 shows the impact of varying accuracy metrics of the failure predictor on the performance of adaptive replication. The figure shows that efficiency is more than 60% for all precision and recall values above 0.5. Failure predictors today [27, 42] have observed precision and recall values in the range of 0.5 to 0.8. Hence, adaptive replication gives high efficiency for all realistic values of precision and recall.

We also observe that the efficiency achieved by adaptive replication depends a lot on the recall of the predictor and does not depend much on its precision. The efficiency achieved is more than 75% for a recall of 0.9 even when the precision is just 0.1. But, if the recall is

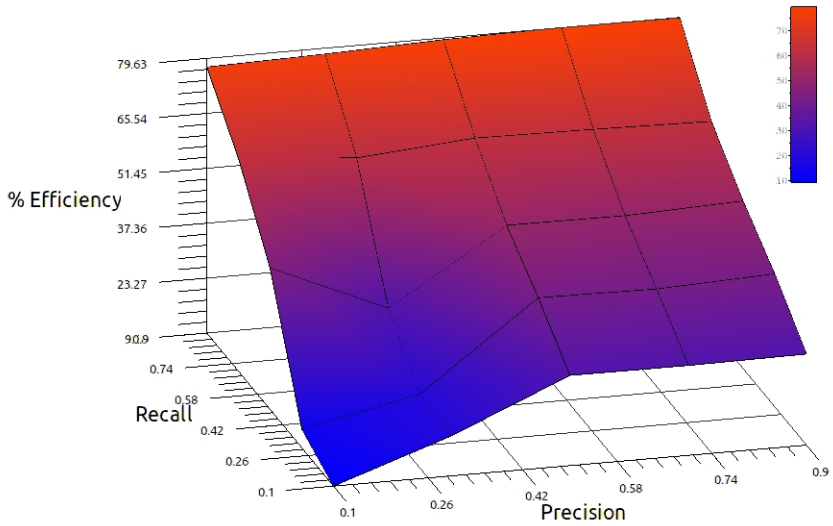


Figure 5.10: Effect of accuracy of failure predictions on adaptive replication

low, then the efficiency is poor even for a very high precision, as shown by the 32% efficiency observed with a precision of 0.9 and a recall of 0.1. Note that for a failure predictor, a high precision means that the number of correct predictions out of the total number of predictions is high, while a high recall means that the number of predicted failures out of the total number of failures is high. Hence application efficiency for adaptive replication is higher for large number of correct failure predictions (true positives, i.e., predictions of failures that actually happen) and is independent on the number of wrong failure predictions (false positives, i.e., predictions of failures that do not happen). For example, suppose 10 nodes are actually going to fail in the next interval. Then no matter whether the predictor gives 20 predictions for the next interval or 200 predictions, if the 10 correct predictions are included in the set of predictions, adaptive replication gives high efficiency.

The above observation provides an interesting insight for future works on failure predictors. Most failure prediction methods today [27, 42] focus on achieving a high value for both precision and recall. Our analysis shows that a strategy like adaptive replication can perform well if the recall is high, even if the precision is very less. Adaptive replication and similar strategies can then make use of a failure predictor that tries to achieve a better recall even at the expense of having a reduced precision. After all, if we have to guess the names of 10 items in a closed box, it will be much easier to get the names of all the 10 items correctly if we are allowed to

make 200 guesses than if we are allowed to make only 20 guesses!

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we have developed two adaptive fault tolerance strategies that make use of node failure prediction mechanisms to provide proactive fault tolerance on large scale systems.

In the first part of this thesis, we have developed ADFT, an adaptive framework that makes runtime decisions on fault tolerance techniques at different points of application execution. Our framework considers malleability of applications and performs rescheduling to different number of processors during execution. We have demonstrated the effectiveness of our framework in realistic scenarios based on cost models using a robust failure simulator. Using larger number of processors with real and synthetic failure traces, and with real and synthetic applications, we showed that our strategy involving malleability outperforms the popular but static periodic checkpointing approach by at least 21%, and also yields up to 23% higher amount of work than the dynamic *FT-Pro* strategy that does not involve malleability. Our results also show that our adaptive strategy yields high performance even for petascale systems and beyond. We also showed that application malleability will have to be considered strongly for future exascale systems.

In the second part of this thesis, we have developed a framework that uses partial replication along with adaptive changing of the set of replicated processes based on failure predictions, to provide effective fault tolerance for both medium and large scale systems. We have also developed an MPI prototype implementation PAREP-MPI, that supports partial replication and

adaptive replica change for MPI applications with minimal runtime overhead. Simulations using failure traces of both exponential and Weibull distributions have shown that our adaptive replication strategy significantly outperforms periodic checkpointing and dual redundancy, providing up to 20% more efficiency even for a projected exascale system with $200,000$ nodes. We have also shown the importance of the recall value of failure predictors in providing better fault tolerance using our adaptive strategy and that adaptive replication can achieve high application efficiency with a good predictor recall, even if the precision is low.

6.2 Future Work

In future, we plan to develop a fault management software suite that will consist of the fault management framework discussed in the thesis, tools for performing various fault tolerance actions, and techniques that give failure predictions. We also plan to enhance our failure simulator to study fault tolerance scenarios in future systems so as to explore alternate and novel fault tolerance techniques that can give high performance for such very large scale future systems based on the study conducted in this work.

To improve our adaptive process replication strategy, we plan to consider application malleability to adaptively change the number of nodes in a replica set, along with adaptive replica change and to develop better fault tolerance frameworks based on both the features. Our current implementation of PAREP-MPI requires that Address Space Layout Randomization (ASLR) feature in the current versions of Linux kernel and Pointer Encryption or Pointer Guard feature supported by the current versions of glibc are turned off. In future, we plan to extend our implementation to overcome these limitations and also support more features for fault tolerance on exascale systems and beyond.

References

[1] N. R Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[2] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *IPDPS '09: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2009.

[3] Address Space Layout Randomization. `http://en.wikipedia.org/wiki/Address_space_layout_randomization/`.

[4] A. Bhatele, P. Jetley, H. Gahvari, L. Wesolowski, W. D. Gropp, and L. V. Kale. Architectural Constraints to Attain 1 Exaflop/s for Three Scientific Application Classes. In *IPDPS '11: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2011.

[5] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *IPDPS '08: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2008.

[6] Marin Bougeret, Henri Casanova, Mikael Rabie, Yves Robert, and Frédéric Vivien. Checkpointing strategies for parallel jobs. In *SC '11: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[7] Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Using replication for resilience on exascale system, 2012. INRIA Research report RR-7830.

[8] K.J. Bowers, E. Chow, H. Xu, R.O. Dror, M.P. Eastwood, B.A. Gregersen, J.L. Klepeis, I. Kolossvary, M.A. Moraes, F.D. Sacerdoti, et al. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters. In *SC 2006: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 43–43. IEEE, 2006.

[9] James Brandt, Frank Chen, Vincent De Sapio, Ann Gentile, Jackson Mayo, Philippe Pébay, Diana Roe, David Thompson, and Matthew Wong. Using Cloud Constructs and Predictive Analysis to Enable Pre-Failure Process Migration in HPC Systems. In *CCGRID '10: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 703–708, 2010.

[10] Bruce Allen. Monitoring Hard Disks With SMART, 2004. Linux Journal.

[11] F. Cappello. Resilience: One of the main challenges for Exascale Computing. INRIA Illinois Joint-Laboratory on Petascale computing.

[12] F. Cappello, H. Casanova, and Y. Robert. Checkpointing vs. Migration for Post-Petascale Supercomputers. In *ICPP '10 Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010.

[13] Community Climate System Model (CCSM). `http://www.ccsm.ucar.edu`.

[14] Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. `http://research.cs.wisc.edu/condor/doc/ckpt97.pdf/`.

[15] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The Community Climate System Model: CCSM3. *Journal of Climate*, 1998.

[16] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, February 2006.

[17] J. M. Dennis, R. Jacob, M. Vertenstein, T. Craig, and R. Loy. Toward an Ultra-High Resolution Community Climate System Model for the Bluegene Platform. *Journal of Physics: Conference Series*, 78, 2007.

[18] M.D. Dikaiakos and J. Stadel. A Performance Study of Cosmological Simulations on Message-Passing and Shared-Memory Multiprocessors. In *Proceedings of the 10th international Conference on Supercomputing*, pages 94–101. ACM, 1996.

[19] A. B. Downey. A Model For Speedup of Parallel Programs, 1997. Technical Report, University of California at Berkeley Berkeley, CA, USA.

[20] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *ICDCS '12: Proceedings of the 32nd International Conference on Distributed Computing Systems*, 2012.

[21] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *SC '11: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[22] B.G. Fitch, A. Rayshubskiy, M. Eleftheriou, T.J.C. Ward, M. Giampapa, M.C. Pitman, and R.S. Germain. Blue Matter: Approaching the Limits of Concurrency for Classical Molecular Dynamics. In *SC 2006: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 44–44. IEEE, 2006.

[23] Failure Trace Archive. `http://fta.inria.fr/apache2-default/pmwiki/index.php?n=Main.DataSets/`.

[24] C. George and S. Vadhiyar. An Adaptive Framework for Fault Tolerance on Large Scale Systems using Application Malleability. In *ICCS '12: Proceedings of the International Conference on Computational Science*, 2012.

[25] F. Gioachin, P. Jetley, C. L. Mendes, L. V. Kale, and T. Quinn. Towards Petascale Cosmological Simulations with ChaNGa, 2007. Technical Report, Parallel Programming Laboratory, University of Illinois.

[26] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems. In *CCGRID '10: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010.

[27] P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White. A Meta-Learning Failure Predictor for Blue Gene/L Systems. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, 2007.

[28] G. Hamerly, C. Elkan, et al. Bayesian Approaches to Failure Prediction for Disk Drives. In *ICML: International Conference on Machine Learning*, pages 202–209, 2001.

[29] J.L. Hellerstein, F. Zhang, and P. Shahabuddin. A Statistical Approach to Predictive Detection. *Computer Networks*, 35(1):77–95, 2001.

[30] Sandia National Laboratory - LAMMPS Molecular Dynamics Simulator. `https://lammps.sandia.gov/`.

[31] Sandia National Laboratory - Mantevo Project. `https://software.sandia.gov/mantevo/`.

[32] Intelligent Platform Managemtn Interface. `http://www.intel.com/design/servers/ipmi`.

[33] William M. Jones, John T. Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 276–279, 2010.

[34] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems, 2008. (P. Kogge, Editor and Study Lead).

[35] Z. Lan and Y. Li. Adaptive Fault Management of Parallel Applications for High-Performance Computing. *IEEE Transactions on Computers*, 57(12), 2008.

[36] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *DSN: International Conference on Dependable Systems and Networks*, pages 425–434. IEEE, 2006.

[37] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *IPDPS '08: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages 1–9, 2008.

[38] Hardware Monitoring By Lm Sensors. `http://en.wikipedia.org/wiki/Address_space_layout_randomization/`.

[39] Frank H. Mathis. A generalized birthday problem. *SIAM Review*, 33(2):265–270, May 1991.

[40] J. Michalakes, J. Hacker, R. Loft, M. O. McCracken, A. Snavely, N. J. Wright, T. Spelce, R. Walkup, and B. Gorda. WRF Nature Run. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[41] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, 2007.

[42] N. Nakka, A. Agrawal, and A. Choudhary. Predicting Node Failure in High Performance Computing Systems from Failure and Usage Logs. In *IPDPS '11: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2011.

[43] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html/`.

[44] L. Oliker, A. Canning, J. Carter, C. lancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific Application Performance on Candidate PetaScale Platforms. In *IPDPS '07: Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2007.

[45] A. Osman, H. Ammar, A. Smirnov, S. Shi, and I. Celik. Scalability Analysis and Domain Decomposition of Large Eddy Simulations of Ship Wakes. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 213–219. IEEE, 2001.

[46] Xiangyong Ouyang, R. Rajachandrasekar, X. Besseron, and D.K. Panda. High Performance Pipelined Process Migration with RDMA. In *CCGRID '11: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 314–323, 2011.

[47] F. Petrini, K. Davis, and J. Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In *IPDPS '04: Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages 209–, 2004.

[48] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance, 1997. Technical Report, University of Tennessee Knoxville, TN, USA.

[49] Pointer Encryption / Pointer Guard. `http://www.kernel.org/doc/man-pages/online/pages/man8/ld.so.8.html`.

[50] 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*, pages 426–435. ACM, 2003.

[51] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[52] B. Schroeder and G. Gibson. A Large-scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, 2006.

[53] Top 500 Supercomputing Sites. `http://www.top500.org/`.

[54] S. Vadhiyar and J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[55] K. Vaidyanathan and K.S. Trivedi. A Measurement-based Model for Estimation of Resource Exhaustion in Operational Software Systems. In *Proceedings of 10th International Symposium on Software Reliability Engineering*, pages 84–93. IEEE, 1999.

[56] R. Vilalta and S. Ma. Predicting Rare Events in Temporal Domains. In *ICDM: Proceedings of IEEE International Conference on Data Mining*, pages 474–481. IEEE, 2002.

[57] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[58] P.K. Weiner and P.A. Kollman. AMBER: Assisted Model Building with Energy Refinement. A General Program for Modeling Molecules and their Interactions. *Journal of Computational Chemistry*, 2(3):287–303, 1981.

[59] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon. Validating the FLASH Code: Vortex-Dominated Flows. *Astrophysics and Space Science*, 298(1):341–346, 2005.

[60] X. Yang, Y. Du, P. Wang, H. Fu, and J. Jia. FTPA: Supporting Fault-Tolerant Parallel Computing through Parallel Recomputing. *IEEE Trasactions on Parallel and Distributed Systems*, 20(10), 2009.

[61] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.

[62] G. Zheng, L. Shi, and L. V. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 93–103, sept. 2004.