THE UNIVERSITY OF SYDNEY

COPYRIGHT AND USE OF THIS THESIS

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

You may infringe the author's moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author's reputation

For further information contact the University's Director of Copyright Services

sydney.edu.au/copyright

A Proactive Fault Tolerance Framework for High Performance Computing (HPC) Systems in the Cloud

Ifeanyi P. Egwutuoha

THE UNIVERSITY OF
SYDNEY

A thesis submitted in fulfillment of the
requirements for the degree of

Doctor of Philosophy

Faculty of Engineering and Information Technologies
School of Electrical and Information Engineering
The University of Sydney
July 2014

Abstract

High Performance Computing (HPC) systems have been widely used by scientists and researchers in both industry and university laboratories to solve advanced computation problems. Most advanced computation problems are either data-intensive or computation-intensive. They may take hours, days or even weeks to complete execution. For example, some of the traditional HPC systems computations run on 100,000 processors for weeks. Consequently traditional HPC systems often require huge capital investments. As a result, scientists and researchers sometimes have to wait in long queues to access shared, expensive HPC systems.

Cloud computing, on the other hand, offers new computing paradigms, capacity, and flexible solutions for both business and HPC applications. Some of the computation-intensive applications that are usually executed in traditional HPC systems can now be executed in the cloud. Cloud computing price model eliminates huge capital investments.

However, even for cloud-based HPC systems, fault tolerance is still an issue of growing concern. The large number of virtual machines and electronic components, as well as software complexity and overall system reliability, availability and serviceability (RAS), are factors with which HPC systems in the cloud must contend. The reactive fault tolerance approach of checkpoint/restart, which is commonly used in HPC systems, does not scale well in the cloud due to resource sharing and distributed systems networks. Hence, the need for reliable fault tolerant HPC systems is even greater in a cloud environment.

In this thesis we present a proactive fault tolerance approach to HPC systems in the cloud to reduce the wall-clock execution time, as well as dollar cost, in the presence of hardware failure. We have developed a generic fault tolerance algorithm for HPC systems in the cloud. We have further developed a cost model for executing computation-intensive applications on HPC systems in the cloud. Our experimental results obtained from a real cloud execution environment show that the wall-clock execution time and cost of running computation-intensive applications in the cloud can be considerably reduced compared to checkpoint and redundancy techniques used in traditional HPC systems.

Dedication

This thesis is dedicated to

Almighty God, the giver of all wisdom,

my wife Nancy Egwutuoha and my children Excel, Esther and Ebube.

Acknowledgements

Most importantly, I acknowledge my supervisors, Associate Professor David Levy, Mr. Bran Selic, Dr. Shiping Chen and Associate Professor Rafael Calvo for their outstanding support, advice, encouragement and contribution to my studies. Without their advice and encouragement, this thesis would not have been completed.

Special thanks to the School of Electrical and Information Engineering, The University of Sydney, for granting me the opportunity to pursue PhD studies with the support of a Norman I. Price Supplementary Scholarship.

It is with great pleasure that I acknowledge all my family members, friends, and members of the Software Engineering group in the School of Electrical and Information Engineering for their outstanding comments and suggestions on my research. In particular, special thanks to Daniel Schragl and Phil Cairns. I would like to thank Dr Cherry Russell, who proofread the final draft of this thesis.

Finally, I am deeply grateful to my wonderful wife Nancy Egwutuoha and my children Excel, Esther and Ebube for their unconditional love, encouragement and everything else.

May Almighty God continuously reward everyone who has made this PhD thesis a success.

Publications

International Journal Publications

1. **Ifeanyi P. Egwutuoha**, David Levy, Bran Selic, and Shiping Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing,* vol. 65, no. 3, pp 1302-1326, 2013.
2. **Ifeanyi P. Egwutuoha**, Daniel Schragl, and Rafael Calvo, "A Brief Review of Cloud Computing, Challenges and Potential Solutions," *Journal of Parallel & Cloud Computing,* vol. 2, no. 1, pp. 7-14, 2013.
3. **Ifeanyi P. Egwutuoha**, Shiping Chen, David Levy, Bran Selic, and Rafael Calvo, "Cost-Oriented Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud," *International Journal of Parallel, Emergent and Distributed Systems,* no. ahead-of-print, pp. 1–16, 2014.

International Conference Publications

1. **Ifeanyi P. Egwutuoha**, Shiping Chen, David Levy, Bran Selic, and Rafael Calvo, "Energy Efficient Fault Tolerance for High Performance Computing (HPC) in the Cloud," *in Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, CLOUD '13,* pp. 762–769, IEEE Computer Society, 2013.
2. **Ifeanyi. P. Egwutuoha**, Shiping Chen, David Levy, and Rafael Calvo, "Cost-effective Cloud Services for HPC in the Cloud: The IaaS or The HaaS?," *in 2013 Int. Conf. Parallel and Distributed*

Processing Techniques and Applications, PDPTA'13, pp. 223–228, 2013.

3. **Ifeanyi P. Egwutuoha**, Shiping Chen, David Levy, Bran. Selic, and Rafael Calvo, "A Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud," *2012 Second in International Conference on Cloud and Green Computing (CGC),* pp. 268–273, IEEE Press, 2012.
4. **Ifeanyi P. Egwutuoha**, Shiping Chen, David Levy, and Bran Selic, "A Fault Tolerance Framework for High Performance Computing in Cloud," *in 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),* pp. 709–710, IEEE Press, 2012.
5. **Ifeanyi P. Egwutuoha**, David Levy, and Bran Selic, "Evaluation of process level redundant checkpointing/restart for HPC systems," *2011 IEEE 30th International in Performance Computing and Communications Conference (IPCCC),* pp. 1-2, IEEE Press, 2011.

Contents

Glossary **xiii**

1 Introduction **1**
- 1.1 Background . . . 5
- 1.2 Motivation . . . 6
- 1.3 Identification of Key Research Problems . . . 7
  - 1.3.1 Fault Tolerance . . . 7
  - 1.3.2 Rollback-Recovery . . . 8
  - 1.3.3 Energy Efficiency . . . 8
- 1.4 Contributions . . . 9
- 1.5 Thesis Outline . . . 10

2 Related Work **12**
- 2.1 Analysis of Failure Rates of HPC Systems . . . 12
  - 2.1.1 Software Failure Rates . . . 13
  - 2.1.2 Hardware Failure Rates . . . 14
  - 2.1.3 Human Error Failure Rates . . . 19
- 2.2 Review of Fault Tolerance Mechanisms for High Performance Computing Systems . . . 19
  - 2.2.1 Migration Method . . . 20
  - 2.2.2 Redundancy . . . 22
  - 2.2.3 Failure Masking . . . 24
  - 2.2.4 Failure Semantics . . . 25
  - 2.2.5 Recovery . . . 26
- 2.3 Rollback-recovery Feature Requirements for HPC Systems . . . 30

2.4 Checkpoint-based Rollback-recovery Mechanisms . . . 31
2.4.1 Log-based Rollback-recovery Mechanisms . . . 33
2.5 Taxonomy of Checkpoint Implementation . . . 35
2.6 Reducing the Time for Saving the Checkpoint in Persistent Storage 37
2.7 Survey of Checkpoint/Restart Facilities . . . 39
2.8 Summary . . . 39

3 HPC Systems in the Cloud, Concepts and Architecture 41
3.1 Cloud Computing Architectures . . . 41
3.1.1 Cloud Providers . . . 43
3.1.2 Cloud Users . . . 43
3.2 Cloud Service Models . . . 43
3.2.1 Software as a Service (SaaS) . . . 45
3.2.2 Platform as a Service (PaaS) . . . 45
3.2.3 Infrastructure as a Service (IaaS) . . . 46
3.2.4 Hardware as a Service (HaaS) . . . 46
3.3 Types of Clouds . . . 47
3.3.1 Public Cloud . . . 47
3.3.2 Private Cloud . . . 47
3.3.3 Community Cloud . . . 49
3.3.4 Hybrid Cloud . . . 49
3.4 HPC Systems in the Cloud . . . 50
3.4.1 Head Node . . . 50
3.4.2 Compute Nodes . . . 51
3.4.3 Message Passing Interface . . . 52
3.4.4 Fail-stop Model . . . 52
3.4.5 Virtualisation . . . 53
3.5 Challenges for HPC Systems in the Cloud . . . 54
3.5.1 System Reliability . . . 54
3.5.2 Cost . . . 56
3.6 Summary . . . 57

4 Proactive Fault Tolerance Approach to HPC Systems in the Cloud **59**
4.1 Introduction . . . 59
4.2 Benefit of HaaS . . . 63
4.3 Proactive Fault Tolerance for HPC Systems in the Cloud . . . 64
4.3.1 Monitoring Module with Lm-sensors . . . 65
4.3.2 Failure Predictor . . . 66
4.3.3 Proactive Fault Tolerance Policy . . . 70
4.3.4 The Controller Module . . . 71
4.4 Proactive Fault Tolerance Algorithm . . . 74
4.4.1 Cost Model . . . 76
4.4.2 Installation/Configuration Cost . . . 77
4.4.3 Execution Cost . . . 78
4.4.4 Communication Cost . . . 78
4.4.5 Storage Cost . . . 79
4.4.6 Redundancy Cost . . . 79
4.4.7 Failure Cost . . . 80
4.4.8 Quantitative Analysis . . . 82
4.5 Evaluation . . . 84
4.6 Related Work . . . 92
4.7 Summary . . . 94

5 Cost-effective Cloud Services for HPC in the Cloud: IaaS or HaaS? **95**
5.1 Introduction . . . 95
5.2 Experimental Setup . . . 96
5.2.1 Cluster Compute Instances from Cloud-A (IaaS) . . . 97
5.2.2 HPC System on HaaS in the Cloud . . . 97
5.3 MPI Applications and Benchmark . . . 101
5.4 Results and Discussion . . . 102
5.5 Cost Analysis . . . 104
5.6 Related Work . . . 106
5.7 Summary . . . 108

6 Summary, Conclusions and Future work **109**

6.1 Summary 109

6.2 Conclusions 112

6.3 Future Work 114

Appendix A **116**

References **154**

List of Tables

2.1 Summary of HPC systems studied by Oliner and Stearley [1] . . . 18

4.1 HPC systems applications 60
4.2 Lm-senors generated events 67
4.3 HPL with different problem sizes and nodes 86
4.4 HPL with different problem sizes, host and compute nodes 88
4.5 Cost analysis with different FT and nodes 90

5.1 Virtual and HaaS Instances from Cloud-A and Cloud-B 98
5.2 The cost analysis virtual machine of IaaS and virtual machine of HaaS 100

A.1 Future failure prediction and warning state table 116
A.2 Checkpoint/restart facilities 117

List of Figures

1.1 Performance development of HPC systems 1993-2012 and projected performance development for 2020 [2] 3
1.2 Performance growth in the number of processors on HPC systems 1993-2012 4
1.3 Thesis outline 11

2.1 A simplified analysis of number of failures for each system according to CFDR data [3] 16
2.2 Failure rate of HPC systems with different CPUs and nodes . . . 17
2.3 An abstract view of fault tolerance techniques with feature modeling 21
2.4 Flat group and hierarchical group masking 25
2.5 Taxonomy of Checkpoint Implementation 37

3.1 Cloud computing architecture 44
3.2 HPC System in the Cloud with computation-intensive application (our illustration shows MPI application executing in HPC system in the cloud) 48
3.3 Host and Xen virtualization 51
3.4 Reliability levels of two systems with MTBF of 10^5 and 10^6 as a function of the number of nodes 56

4.1 Rule-based predictor model 68
4.2 System architecture 73
4.3 Performance of HPL benchmarking without checkpointing, with checkpointing, and with FTDaemon. 87
4.4 HPL with different problem sizes and nodes 91

5.1 Computational performance of High Performance Linpack on 1 node with 16 processors 103
5.2 Performance of ClustalW-MPI application on 16 processors 104
5.3 The cost analysis; virtual machine of IaaS and virtual machine of HaaS 106

Glossary

Amazon EC2	Amazon Elastic Compute Cloud
CFDR	Computer Failure Data Repository
CPU	Central Processing Unit
Dom0	Privileged Domain
DomU	Unprivileged Domain
FER	Forward Error Recovery
FT	Fault Tolerance
HaaS	Hardware as a Service
HPC	High Performance Computing
I/O	Input/Output
IaaS	Infrastructure as a Service
IT	Information Technology
LANL	Los Alamos National Laboratory
MPI	Message Passage Interface
MTBF	Mean Time Between Failure
NAND	Negated AND or NOT AND
NCSA	National Center for Supercomputing Applications

NUMA	Non-Uniform Memory Architecture
OpenMPI	Open Message Passage Interface
OS	Operating System
PaaS	Platform as a Service
PC	Personal Computer
PhD	Doctor of Philosophy
RAM	Random-access memory
RAS	Reliability, availability, and serviceability
SaaS	Software as a Service
SMP	Symmetric Multi-Processor
SNL	Sandia National Labs
SSD	Solid State Device
SSH	Secure Shell
TMR	Triple Modular Redundancy
VM	Virtual Machine

Chapter 1

Introduction

In the scientific research domain, High Performance Computing (HPC) refers to the use of supercomputers, grid environments and/or clusters of computers to solve advanced computational problems. A computer cluster is a combination of commodities unit (PCs, processors, networks or symmetric multi-processor (SMPs)).

HPC systems play important roles in today's society. Common applications include weather forecasting, aircraft crash simulation, computational fluid dynamics for studies in aerodynamics, bioinformatics, protein folding for molecular modelling in biomedical research, and many others [4, 5]. It has been estimated that improvements in HPC systems leading to more accurate seismic modelling of oil reservoirs could increase oil recovery by 50-75% [6].

Today, HPC systems also offer new opportunities in business [7]. For example, financial institutions currently use HPC systems in real time modelling to make informed investment decisions.

Analysis of the Top500 [2] HPC systems shows that the number of processors and nodes in HPC systems has increased over time in the quest for greater performance levels. Top500 is a detailed statistical ranking of the world's 500 most

powerful supercomputers. The list is compiled twice a year. Figure 1.1 shows the performance development of Top500 HPC systems between 1993 and 2012, and the projected performance development for 2020. HPC systems continue to grow exponentially in scale, from petascale computing floating point operations per second to exascale computing floating point operations per second). As can be seen, performance almost doubles each year due in large part to the steady increase in the density of transistors in integrated circuits.

As noted above, this performance development is associated with growth in the number of processors in each HPC system. Figure 1.2 shows the growth in the number of processors between 1993 and 2012, based on an analysis of HPC systems development data provided in the Top500 [2].

There are, however, a number of technical challenges for HPC systems as they grow to exascale [8, 9]. Although the Mean Time Between Failure (MTBF) of individual system components may be high, the overall system Mean Time Between Failure is reduced to just a few hours due to the increased number of processors involved [10, 11]. For example, the IBM Blue Gene/L was built with 131,000 processors. To calculate its MTBF, we assume that each processor has a constant failure rate. If the MTBF of each processor is 876000 hours (100 years), a cluster of 131,000 processors will have an expected MTBF of 876000/131000 = 6.68 hours.

The importance of fault tolerance for HPC systems has been widely recognised by various research communities in HPC systems. Many approaches have been proposed to provide fault tolerance in HPC systems [12, 13, 14, 11, 15, 16, 17]. Approaches such as [15, 16], explore redundancy and rollback-recovery techniques respectively.

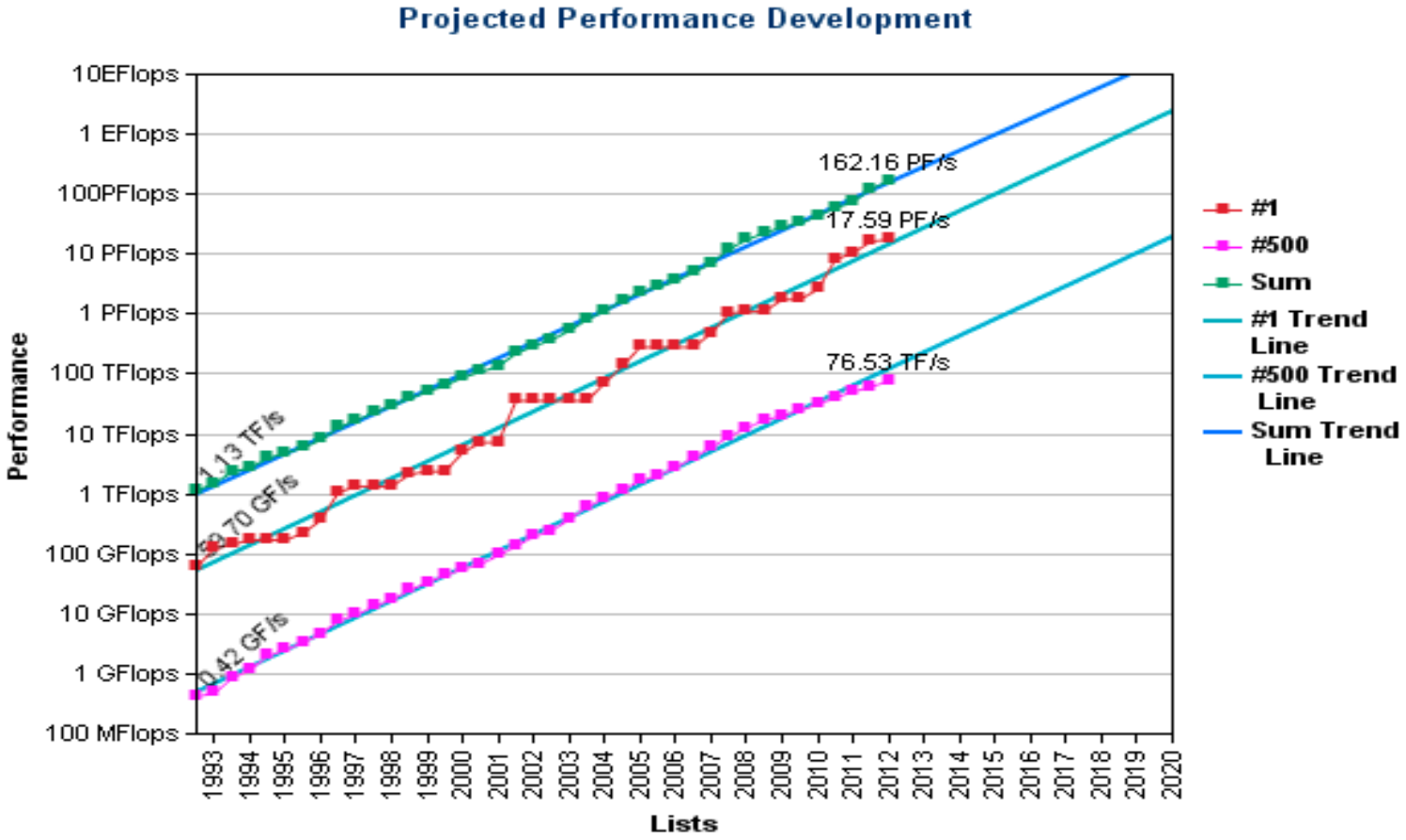

Figure 1.1: Performance development of HPC systems 1993-2012 and projected performance development for 2020 [2]

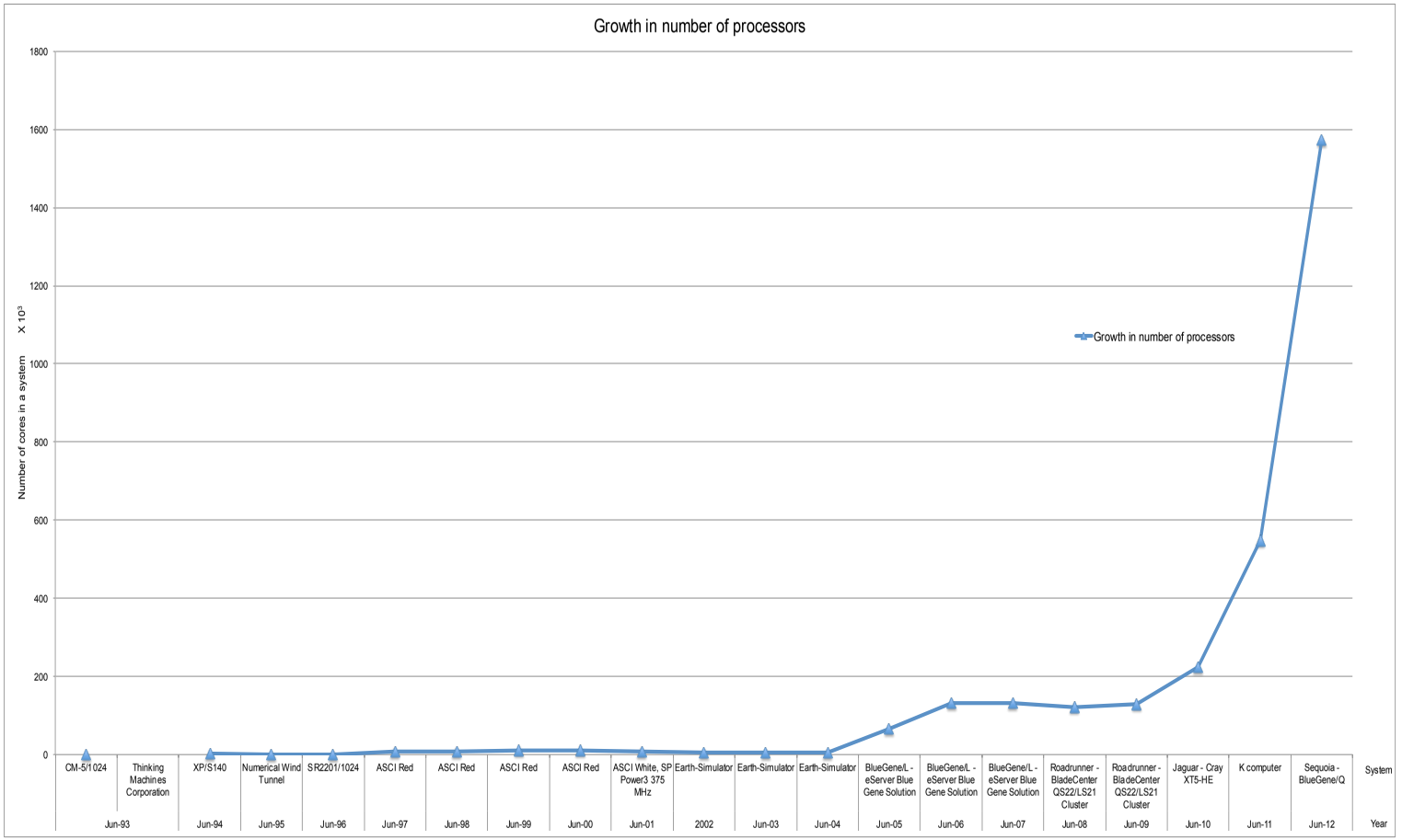

Figure 1.2: Performance growth in the number of processors on HPC systems 1993-2012

Rollback-recovery is one of the most widely used fault tolerance mechanisms for HPC systems. Rollback-recovery consists of checkpoint, failure detection and recovery/restart. However, rollback-recovery usually increases the wall clock execution time of HPC applications, thereby increasing the electrical energy used and the dollar cost of running HPC applications in both traditional HPC system and in HPC systems in the cloud.

Cloud computing offers new computing paradigms, capacity, and flexibility to HPC applications through provisioning of a large number of Virtual Machines (VMs) for computation-intensive applications using cloud services. It is expected that computation-intensive applications will increasingly be deployed and run in HPC systems in the cloud [18, 19]. For example, the Amazon Elastic Compute Cloud (Amazon EC2 [20] cluster recently appeared in the Top500 list.

The aim of this thesis is to provide fault tolerance for HPC systems in the cloud, particularly when Hardware as a Service (HaaS) is leased. We propose a framework, develop and implement the associated algorithms and test a solution using the C programming language. The solution proposed in this thesis covers hardware failures that occur after warning window.

1.1 Background

The history of HPC systems dates back to the 1960s when parallel and distributed computing were used to achieve high computational performance. Parallel computing uses shared memory to exchange information between processors while distributed computing uses distributed memory, with information shared between processors by message passing. Recently, it has become difficult to

distinguish between parallel and distributed systems, since parallel computers often have some distributed characteristics. This is clearly demonstrated in the Top500 list

Today, parallel and distributed computing systems with large numbers of processors are commonly known as HPC systems. HPC systems scale from a few hundred processors to hundreds of thousands of processors, such as the Clay Titan [2].

Computation-intensive applications are sometimes referred to as long-running applications and are most often scientific computations using mathematical models and quantitative analysis techniques running on HPC systems to analyse and solve large and complex scientific problems. Some of the scientific computations would have been too difficult to carry out without HPC systems, due to the capital cost, complexity or financial risk involved (for example, in nuclear weapons experiments).

With cloud computing, HPC systems are no longer limited to large organisations but are also available to individuals. Cloud computing [18] promises numerous benefits, including the fact that there is no need for up-front investment in the purchase and installation of equipment and software. An HPC system in the cloud is a good alternative to a traditional HPC system.

1.2 Motivation

As noted above, fault tolerance is one of the major challenges faced by cloud services for HPC applications. Evidence shows that a system with 100,000 processors will experience a processor failure every few hours [21, 9]. A failure occurs when a

hardware component fails and needs replacement, a software component fails or a node/processor halts or is forced to reboot, or software fails to complete its run. In such cases, an application utilising the failed component will fail.

In addition, HPC applications deployed in the cloud run on virtual machines, which are more likely to fail due to resource sharing and contention [19, 22]. Therefore, fault tolerance (FT) technology is particularly important for HPC applications running in cloud environments, because fault tolerance means there is no need to restart a long-running application from the beginning in the event of a failure, thereby reducing operational costs and energy consumption.

1.3 Identification of Key Research Problems

This section describes the three key challenges for HPC systems that are addressed by this thesis: 1) fault tolerance, 2) rollback-recovery and 3) energy efficiency. These are significant problems for HPC system communities. The thesis also aims to contribute to improvements in HPC systems in the cloud by providing a detailed study of current research in the field.

1.3.1 Fault Tolerance

Recent studies by Schroeder and Gibson [23, 9], Egwutuoha, *et al.* [11], and Yigitbasi, *et al.* [24] and the data sets provided in [3], show that hardware (processors, hard disk drive, integrated circuit sockets, and memory) causes more than 50% of the failures on HPC systems. These works also show that:

1. The failure rate is almost proportional to the number of CPUs (failure increases with the number of nodes and/or processors).

2. The intensity of the workload affects the failure rate [23, 9].

3. There is a time varying correlation with failure rate [24].

1.3.2 Rollback-Recovery

The rollback-recovery fault tolerance technique is commonly used by HPC systems communities [16, 25]. It tends to minimise the impact of failure on computation-intensive applications running in a HPC system when one or more computational nodes fail. A good example of rollback-recovery fault tolerance is checkpoint and restart. Checkpoint and restart allows computation-intensive problems that may take a long time to execute in HPC systems to be restarted from the checkpoint prior to failure in the event of errors or failures.

However, recent publications [23, 10, 26, 14] show that, with the steadily increasing number of components in today‘s HPC systems, applications running on HPC systems may not be able to achieve meaningful progress with the basic checkpoint and restart approach. This is because the system will spend most of its computational time in checkpoint, which is not part of the computational activities. A fault tolerant solution that will reduce the overhead of rollback-recovery is particularly important.

1.3.3 Energy Efficiency

HPC systems utilize large amounts of electrical power to operate their large numbers of processors, electronic components and other electrical parts that support constitute them. For example, it is estimated that IBM BlueGene/L which is built to use low power components, costs between $200,000 to $1.2 million

annually [27].

Because of this high energy utilisation, the HPC community created the Green500 [28] list. The performance of HPC systems is now measured not only in floating point operations per second (FLOPS) but also in FLOPS per watt. Clearly, the high energy utilisation of HPC systems presents a challenge that cannot be ignored in the design of fault tolerance for HPC systems. With energy efficient fault tolerant solution, electrical energy would be utilised in computational activities with minimisation of rework activities due to failure. Hence, the development of energy efficient fault tolerant HPC systems will make a significant improvement in terms of the energy utilisation and operational cost of HPC systems, particularly HPC systems in the cloud.

1.4 Contributions

In addressing these key research problems, the thesis makes the following major contributions to the field of HPC systems in the cloud:

1. We propose a fault tolerance framework for High Performance Computing (HPC) in the cloud [29] that can be used to build a reliable HPC systems in the cloud.

2. We propose a proactive fault tolerance approach to High Performance Computing (HPC) in the cloud. We develop an algorithm and corresponding software solution for HPC systems in the cloud to support users and researchers who lease HaaS [30].

3. We present an empirical analysis of the proposed solution that demonstrates

its cost effectiveness and electrical energy efficiency [31, 32]. The empirical analysis presented on this thesis is general and can be used to analysed similar cloud-based systems.

4. We analyse two cloud services that are usually used for HPC in the cloud and identify the most cost effective cloud service for a HPC system in the cloud [33]. This can assist in choosing HPC systems in the cloud that are cost effective.

1.5 Thesis Outline

The thesis is divided into six chapters, as shown in Figure 1.3. This chapter has described the background to, and motivation for, the research and has identified key research problems and contributions. Chapter 2 surveys fault tolerance approaches, analysis of failure rates and rollback feature requirements for modern HPC systems and evaluates over 20 previous works on checkpoint/restart facilities. Chapter 3 explains the theoretical background of the study, including definitions of the HPC in the cloud concept, architecture, related services and fault tolerance deficiencies. Chapter 4 presents a new proactive fault tolerance approach to High Performance Computing (HPC) in the cloud, including the algorithms, design approaches, implementations, evaluations and test results. Chapter 5 reports the cost analysis and energy utilisation of the approach, and Chapter 6 concludes the discussion and makes recommendations for future work.

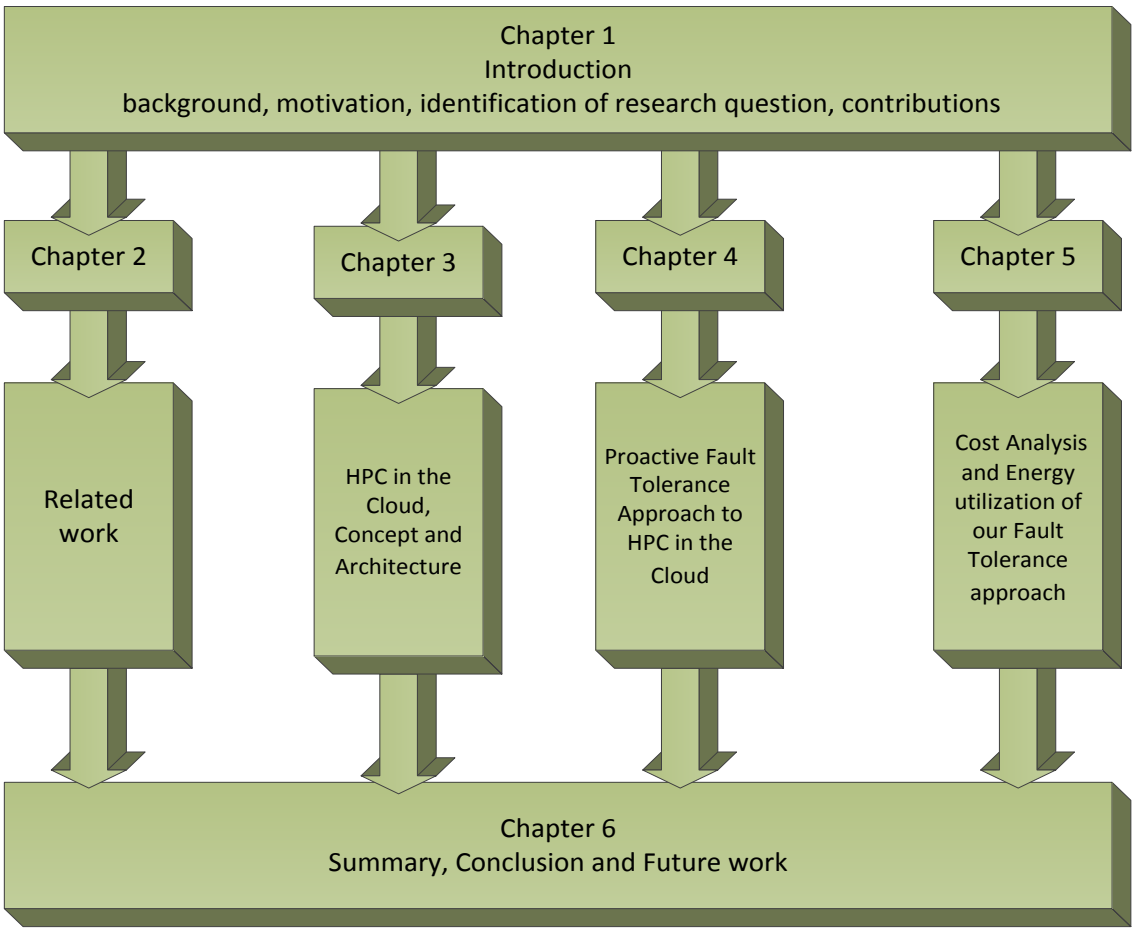

Figure 1.3: Thesis outline

Chapter 2

Related Work

"Fault tolerance is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components" [34]. Fault tolerance is highly desirable in HPC systems because it may ensure that computation-intensive applications are completed in a timely manner. In some fault tolerant systems, a combination of one or more techniques is used.

This chapter begins with an analysis of the failure rates of HPC systems. This is followed by a review of fault tolerance approaches that examines the issues associated with each approach in the context of HPC systems. Research efforts directed at reducing the time to checkpoint in persistent storage are briefly discussed. Much of the content of this chapter appears in previously published work [11, 32].

2.1 Analysis of Failure Rates of HPC Systems

Generally, failures occur as a result of hardware or software faults, human factors, malicious attacks, network congestion, increased intensity of workload (overload),

and, possibly, other unknown causes [35, 36, 1, 37]. These failures may cause computational errors. They may be *transient or intermittent*, but can still lead to *permanent failures* [38]. A *transient failure* causes a component to malfunction for a certain period of time, but it then disappears and the functionality of that component is fully restored. *An intermittent* failure appears and disappears sporadically; it never goes away completely, unless it is resolved. *A permanent failure* causes the component to malfunction consistently until it is replaced. A great deal of work has been done on understanding the causes of failure and we briefly review the major contributors to failure in this section. We also include our findings from the present study.

2.1.1 Software Failure Rates

Gray [35] analyzed outage/failure reports of Tandem computer systems between 1985 and 1990, and found that software failure caused about 55% of outages. Tandem systems were designed to be single-fault fault-tolerant systems, that is, systems capable of overcoming the failure of a single element (but not overlapping multiple failures). Each Tandem system consisted of 4 to 16 processors, 6 to 100 discs, 100 to 1000 terminals and their communication equipment. Systems with more than 16 processors were partitioned to form multiple systems and each of the multiple systems had 10 processors linked together to form an application system.

Lu [37] studied the failure log of three different architectures at the National Center for Supercomputing Applications (NCSA). The systems were:

1. A cluster of 12 SGI Origin 2000 NUMA (Non-Uniform Memory Architecture)

distributed shared memory supercomputers with a total of 1,520 Central Processing Units (CPUs),

2. "Platinum", a PC cluster with 1,040 CPUs and 520 nodes, and

3. "Titan", a cluster of 162 two-way SMP 800 MHz Itanium-1 nodes (324 CPUs).

In the study, five types of outages/failures were defined: software halt, hardware halt, scheduled maintenance, network outages, and air conditioning or power halts. Lu found that software failure was the main contributor to outage (59-83%), suggesting that software failure rates are higher than hardware failure rates.

Similarly, El-Sayed and Schroeder [39] studied field failure data of HPC systems available at [40]. The failure data studies were collected over 9 years period. They observed that there is significant relationship between network, environmental and software failures. They also observed that there is a significant increase in the probability of software failure after power issues had occurred.

Nagappan, *et al* [41] studied the failure log files of on-demand virtual computing lab at North Carolina State University [42]. Virtual computing lab at North Carolina State University is operated as private cloud with more than 2000 computers. In this study, they observed that system software contributes relatively small role in system failure. According to their finding, most of the recorded failures were caused by workload, license exhaustion and hardware failures.

2.1.2 Hardware Failure Rates

A large set of failure data, the computer failure data repository (CFDR) was also released by the USENIX Association [3]. It comprised the failure statistics of 22

HPC systems, including a total of 4,750 nodes and 24,101 processors collected over a period of 9 years at Los Alamos National Laboratory (LANL). The workloads consisted of large-scale long-running 3D scientific simulations that take months to complete. We have further analysed the data in order to reveal the systems failure rates in more detail. Figure 2.1 shows systems (2 to 24) with different configurations and architectures, with the number of nodes varying from 1 to 1024 and the number of processors varying from 4 to 6152. System 2 with 6152 processors recorded the highest number of hardware failures. Figure 2.1 also shows the number of failures and corresponding causes recorded over the period, represented by a bar chart. From the bar chart, it can be clearly seen that the failure rates of HPC systems increase linearly as the number of nodes and processors increases.

To further examine the failure rates, we selected seven HPC systems from the failure data repository [3]. The systems selected consist of five clusters with the highest number of totals CPUs and/or compute nodes, one Symmetric MultiProcessing (SMP) system with the highest number of CPUs, and the only Non-Uniform Memory Access (NUMA) system in the data repository. Figure 2.2 shows the analysis of failure rates of HPC systems with their system IDs.

As can be seen from Figures 2.1 and 2.2, more than 60% of the recorded failures that occurred on HPC systems are hardware failures.

Schroeder and Gibson [9, 23] analysed failure data collected at two large HPC sites: the data set from LANL RAS [3] and the data set collected over a period of one year at a large supercomputing system with 20 nodes and more than 10,000 processors. Their analysis suggests that:

1. The mean repair time across all failures (irrespective of failure types) is

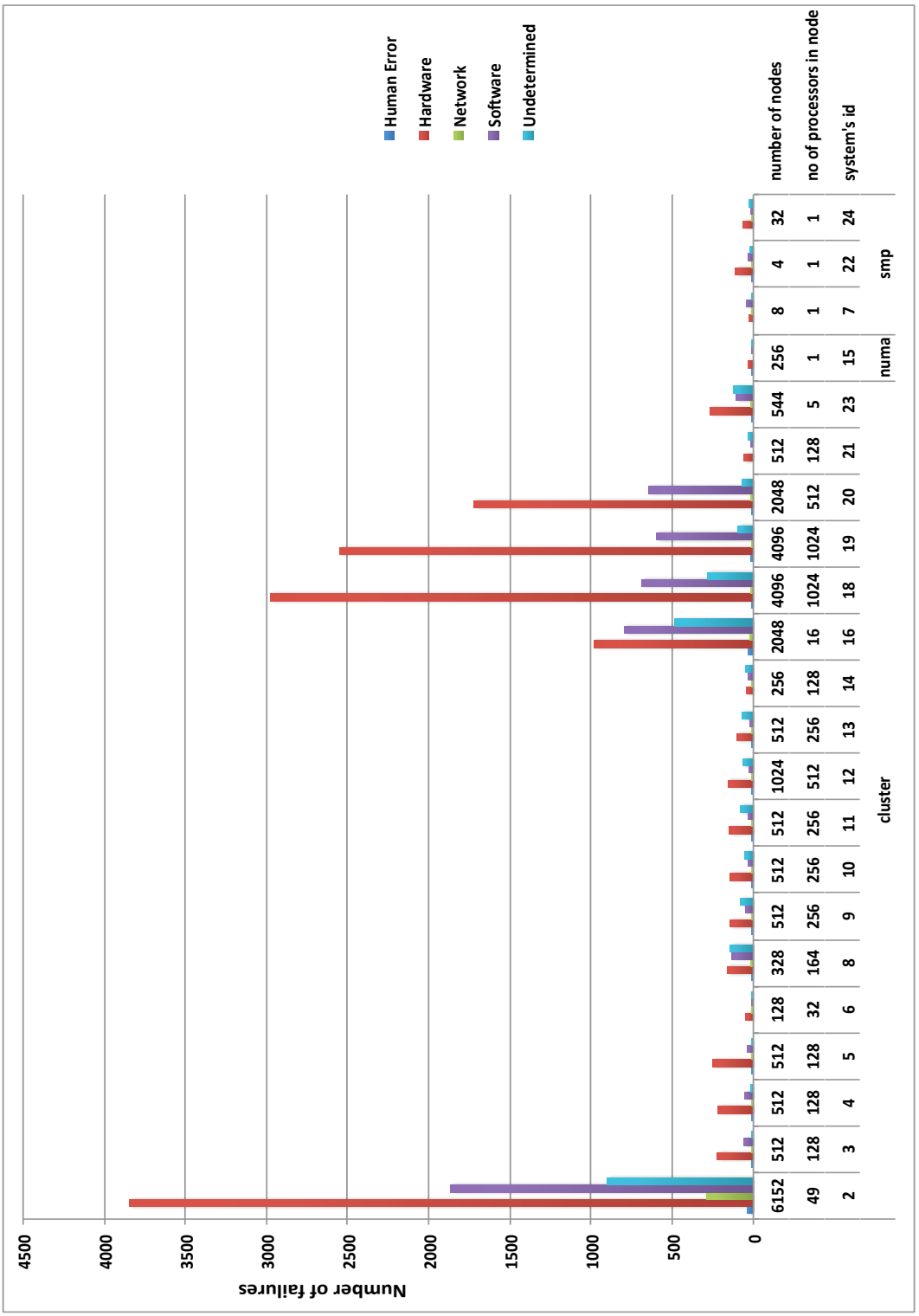

Figure 2.1: A simplified analysis of number of failures for each system according to CFDR data [3]

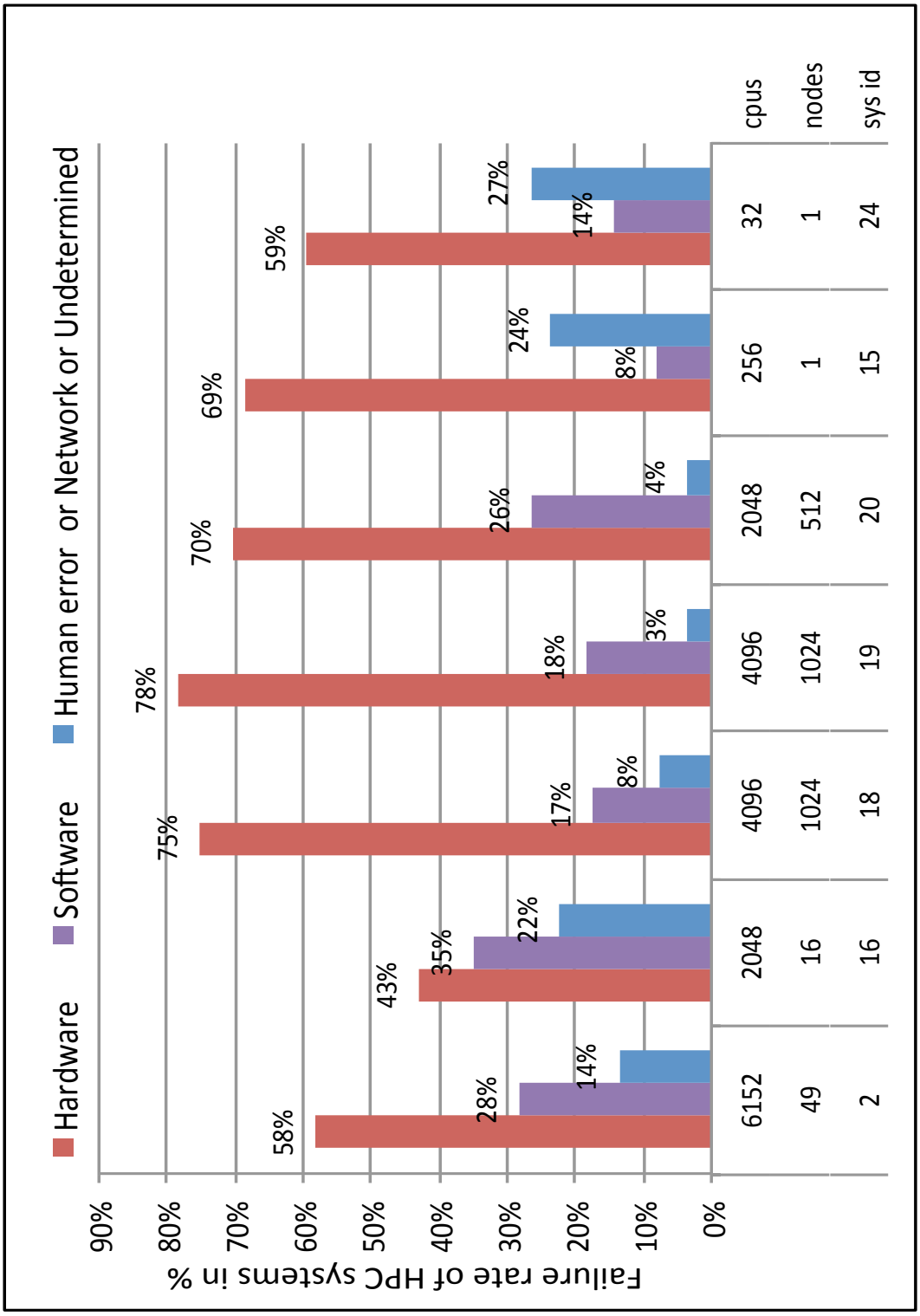

Figure 2.2: Failure rate of HPC systems with different CPUs and nodes

about 6 hours.

2. There is a relationship between the failure rate of a system and the applications running on it.

3. As many as three failures may occur on some systems within 24 hours.

4. The failure rate is almost proportional to the number of processors in a system.

Oliner and Stearley [1] studied system logs from five supercomputers installed at Sandia National Labs (SNL) as well as Blue Gene/L, which is installed at Lawrence Livermore National Labs (LLNL). The five systems were ranked in the Top500 supercomputers at the time of their study. The systems were structured as follows: 1) Blue Gene/L with 131,072 CPUs and a custom interconnect; 2) Thunderbird with 9,024 CPUs and an InfiniBand interconnect; 3) Red Storm with 10,880 CPUs and a custom interconnect; 4) Spirit (ICC2) with 1,028 CPUs and a GigEthernet (Gigabit Ethernet) interconnect; and 5) Liberty with 512 CPUs and a Myrinet interconnect. A summary of the systems is provided in Table 2.1. Although the raw data appeared to show that 98% of the failures were due to hardware, after the data were filtered, the analysis revealed that 64% of the failures were due to software.

Table 2.1: Summary of HPC systems studied by Oliner and Stearley [1]

No	System name	System configuration
1	Blue Gene/L	131,072 CPUs and custom interconnect
2	Thunderbird	9,024 CPUs and an InfiniBand interconnect
3	Red Storm	10,880 CPUs and a custom interconnect
4	Spirit (ICC2)	1,028 CPUs and a GigEthernet (Gigabit Ethernet) interconnect
5	Liberty	512 CPUs and a Myrinet interconnect

2.1.3 Human Error Failure Rates

Oppenheimer and Patterson [36] report that operator error is one of the largest single root causes of failure. According to their report, *Architecture and Dependability of Large-Scale Internet Services*, the failures occurred when operational staff made changes to the system, such as replacement of hardware, reconfiguration of the system, deployment, patching, software upgrade and system maintenance. They attributed 14-30% of failures to human error.

Thus we can conclude that almost all failures of computation-intensive applications are due to hardware failures, software failures and human error. It is difficult, however, to specify the single major cause of failures, since the analyses reported above were carried out:

1. using different systems on which different applications were running;
2. under different environmental conditions; and
3. using different data correlating periods and methods.

Consequently, effective fault tolerant HPC systems should address hardware and software failures as well as human error.

2.2 Review of Fault Tolerance Mechanisms for High Performance Computing Systems

Figure 2.3 shows an abstract view of the fault tolerance techniques used in this review. We use the feature modelling technique [43] to model this abstract

view because of its conceptual simplicity and because it makes it easy to map dependencies in an abstract representation. The most widely used fault tolerance techniques in HPC systems are migration methods, redundancy (hardware and software), failure masking, failure semantics and rollback-recovery techniques [44, 13, 26]. Each is briefly summarised below.

2.2.1 Migration Method

With recent advances in virtualisation technologies, migration can be categorised into two major groups: process-level migration and Virtual Machine (VM) migration. *Process-level migration* is the movement of an executing process from its current node to a different node. The techniques commonly used in process-level migration are *eager, pre-copy, post-copy, flushing and live* migration techniques [45]. *VM migration* is the movement of a VM from one node/machine to a new node. *Stop-and-copy* and *live* migration of VMs are the most commonly used techniques [46].

In the migration approach, the key idea is to avoid an application failure by taking preventive action. When a part of an application running on a node seems likely to fail (which may lead to failure of the whole application), that part of the application that is likely to fail is migrated to a safe node and the application continues. This technique relies primarily on accurate prediction of the location, time, and type of failure that will occur. Reliability, availability, and serviceability (RAS) log files are commonly used to develop the prediction algorithm [47]. RAS log files contain features that will assist in accomplishing RAS goals - minimal downtime, minimal unplanned downtime, rapid recovery

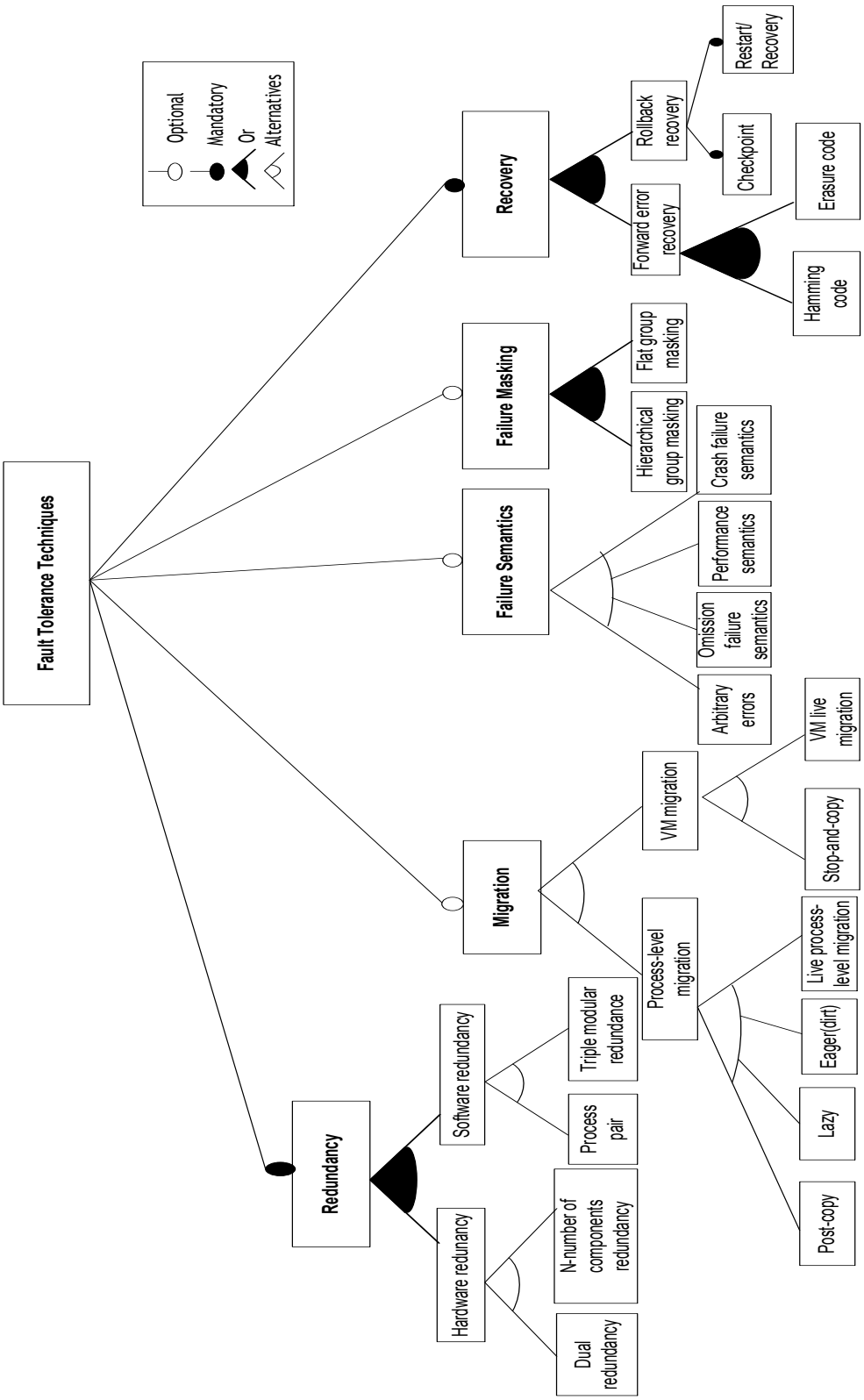

Figure 2.3: An abstract view of fault tolerance techniques with feature modeling

after a failure, and manageability of the system (the ease with which diagnosis and repair of problems can be carried out). Error events and warning messages are examples of information contained in a RAS log.

Failure types that have not been recorded in RAS log files will not be correctly predicted. It is still a challenge to build accurate failure predictors for petascale and exascale systems with thousands of processors and nodes [10]. A failure predictor may predict failures that will never occur; these are called ”false positives” and may fail to predict failures that do occur. Therefore, the migration method should be used with other fault tolerance techniques such as checkpoint/restart facilities in order to build robust fault tolerance HPC systems.

2.2.2 Redundancy

With physical redundancy techniques, redundant components or processes are added to make it possible for the HPC system to tolerate failures [13, 48]. The critical components are replicated (as spare) as, for example, in the Blue Gene/L and Tandem nonstop systems. In the event of hardware failure of one component, other components that are in good working order continue to perform until the failed part is replaced. Hardware redundancy is commonly used to provide fault tolerance to hardware failures. The process of voting may be employed as proposed in n (n >2) modular redundancy [12]. Usually, n=3, but some systems use n >3, along with majority voting.

Software redundancy can be grouped into two major approaches: process pairs and Triple Modular Redundancy (TMR). In the process pair technique, two types of processes are created, a primary (active) process and a backup (passive) process.

The primary and backup processes are identical but execute on different processors and the backup process takes over when the primary process fails. It uses module signal failure techniques or 'I am alive message' to detect failure [49]

In the TMR approach, three modules are created, they perform a process and the result is processed by a voting system to produce a single output. If any one of the three modules fails, the other two modules can correct and mask the fault. A fault in a module may not be detected if all three modules have identical faults because they will all produce the same erroneous output. To address that, N-version programming [50], and N self-checking [51] have been proposed. There are other methods as well, such as recovery blocks, reversible computation, range estimation, and post-condition evaluation [38]. N-version programming is also known as multiple version programming. In this approach, different software versions are developed by independent development teams, but with the same specifications. The different software versions are then run concurrently to provide fault tolerance to software design faults that escaped detection. During runtime, the results from different versions are voted on and a single output is selected. In recovery block techniques, N unique versions of the software are developed, but they are subjected to a common acceptance test. The input data are also checkpointed, before the execution of the primary version. If the result passes the acceptance test, the system will use the primary version, or else it will rollback to the previous checkpoint to try the alternative versions. The system fails if none of the versions passes the acceptance test. In N self-checking programming, N unique versions of the software are also developed, but each has its own acceptance test. The software version that passes its own acceptance test is selected through an acceptance voting system.

Software systems usually have a large number of states (upward of 10^{40}) [52] which implies that only a small part of the software can be verified for correctness.

2.2.3 Failure Masking

Failure masking techniques provide fault tolerance by ensuring that services are available to clients despite failure of a worker, by means of a group of redundant and physically independent workers. In the event of failure of one or more members of the group, the services are still provided to clients by the surviving members of the group, often without the clients noticing any disruption. There are two masking techniques used to achieve failure masking: hierarchical group masking and flat group masking [44]. Figure 2.4 illustrates the flat group and the hierarchical group masking methods.

Flat group masking is symmetrical and does not have a single point of failure; the individual workers are hidden from the clients, appearing as a single worker. A voting process is used to select a worker in event of failure. The voting process may introduce some delays and overhead because a decision is only reached when inputs from various workers have been received and compared.

In hierarchical group failure masking, a coordinator of the activities of the group decides within a group which worker may replace a failed worker in the event of failure. This approach has a single point of failure; the ability to effectively mask failures depends on the semantic specifications implemented [53].

Fault masking may create new errors, hazards and critical operational failures when operational staff fails to replace already failed components [54]. When failure masking is used, the system should be regularly inspected. However, there are

costs associated with regular inspections.

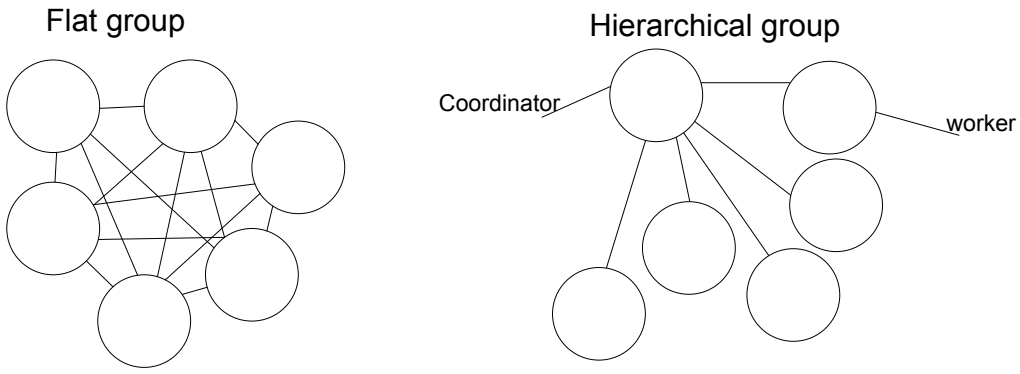

Figure 2.4: Flat group and hierarchical group masking

2.2.4 Failure Semantics

Failure semantics refers to the different ways in which a system designer anticipates the system can fail, along with failure handling strategies for each failure mode. This list is then used to decide what kind of fault tolerance mechanisms to provide in the system. In other words, with failure semantics [44], the anticipated types of system failure are built within the fault tolerance system and the recovery actions are invoked upon detection of failures. Some of the different failure semantics are omission failure semantics, performance semantics, and crash failure semantics.

Fail-stop failure semantics apply if the only failure that the designers anticipate from a component is for it to stop processing instructions, while behaving correctly prior to that [55]. Omission failure semantics apply if the designers expect a communication service to lose messages, with negligible chances that messages are delayed or corrupted. Omission/performance failure semantics apply when the designers expect a service to lose or delay messages, but with lesser probability

that messages can be corrupted.

The fault tolerant system is built based on foreknowledge of the anticipated failure patterns and it reacts to them when these patterns are detected; hence the level of fault tolerance depends on the likely failure behaviours of the model implemented. Broad classes of failure modes with associated failure semantics may also be defined (rather than specific individual failure types). This technique relies on the ability of the designer to predict failure modes accurately and to specify the appropriate action to be taken when a failure scenario is detected. It is not feasible, however, in any system of any complexity such as HPC systems, to predict all possible failure modes. For example, a processor can achieve crash failure semantics with duplicate processors. Failure semantics may also require hardware modifications [56]. Similarly, some of the nodes and applications failures which occur in HPC systems may be unknown to the fault tolerance in place. For example, a new virus may exhibit a new behaviour pattern which would go undetected even though it could crash the system [57]. An unidentified failure could also lead to stoppage.

2.2.5 Recovery

Generally, fault tolerance implies recovering from an error which otherwise may lead to computational error or system failure. The main idea is to replace the erroneous state with a correct and stable state. There are two forms of error recovery mechanisms: forward and backward error recovery.

Forward Error Recovery: With Forward Error Recovery (FER) [58] mechanisms, an effort is made to bring the system to a new correct state from which

it can continue to execute, without the need to repeat any previous computations. FER, in other words, implies detailed understanding of the impact of the error on the system, and a good strategy for later recovery. FER is commonly implemented where continued service is more important than immediate recovery, and high levels of accuracy in values may be sacrificed; that is, where it is required to act urgently (in, e.g., mission-critical environments) to keep the system operational.

FER is commonly used in flight control operation, where future recovery may be preferable to rollback-recovery. A good example of forward correction is fault masking, such as the voting process employed in triple modular redundancy and in N-version programming.

As the number of redundant components increases, the overhead cost of FER and of the CPU increases because recovery is expected to be completed in the degraded operating states, and the possibility of reconstruction of data may be small in such states [59]. Software systems typically have large numbers of states and multiple concurrent operations [60], which implies that there may be low probability of recovery to a valid state. It may be possible in certain scenarios to predict the fault; however, it may be difficult to design an appropriate solution in the event of unanticipated faults. In complex systems, FER cannot guarantee that state variables required for the future computation are correctly re-established following a fault; therefore, the result of the computations following an error occurrence may be erroneous. FER is also more difficult to implement compared to rollback-recovery techniques, because of the number of states and concurrent operations. In some applications, a combination of both forward and rollback-recovery may be desirable.

Rollback-recovery: Rollback-recovery consists of checkpoint, failure detection

and recovery/restart. A checkpoint [38] is a snapshot of the state of the entire process at a particular point such that the process could be restarted from that point in the event that a subsequent failure is detected. Rollback-recovery is one of the most widely used fault tolerance mechanism for HPC systems, probably because:

1. Failures in HPC systems often lead to fail-stop of the application execution.
2. Rollback-recovery technique uses a fail-stop model (discussed in Chapter 3) whereby a failed process can be restarted from saved checkpoint data.

In addition, rollback-recovery is used to protect against failures in parallel systems because of the following major advantages [61]:

1. It allows computational problems that take days to execute in HPC systems to be checkpointed and restarted (from the last saved) in event of failures instead of restarting the application from the beginning.
2. It allows load balancing and for applications to be migrated to other nodes, or even another system where computation can be resumed if an executing node fails.
3. It has lower implementation cost compared to hardware redundancy.
4. It reduces electrical power consumption compared to hardware redundancy.

The major disadvantage is that rollback-recovery does not protect against design faults (software fault). After rollback the system continues processing as it did previously. This will recover from a transient fault, but if the fault was caused by a design fault, then the system will fail and recover endlessly, unless

an alternate computational path is provided during the recovery phase. Note that some states cannot be recovered, if all components use checkpointing, an invalid message can be sent to other applications, causing them to roll back and then consume fresh, correct results. This is similar to invalidation protocols in distributed caches as discussed in [62]. Despite these limitations, the need to ensure that computation-intensive parallel applications complete successfully necessitates its use. Two major techniques are used to implement rollback-recovery: checkpoint-based rollback-recovery and log-based rollback-recovery. These techniques and over 20 checkpoint/restart facilities are discussed in detail in our earlier work [11].

A great deal of research has been carried out on checkpoint and restart but some issues [26, 8] are yet to be addressed:

1. The number of transient errors could increase exponentially because of the exponential increase in the number of transistors in integrated circuits in HPC systems.

2. Some faults may go undetected (e.g., software errors), which would lead to further erroneous computations in long-running applications, potentially resulting in complete failure of an HPC system.

3. Correctable errors may also lead to software instability due to persistent error recovery activities.

4. How to reduce the time required to save the execution state, which is one of the major sources of overhead.

2.3 Rollback-recovery Feature Requirements for HPC Systems

We define the following rollback-recovery feature requirements which are important to HPC fault tolerance systems [63, 64, 65]. We do not claim that these features are necessary or sufficient, since future technological developments may force additional requirements or, conversely, eliminate some of them from the list. These feature requirements will be used to evaluate the applicability of different checkpointing/restart facilities listed in this survey.

1. *Transparency:* A good fault tolerance approach should be transparent; ideally, it should not require source code or application modifications, nor re-compilation and re-linking of user binaries, because new software bugs could be introduced into the system.

2. *Application coverage:* The checkpointing solution must have a wide range of applications coverage, to reduce the likelihood of implementing and using multiple different of checkpointing/restart solutions which may lead to software conflicts and greater performance overhead.

3. *Platform portability:* It must not be tightly coupled to one version of an operating system or application framework, so that it can be ported to other platforms with minimal effort.

4. *Intelligence/Automatic:* It should use failure prediction and failure detection mechanisms to determine when checkpointing/restart should occur without the user's intervention. Whenever this feature is lacking, users are required to initialise the checkpointing/restart process. Although system users may

be trained to carry out the checkpoint/restart activities, human error can still be introduced if system users have to initiate the checkpoint or recovery processes [66].

5. *Low overhead:* Low overhead: The time to save checkpoint data should be significantly shorter compared to the 40 to 60 minutes that has been recorded on some of the top500 HPC systems [26]. The size of the checkpoint should be minimised.

2.4 Checkpoint-based Rollback-recovery Mechanisms

In checkpoint-based rollback-recovery, an application is rolled back to the most recent consistent state using checkpoint data. Due to the global consistency state issue in distributed systems [16], checkpointing of applications running in this type of environment is quite difficult to implement compared to uniprocessor systems. This is because different processors in the HPC system may be at different stages in the parallel computation and thus require global coordination, but it is difficult to obtain a consistent global state for checkpointing. (Due to drift variations in local clocks it is generally not practical to use clock-based methods for this purpose.) A consistent global checkpoint is a collection of local checkpoints, one from every processor, such that each local checkpoint is synchronised to every other local checkpoint [67]. The process of establishing a consistent state in distributed systems may force other application processes to roll back to their checkpoints even if they did not experience failure, which, in turn, may cause other processes to roll back to even earlier checkpoints. This effect is called

the *domino effect* [68]. In the most extreme case this domino effect may lead to the only consistent state being the initial state, clearly something that is not very useful. There are three main approaches to dealing with this problem in HPC systems: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing. We briefly discuss each of these below.

Uncoordinated checkpointing allows different processes to do checkpoints when it is most convenient for each process thereby reducing overhead [69]. Multiple checkpoints are maintained by the processes, which increase the storage overhead [70]. With this approach it might be difficult to find a globally consistent state, rendering the checkpoint ineffective. Therefore uncoordinated checkpointing is vulnerable to the domino effect and may lead to undesirable loss of computational work.

Coordinated checkpointing guarantees consistent global states by enforcing each of the processes to synchronize their checkpoints. Coordinated checkpointing has the advantages that it makes recovery from failed states simpler and is not prone to the domino effect. Storage overhead is also reduced compared to uncoordinated checkpointing, because each process maintains only one checkpoint on stable permanent storage. However, it adds overhead because a global checkpoint needs internal synchronization to occur prior to checkpointing. A number of checkpoint protocols have been proposed to ensure global coordination. A non-blocking checkpointing coordination protocol was proposed [71] to ensure that applications which would render coordinated checkpointing inconsistent, are prevented from running. Checkpointing with synchronised clocks [72] has also been proposed. The DMTCP [65] checkpointing facility is an example that implements a coordinated checkpointing mechanism.

Communication-induced checkpointing (CIC) (also called *message induced checkpointing)* protocols do not require that all checkpoints be consistent, and still avoids the domino effect. With this technique, processes perform two types of checkpoints: local and forced checkpoints. A local checkpoint is a snapshot of the local state of a process, saved on persistent storage. Local checkpoints are taken independently of the global state. Forced checkpoints are taken when the protocol forces the processes to make an additional checkpoint. The main advantage of CIC protocols is that they allow independence in detecting when to checkpoint. The overhead is reduced because a process can take local checkpoints when the process state is small. CIC, however, has two major disadvantages: (1) it generates large numbers of forced checkpoints with resulting storage overhead; and (2) the data piggybacked on the messages generates considerable communications overhead.

2.4.1 Log-based Rollback-recovery Mechanisms

Log-based rollback-recovery mechanismsare similar to checkpoint-based rollback-recovery except that messages sent and received by each process are recorded in a log. The recorded information in the message log is called a *determinant*. In the event of failure, the process can be recovered using the checkpoint and reapplying the logged determinants to replay its associated non-determinants events and to reconstruct its previous state. There are three main mechanisms: *pessimistic, optimistic,* and *casual message logging mechanisms*. A complete review of these techniques can be found in [16]. *Pessimistic message logging protocols* record the determinant of each event to stable storage before it is allowed to trigger the execution of the application. The main advantages of

this method are: (1) that the recovery of the failed application is simplified by allowing each process of the failed application to recover to the known state in relationship with other applications; and (2) that only the latest checkpoint is stored, while older ones are discarded. However, the process is blocked while the event determinant is logged to a stable state, which incurs an overhead.

In *optimistic logging protocols*, the determinant of each process is logged to volatile storage; events are allowed to trigger the execution of application before logging of the determinant is concluded. This method is good as long as the fault did not occur between the non-determinant event and subsequent logging of the determinant event. Consequently, overhead is reduced because volatile storage is used; however, the recovery process may not be possible if the volatile store loses its content due to power failure.

Casual message logging protocols utilise the advantages of both pessimistic and optimistic message logging protocols. Here the messages logs are stored in stable storage when it is most convenient for the process to do so. In casual message logging protocols, processes piggyback the non determinant messages on the local storage. Therefore only the most recent message log is required for restarting and multiple copies are kept, making the logs available in event of multiple machine failure. Further discussion of the piggyback concept in casual message logging protocols is found in [46, 16]. The main disadvantage of the casual message logging protocol is that it requires a more complex recovery protocol.

2.5 Taxonomy of Checkpoint Implementation

In this section, three major approaches to implementing checkpoint/restart systems are described: application-level implementation, user-level implementation and system level implementation. The implementation level refers to how it integrates with the application and platform. Figure 2.5 shows the taxonomy of checkpoint implementation.

In application-level implementations, the programmer or some automated pre-processor injects the checkpointing code directly into the application code. The checkpointing activities are carried out by the application. Basically it involves inserting checkpointing code where the amount of state that needs to be saved is small, saving the checkpoint in persistent storage, and restarting from the checkpoint if a failure had occurred [73]. Application-level checkpointing accommodates heterogeneous systems, but lacks transparency, which is usually available with a kernel-level or a user-level approach. The major challenge in this approach is that it requires the programmer to have a good understanding of the applications to be checkpointed. (Note that programmers (users) may not always have access to the application source code.) The Cornell Checkpoint(pre) Compiler (C3) [74] is an excellent implementation of application-level checkpointing.

With user-level implementations, a user-level library is used to do the checkpointing and the application programs are linked to the library. Some typical library implementations are Esky [75], Condo [76], and libckpt [77]. This approach is usually not transparent to users because applications are modified, recompiled and re-linked to the checkpoint library before the checkpoint facility is used. The major disadvantages of these implementations are that they impose limitations

on which system calls applications can make. Some shell scripts and parallel applications may not be checkpointed even though they should be because the library may not have access to the system files [25].

Checkpoint/restart may also be implemented at the system level, either in the OS kernel or in hardware. When implemented at the system level, it is can be transparent to the user and usually no modification of application program code is required. Applications can be checkpointed at any time under control of a system parameter that defines the checkpoint interval. Examples of system-level implementations include CRAK [78], Zap [79], and BLCR [63]. These offer a choice of periodic and non-periodic mechanisms. It may be challenging to checkpoint at this level because not all operating system vendors make the kernel source code available for modification, but if a package for a particular OS exists, then it is very easy to use, as the user does not have to do anything once the package is installed. One drawback, however, is that a kernel level implementation is not portable to other platforms [74].

Hardware-level checkpointing uses digital hardware to customise a cluster of commodity hardware for checkpointing. It is transparent to users. Different hardware checkpointing approaches have been proposed, including SWICH [80]. Hardware checkpointing could be implemented with FPGAs [81]. Additional hardware is required and there is the overhead cost of building specialised hardware if this approach is selected. Hardware-level checkpointing is also not portable, which is a significant disadvantage.

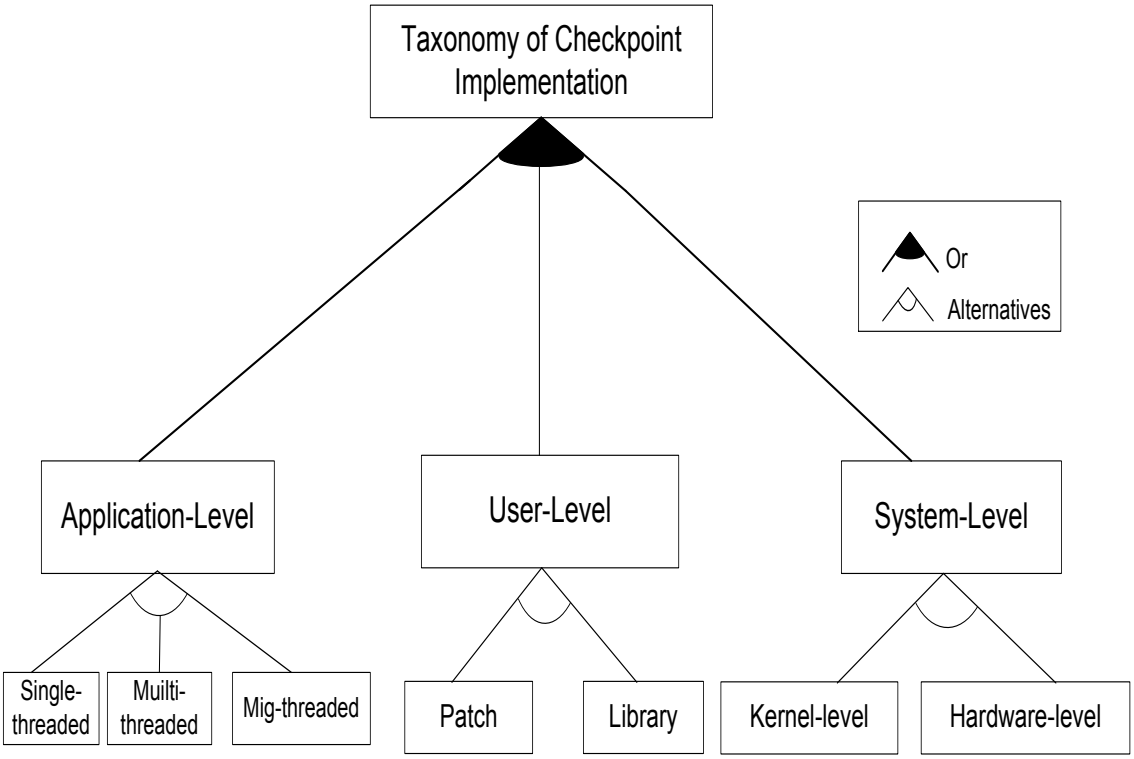

Figure 2.5: Taxonomy of Checkpoint Implementation

2.6 Reducing the Time for Saving the Checkpoint in Persistent Storage

There are techniques designed to reduce the overhead cost in saving the checkpoint data when writing the state of a process to persistent storage. This is, of course, one of the major sources of increased performance overhead. We briefly discuss here some of these techniques.

In *incremental checkpointing,* only the portion of the program that has changed since the last saved process [77] is saved. The unchanged portion can be restored from previous checkpoints. The overhead of checkpointing is reduced in this

process. However, the recovery could be complex because the multiple incremental saved files are kept and grow with each checkpoint. This can be limited to at most n increments, after which a full checkpoint is saved as is commonly done in storage backup systems.

Flash-based *Solid State Disk (SSD)* memory may also be used as a persistent store for the checkpoint data. SSD is based on semiconductor chips rather than magnetic media technology such as hard drives, to store persistent data. SSD has lower access times and latency compared to hard disks, however it has limited read/write cycles of about 100,000 times and data may be difficult to access after this threshold has been exceeded [82]. Wear leveling is used to minimise this problem [83].

Copy-on-write[77] techniques reduce the checkpoint time by allowing the parent process to fork a child process at each checkpoint. The parent process continues execution while the child process carries out checkpointing activities. The technique is useful in reducing checkpoint time when the checkpoint data is small. However, there is a performance degradation if the size of the checkpoint data is large because the child and parent processes will compete for computer resources (e.g., memories and network bandwidth).

Data compression reduces the size of checkpoint data to be saved on the storage, thereby reducing the time to save the checkpoint data. However it takes time and computer resources to carry out the compression. Plank [84] showed that checkpointing can benefit from data compression techniques. However, data compression depends on the compression ratio and application state. If the amount of data to compress is large it consumes more memory, which will result in performance degradation of the executing application. When data are compressed,

it will require more time to restart the application due to decompression time.

2.7 Survey of Checkpoint/Restart Facilities

A number of surveys of checkpoint/restart facilities have been carried out, such as checkpoint.org [85], Kalaiselvi and Rajaraman [67], Byoung-Jip [86], Roman [61], Elnozahy *et al* [16], and Maloney and Goscinski [64]. None presents summarised information of currently available facilities that would easily aid research in this area. Hence, we summarise and tabulate our findings in Appendix A, Table A.2. This provides an overview of existing checkpoint/restart facilities that have been proposed by researchers for different computing platforms (the website addresses of the checkpoint facilities surveyed are included in the table). The criteria used in this survey were based on the rollback-recovery feature requirements for HPC systems discussed above. Table A.2 is concise and includes information that provides the HPC checkpointing research community with a good overview of the systems that have been proposed. The selected checkpoint/restart facilities covered include recent work that is currently widely used.

2.8 Summary

In this chapter, we have provided an overview and analysis of failure rates of HPC systems. Although it is difficult to determine the single most common root cause of failure, we also conclude that computation-intensive applications are most frequently interrupted by hardware failures, software failures or human error. We conclude that a good fault tolerance mechanism should be able to mitigate or, in

some cases, even eliminate the consequences of failure. We have surveyed fault tolerance mechanisms (redundancy, migration, failure masking and recovery) for HPC and identified the pros and cons of each technique.

Recovery techniques are discussed in detail, and over 20 checkpoint/restart facilities have been surveyed. Research efforts directed at reducing the time for saving the checkpoint in persistent storage are presented. The rollback feature requirements that were identified are used to evaluate them and the results are provided in a tabular format for ease of reference. The web site of each surveyed checkpoint/restart facility is also provided for further investigation.

While much work has been done on fault tolerance in traditional HPC systems [77, 44, 13, 45], in particular on checkpoint and restart techniques [16, 61], recent publications [10, 23, 87, 88] have shown that present fault tolerance (e.g., checkpoint and restart techniques) may not be effective in HPC systems with more than 10,000 nodes.

Fault tolerance in the context of HPC systems in the cloud is essentially different in nature because computation-intensive applications are executed in virtual machines. Since cloud computing is relatively new, there was no available published work on fault tolerance for HPC systems in the cloud, particularly when HaaS is leased.

Chapter 3

HPC Systems in the Cloud, Concepts and Architecture

Cloud computing is a new computing paradigm. In this chapter, we provide the theoretical background to this thesis by defining the HPC in the cloud. We examine relevant concepts, architecture, cloud services and fault tolerance issues, with particular focus on HPC systems in the cloud. This chapter contains extracts from our previous work [30, 29] but also includes refinements and new contributions.

3.1 Cloud Computing Architectures

The published literature [89, 18, 90, 91] and other sources [92] contain different definitions of cloud computing. We adopt the definition of Foster *et al.* [91], which captures the four-layer architecture of cloud computing:

> *"Cloud computing is a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet."*

Cloud computing promises numerous benefits [18, 93], including no up-front investment, scalability and greater research collaboration efficiency. It reduces development and deployment time, staff (e.g., administrators) and hardware requirements, which can result in significant cost saving. With cloud computing, the need to plan ahead for provisioning Information Technology (IT) infrastructure is greatly reduced, because it is difficult to predict IT service demand, which can depend on unforeseeable events. HPC systems can also be more easily scaled in cloud computing than in traditional HPC systems.

With the cloud computing pay-as-you-go pricing model, scientists and researchers can lease cloud services such as Infrastructure as a Service (IaaS) and Hardware as a Service (HaaS) for computation-intensive applications. These service models avoid job queuing, which is common in traditional HPC systems. The price model is also attractive when compared to traditional HPC systems that involve large capital investment, administrative issues and allocation policies. The infrastructure costs mean that only large universities and research communities can justify the expense of acquiring HPC systems.

A number of publications [94, 18, 95 96, 97] has shown that HPC systems in the cloud are a good alternative to HPC systems. It is expected that more computation-intensive applications will be deployed and run in HPC systems in the cloud. As a case in point, the Amazon Elastic Compute Cloud (Amazon EC2) cluster recently appeared in the Top500 list [2], which suggests likely future demand for HPC systems in the cloud.

The two major key players in cloud computing:

1. Cloud providers and

2. Cloud users.

3.1.1 Cloud Providers

Cloud providers (e.g., Baremetalcloud [98] and Amazon EC2 [20]) create and provide cloud services (hardware, virtual machines, storage, etc.) to which consumers can subscribe, usually on a pay-as-you-go basis.

3.1.2 Cloud Users

Cloud users are organisations or individuals (e.g., scientists) who subscribe to a selection of cloud services. This allows them either to operate their IT in the cloud at a reduced cost or to create products (or perform experiments) that would have been difficult without the possibilities inherent in the cloud.

3.2 Cloud Service Models

Cloud service models allow cloud providers to offer specialised services that can represent significant savings for some users. The benefits of running computation-intensive applications in HPC systems in the cloud include scalability, a pay-as-you-go price model and easy access. Although there are different views on cloud computing architectures [18, 90, 91, 99], we favour a four-layer architecture for cloud computing that is consistent with the definition provided in [91] and with the four major cloud computing service models [18, 100, 32]. The four-layer architecture for cloud computing model gives a broad and conceptually simple yet comprehensive view of major cloud services. This four-layer architecture for cloud computing, shown in Figure 3.1, consists of:

1. Software as a Service (SaaS)
2. Platform as a Service (PaaS)
3. Infrastructure as a Service (IaaS)
4. Hardware as a Service (HaaS).

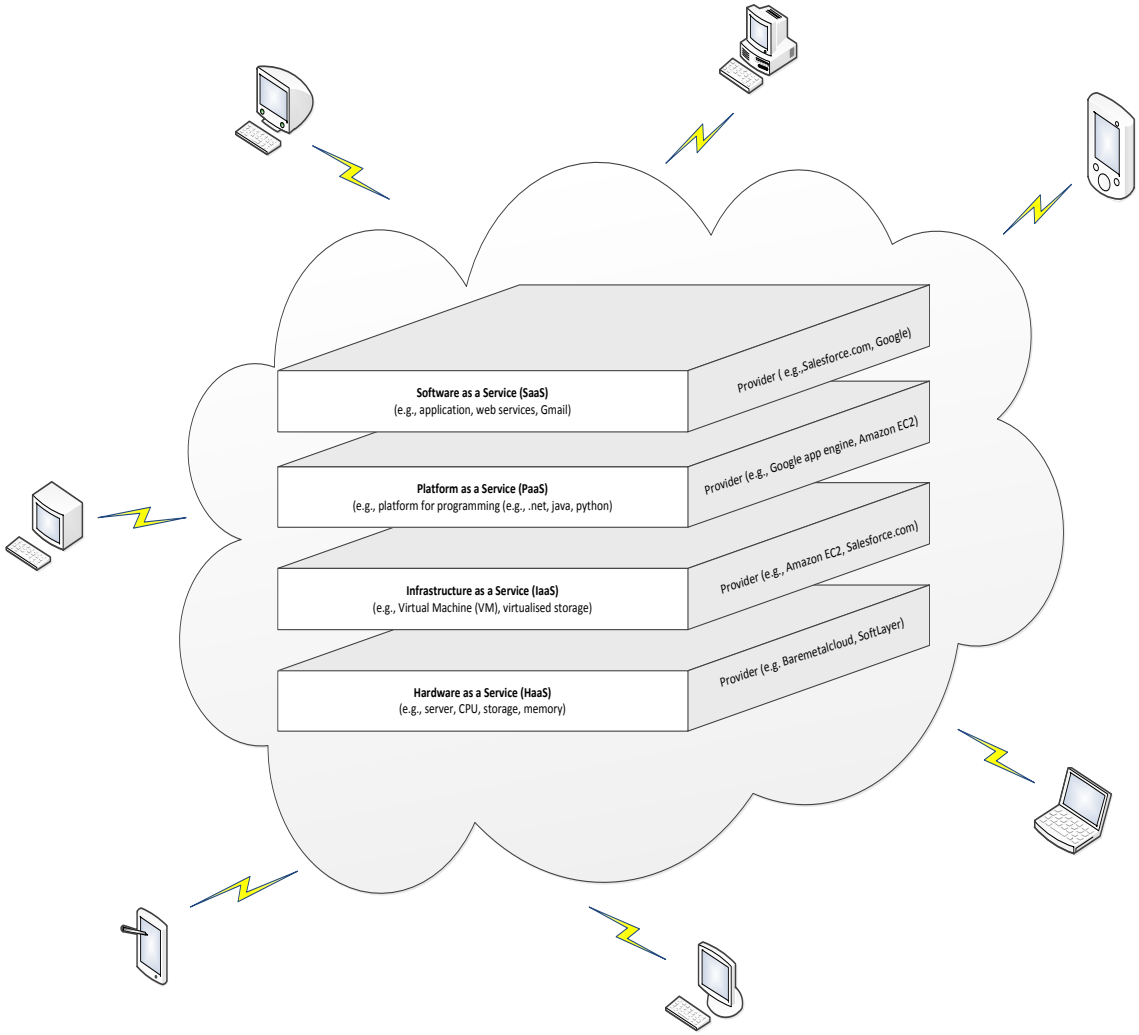

Figure 3.1: Cloud computing architecture

3.2.1 Software as a Service (SaaS)

SaaS is the highest abstraction level in the cloud. It offers cloud users ready-to-use online applications that are already deployed in the cloud. This layer is hidden from users and is managed by the SaaS providers. Users do not know where or how these applications are deployed; they simply use them. SaaS cloud applications can be accessed via the Internet with any Internet-ready device such as a laptop, smartphone, or tablet. This enables relatively unsophisticated clients to perform complex tasks by shifting the real work, transparently to the user, into the cloud. Good examples of SaaS include Microsoft Word online provided by Microsoft and the commonly used Gmail (email services) provided by Google.

3.2.2 Platform as a Service (PaaS)

PaaS provides cloud users with a fully configured and managed computing platform for ready-to-run custom software developed by users. Each PaaS platform is targeted to software developed in a specific programming language or software framework and is ready to execute corresponding builds. Good examples of PaaS are Microsoft Azure and Google App Engine [101, 102]. PaaS cloud users deploy and run their software without the need to set up virtual machines and software stacks or to think about scalability or clustering, and often without even knowing how many computers or CPUs their application will run on. Fault tolerance is provided for this level of service.

3.2.3 Infrastructure as a Service (IaaS)

IaaS is similar to HaaS but virtual machines, rather than real hardware, are rented out. IaaS cloud users have to install, configure, and maintain the virtual machines they rent and are free to choose the operating system and software stack they install in their virtual machines. Often IaaS users make use of pre-installed and preconfigured virtual machine images supplied by their provider as base installation. Cloud users of IaaS do not have root access to the hardware but only to the virtual machines they lease. A good example of a cloud provider who offers IaaS for HPC applications is Amazon Elastic Compute Cloud (EC2). Amazon EC2 offers cluster compute instances for HPC applications. IaaS providers usually provide fault tolerance to users.

3.2.4 Hardware as a Service (HaaS)

In this case, the cloud provider basically rents out 'bare-metal' hardware (e.g., server/host and storage). Notable HaaS providers include Baremetalcloud [98], and SoftLayer [103]. Cloud users connect to HaaS via the Internet, install and configure (e.g., virtual machines) the server or service they have leased. Cloud users choose HaaS because it gives them full control of the server (host), operating system, and software/hardware stack, as well as the number of virtual machines they execute on it. Research communities lease HaaS for computation-intensive and/or data-intensive applications and configure HPC systems according to their needs. Consequently, computation-intensive applications that were traditionally run on HPC systems can now be executed in the cloud. However, fault tolerance is not provided at this service level.

3.3 Types of Clouds

One way of classifying cloud computing is: public cloud, private cloud, community cloud, and hybrid cloud [93, 104]. Each of these cloud types offers different advantages to cloud users. We briefly explain these types and the advantages that each offers in relationship to HPC systems in the cloud. Figure 3.2 shows the types of cloud computing with an illustrative diagram of an HPC system in the cloud.

3.3.1 Public Cloud

In public cloud computing, computer resources (e.g., virtual machines, CPU, storage, bandwidth, databases) are made available to the public by a cloud service provider such as Amazon Web Services and Googles App Engine. These resources may be provided free (e.g., Cloudo [105]) or based on a pay-as-you-go model, such as the services from Amazon and Google. Public clouds promise the major benefits of cloud computing [18]. In this thesis, we deal with HPC systems in the cloud (public cloud).

3.3.2 Private Cloud

A private cloud is hosted in the data centre of a company and provides its cloud services only to internal users or its partners. A private cloud may offer higher security than public clouds and can provide cost savings if it utilises otherwise unused capacity in an existing data centre [106]. Large organisations, including large universities, can host HPC systems in their private cloud.

Figure 3.2: HPC System in the Cloud with computation-intensive application (our illustration shows MPI application executing in HPC system in the cloud)

3.3.3 Community Cloud

In community cloud computing, cloud services are provided to a specific group of organisations that share common goals or missions (such as security requirements, policy, or compliance considerations) [89]. The cost of running the cloud services is shared among the participants. Community clouds can leverage service compliance to provide highly secure cloud environments among trusted communities [107]). Community clouds offer higher security and cost savings than public clouds, if managed properly, but less security than a private cloud.

3.3.4 Hybrid Cloud

This is a combination of private, public, or community clouds, as shown in Figure 3.2. Hybrid clouds allow organisations to find an optimal balance between cost of IT operations and inherent security risks by running highly confidential applications on private clouds and utilising public clouds for peak loads or other computations. For example, a healthcare organisation may use a private cloud managed by its internal IT unit to meet security requirements for healthcare data and public cloud services to fulfill organisational goals with lower security requirements. Hybrid clouds are still at an early stage of development, and inter-operability among clouds is a major challenge that has to be overcome so that users can manage their hybrid cloud environments without added complexity with the very tools they use to manage their private clouds.

3.4 HPC Systems in the Cloud

Cloud computing presents a significant opportunity for HPC systems. The price model means that HPC systems are no longer limited to large organisations but are also available to smaller enterprises, and even individuals, for computation-intensive computation. An HPC system in the cloud is deployed on top of virtualisation technology, as shown in Figure 3.3.

Virtualisation allows installation of multiple operating systems and software stacks (virtual machines) on one physical computer, allowing them to execute simultaneously but fully segregated from each other.

HPC in the cloud typically runs jobs (computation-intensive applications) and other services (e.g., resource management and file system services) on a head node and compute nodes using a Message Passing Interface (MPI) to communicate between different nodes.

3.4.1 Head Node

The head node on an HPC system in the cloud is usually a virtual machine that is configured to act as a point of contact between the HPC system and the outside network. It is usually seen as the heart of the cluster (HPC system) and performs the roles of job submission, network management, user login, access point to compute nodes, and control. In theory, users do not need to access individual compute nodes of an HPC system; rather, they depend on the head node and the job submission and other resource management tools to monitor and retrieve data and results. A head node failure may result in the loss of the application and failure of the entire HPC system. The possibility of network partition and

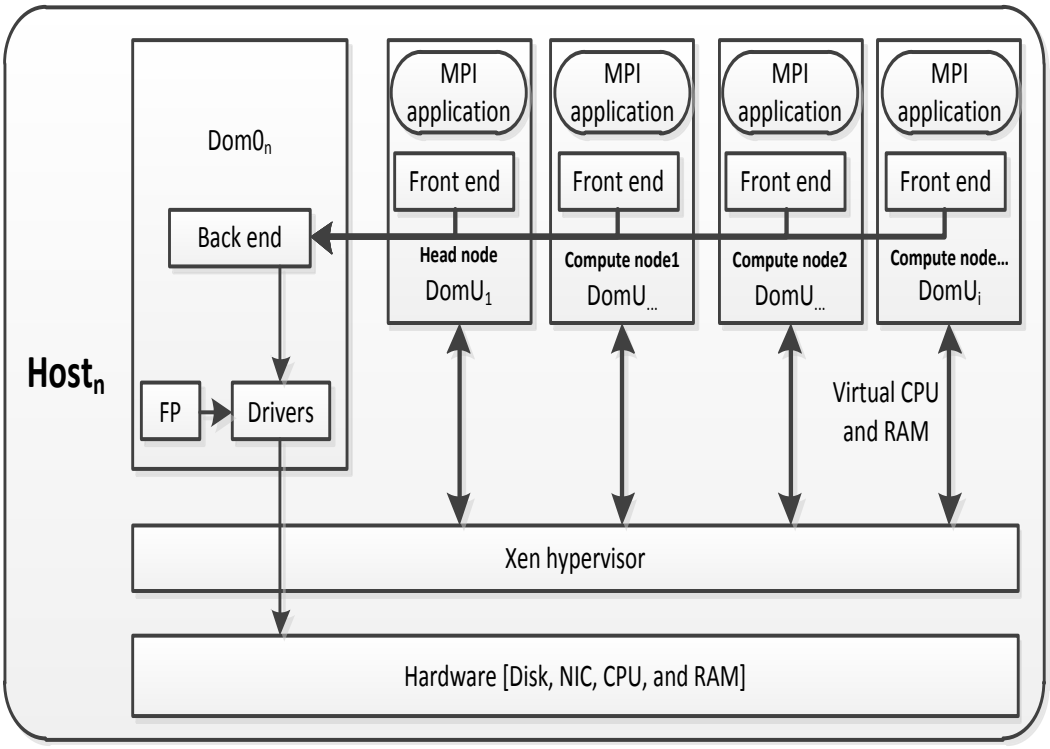

Figure 3.3: Host and Xen virtualization

how it impacts the notion of a head node is out of the scope covered in this thesis. The interested reader is referred to [15].

3.4.2 Compute Nodes

A compute node in an HPC system in the cloud is essentially an independent virtual machine configured to accept jobs from the head node and produce corresponding results. In theory, only the head node has direct access to the compute node. The head node may use secure protocol such as Secure Shell (SSH) transport layer protocol [108] to connect to the compute node. Each compute node running a computation-intensive application on HPC in the cloud is a single point of failure for the HPC system; that is, failure of a compute node running

a computation-intensive application may lead to loss of system and application state.

3.4.3 Message Passing Interface

The Message Passing Interface (MPI) [109] is the common parallel programming industry standard on which most parallel applications are written. With MPI, distributed and parallel virtual machines with distributed memories on HPC systems in the cloud communicate and share their data with other virtual machines by sending and receiving messages. MPI (e.g., OpenMPI, MPICH) provides two modes of operation: running or failed. That is, it follows the fail-stop model. An example of MPI application is Portable, Extensible Toolkit for Scientific Computation (PETSc) [110] that is used for modelling in scientific applications such as acoustics, aerodynamics, brain surgery, medical imaging, ocean dynamics and oil recovery. In this thesis we used OpenMPI implementation [111] because OpenMPI is an open source MPI-2 implementation that is widely used by HPC community across all platform. OpenMPI is developed and maintained by a collaboration of academic, research and industry partners.

3.4.4 Fail-stop Model

The fail-stop model assumes that systems components (e.g., compute node, processor) have failed completely when they are not producing output, have halted, restarted or stopped. A heart-beat mechanism is commonly used to detect failures. Most HPC systems assume a fail-stop model [55]. MPI applications running in HPC systems in the cloud can fail at any point of execution due to the hard-

ware, software and virtual machines that make up HPC systems in the cloud. This thesis focuses on the fail-stop model because HPC systems in the cloud, particularly when HaaS is leased, do not have fault tolerance in place. Fail-stop model is the most commonly experienced failure in HPC systems [15]

3.4.5 Virtualisation

Virtualisation is the enabling technology for HPC systems in the cloud. Virtualisation allows installation of multiple operating systems and software stacks (virtual machines) on physical computers (hosts) and configured virtual machines to form HPC systems in the cloud. There are several competing virtualisation technologies, most notably Xen [112] and the Linux Kernel Virtual Machine (KVM) [113]. In this study we used Xen virtualisation technology because of its features, which are briefly described below.

Xen hypervisor[112] is an open-source industrial standard virtualisation technology that is widely used by researchers [114]. As shown in Figure 3.3, Xen hypervisor provides a low level interaction between virtual machines usually called *DomainU* (guest machine), and the physical hardware. *Domain0* or the control domain is the virtual host environment. Xen hypervisor supports *full virtualisation* and *para-virtualization* modes. The full virtualisation mode allows virtual machines to run unmodified operating systems (e.g., Windows XP), but the hardware must support hardware-assisted virtualisation technology. Para-virtualisation allows the guest operating system to be modified for performance reasons [114].

3.5 Challenges for HPC Systems in the Cloud

In addition to the challenges mentioned in Chapter 1, HPC systems in the cloud face two additional challenges:

1. System Reliability
2. Cost

3.5.1 System Reliability

With an increase in the number of processors, integrated circuit sockets and compute nodes, the overall system Mean Time Between Failure (MTBF) in large-scale HPC systems may typically be reduced to just a few hours [115, 11, 23].

MTBF is a primary measure of system reliability and is defined as the probability that the system performs without deviations from agreed-upon behaviour for a specific period of time [116]. The reliability of a component is given as

$$\textit{Reliability function} = \frac{n(t)}{N} = \frac{\textit{failure free elements}}{\textit{number of elements at time=0}} \tag{3.1}$$

The reliability of elements connected in series

$$R_s = \prod_{n=1}^{m} e^{-\lambda_i t} \tag{3.2}$$

and, the reliability of elements connected in parallel is given as

$$R_p = 1 - \prod_{n=1}^{m} (1 - e^{-\lambda_i t}) \tag{3.3}$$

If we assume that in a system of m components, the $MTBF$ of any component i is independent of all other components, the reliability R of the system is

$$R = \frac{1}{MTBF_1} + \frac{1}{MTBF_2} + ... + \frac{1}{MTBF_m} \tag{3.4}$$

If $MTBF_1 = MTBF_2 = ... = MTBF_m$ then,

$$R = \frac{component\ MTBF}{m} \tag{3.5}$$

Availability is the degree to which a system or component is operational and able to perform its designed function [87].

$$\frac{Availability}{(MTBF + MTTR)} \tag{3.6}$$

where $MTTR = Mean\ Time\ To\ Repair$

For example, following a certain threshold, in a system with a large number of components, the system reliability can decrease, as illustrated in Figure 3.4. The diagram also shows how the value of the MTBF affects reliability (e.g., MTBFs of 100,000 and 1,000,000 hours).

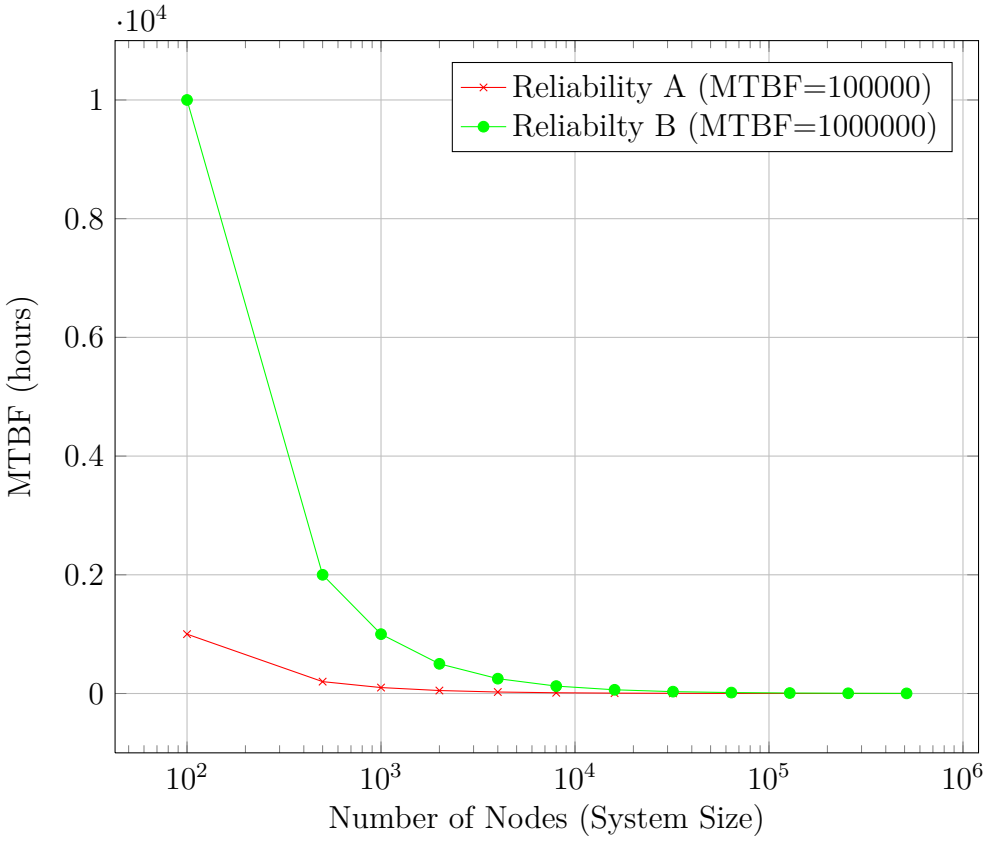

Figure 3.4: Reliability levels of two systems with MTBF of 10^5 and 10^6 as a function of the number of nodes

HPC applications deployed in cloud environments run on virtual machines, which are more likely to fail due to resource sharing and contention. Therefore, HPC systems in the cloud face three major types of failures;

1. Hardware failures
2. Software failures
3. Virtual machine failures

3.5.2 Cost

The price model for HPC systems in the cloud is attractive, particularly when compared to traditional HPC systems that require huge investments of capital and

involve administrative issues and allocation policies. However, the cost of running HPC application on HPC systems in the cloud may be high if the cloud services are not well understood or if non-cost-effective cloud services are chosen. If the dollar cost of running HPC applications in the cloud (when a non-cost-effective cloud service is chosen) is high compared to that of a traditional HPC system, then the benefits of running computation-intensive applications on the cloud may not be realised. HPC systems research communities are concerned about the cost and computational performance of different cloud services.

3.6 Summary

This chapter has presented the conceptual background of the types of HPC systems in the cloud that are the subject of this thesis. We have explained cloud computing architectures, described their benefits and identified the two main categories of cloud players (cloud providers and cloud users). We have explained the various cloud service models, SaaS, PaaS, IaaS and HaaS, discussed their challenges with respect to HPC systems in the cloud, and presented the four types of cloud computing available for HPC systems.

HPC systems in the cloud usually consist of a head node and compute nodes. The head node and compute nodes are virtual machines configured to fit their roles. The head node usually handles the user login, resource management, jobs assignment and network file system. Compute nodes are normally accessed through the head node. Compute nodes perform the computation-intensive job assigned by the head and produce results. The head node and compute nodes communicate through the Message Passing Interface. Failures of the head node and/or compute

node(s) commonly lead to a failure of the HPC system.

We also identified two additional challenges: system reliability (hardware failures, software failures and virtual machine failures) and cost (what cloud service is cost effective for HPC systems?). This thesis addresses cost, reliability and the other HPC challenges identified in Chapter 1.

Chapter 4

Proactive Fault Tolerance Approach to HPC Systems in the Cloud

This chapter describes a proactive fault tolerance approach to HPC systems in the cloud. It presents an account of our design and implementation approaches, cost model, mathematical analysis and test results. Some of the content is extracted from our earlier work [31, 29], with some additions and modifications.

4.1 Introduction

HPC systems are currently used by scientists and researchers in both industry and university laboratories. Computation-intensive applications requiring large amounts of computing power are executed in HPC systems. Fields of application [117, 4, 118] of HPC systems are shown in Table 4.1. The table shows that HPC system has wide range of applications.

Table 4.1: HPC systems applications

Field	**Application**
Medicine	Medical imaging (e.g., 3D ultrasound real-time X-ray) Monitoring brain activities (e.g., Magneto Encephalography (MEG)) Brain data analysis
Financial	Investment decision Derivatives trading Monte Carlo simulations
Pharmaceutical	Molecular modelling Drug discovery Drug design
Oil and Gas	Reservoir modelling and simulation Deep oil recovery 3D imaging processing Upstream oil and gas exploration
Bioinformatics	Storing, retrieving large data set Development of genetic algorithms Statistical analysis of large data set Image processing, sequence analysis, sequence alignments
Weather	Climate change Weather forecast Big data analysis Meteorology
Research	Nuclear weapons research Auto crash simulations Aerodynamics research

In general, most of these are parallel and/or distributed applications that are

computation-intensive. Drug recovery processes that can take up to 15 years from compound synthesis in the pharmaceutical laboratory to drug production can be reduced to a few weeks with HPC systems [4]. Because HPC systems usually require huge investment capital in hardware [2]. Scientists and researchers must sometimes wait in long queues to access shared HPC systems.

Amazon EC2 [20] and recent publications [119, 120, 19] have shown that HPC systems in the cloud are a good alternative for computation-intensive applications that are usually executed in traditional HPC systems. Cloud service providers, however, do not manage the virtual machines provided on the HaaS. Due to the large number of virtual machines and electronic components in HPC systems in the cloud, any fault during the execution would result in re-running the applications, which costs time, money and energy. Fault tolerance is one of the major challenges that HPC systems in the cloud face today, particularly when HaaS is leased.

Moreover, HPC applications deployed in cloud environments run on virtual machines, which are more likely to fail due to resource sharing and contention. Therefore, fault tolerance technology is particularly important for HPC applications running in cloud environments, because it can prevent restarting, thereby reducing operational costs and energy consumption.

Hardware redundancy is used to provide fault tolerance to hardware failures. In the event of hardware failure of one component, other components that are in good working order continue to perform until the failed part is replaced. With hardware redundancy fault tolerance techniques, redundant compute nodes are added to make it possible for the HPC systems to tolerate failures [121, 14, 122]. Riesen *et al* proposed a concept of redundant computing for HPC systems in [14]. In redundant computing, all compute nodes are replicated. Hsieh [123]

proposed an optimal task allocation and hardware redundancy policy, while [122] and [121] proposed a fixed-number of hardware redundancy schemes to achieve higher reliability in HPC systems. Consequently, the dollar cost of running computation-intensive application in the cloud with redundant computing fault tolerance will be high.

A reactive fault tolerance technique is commonly used for computation-intensive applications in classical grid computing through checkpoints and restart [124, 125]. However, it usually increases the wall clock execution time of HPC applications. Reactive fault tolerance techniques allow computation-intensive applications (which may take hours or days to complete) to log their intermediate results and states at checkpoints during their execution. Once a failure occurs, the application can be restarted from the checkpoint prior to the point of failure, rather than from the beginning. The frequency at which a component or application fails is an important measure in fault tolerance. It has been predicted that in peta-scale computing the MTBF is short; i.e., an application running on a peta-scale system will be interrupted by failure more often, with the MTBF decreasing as the reciprocal of the number of nodes [10, 23, 11].

In this study, we develop a proactive fault tolerance approach to HPC systems in the cloud to reduce the wall clock execution time in the presence of faults and dollar cost of running computation-intensive application. In particular, we make the following contributions:

1. We develop a generic fault tolerance algorithm for HPC systems in the cloud that does not rely on a leased spare node prior to prediction of a hardware failure.

2. We analyse and compare the dollar costs of four different fault tolerance solutions for running computation-intensive applications: (a) an ideal fully reliable system (reliability = 1), (b) a fault tolerance checkpoint scheme, (c) a migration-based scheme, and (d) the scheme we proposed. Our analysis allows HPC users leasing HaaS from a service provider to select a particular fault tolerance policy in line with budget and plans.

3. We derive a cost model for executing computation-intensive applications on HPC systems in the cloud. To the best of our knowledge, this is the first detailed cost model for executing computation-intensive application on HPC systems in the cloud particularly when HaaS is leased.

4. We analyse the dollar cost of provisioning spare nodes to assess the value of our approach.

4.2 Benefit of HaaS

This study focuses on Hardware as a Service (HaaS) because of the benefits HaaS offers [31]. In HaaS, the cloud provider basically rents out 'bare-bones' hardware (e.g., server/host and data). Cloud users connect to this service via the Internet, install and configure (e.g., VMs) the server they have leased. Cloud users choose HaaS because it gives them full control of the server, operating system, and software stack, as well as the number of VMs they execute on it. Research communities can easily lease HaaS for computation-intensive and/or data-intensive applications and configure HPC systems according to their needs. It has been shown that HaaS is a cost effective service for HPC systems in the

cloud [31].

4.3 Proactive Fault Tolerance for HPC Systems in the Cloud

The cloud provides pools of computing resources as services via the Internet using a pay-as-you-go price model that eliminates initial costly capital investments in hardware and infrastructure procurement. Until recently, HPC systems have been out of the reach of most research communities. Research and academic communities can now leverage the benefits of the cloud price model for their computation-intensive applications that traditionally run in dedicated HPC environments. HaaS providers lease out the 'bare bone' hardware, such as computers, data servers, and storage, while cloud users are responsible for configuring and maintaining the services. Performance and other types of trade-off can be more easily made when there is more complete control of the applications that execute in the HPC environment (both software and hardware stacks). Cloud service providers do not commonly provide fault tolerance mechanisms at this level.

Proactive fault tolerance uses an avoidance mechanism to tolerate faults. It achieves this by relying on a system log and health monitoring facilities. The system log and health monitoring provide information about the hardware/software state [126]. Health monitoring of hardware has recently attracted attention in fault tolerance communities because sensors are installed on modern hardware to monitor, for example, the processor temperature and fan speeds. This information is used to predict future failures.

Our proactive fault tolerance for HPC systems in the cloud requires four types of modules:

1. Monitoring module with lm-sensors
2. Failure predictor
3. Proactive fault tolerance policy module
4. Controller module.

These are explained in the following sections:

4.3.1 Monitoring Module with Lm-sensors

Many hardware failures can be predicted/detected in recent processors. Sensors installed in modern processors can be used to monitor CPU temperature, fan speeds and other parameters [127, 126]. Variations in the monitored parameters can adversely affect the performance and reliability of systems. We use the lm-sensors [128] package that provides tools, libraries, and drivers for monitoring these parameters. The libsensors library is used to access the values of the monitored parameters. It provides user-space support for the hardware monitoring drivers and console tools that report sensor readings. Lm-sensors allows easy setting of sensor limits. We selected lm-sensors because most HPC systems run Linux, and lm-sensors use Linux operating system kernel drivers. We used lm-sensors to develop FTDaemon which can be easily deployed on an HPC system in the cloud. Our methods, however, may easily be generalised to other operating system platforms.

Centrally monitoring the health of all the nodes in an HPC system with over 100,000 processors would a impose heavy overhead on the network as well as on the

HPC system. Therefore, we have designed FTDaemon to reduce the monitoring overhead, by having each node/host monitor its hardware by periodically reading its parameters. In our prototype, the FTDaemon running on each computing node/host collects lm-sensors information (e.g., processor temperature) every 600 milliseconds (the user can also set this interval to a higher value). An alarm is triggered whenever the monitored parameters exceed the maximum set values. The alarm prompts the reading of the sensors' values and a computation to determine if failure is likely to occur.

4.3.2 Failure Predictor

The FTDaemon runs on each host in user space. It uses rule-based prediction techniques [129, 130, 131], in which the future failure situation is determined by periodically reading the sensors' values. The current values are compared against the set maximum operating conditions obtained from sensor data repository.

The sensor data repository provides reliability information on all sensors installed on the system (e.g., temperature, fan speed and voltage). The basic idea in reliability information provided in the sensor data repository, derived from probability theory, is that a given component such as a CPU has operating conditions (normal, maximum and critical values) which, if violated, may result in the failure of the CPU [116]. Severity weight values are used to determine the severity level, as shown in Table 4.2. All the monitored sensors events are assigned a severity weight of -1, 0, 1 or 2 to represent normal, maximum, critical and error respectively. The info events are generated when the sensor is operating at normal state. The warning event is associated with maximum state value while

the critical event is associated with prediction of future failure. The error reading is associated with sensor data that are not available. For example, when the predictor wants to read the temperature of CPU4 and this is not available, it generates an error.

Table 4.2: Lm-senors generated events

Sensor State	**Severity weight**	**Severity level**
Normal	-1	Info
Maximum	0	Warning
Critical	1	Critical
Error	2	Error reading

The result obtained by comparing the current sensor's values with maximum set thresholds is used to determine if a failure is likely to occur soon. The predictor has a vector input of the four sensors: temperature T, voltage V and fan speed F; and CPU utilisation C. We chose to monitor CPU temperature, voltage, fan speed and CPU utilisation because high values of these parameters adversely affect performance of the HPC systems which may results to system failure. For example, overheating of CPU temperature would lead to automatic shutdown of CPU [39]. The combination of the four parameters increases positive failure prediction [132, 133]. Figure 4.2 illustrates our failure predictor model.

We use set theory [134] and associate the input valuables. The possible severity weight values that can be obtained from sensors' reading can be represented as:

- $T_i \in$ {-1, 0, 1, 2}
- $F_i \in$ {-1, 0, 1, 2}

- $V_i \in$ {-1, 0, 1, 2}
- $C_i \in$ {-1, 0, 1, 2}

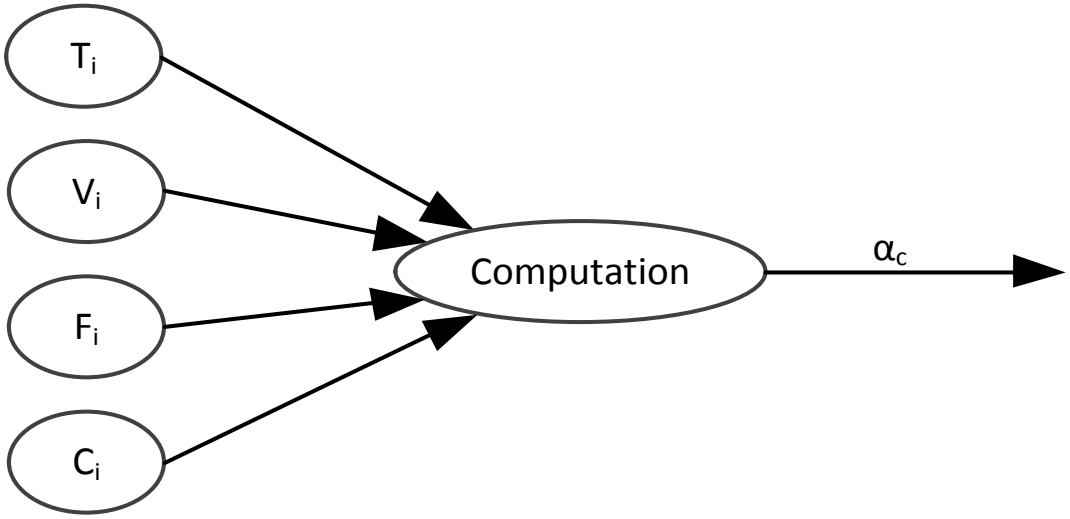

Figure 4.1: Rule-based predictor model

The sample algorithm 1 monitors the CPU temperature. The CPU fan speed, CPU voltage and CPU utilisation are monitored with a similar algorithm. In formulating the prediction rules, we use Boolean function which can be represented with two values: 0 and 1 to represent warning and critical operating state. Applying the Boolean function, a potential failure is predicted when three or four of the four CPU parameters are operating in a critical state. Similarly, a warning message is issued when two of the monitored parameters are in critical states. The construction of the prediction rule is shown in Appendix A, Table A.1. The critical operating condition occurs when three of the measured parameters are operating at critical states. The equivalent representations for future failure prediction and warning state are shown below using set theory [134] and a fault tree technique [135]:

$$\alpha_c = \textit{Future failure state} = \begin{cases} T_{ci} \cap V_{ci} \cap F_{ci} \cap C_{ci} \\ (T_{ci} \cap V_{ci} \cap F_{ci}) \cup C_{ci} \\ (T_{ci} \cap V_{ci} \cap C_{ci}) \cup F_{ci} \\ (T_{ci} \cap F_{ci} \cap C_{ci}) \cup V_{ci} \\ (V_{ci} \cap F_{ci} \cap C_{ci}) \cup T_{ci} \end{cases}$$

where:

- T_{ci} = CPU critical temperature
- V_{ci} = CPU critical voltage
- F_{ci} = CPU critical fan speed
- C_{ci} = CPU critical utilisation

For warning state:

$$\alpha_w = \textit{Warning State} = \begin{cases} T_{wi} \cap V_{wi} \\ V_{wi} \cap C_{wi} \\ C_{wi} \cap F_{wi} \\ T_{wi} \cap C_{wi} \\ T_{wi} \cap F_{wi} \end{cases}$$

where:

- T_{wi} = CPU maximum reliable temperature
- V_{wi} = CPU maximum reliable voltage
- T_{wi} = CPU maximum reliable fan speed
- C_{wi} = CPU maximum reliable utilisation

Algorithm 1

1: set timer **TRUE**
2: **while** timer **TRUE** # for all the FTDaemon running on the computer hosts
3: **do**
4: read sensor value T_i; # CPU temperature
5: compute for C_iT_i; # C_iT_i = measured variables
eg CPU temperature, voltage, fan speed and utilisation.
Compute the operating state
6: **if** $C_iT_i = 0$ or 1 **then**; # If operating at maximum or critical state exit
7: **break**

4.3.3 Proactive Fault Tolerance Policy

The goal of the proactive fault tolerance policy is to reduce the impact of failure on the execution of a computation-intensive application. We defined and implemented three policies:

1. Lease an additional node/host from the service provider
2. Relinquish the unhealthy node
3. Notify the administrator to take action.

When failure is predicted, the FTDaemon can proceed either to lease an additional host or to inform the administrator. The default policy is to lease an additional host/server and to log the details of the newly leased host with the head host. The head host maintains a database of all hosts. The functionality of the head host is transferred to newly leased host in the event of head host being predicted to fail. The 'relinquish the unhealthy host policy' is executed after migration of virtual machines from the unhealthy to the newly leased host.

4.3.4 The Controller Module

The controller module implements the policies listed above. A controller module is installed on all hosts. This technique provides future failure location information and allows immediate action by the host that is about to fail. The FTDaemon invokes this controller module when a failure is predicted. The controller module contacts the service provider and provides the service provider with the credentials (e.g., user name and password) that are required in the leasing process. FTDaemon collects the required credentials from the user during the startup of FTDaemon. Our implementation requires only the user name and the password,although more credentials information can be added.

After leasing the additional host, the controller module carries out live migration of the virtual machines from the unhealthy host to the newly leased host. It also logs the details of the additional node with the head host. On completion of migration of the virtual machines, the controller module also installs the FTDaemon on the newly leased host.

Figure 4.3 shows the architecture of our system. Xen hypervisor is a virtualisation software which is installed on each of the host. This allows multiple para-virtualised Operating Systems (OS) to be installed on each host. A para-virtualised OS is an operating system which its kernel is modified to work in virtual machine [136]. The $Dom0_0, \ldots, Dom0_n$, usually called domain zeros are the host operating systems. They run the management console and have special privileges to access the hardware. FTDaemon runs on the host operating systems. The backend and FTDaemon, as shown in Figure 4.3, communicate to the hardware through the drivers. The $DomU_0, \ldots, DomU_n$ (unprivileged domains) are the guest virtual

machines. The guest VMs (compute nodes) are configured to form a cluster with head node and compute nodes. The guest virtual machines execute the computation-intensive applications.

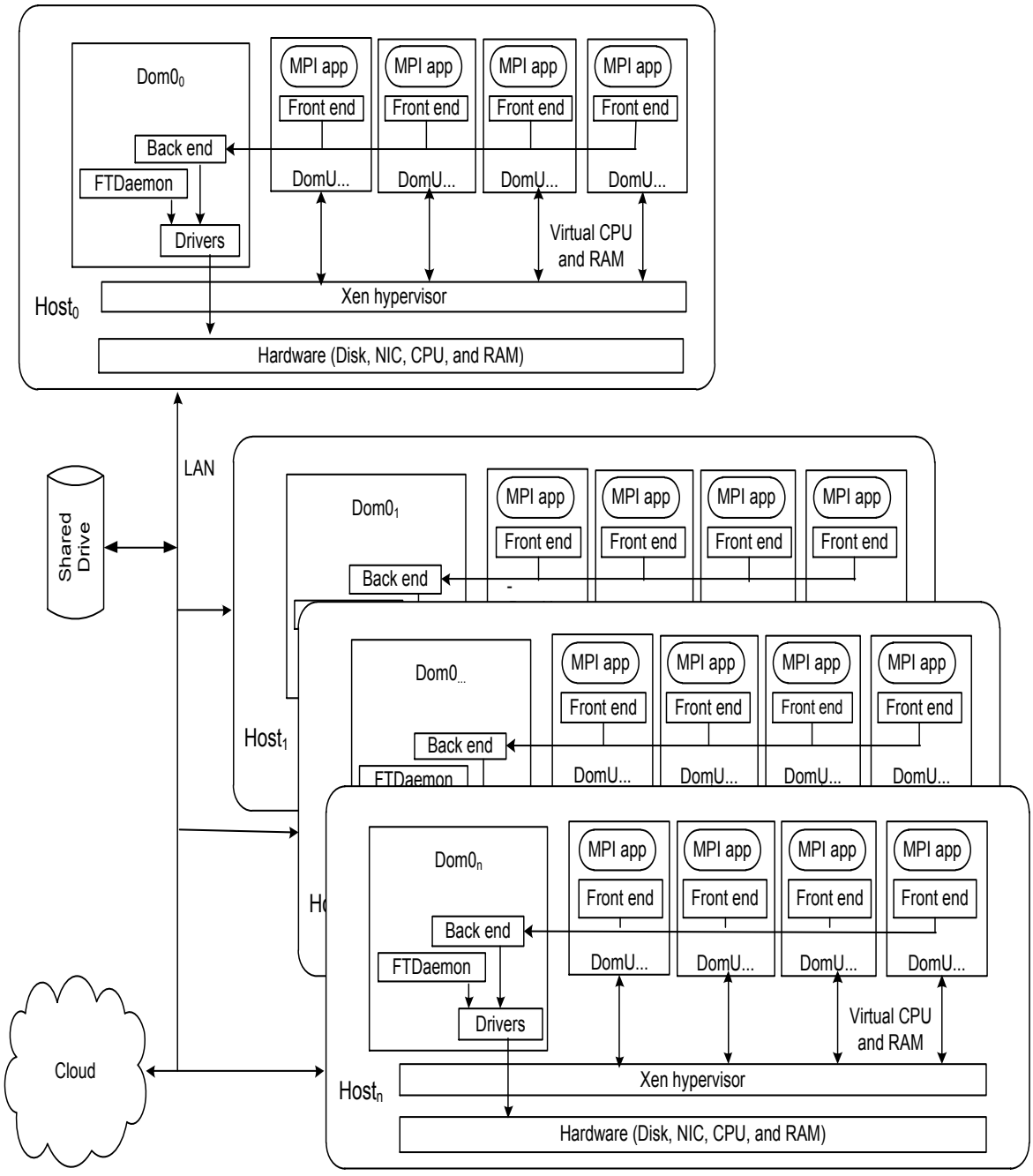

Figure 4.2: System architecture

4.4 Proactive Fault Tolerance Algorithm

In this section, we describe our algorithm and provide a quantitative analysis of its properties. The current information from the sensors is used to determine the state of monitored parameters. The algorithm predicts future failure, and takes action to reduce the impact of failure on the application. Finally, it also relinquishes the unhealthy node and installs an FTDaemon on the newly leased node. Algorithm 2 is given as follows:

Algorithm 2

FTDaemon running on all host C_i(i = 0, 1,. . . , n);
Monitored parameters: V= temperature, fan speed, voltages and utilization;
Variables = operating conditions V_w: weight (-1, 0, 1, 2);
where:
#-1 = operating at normal values of all parameters; V;
0 = operating at max values of one or more parameters
1 = operating at critical value;
2 = error value;
α_w = Warning State;
α_c = Future failure State;
Compute for host current state and take action;
if future failure is predicted;

1: **FTDaemon:**
2: **begin**
3: *record the hostname of all guest VMs active on host C_i;*
4: *get critical values of C_i V;*
5: *set timer* **TRUE;**
6: **while TRUE** do;
7: *read parameters C_i V;*
8: *compute for severity weight values;*
9: *compute for α_c and α_w;*
10: **if** $\alpha_c = 1$ then;
11: *break;* # exit loop
12: **elseif** $\alpha_w = 1$;
13: *send warning message;*
14: *delay;*
15: **else**
16: *check if alarm trigger is received;*
17: **end while;**
18: **controller module:**
19: *lease additional host;*
20: *live migration of <VM_1, . . . , VM_n >;*
21: *install FTDaemon on newly leased host;*
22: *send details of newly leased host to head host;*
23: *relinquish the unhealthy host;*
24: **end**

4.4.1 Cost Model

We develop a cost model to show that our proposed solution is a cost-oriented approach. Cost-oriented fault tolerance for HPC systems in the cloud is particularly important because a high dollar cost would discourage scientists and researchers from using HPC systems in the cloud. A good understanding of cost implications and reliability of HPC systems in the cloud enables users to choose a suitable fault tolerance strategy at a minimum dollar cost in running the application in the cloud environment. From the cost management viewpoint, it is a tool used to cut costs and to choose fault tolerance in line with a project's budget and strategy. It also helps to choose cloud computing resources to achieve a particular reliability level, and compare alternative fault tolerance solutions. We derive a cost model C_{ca} for running computation-intensive applications on HPC systems in the cloud. The cost can be determined with the following cloud computing parameters [22, 18, 123, 121, 122]:

- Installation/configuration cost C_{in}
- Execution cost C_e
- Communication cost C_c
- Storage cost C_s
- Redundancy cost C_r
- Failure cost C_f

Hence, we have;

$$C_{ca} = C_{in} + C_e + C_c + C_s + C_r + C_f \tag{4.1}$$

Below, we briefly discuss the above parameters and show how we derive the cost model.

4.4.2 Installation/Configuration Cost

To derive the cost-oriented fault tolerance model, we include the installation and/or configuration cost. The installation and/or configuration cost has been overlooked in the past [18, 22]. There are few HPC systems in the cloud services that do not require installation and configuration of the necessary software tools before running application. The level of installation and configuration depends on the cloud services leased and the application's requirements. For instance, when HaaS is leased from a service provider, it is the sole responsibility of the user to set up the HPC system environment, install and configure the HPC system tools in line with the computation-intensive application's requirements. We model the installation and configuration cost as;

$$C_{in} = C_0 + \sum_{g=1}^{m} C_{ig} + \sum_{h=1}^{n} C_{ih} \tag{4.2}$$

where:

C_0 is the initial/fixed cost to set up a "standard" HPC system environment in a cloud (defined number of computer nodes in HPC system), C_{ig} (g = 1, 2, 3,. . . , m) is the set up unit cost of each computing node when it exceeds the "standard", while C_{ih} (h = 1, 2, 3,. . . , n) is the unit cost to setup and configure the storage resources. In most HPC systems, performance tests are typically carried out with 4 to 32 compute nodes. The "standard" we used here are 32 compute nodes, the linux operating system and a MPI environment. For scalability analysis we may require much larger systems consisting of more than 1000 nodes.

4.4.3 Execution Cost

The execution cost C_e, is defined as the cost incurred during the execution of the computation-intensive application with computation resources (CPU and RAM) in the cloud. The execution cost is usually charged by the hour or month. This can be represented as the product of the cost of leased host and the time to complete execution of computation-intensive application in HPC systems in the cloud. We can represent the execution cost as:

$$C_e = \sum_{i=1}^{p} C_{ei}.E_{ti} \tag{4.3}$$

where:

C_{ei} is cost per host, i = 1, 2, 3,. . . , p is the number of hosts used to set up the HPC systems in the cloud for the computation-intensive application. E_{ti} is the execution time of the computation-intensive application in the HPC system with host i.

4.4.4 Communication Cost

The communication cost is the cost associated with transferring data into its cloud storage resources and the cost associated with transferring data out of its cloud storage resources. Some cloud providers, for example, Baremetalcloud [98] do not charge for communication within the same cloud. Consequently, the communication cost is directly proportional to the rate at which data are moved in and out of the cloud storage resources. For data intensive applications, this

cost maybe high [22]. The model for the communication cost is as follows:

$$C_c = \sum_{j=1}^{q} C_{cj}^{in} + \sum_{k=1}^{r} C_{ck}^{out} \tag{4.4}$$

where:

C_{cj}^{in} (j = 1, 2, 3,. . . , q) is the cost associated with transferring q bytes into the cloud storage system. C_{ck}^{out}(k = 1, 2, 3,. . . , r) is the cost associated with transferring r bytes out of the cloud storage system.

4.4.5 Storage Cost

There is a cost associated with cloud storage utilisation. At the time of writing of this thesis, the cost for cloud storage HaaS is usually around 3.5 cents per GB per month [98]. We model the storage cost with the following;

$$C_s = \sum_{l=1}^{s} C_{sl}.C_{tl} \tag{4.5}$$

where:

C_{sl} (l = 1, 2, 3,. . . , s) is the cost associated with s cloud storage system. C_{tl}, is the time associated with usage of the s storage system per GB.

4.4.6 Redundancy Cost

Redundancy is commonly used in HPC systems to improve reliability. For instance, in the fault tolerance techniques we investigated [127], virtual machines are migrated from unhealthy hosts to redundant hosts when a future failure of host

is predicted. In redundant computing (discussed in section 4.1 above), compute nodes are replicated twice. Redundant compute nodes may increase the reliability of HPC systems but at the same time increase the cost of running applications in the cloud. The cost of providing redundant hosts can be expressed with the following equation. We assume that the redundant hosts are in parallel with virtual machine hosts $Dom0_0, \ldots, Dom0_n$.

$$C_r = \sum_{u=1}^{v} C_{ru}.C_{tu} \tag{4.6}$$

where:

C_{ru} (u= 1, 2, 3,. . . , v) is the cost associated with the v number of redundant hosts. t is the time associated with the usage of the v compute hosts running virtual machines.

4.4.7 Failure Cost

It is unlikely that an HPC system in the cloud would achieve a reliability value of 1. This may be attributed to the number of shared resources comprising hardware components, communication links and virtual machines. Different fault tolerance approaches tend to reduce this cost. We factor failure cost into the cost-oriented fault tolerance model. Based on MPI model, we assume that failures of compute hosts are statistically independent, hence, compute hosts are in either operational or failed states. The failure cost associated with not using fault tolerance or using appropriate fault tolerance techniques is high, therefore, failure cannot be

neglected in the cost model.

$$C_f = C_{loss}[1 - R_x] \tag{4.7}$$

where:

R_x is the HPC system reliability, while C_{loss} can be obtained from a failure data repository such as CFDR [3] or can be predicted by an expert as proposed in [121]

Substituting equations (4.2), (4.3), (4.4), (4.5), (4.6) and (4.7) into equation (4.1), the total cost for running computation-intensive application in HPC systems in the cloud C_{ca} is expressed by the following:

$$C_{ca} = C_0 + \sum_{g=1}^{m} C_{ig} + \sum_{h=1}^{n} C_{ih} + \sum_{i=1}^{p} C_{ei}.E_{ti} + \sum_{j=1}^{q} C_{cj}^{in} + \sum_{k=1}^{r} C_{ck}^{out} + \sum_{l=1}^{s} C_{(ss)l}.C_{tl} + \sum_{u=1}^{v} C_{ru}.C_{tu} + C_{loss}[1 - R_x] \tag{4.8}$$

From equation (4.8) we can deduce the following:

1. Compute host redundancy fault tolerance techniques that are commonly used in traditional HPC systems increase the cost of running computation-intensive applications in HPC systems in the cloud.

2. Using checkpoint and restart fault tolerance techniques in HPC system in the cloud improves the reliability of the HPC systems; however, the cost of running the computation-intensive application increases. This is attributed to the increase in the wall clock execution time of the application caused by checkpoint and restart fault tolerance techniques. This is in line with existing

results on the effect of checkpoint and restart fault tolerance techniques [26]

3. Live migration of virtual machines from unhealthy hosts to redundant hosts can improve the reliability of HPC systems and reduce the cost of running computation-intensive applications if the spare hosts are not provided in advance, that is, before a failure prediction is made [31, 87].

4.4.8 Quantitative Analysis

Case 1:

We first analyse the total dollar cost of the proactive fault tolerance algorithm used in [127] using equation (4.8), when a spare host is provisioned ahead of prediction of failure and algorithm proposed in redundant computing [14]. With these models the cost of running computation-intensive applications in HPC systems in the cloud will be relatively high, due to the cost of the spare nodes/hosts. The cost in redundant computing would be twice that of the compute hosts used to execute computation-intensive application. The cost using the algorithm proposed in [127] has been computed and is shown in Figure 4.4. The cost implication of this model is shown in case 2.

Case 2:

A configuration is established for which the leased host is operating in normal state (as described above). In this state, there is no need to keep a spare host. From observations and records, HPC systems operate in this region most of the time, except when failure is about to occur (when a host enters its critical state (i.e., $\alpha_w = 1$)). As already stated eariler, critical state is established when three of

prediction parameters are operating at critical states, from our observation system failure mostly occur at this state. Only in this state does the controller model lease an additional node/host from the service provider as well as relinquish the unhealthy one. We assume that the time window from prediction of host failure to actual failure is enough to migrate virtual machines from the unhealthy host to the newly leased host based on similar experiments [130]. For failures that occur without prior time window warning, for example, power failure, reactive fault tolerance technique such as checkpoint/restart is used to recover from such failure. Using equation (4.8), the operating cost of the spare node/host is close to zero, because the unhealthy host is relinquished immediately after migration of the virtual machines. Our experimental results show that provision of a node/host and live migration of virtual machines varies between 20 seconds and 60 seconds on our test system.

$$C_r = \sum_{u=1}^{v} C_u.R_{tu} \approx 0 \tag{4.9}$$

Therefore:

$$C_{ca} = C_0 + \sum_{g=1}^{m} C_{ig} + \sum_{h=1}^{n} C_{ih} + \sum_{i=1}^{p} C_{ei}.E_{ti} + \sum_{j=1}^{q} C_{jh}^{in} + \sum_{k=1}^{r} C_{ck}^{out} + \sum_{l=1}^{s} C_{sl}.C_{tl} + C_{loss}[1 - R_x] \tag{4.10}$$

There is a significant dollar saving of about 20% with our model as can be seen by comparing Equations (4.8) and (4.10). We further show the significant difference

with the cost model in the next section.

4.5 Evaluation

We evaluate the wall clock execution time in the presence of faults and the dollar cost of providing spare node in advance as it is applied in traditional HPC systems using our proposed technique (Algorithm 2). We assume that all corresponding factors in equations (4.8) and (4.10) are the same.

We have experimented with the characteristics of our fault tolerance design in a real cloud environment. We leased two to sixteen servers/hosts from a cloud service provider of HaaS [98]. Each compute host/server had the following configuration; Intel Dual Xeon core processor (2 x 3.5GHz), 2GB memory, PC3200 3.5 SCSI 10000rpm and 100GB network drive using Internet Small Computer System Interface (iSCSI) SAN configuration [137]. The hosts/servers use Local Area Network (LAN) connections. Details of the HaaS provided by the cloud service provider can be found in [98].

We installed Xen hypervisor [112] on each of the hosts. Therefore Xen hypervisor runs on all the hosts as shown in Figure 4.3. Xen hypervisor is an open source, industrial standard virtualisation technology. The Linux operating system runs on top of the Xen hypervisor. We installed a para-virtualised guest operating system on the hosts. A para-virtualised operating system uses a modified kernel, and reduces the size of the image.

Wall clock execution time:

For the experiment to determine the wall clock execution time, each host is con-figured to host 1 to 4 virtual machines (compute node). Each virtual machine

is configured to have one processor, 250MB memory and 5GB hard drive. With the 4 hosts/servers we leased, we formed a cluster of 16 compute nodes to test our algorithm.

We conducted three sets of experiments with 2, 4, 8, and 16 nodes per cluster. We ran a HPC application, the High Performance Linpack benchmark (HPL) [138] in an OpenMPI [111] environment. HPL is widely used in benchmarking Top500 supercomputing. OpenMPI is an open source implementation of MPI-2.

We executed the HPL application with four different problem sizes of 2000, 4000, 6000 and 8000 on 2, 4, 8, and 16 nodes respectively. The wall clock execution time of each the problem size was recorded without checkpoint, with checkpoint (the checkpoint and restart solution that is package in OpenMPI implementation [139, 17]), and with the FTDaemon (our proposed solution). For the tests with checkpoints, the number of checkpoints used is shown in Table 4.4, which also shows the number of live migration associated with each problem size. We observed that our algorithm significantly improved application resiliency at a reduced cost compared to more common reactive approaches. This helps to determine the effect of checkpointing on computation-intensive applications running in a cloud.

Table 4.3: HPL with different problem sizes and nodes

HPL Problem Sizes	**Number of Nodes**	**Number of Checkpoints**	**Number of Migrations**
2000	2	2	2
4000	4	3	3
6000	8	4	4
8000	16	6	6

The time to lease and provision a node is about 18 seconds. The average migration down time obtained from our experiment was 0.315 seconds. The performance results are shown in Figure 4.5.

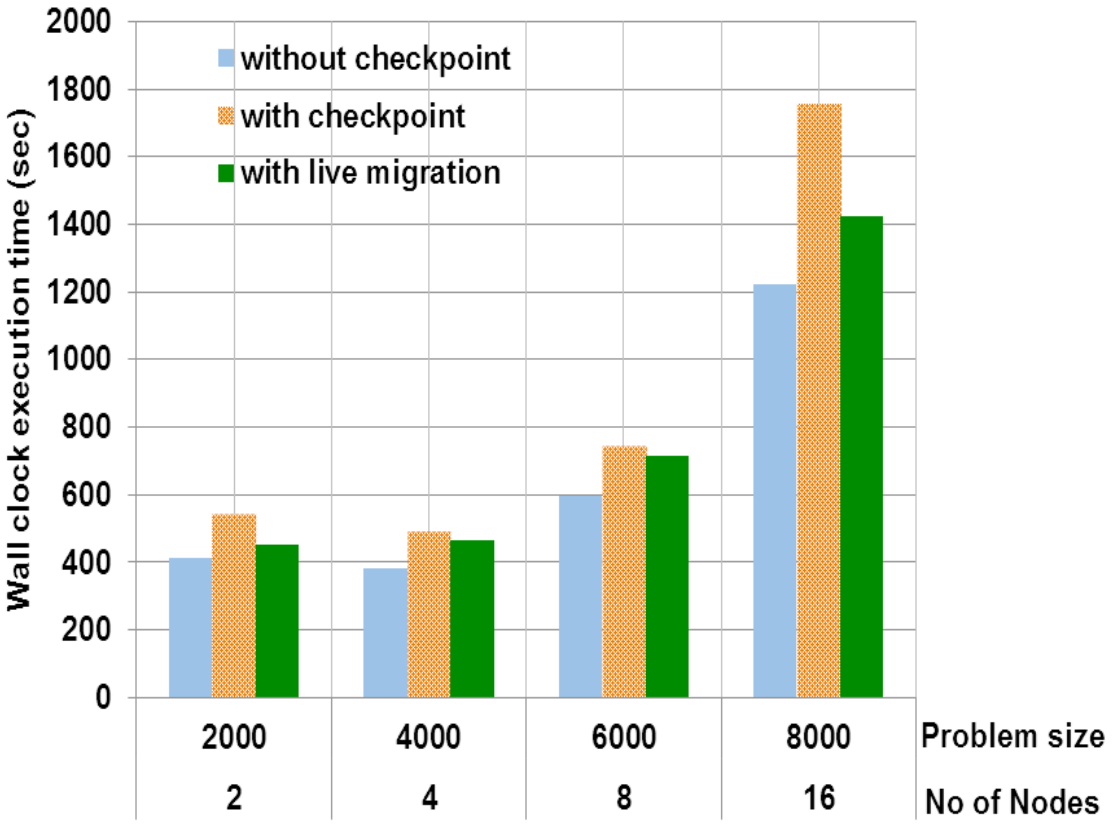

Figure 4.3: Performance of HPL benchmarking without checkpointing, with checkpointing, and with FTDaemon.

Cost Analysis:

To show that our proposed fault tolerance solution is a cost-oriented approach, we conducted four sets of experiments with 2, 4, 8, 16 and 32 nodes per cluster. We ran a real HPC application, the High Performance Linpack benchmark (HPL) [138] in an OpenMPI environment.

We executed the HPL application with five different problem sizes of 2,000, 4,000, 6,000, 8,000 and 10,000 on 2, 4, 8, 16 and 32 nodes respectively. The wall

clock execution time of each problem size was recorded when run:

1. without checkpoints (assume no fault tolerance provided and no fault occurred during execution),
2. with checkpoints taken (used checkpoint and restart solution provided in OpenMPI implementation [139, 17]),
3. with migration (migration solution proposed in traditional HPC systems [127]) and
4. with the FTDaemon (our proposed solution).

For the tests with checkpoints, the number of checkpoints used is shown in Table 4.4, which also shows the number of live migrations associated with each problem size as well as the minimum number of redundant hosts. We selected the minimum number of redundant nodes based on [123, 121]. This helps to determine the effect of checkpointing on computation-intensive applications running in a cloud. Furthermore, the dollar cost implications of redundancy nodes proposed in [14, 127] can be easily analysed with this setup.

Table 4.4: HPL with different problem sizes, host and compute nodes

$\mathbf{P}_{SIZE}$	$\mathbf{N}_{H}$	$\mathbf{N}_{RH}$	$\mathbf{N}_{VMs}$	$\mathbf{N}_{C}$	$\mathbf{N}_{M}$
2000	2	0	2	2	2
4000	2	1	4	3	3
6000	4	1	8	4	4
8000	8	2	16	6	6
10000	16	3	32	7	7

where:

- P_{SIZE} = problem size
- N_H = number of host/server
- N_{RH} = number of redundant host/server
- N_{VMs} = number of VMs in a cluster
- N_C = number of checkpoints taken
- N_M = number of live migrations

At the time of writing, the cost to lease each compute node with the above configuration is $0.09 per hour while it costs $0.03 per hour to lease a 100GB network drive. We use these figures to compare the cost of using checkpointing, migration with provision of redundant hosts ahead of prediction of failure, and our solution.

In the prototype implementation, we monitored the real-time CPU temperature, CPU voltage, CPU fan speed, and CPU utilisation as a system reliability metric with the FTDaemon running on each host. High temperature variations on the nodes affect system reliability, degrade performance, and cause failure of CPUs and circuits [126, 140]. We induced high temperatures on the CPU with the running HPL. From our experiment conducted with different HPL problem sizes, live migration of VM from one host/server to another host/server within the same local area network (LAN) varies between 2 to 16 seconds (from start to finish). It should also be noted that the live migration time and down time depend on the configuration of the physical machine (host/server) and the configuration of the virtual machine [46].

Although the service provider charged on a cost per hour basis, in our cost analysis shown in Table 4.5, we assume cost per second. The cost per second is

equivalent to computation-intensive applications that run for hours, weeks and months. This enabled us to quantify the costs of checkpointing, providing a spare node ahead of prediction failure of a compute node as proposed in [127], redundant computing [14] and with our FT techniques.

Table 4.5: Cost analysis with different FT and nodes

$\mathbf{N}_n$	$\mathbf{N}_{cr}$	$\mathbf{P}_{size}$	$\mathbf{\$}_{ch}$	$\mathbf{\$}_{r=1}$	$\mathbf{\$}_{rn}$	$\mathbf{\$}_{FTDaemon}$
2	2	2000	97.81	74.14	81.14	81.14
2	3	4000	88.20	68.45	125.20	83.47
4	5	6000	268.28	214.86	321.07	256.85
8	10	8000	1266.14	879.58	1153.46	1025.29
16	18	10000	4102.82	2669.52	3857.88	3429.23

N_n = number of compute nodes

N_{cr} = number of compute and redundant nodes

P_{size} = problem size

$\$_{ch}$ = dollar cost of running application with checkpointing (No redundant nodes)

$\$_{r=1}$ = dollar cost of running application (reliability = 1)

$\$_{rn}$ = dollar cost of running application with redundant nodes provisoned

$\$_{FTDaemon}$ = dolar cost of running with our approach (redundant nodes not provisioned ahead of prediction)

The performance results of the three different fault tolerance cost models are shown in Figure 4.5. As already stated above, we assume that all related factors in equations (4.8) and (4.10) are the same. The result shows that the proposed proactive fault tolerance approach to HPC in the cloud significantly reduces

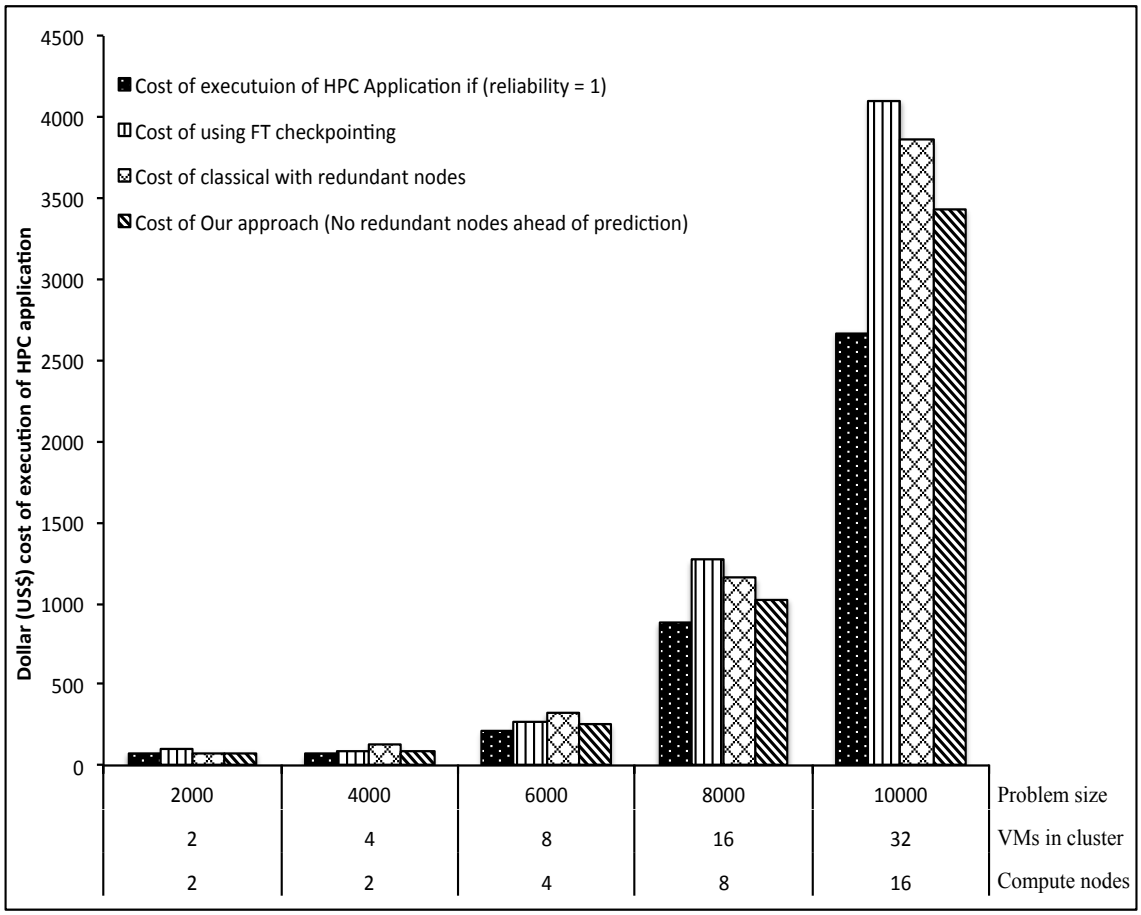

Figure 4.4: HPL with different problem sizes and nodes

the wall clock execution time and therefore the cost of computation-intensive applications running in the cloud. We observed that our algorithm significantly improved application resiliency at a reduced cost compared to more common reactive approaches.

Periodic checkpointing can be used to provide FT for unpredictable failure. However, the rate of checkpointing of computation-intensive application can be reduced by as much as 50% to reduce the cost due to prediction.

4.6 Related Work

Fault tolerance techniques for HPC applications with MPI can be classified into two major groups:

1. Reactive fault tolerance techniques
2. Proactive fault tolerance techniques.

A reactive fault tolerance technique tends to minimise the impact of failure on the computation-intensive applications in the presence of failure of one or more computational nodes. A good example of reactive fault tolerance is checkpoint and restart. Checkpoint and restart allows computation-intensive problems that may take a long time to execute in HPC systems to be restarted from the point of failure in the event of errors or failures. Checkpoint and restart techniques have received considerable attention in the past [16, 85, 11, 125, 77, 124]. One of the major issues with checkpoint and restart techniques is the overhead involved in saving the execution state (reducing this overhead will ensure that a computation-intensive application does not spend most of its time doing checkpointing).

To reduce the time required for checkpointing, several checkpointing optimisation techniques have been proposed [125, 77, 141, 141, 77, 84]. The overall aim is to reduce the actual elapsed time for execution of applications in the presence of spontaneous failures because computer resources and time are wasted when the system is executing fault tolerance strategies such as checkpointing

These efforts to reduce the overhead caused by checkpoint and restart fault tolerance techniques in computation-intensive applications have not been effective. Recent publications [10, 23, 87 88] show that with steadily increasing numbers of components in today's HPC systems, applications running on HPC systems may

not be able to achieve meaningful progress with the basic checkpoint and restart approach.

Proactive fault tolerance mitigates the effect of failure during the lifetime of a computation-intensive application by taking proactive measures using failure prediction techniques. The commonly used failure prediction techniques include analysis of the RAS log and monitoring hardware parameters such as processor temperatures, fan speeds and voltages. In the pioneering work of Nagarajan, *et al.* [127] on proactive fault tolerance for HPC with Xen virtualisation, processes are migrated from unhealthy nodes to spare nodes. However, this requires spare nodes to be always available. This technique may not be cost efficient in cloud computing because the spare nodes will be billed, and the cost of running the application in the cloud will be higher.

Redundant computing has recently been proposed [14] to reduce wall-clock time of computation-intensive application running in HPC systems in the presence of failure. In redundant computing, each process is replicated a defined number of times to counter the effect of component or node failure. The replicated processes run in parallel. The replicated processes keep running when the primary processes fail. In redundant computing, compute nodes are also replicated. The replication of the compute nodes increases the energy utilisation and dollar cost of running computation-intensive applications in HPC systems in the cloud.

Our work differs from previous work in that our fault tolerance algorithm provides fault tolerance to HPC in the cloud at the hardware level at reduced cost, while running in user space (under user's control). It does not rely on the existence of pre-configured spare nodes. In addition, our solution does not also rely on redundant computing techniques. The proposed solution is cost effective

compared to both redundant computing and the approach in [127, 14]. We derived the cost model for running computation-intensive application in HPC systems in the cloud. With our cost model, users may be able to weigh the cost of different fault tolerance techniques. Our fault tolerance solution is particularly suited to users who lease HaaS.

4.7 Summary

In this chapter, we have presented the design and implementation of a proactive fault tolerance framework for High Performance Computing (HPC) in the Cloud. We derived the cost model for running computation-intensive application in HPC systems in the clouds. We analysed the dollar cost of providing spare nodes ahead of prediction of failure. We showed that our solution does not rely on the provision of spare nodes ahead of the prediction of failure. We presented experimental results carried out on a real cloud environment. The experimental results clearly show that the proposed proactive fault tolerance approach to HPC systems in the cloud can significantly improve the execution time of computation-intensive applications by upto 30% and thereby reduce the dollar cost for running them by as much as 30%. The frequency of checkpointing of computation-intensive applications can also be reduced by 50% with our FTDaemon. Our approach can help reduce energy consumption by reducing the wall execution time of computation-intensive HPC applications in the presence of failure of one or more computational nodes.

Chapter 5

Cost-effective Cloud Services for HPC in the Cloud: IaaS or HaaS?

In this chapter, we address the research question posed in Section 3.5.2 (HPC systems research communities are concerned about the cost and computational performance of different cloud services). The content of this chapter, with minor changes, is extracted from our previous work, which has been presented at the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13) [33].

5.1 Introduction

In the scientific research domain, traditional High Performance Computing (e.g., Blue Gene/L, clusters of computers) is used to solve computation-intensive and/or data-intensive problems. Traditional HPC systems are expensive and sometimes involve huge start-up investment, technical and administrative support, and job queuing. With the benefit of cloud computing, cloud services, such as

Infrastructure as a Service (IaaS) and Hardware as a Service (HaaS), enable scientists and researchers to run their HPC applications in the cloud without the upfront investment associated with traditional HPC infrastructure.

However, the cost of running HPC applications on the cloud may be high if the cloud services are not well understood and cost-effective cloud services are not chosen. If the dollar cost of running HPC applications in the cloud is high compared to traditional HPC systems, the benefits of running computation-intensive application on the cloud may be negated. HPC research communities are concerned about the cost and computational performance of different cloud services.

In this chapter we analyse the computational performance and dollar cost of running computation-intensive applications in HPC systems in the cloud when IaaS and HaaS are leased. We find that the cost of executing computation-intensive application when HaaS is leased is significantly lower than that of the IaaS model. We show that there is significant improvement in computational performance of the application on HaaS if the computation-intensive application is not a network intensive application. Our experimental setup uses the Message Passing Interface (MPI) implementation [111]. We provide our test results, but do not reveal the identity of the cloud providers in order to avoid any head-to-head comparisons. However, we do include the relevant technical details of the cloud instances.

5.2 Experimental Setup

We setup experimental environments to evaluate the computational performance and dollar cost of running computation-intensive application on IaaS and HaaS. Our experimental setup includes two services we have leased from two cloud

service providers, referred to here as Cloud-A and Cloud-B. Cloud-A offers IaaS in different kinds of cluster instances for HPC applications (for example, cluster compute instances). Cloud-B offers HaaS that can be configured to run HPC applications.

5.2.1 Cluster Compute Instances from Cloud-A (IaaS)

Cloud-A is one of the major cloud service providers. They offer IaaS in different instances for HPC applications. Table 5.1 shows a sample of cluster compute instances with price details of on-demand instances from cloud providers. The cluster compute instances are available with commonly used Operating Systems (OS) (Windows and Linux) in 32-bit and 64-bit platforms. For our experiments, we chose the Ref-C virtual instance in Table 5.1 because it is widely used for HPC applications. The instances use Xen full virtualisation (see Section 3.4.5). The I/O network communication between the cluster instance is 10 Gigabit Ethernet.

In order to compare the computational performance and dollar cost of running HPC applications when IaaS and HaaS services are leased, we leased a cluster compute instance with a total of 16 processors. Details of the leased cluster compute instance are shown in Table 5.2. We installed OpenMPI 1.6 [111] on the node/virtual machine. OpenMPI is an open source implementation of the Message Passage Interface (MPI).

5.2.2 HPC System on HaaS in the Cloud

As explained in Section 4.2, HaaS allows users to have full control of the system and control environment for measuring system performance and other experiments.

Table 5.1: Virtual and HaaS Instances from Cloud-A and Cloud-B

Instance type	Memory	CPU	Disk	Cost of instance for Linux	Cost of instance Window
Ref-A Virtual instance	30 GB	2x2.0 GHz (sixteen-core)	500 GB	$1.600 per hour	$1.800 per hour
Ref-B Virtual instance	244 GB	2 x Intel Xeon E5-2670 (eight-core)	240 GB	$3.500 per hour	$3.831 per hour
Ref-C Virtual instance	22 GB	2 x Intel Xeon X5570 (quad-core)	1690 GB	$2.100 per hour	$2.600 per hour
Ref-D Virtual instance	23.00 GB	2 x Intel Xeon X5570 (quad-core)	1690 GB	$1.30 per hour	$1.610 per hour
Ref-E Hardware Instance	96 GB DDR3-1333	2x2.13 GHz E5606 (eight-core)	1000.0GB, 7200RPM	0.99 per hour	$1.19 per hour
Ref-F Hardware Instance	48 GB DDR3-1066	2x2.66 GHz X5650 (twelve-core)	300GB, 10000RPM	0.73 per hour	$0.93 per hour
Ref-G Hardware Instance	64 GB	2x2.0 GHz E5-2650-OctoCore (sixteen-core)	500.0GB	1.54 per hour	$1.59 per hour
Ref-H Hardware Instance	32 GB	2x2.0 GHz (eight-core)	250.0GB	1.25 per hour	$1.3 per hour

This enables users to determine the number of VMs to be deployed for HPC applications. We leased a HaaS instance (Ref-G) with 64GB RAM from Cloud-B. Table 5.1 shows some of the cloud services offered by the HaaS providers that are similar to cluster compute instances offered by Cloud-A. The table also gives a summary of HaaS and the price of the leased service. The communication network between each HaaS is a 1 Gigabit Ethernet.

A summary of the virtual machines we provisioned on the HaaS is shown in Table 5.2. We installed Xen hypervisor [112] on the host. Xen hypervisor is an open source, industry standard virtualisation technology. Linux Operating System (Ubuntu 12.4 64-bit) runs on top of the Xen hypervisor. We imported our pre-configured para-virtualised guest operating system (Ubuntu 12.4 64-bit) on the HaaS instance. The pre-configured para-virtualised guest reduces the time needed to setup the HPC system on the HaaS instance. A para-virtualised OS uses a modified kernel, reduces the size of the image and improves performance [136, 114]. The virtual machine is configured to have 16 processors with 60GB memory and 200GB hard drive. We installed OpenMPI on the node. This setup is almost equivalent to the cluster compute instances we leased from Cloud-A. The setup also provides a good comparison environment for IaaS and HaaS in terms of computational performance and dollar cost. Table 5.2 shows both the IaaS and HaaS environments we used.

Table 5.2: The cost analysis virtual machine of IaaS and virtual machine of HaaS

Cloud-A, VM of IaaS	**Cloud-B, VM of HaaS**
Architecture: x86-64	Architecture: x86-64
RAM: 22GB	RAM: 60 GB
CPU op-mode(s): 32-bit, 64-bit	CPU op-mode(s): 64-bit
CPU(s): 16	CPU(s): 16
NUMA node(s): 1	NUMA node(s): 1
Vendor ID: GenuineIntel	Vendor ID: GenuineIntel
CPU family: 6	CPU family: 6
Model: 26	Model: 26
Stepping: 5	Stepping: 5
CPU MHz: 2933.440	CPU MHz: 2933.468
Hypervisor vendor: Xen	Hypervisor vendor: Xen
Virtualisation type: full	Virtualisation type: para
L1d cache: 32K	L1d cache: 32K
L1i cache: 32K	L1i cache: 32K
L2 cache: 256K	L2 cache: 256K
L3 cache: 8192K	L3 cache: 8192K

5.3 MPI Applications and Benchmark

We used a commonly used HPC benchmark and real HPC application to analyse and evaluate the MPI applications running on IaaS and HaaS services. The benchmark was the High Performance Linpack (HPL) benchmark [138] and the application was ClustalW-MPI [117]. We describe these below.

HPL [138] is a benchmark that is commonly used to evaluate the compu[c1]tational[c1] - performance of HPC systems such as top500 [2]. It measures the floating execution rate of linear equations based on the problem size. We executed the HPL benchmark with five different problem sizes of 2,000, 4,000, 6,000, 8,000 and 10,000 on both cloud services on virtual machines from IaaS and on HaaS. The execution of each problem size was carried out twice and the average execution time was calculated. The five different problem sizes enable us to obtain different wall clock execution times of HPL. We recorded the wall clock execution time for each problem size. We used the wall clock execution time to analyse the dollar cost and computational performance of both platforms. Figure 5.1 shows the results obtained on computational-performance.

ClustalW-MPI [117] is a parallel implementation of ClustalW [142] which is based on MPI. ClustalW is a tool that is widely used in bioinformatics for multiple alignments of nucleic acid and protein sequences. It uses three alignment steps: pairwise alignment, guide-tree generation and progressive alignment. We ran a sample of 'A full multiple sequence alignment', 'A guide tree only sequence alignment', and 'A multiple sequence alignment out of an existing' on nodes from IaaS and from HaaS. We recorded the execution time of the three alignment steps to compare time to finish executions with both IaaS and HaaS. The results are

shown in figure 5.2.

5.4 Results and Discussion

One of the major attractions of the Cloud-A cluster compute instance is that it is relatively easy to setup the clusters compared to setting up a cluster in HaaS. However, some level of technical knowledge is required to setup clusters on Cloud-A that will run HPC applications, due to the varying needs of HPC applications. In order to reduce the time to setup an HPC system on HaaS instances in the cloud, we uploaded our pre-configured para-virtualised image to the cloud. Similar VM images can be downloaded from different sites. We estimated that this technique reduces the setup time by up to 80%. We did not compare the setup time for the HPC system between Cloud-A (IaaS) and Cloud-B (HaaS) because setup time varies with the individual user's technical experience.

From the computational performance result of the HPL benchmark shown in Figure 5.3, we can see that the wall clock execution time of the HPL benchmark on a provisioned instance on HaaS is shorter than that of IaaS provided by Cloud-A. We achieved this because the memory of the virtual instances deployed on HaaS is 60GB. We chose to allocate this amount of memory to our virtual instance because we can predict the memory needed. This option is not available for the IaaS instance (users cannot change the memory of the virtual instance chosen). We also had full control of the hardware instance and virtual instances.

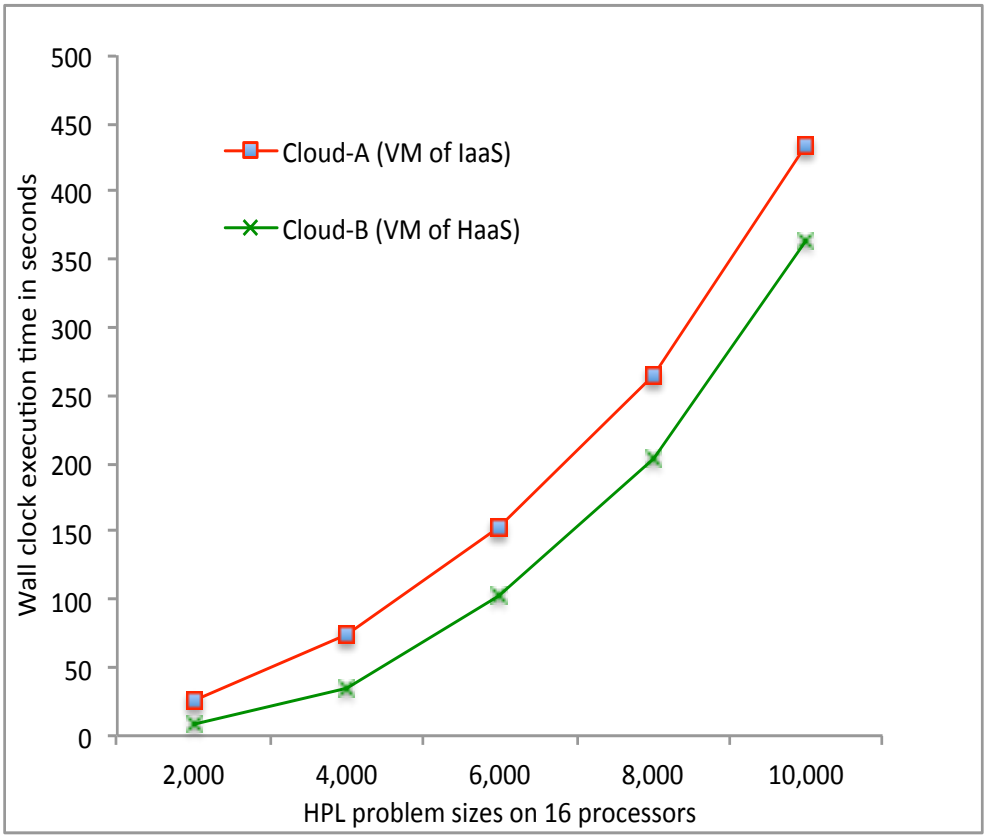

Figure 5.1: Computational performance of High Performance Linpack on 1 node with 16 processors

As shown in Figure 5.3, executing the HPL on 1 node with 16 processors eliminates the bandwidth inequality on both providers. The virtual instance on HaaS outperforms IaaS. This is because we had full control of the applications running on our HaaS instance and we allocated higher memory to the virtual machine on HaaS. On IaaS, other virtual instances may have been hosted on the hardware, which may have affected the performance of the application running on our leased IaaS instance. As shown in [135], high resource allocations on infrastructure affect applications running on virtual machines.

The ClustalW-MPI results are shown in Figure 5.2. Cloud-A IaaS uses a 10

Gigabit Ethernet network, whereas the HaaS we leased uses a 1 gigabit Ethernet network. We could have used benchmarks with the same bandwidths, however the two major providers of HaaS do not have a 10 gigabit Ethernet network. The results in Figure 5.2 show that there is no significant impact on application running on IaaS and on virtual instances on HaaS.

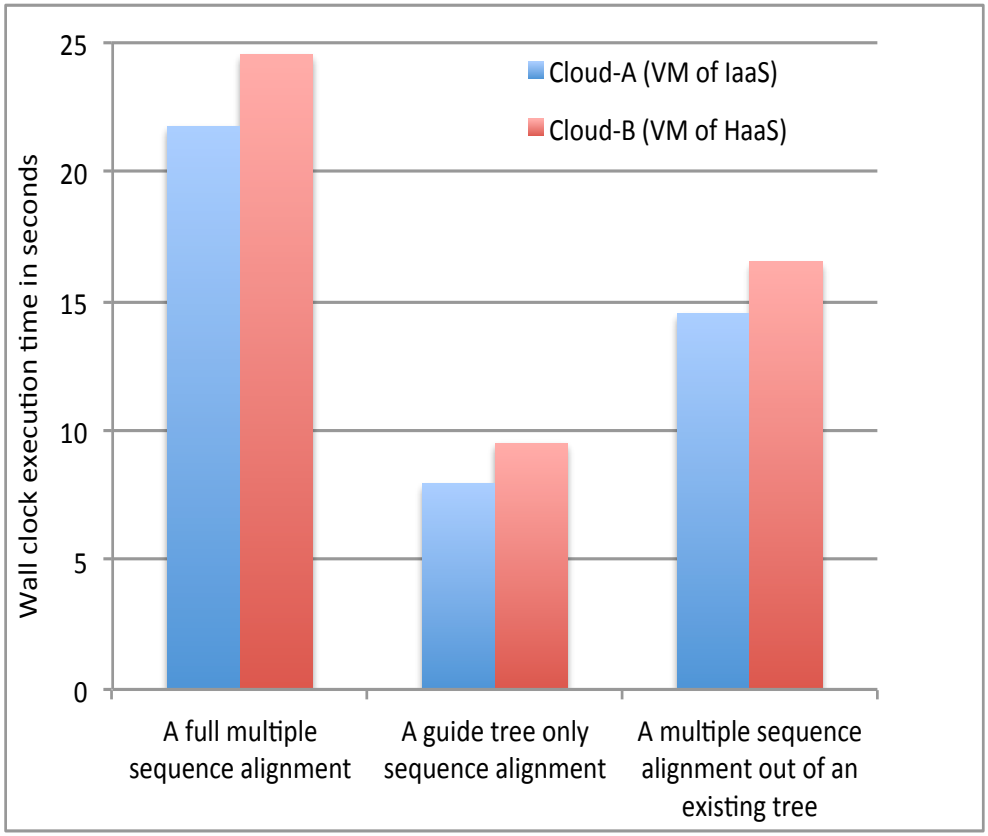

Figure 5.2: Performance of ClustalW-MPI application on 16 processors

5.5 Cost Analysis

At the time of writing, Cloud-A offers different price models to their cluster compute instance customers. The most widely used price model is called 'on-demand

instances'. The on-demand instances price model allows users to pay hourly without a contract, whereas other price models may require up-front payments and/or contracts.

Cloud-B offers their customers a pay-as-you-go price model, which is similar to on-demand instance prices offered by Cloud-A. Therefore we use on-demand price instance to compare the cost of running computation-intensive applications on both cloud services. There are additional charges for some cloud services, such as network bandwidth and IP addresses, but we do not consider these to avoid complexity. Both cloud providers, however, charge a similar rate for such services. We used the results obtained from HPL benchmarking to analyse the cost. Previous similar cost analyses [143, 22], assumed that 1 second is equal to the hourly rate, and we use the same assumption here (although both cloud providers offer an hourly rate). This also allows us to do the analysis without paying for the hours the experiment would have cost. We used the prices of the leased services as shown in Table 5.1. The cost analysis computation of IaaS and HaaS is shown in Figure 5.3.

Based on the computational performance and cost analysis, it appears that it is more cost effective to lease HaaS and configure the HPC systems. Cloud service users of HaaS have full control of the hardware as well as the virtual machines they provisioned. Application performance and other metrics can be easily measured. From the result, the cost of running HPC applications can be reduced by as much as 20% when HaaS is leased.

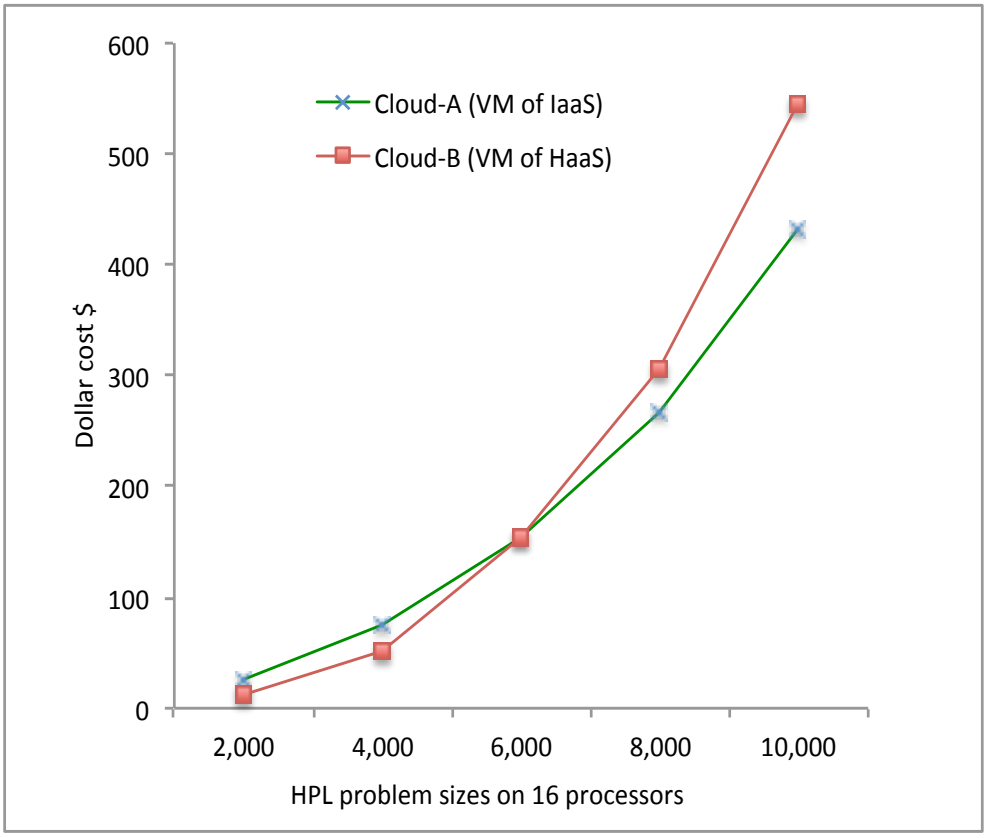

Figure 5.3: The cost analysis; virtual machine of IaaS and virtual machine of HaaS

5.6 Related Work

Cloud computing promises numerous benefits, which includes no up front investments for HPC applications, which is attractive, compared to traditional HPC systems. Many studies have evaluated the suitability of HPC systems in the cloud and showed that it is expected that more computation-intensive HPC applications will be run in the cloud HPC than traditional HPC systems [19, 22, 144, 145]. Furthermore, the Amazon Elastic Compute Cloud (Amazon EC2) cluster recently appeared in Top500 list [2] in year 2010, which shows that there is a viable future for HPC systems in the cloud.

Many past researches evaluating of HPC applications on HPC systems in the Cloud with emphasis on Amazon EC2 have been carried out. These investigations focus on the performance of Amazon EC2 and Traditional HPC systems [143, 22, 144, 19].

Carlyle *et al.* [143] studied the cost effective HPC System. They show that it is more cost effective for institutions like Purdue University to operate a community/traditional cluster than to lease HPC resources from Amazon EC2. This study clearly shows that Amazon on-demand cluster compute instances prices are not cost effective for HPC applications for some institutions. Their work focuses on Amazon EC2 service IaaS and traditional HPC systems.

Deelman *et al.* [22] in their work on 'The Cost of Doing Science on the Cloud: The Montage examples'; show that the cost of cloud services could be significantly reduced without significant impact on application performance, if the right storage and compute resources are provisioned. However, they did not consider different platforms like HaaS. We extended their work, demonstrating that HaaS can significantly reduce the cost of running computation-intensive application on HPC in the cloud.

Ekanayake and Fox [144] compare HPC applications with different needs and showed the performance of applications with latency. However, they did not compare the cost of executing computation-intensive application on different services such as IaaS and HaaS.

Yao *et al.* [146] showed that optimal cost-performance ratio can be achieved with the appropriate cloud instance. However, they did not consider cost and computational performance when IaaS and HaaS are leased.

Although others [143, 147, 144, 19, 148] have compared the computational

performance and dollar cost of using different cloud service such PaaS, IaaS and traditional HPC systems. However, our work is different from other work in that we study and compare the computational performance and dollar cost of running computation-intensive application in HPC in the cloud when IaaS and HaaS are leased. We experimentally demonstrated that the dollar cost of running computation-intensive application can be reduced as much as 20% with HaaS without significant impact to performance. With this experimental results, HPC system communities are equipped with cost analysis that would help them to select cost effective cloud service for HPC systems in the cloud.

5.7 Summary

Due to the huge capital investment required to own a traditional HPC systems which typically involves job queuing, using an HPC system in the cloud is a good alternative. Cloud computing offers IaaS and HaaS for deployment of cluster instances, which can be used to run computation-intensive applications. IaaS provides almost ready to use clusters with minimal deployment installation tasks. With HaaS, virtual machines can be provisioned to run computation-intensive application. We have conducted experimental analysis to determine the performance and cost when cloud services IaaS and HaaS are leased to run computation-intensive application. We showed that the dollar cost of running computation-intensive application in the cloud can be reduced by as much as 20% when HaaS is leased. We showed that there is no significant impact in performance of the applications when executed on the leased HaaS.

Chapter 6

Summary, Conclusions and Future work

This chapter summarises the thesis, presents our conclusions and makes recommendations for future work.

6.1 Summary

Key points from each chapter are summarised below.

In Chapter 1, we presented an overview of the research background and motivation, and identified key research problems and contributions. The role of HPC systems in today's society, for instance in weather forecasting, were discussed. We showed that the number of processors and nodes in HPC systems has increased over time in a quest to achieve greater performance levels and that, with this increase, the overall system MTBF has been reduced to just a few hours. Traditional HPC systems require huge capital investment, and we argued that HPC systems in the cloud offer a good alternative. We identified the key research problems and stated our motivation to provide a proactive fault tolerance

framework for HPC systems in the cloud. The major contributions of this thesis were identified.

In Chapter 2, we reviewed related work on fault tolerance and analysed failure rates in HPC systems. Although it is difficult to determine the single most common root cause of failure, we showed that computation-intensive applications are most frequently interrupted by hardware failures, software failures, or human error. We proposed that a good fault tolerance mechanism should be able to mitigate or, in some cases, even eliminate the consequences of failure. We surveyed fault tolerance mechanisms (redundancy, migration, failure making and recovery) for HPC and identified the pros and cons of each technique. Research efforts directed at reducing the time required for saving the checkpoint in persistent storage were described.

Recovery techniques, which are commonly used in HPC systems, were discussed in detail and over 20 checkpoint/restart facilities were surveyed. The rollback feature requirements identified were used to evaluate them and the results were provided in tabular format as an aid for research in this area. The web site of each surveyed checkpoint/restart facility was provided to facilitate further investigation.

Chapter 3 presented the conceptual background of HPC systems in the cloud that are the subject of this thesis. We explained cloud computing architectures and their benefits and identified the two main categories of cloud players (cloud provides and cloud users). The different cloud service models, SaaS, PaaS, IaaS and HaaS, were explained and their challenges with respect to HPC systems in the cloud were presented. The four types of cloud computing available for HPC

systems were discussed.

HPC systems in the cloud usually consist of a head node and compute nodes. The head node and compute nodes are virtual machines configured to fit their roles. The head node usually handles the user login, resource management, jobs assignment and network file system. Compute nodes are normally accessed through the head node. Compute nodes perform the computation-intensive job assigned by the head and produce results. The head node and compute nodes communicate through the Message Passing Interface. Failure of a head node and/or compute node(s) commonly leads to failure of the HPC system.

We identified two additional challenges: system reliability (hardware failures, software failures and virtual machine failures) and cost (which cloud service is cost effective for HPC systems?). This thesis addresses these challenges.

Chapter 4 we described the design and implementation of a proactive fault tolerance framework for High Performance Computing (HPC) systems in the cloud. We derived the cost model for running computation-intensive applications in HPC systems in the cloud. We analysed the dollar cost of providing spare nodes ahead of prediction of failure. We showed that our solution does not rely on the provision of spare nodes ahead of the prediction of failure. We presented experimental results carried out in a real cloud environment.

The experimental results clearly show that the proposed proactive fault tol-erance approach to HPC systems in the cloud can significantly improve the execution time of computation-intensive applications and thereby reduce the dollar cost of running them by as much as 30%. The frequency of checkpointing of computation-intensive applications can also be reduced by 50% with our

FT-Daemon. Our approach can help reduce energy consumption by reducing the wall execution time of computation-intensive HPC applications in the presence of failure of one or more computational nodes.

In Chapter 5, we addressed the research question posed in Chapter 3 (the cost and computational performance of different cloud services). Due to the huge capital investment required to own a traditional HPC system, which typically involves job queuing, using an HPC system in the cloud is a good alternative. Cloud computing offers IaaS and HaaS for deployment of cluster instances, which can be used to run computation-intensive applications. IaaS provides almost ready-to-use clusters with minimal deployment installation tasks. With HaaS, virtual machines can be provisioned to run computation-intensive applications. We have conducted experimental analysis to determine the performance and cost when cloud services IaaS and HaaS are leased to run computation-intensive applications. We showed that the dollar cost of running computation-intensive applications in the cloud can be reduced by as much as 20% when HaaS is leased. We showed that there is no significant impact in performance of the applications when executed on the leased HaaS.

6.2 Conclusions

The theoretical concepts behind the architecture of HPC systems in the cloud that are discussed in this thesis provide a significant resource for researchers investigating the relatively new field of HPC systems in the cloud.

The analysis of failure rates of HPC systems presented in this work shows that

failure is one of the major concerns of the HPC systems community, particularly when the number of processors increases to thousands. The analysis provides results that can be used for further investigation.

The experimental results from the comparison of existing fault tolerance solutions and the proposed proactive fault tolerance approach to HPC systems in the cloud show that the proposed solution can significantly improve the execution time of computation-intensive applications and thereby reduce the dollar cost for running them by as much as 30%. In this proposed solution, the frequency of checkpointing of computation-intensive applications can also be reduced by 50% with our FTDaemon. Based on the experimental results, the proposed approach can help reduce energy consumption by decreasing the wall execution time of computation-intensive HPC applications in the presence of failure of one or more computational nodes.

The experimental analysis to determine the performance and cost when cloud services IaaS and HaaS are leased to run computation-intensive application shows that HaaS is more cost effective for HPC applications. These results indicate to the HPC community that the cost of running HPC applications can be reduced with HaaS without significant computational performance issues.

The solution proposed in this thesis cannot predict failures that occur without prior warning window. For example, sudden power failure and operator's error. Consequently, reactive fault tolerance may be used to recover from the hardware failure.

In summary, we have provided an analysis of the failure rate of HPC systems, an evaluation of checkpoint/restart facilities, and an overview of HPC systems in the cloud (theoretical concepts and architecture). We proposed a proactive

fault tolerance solution for HPC systems in the cloud and identified the most cost effective cloud service for HPC systems in the cloud. In doing so, we have made a significant contribution to the community of HPC systems users through the provision of tools that provide solutions to current problems and useful insights to facilitate further investigation on fault tolerance and other potential applications.

6.3 Future Work

A guide for future research has been provided in each chapter. This section summarises these recommendations and adds some new suggestions.

One area of future work is to use the rollback feature requirements identified in subsection 2.2.6 and the results of the evaluation of checkpoint/restart provided in section 2.6 to investigate how rollback would be applied in HPC systems in the cloud with reduced overheads. At present, the overheads associated with checkpoint/restart activities somewhat limit its usability in HPC systems in the cloud.

Another area that we would suggest for future research involves the creation of efficient tools to reduce the setup time for HPC systems in the cloud, particularly when HaaS is leased. This would reduce the challenges faced by users of HPC systems in the cloud, particularly when they lease HaaS.

We recommend further investigation and experiments to determine the time window between prediction of failure and the time the predicted failure occurs. Such research may help to identify the optimal time window for accessing the risk of failure. A time window is useful for determining if a deferred action policy might be a desirable option. For example, if an HPC system in the cloud is running a

computation-intensive application and the application is likely to finish before the predicted failure occurs, there is no point taking proactive action because the application would have finished execution and the lease would have been relinquished before the failure occurs.

HPC systems in the cloud are relatively a new field. We would recommend that a decade worth of field failure data from HPC systems in the cloud should be collated and analysed. Currently, such field failure data is not available. HPC systems in the cloud field failure data analysis would provide comprehensive insights on the reliability of HPC systems in the cloud. It would also provide detailed information on the factors that causes HPC systems in the cloud failures and how to mitigate unavoidable failures.

The behaviour of VMs, when used in large-scale such as HPC systems in the cloud may be unpredictable. This is particularly notable when IaaS is leased because the VMs running HPC applications may be sharing computer resources with other VMs leased by other cloud users. This can significantly have impact on performance of computation-intensive application running on HPC systems in the cloud. We recommend that mathematical performance model that would focus on predicting and comparing computation-intensive application's performance on HPC systems in the cloud based on HaaS and IaaS infrastructure is investigated.

Appendix A

Table A.1: Future failure prediction and warning state table

T_i	F_i	V_i	C_i	State
0	0	1	1	warning
0	1	0	1	warning
0	1	1	0	warning
0	1	1	1	future failure predicted
1	0	0	1	warning
1	0	1	0	warning
1	0	1	1	future failure predicted
1	1	0	0	warning
1	1	0	1	future failure predicted
1	1	1	0	future failure predicted
1	1	1	1	future failure predicted

Table A.2: Checkpoint/restart facilities

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Zhong and Niel, 2001)[78]	CRAK http://systems.cs.columbia.edu/archive/pub/2001/11/	It requires no modification of OS or application code. Target's processes are stopped before they are checkpointed.	Transparent Kernel module utilities	Supports migration of networked processes, however; it does not support virtualization and multi-threaded process. It works on Linux 2.2 and 2.4 kernel platform	User initiated	It supports TCP/UDP sockets

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Pinheiro, 2001) [149]	Epckpt http://www.research.rutgers.edu~edpin/epckpt/	Supports symmetric multiprocessors and does not require modification of OS or application code in other to use the facility.	Transparent, Kernel level implementation	Supports system V IPC (Semaphores and Shared Memory), fork parallel applications, dynamic load libraries. Linux 2.0, 2.2 and 2.4 kernels.	User initiated and non-periodic.	Cannot checkpoint sockets, timers (sleeping processes will be awakened) and System V IPC Messages Queues

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Condor Team, 2010) [76]	Condor http://www.cs.wisc.edu/condor/	Enabled by the user through linking the program source code with the condor system call library.	Not transparent, library implementation	Supports single processes but multi-process jobs and system calls are not supported. Multiple kernel-level threads and memory mapped programs are not allowed. Works on kernel 2.4 and later	Periodic and user initiated	Interprocess communication is not allowed (e.g., pipes, semaphores, and shared memory)

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Plank *et al*, 1995) [77]	Libckpt http://web.eecs.utk.edu/~plank/plank/www/libckpt.html	It is implemented in user space. It uses copy-on-write and incremental checkpointing mechanism but requires recompiling of the source code.	Not completely transparent. Library implementation	It support files and multiprocessor. It does not provide support for multithread, pipes, Sys V IPC or distributed application	Periodic	Does not support sockets

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Stellner, 1996) [150]	CoCheck http://www.lrr.in.tum.de/Par/tools/Projects.Old/CoCheck.html	User level MPI implementation. CoCheck uses a special process to coordinate checkpoints.	Transparent, library implementation	Supports parallel processes running on multicomputer; CoCheck can be ported to different machine platforms. CoCheck cannot process a checkpoint request when a send operation is in progress [70]	Periodic	Supports TCP sockets

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Ansel *et al*, 2009) [65]	DMTCP http://dmtcp.sourceforge.net/	Coordinated transparent user level checkpointing for distributed applications.	Transparent, Library implementation.	Supports distributed and multithreaded applications. It support Linux 2.4.x and later	Periodic and manually initiated	Provides supports for sockets but does not support multicast and RDMA (remote direct memory access).

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Duell *et al*, 2002) [63]	BLCR https://ftg.lbl.gov/projects/CheckpointRestart/CheckpointDownloads	System-level and MPI implementation for clusters	Transparent	Supports serial and parallel job. It also support single machine or parallel jobs that run across multiple machines on cluster node. It partially supports multithread applications. Its kernel modules are portable across difference CPU architectures. BLCR works on kernel 2.4.x and later.	User initiated	Does not checkpoint or restore open sockets or files like TCP/UDP

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Zandy, 2002) [151]	Ckpt http://pages.cs.wisc.edu/~zandy/ckpt/	Implemented at user-level. Supports asynchronous checkpoints and does not require re-link to programs	Transparent, library implementation	Provides checkpointing functionality to an ordinary program. Linux 2.4 and later	Periodic checkpoint or manual initiation	Does not support TCP/UDP sockets
(Overeinder *et al*, 1996) [152]	Dynamite http://www.science.uva.nl/research/scs/Software/ckpt/#references	User level implementation	Not transparent â€“ requires re-linking of libraries	Supports open files, dynamically loaded libraries and parallel processes (PVM/MPI) but does not support multithread applications. Linux 2.0, 2.2 and later	Periodic	Supports TCP/UDP sockets

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Osman, 2002) [79]	Zap http://www.ncl.cs.columbia.edu/research/migrate/	Uses partial OS virtualization to allow the migration of process domains. It uses Checkpoint-restart mechanism of CRAK using a modified Linux kernel.	Transparent, Kernel module, library.	Supports single-thread and multithread process. It also supports SYS V IPC. Linux 2.4 and later	User initiated	Supports TCP/UDP sockets, devices files.
(Sudakov *et al*, 2007) [153]	CHPOX http://freshmeat.net/projects/chpox/	Systems-level implementation and does not require modification of OS or user programs	Transparent and uses kernel module	Supports files and pipes however multithreaded programs are not supported. Linux2.4 and later	User initiated	Network sockets are not supported

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Ramkumar and Strumpen, 1997) [154]	Porch http://supertech.csail.mit.edu/porch/	Implemented at user level space.	Uses source to source compilation to provide checkpointing solution in heterogeneous environment	Not transparent, recompiling Multithread and distributed applications are not supported.	Periodic checkpoint	File I/O and socket I/O are not supported.
(Gibson) [75]	Esky http://esky.sourceforge.net/	User-level checkpointing and use job freezing techniques (checkpoint/resume) for Unix processes.	Transparent library implementation	Has limited application coverage. Esky can cope with programs that open or mmap() files. Linux 2.2 and later; and Solaris 2.6.	User initiated	Currently works on a limited opening shared libraries with dlopen().

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Sankaran *et al*, 2004) [70]	OpenMPI (LAM/MPI LA-MPI) http://www.lam-mpi.org/	A user-level facility that uses coordinated protocol and BLCR library to checkpoint MPI applications	Not transparent	Uses BLCR facility to checkpoint parallel MPI applications. It works on recent kernels	Periodic	Supports Ethernet, InfiniBand, Myrinet
(Blackham, 2005) [155]	CryoPID http://cryopid.berlios.de/	Uses freeze techniques in checkpointing. It copy the state of a running process and writes it into a file.	Transparent, utilizes dynamically linked library	Supports single thread process. However, it does not support multithread processes. Linux 2.4 and later.	User initiated	Partial support to file descriptors, sockets and X applications

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(William and James, 2001) [156]	Libtckpt http://mtckpt.sourceforge.net/	Implemented at user-level and requires recompiling	Not transparent, Library	Supports multithreaded applications and Linux and Solaris	Periodic	UDP sockets not supported
(Takahashi *et al*, 2000) [157]	Score	No modifications to the application source is required	Transparent, library	Supports parallel applications	Periodic	Supports for Myrinet and Ethernet.

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Fagg and Dongarra, 2000) [158]	FT-MPI http://icl.cs.utk.edu/ftmpi/index.html	Coordinated checkpointing facility and uses messages logging protocol to checkpoint applications.	Not transparent	Supports parallel applications	Semi-automatic	Ethernet, Infiniband, Myrinet

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Ruscio *et al*, 2007) [159]	DejaVu	Coordinated checkpointing facility and implemented in user space. Virtualizes at the OS interface.	Transparent and library implementation	DejaVu supports parallel and distributed applications. Supports forked processes. Permits completely asynchronous checkpoints, it also support anonymous mmap() and incremented checkpointing	Periodic	Supports communication sockets. It supports Infiniband through custom MVAPICH.

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Schulz *et al*, 2004) [74]	C3 (Cornell Check-point(pre) Compiler) http://www.psc.edu/science/2005/pingali/	Application-level checkpointing and does require program modification	Not transparent	Supports single-thread and distributed application. C3 system is easily ported among different architectures and operating systems.	Program initiated	Does not support infiniband and Myrinet

Continued on next page

Continued on next page

Author	**Checkpoint name and link**	**Brief description of the checkpoint**	**Transparent**	**OS/Application coverage**	**Automatic**	**Sockets**
(Bosilca *et al*, 2002) [160]	MPICH-V http://mpich-v.lri.fr/	Its implementation is based on uncoordinated and distributed message logging techniques. MPICH-V relies on the Channel Memory(CM) techniques	Partial Transparent	Supports parallel applications. It uses Checkpoint server scheduler which is not synchronized with checkpoint server. Works in all Unix flavor. It also works in Windows x86 and x64	Automatic	Supports Ethernet and Myrinet

References

[1] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN'07*, pp. 575–584, IEEE Press, 2007.

[2] H. W, Meuer, and others., "TOP500 Supercomputer Sites." http://www.top500.org/. Online: accessed in April, 2013.

[3] USENIX, "Computer failure data repository (cfdr)." https://www.usenix.org/cfdr. Online: accessed in April, 2013.

[4] R. Buyya, K. Branson, J. Giddy, and D. Abramson, "The Virtual Laboratory: A toolset to enable distributed molecular modelling for drug design on the World-Wide Grid," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 1, pp. 1–25, 2003.

[5] R. Buyya *et al.*, "High Performance Cluster Computing: Architectures and Systems (Volume 1)," *Prentice Hall, Upper SaddleRiver, NJ, USA*, vol. 1, 1999.

[6] A. Reuther and S. Tichenor, "Making the Business Case for High

Performance Computing: A Benefit-Cost Analysis Methodology," *CTWatch Quarterly*, vol. 2, no. 4A, pp. 2–9, 2006.

[7] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*. Access Online via Elsevier, 2003.

[8] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010*, pp. 1–25, Springer, 2011.

[9] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *International Conference on Dependable Systems and Networks, 2006. DSN 2006.*, pp. 249–258, IEEE Press, 2006.

[10] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[11] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," in *The Journal of Supercomputing*, vol. 65, pp. 1302–1326, Springer, 2013.

[12] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[13] J. Bartlett, J. Gray, and B. Horst, "Fault tolerance in tandem computer

systems," in *The Evolution of Fault-Tolerant Computing*, pp. 55–76, Springer, 1987.

[14] R. Riesen, K. Ferreira, and J. Stearley, "See applications run and throughput jump: The case for redundant computing in HPC," in *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 29–34, IEEE Press, 2010.

[15] C. Engelmann, *Symmetric Active/Active High Availability for High-Performance Computing System Services*. PhD thesis, Department of Computer Science, University of Reading, UK, 2008.

[16] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[17] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pp. 1–8, IEEE Press, 2007.

[18] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[19] C. Evangelinos and C. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon EC2," *ratio*, vol. 2, no. 2.40, pp. 2–34, 2008.

[20] Amazon EC2, "Amazon Elastic Compute Cloud (Amazon EC2)." http://aws.amazon.com/ec2/. Online: accessed in April, 2013.

[21] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors," *Submitted to Journal of Parallel and Distributed Computing*, 2002.

[22] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2008.

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

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

[25] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current practice and a direction forward in checkpoint/restart implementations for fault tolerance," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, p. 300.2, IEEE Press, 2005.

[26] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal*

of High Performance Computing Applications, vol. 23, no. 3, pp. 212–226, 2009.

[27] W.-C. Feng and K. W. Cameron, "The green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, 2007.

[28] Green500, "Green500 list." http://www.green500.org/. Online: accessed in April, 2013.

[29] I. P. Egwutuoha, S. Chen, D. Levy, and B. Selic, "A Fault Tolerance Framework for High Performance Computing in Cloud," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),*, pp. 709–710, IEEE Press, 2012.

[30] I. P. Egwutuoha, S. Chen, D. Levy, B. Selic, and R. Calvo, "A Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud," in *International Conference on Cloud and Green Computing (CGC), 2012 Second*, pp. 268–273, IEEE Press, 2012.

[31] I. P. Egwutuoha, S. Chen, D. Levy, B. Selic, and R. Calvo, "Cost-oriented Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud," *International Journal of Parallel, Emergent and Distributed Systems*, no. ahead-of-print, pp. 1–16, 2014.

[32] I. P. Egwutuoha, S. Cheny, D. Levy, B. Selic, and R. Calvo, "Energy Efficient Fault Tolerance for High Performance Computing (HPC) in the Cloud," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pp. 762–769, IEEE Computer Society, 2013.

[33] I. P. Egwutuoha, S. Chen, D. Levy, and R. Calvo, "Cost-effective Cloud Services for HPC in the Cloud: The IaaS or The HaaS?," in *2013 Int. Conf. Parallel and Distributed Processing Techniques and Applications, PDPTA'13*, pp. 223–228, 2013.

[34] Wikipedia.org, "Fault-tolerant system." http://en.wikipedia.org/wiki/Fault_tolerant_system. Online: accessed in April, 2013.

[35] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409–418, 1990.

[36] D. Oppenheimer and D. A. Patterson, "Architecture and dependability of large-scale internet services," *IEEE Internet Computing*, vol. 6, no. 5, pp. 41–49, 2002.

[37] C.-D. Lu, *Scalable diskless checkpointing for large parallel systems*. PhD thesis, University of Illinois, 2005.

[38] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann Publishers, 2010.

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

[40] Los Alamos National Laboratory., "Operational Data to Support and Enable Computer Science Research." http://institute.lanl.gov/data/fdata/, 2014. Online: accessed in June, 2014.

[41] M. Nagappan, A. Peeler, and M. Vouk, "Modeling cloud failure data: a case study of the virtual computing lab," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, pp. 8–14, ACM, 2011.

[42] North Carolina State University, "Virtual Computing Lab (VCL)." https://vcl.ncsu.edu. Online: accessed in June, 2014.

[43] K. Czarnecki, K. Østerbye, and M. Völter, "Generative programming," in *Object-Oriented Technology ECOOP 2002 Workshop Reader*, pp. 15–29, Springer, 2002.

[44] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[45] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000.

[46] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[47] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo, "BlueGene/L failure analysis and prediction models," in *International Conference on Dependable Systems and Networks, 2006. DSN 2006*, pp. 425–434, IEEE Press, 2006.

[48] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 87–96, 2004.

[49] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, 1986.

[50] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, no. 12, pp. 1491–1501, 1985.

[51] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware-and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39–51, 1990.

[52] Safeware Engineering Corporation, "Large software state." http://www.safeware-eng.com/White_Papers/Software%20Safety.htm. Online: accessed in April, 2013.

[53] S. Poledna, *Fault-tolerant real-time systems: The problem of replica determinism*. Springer, 1996.

[54] C. Johnson and C. Holloway, "The dangers of failure masking in fault-tolerant software: aspects of a recent in-flight upset event," in *2nd IET International Conference on System Safety*, pp. 60–65, IET Press, 2007.

[55] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 3, pp. 222–238, 1983.

[56] C. Hobbs, H. Becha, and D. Amyot, "Failure semantics in a SOA environment," in *e-Technologies, 2008 International MCETECH Conference on*, pp. 116–121, IEEE Press, 2008.

[57] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," tech. rep., DTIC Document, 2006.

[58] T. Slivinski, C. Broglio, C. Wild, J. Goldberg, K. Levitt, E. Hitt, and J. Webb, "Study of fault-tolerant software technology," tech. rep., National Aeronautics and Space Administration, 1984.

[59] J. Ghaeb, M. Smadi, and J. Chebil, "A high performance data integrity assurance based on the determinant technique," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 614–619, 2011.

[60] I. Courtright, V. William, and G. A. Gibson, "Backward error recovery in redundant disk arrays," tech. rep., Carnegie Mellon University, Pittsburgh Pennsylvania, 1994.

[61] E. Roman, "A survey of checkpoint/restart implementations," tech. rep., Lawrence Berkeley National Laboratory, Tech, 2002.

[62] J. Gwertzman and M. Seltzer, "World-wide web cache consistency," in *Proceedings of the 1996 USENIX Technical Conference*, vol. 8, San Diego, CA, 1996.

[63] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006.

[64] A. Maloney and A. Goscinski, "A survey and review of the current state of rollback-recovery for cluster systems," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 12, pp. 1632–1666, 2009.

[65] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009*, pp. 1–12, IEEE Press, 2009.

[66] A. B. Brown and D. A. Patterson, "To err is human," in *Proceedings of the First Workshop on Evaluating and Architecting System Dependability (EASY'01)*, Citeseer, 2001.

[67] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," in *Sadhana (Academy Proceedings in Engineering Sciences)*, vol. 25, pp. 489–510, Indian Academy of Sciences, 2000.

[68] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, no. 2, pp. 220–232, 1975.

[69] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs, "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 546–554, 1995.

[70] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework:

System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[71] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[72] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," in *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*, pp. 12–20, IEEE Press, 1991.

[73] J. P. Walters and V. Chaudhary, "Application-level checkpointing techniques for parallel programs," in *Distributed Computing and Internet Technology*, pp. 221–234, Springer, 2006.

[74] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, p. 38, IEEE Computer Society, 2004.

[75] D. Gibson, "esky: A slightly portable user-space checkpointing system," *CAP Research, Australian National University, Canberra*, 1999.

[76] C. Team, *Condor® Version 7.3. 0 Manual*. Citeseer, 2000.

[77] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. University of Tennessee, Computer Science Department, 1994.

[78] H. Zhong and J. Nieh, "CRAK: Linux checkpoint/restart as a kernel module," tech. rep., Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.

[79] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[80] R. Teodorescu, J. Nakano, and J. Torrellas, "SWICH: A prototype for efficient cache-level checkpointing and rollback," *IEEE Micro*, vol. 26, no. 5, pp. 28–40, 2006.

[81] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pp. 188–196, ACM Press, 2007.

[82] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pp. 181–192, ACM Press, 2009.

[83] K. M. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM systems," May 8 2001. US Patent 6,230,233.

[84] J. S. Plank and K. Li, "ickp: A consistent checkpointer for multicomputers," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 2, no. 2, pp. 62–67, 1994.

[85] Checkpointing.org, "The home of checkpointing packages." http://checkpointing.org/, march. Online: accessed in March, 2013.

[86] B.-J. Kim, "Comparison of the existing checkpoint systems," tech. rep., Technical report, IBM Watson, 2005.

[87] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The Reliability Wall for Exascale Supercomputing," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 767–779, 2012.

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

[89] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)," *NIST special publication 800-145*, vol. 800, pp. 1–7, 2011.

[90] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC'08*, pp. 825–830, IEEE Press, 2008.

[91] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE'08*, pp. 1–10, IEEE Press, 2008.

[92] SYS-CON Media Inc, "Twenty-One Experts Define Cloud Computing." http://cloudcomputing.sys-con.com/node/612375/print. Online: accessed in April, 2013.

[93] I. P. Egwutuoha, D. Schragl, and R. Calvo, "A Brief Review of Cloud Computing, Challenges and Potential Solutions," vol. 2, no. 1, pp. 7–14, 2013.

[94] V. Mauch, M. Kunze, and M. Hillenbrand, "High performance cloud computing," *Future Generation Computer Systems*, 2012.

[95] S. M. S. P. Ahuja and S. Mani, "The state of high performance computing in the cloud," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, no. 2, pp. 262–266, 2012.

[96] A. Gupta and D. Milojicic, "Evaluation of hpc applications on cloud," in *Open Cirrus Summit (OCS), 2011 Sixth*, pp. 22–26, IEEE Press, 2011.

[97] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pp. 17–20, ACM Press, 2009.

[98] Baremetalcloud, "HaaS provider." http://www.baremetalcloud.com/index.php/en/. Online: accessed in March, 2013.

[99] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *Fifth International Joint Conference on INC, IMS and IDC, 2009. NCM'09*, pp. 44–51, IEEE Press, 2009.

[100] N. Carr, "Here comes haas." http://www.roughtype.com/?p=279. Online: accessed in April, 2013.

[101] Wikipedia, the free encyclopedia, "Microsoft Azure."

[102] Wikipedia, the free encyclopedia, "Google App Engine."

[103] SoftLayer, "HaaS provider." http://www.softlayer.com/. Online: accessed in March, 2013.

[104] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud computing: Principles and Paradigms*, vol. 87. Wiley, 2010.

[105] Xen project, "Cloudo." http://cloudo.com/. Online: accessed in March, 2013.

[106] Aberdeen Group, "Cloudo." http://www.aberdeen.com/Research/Research-Library.aspx?search=private%20could. Online: accessed in June, 2013.

[107] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic, "Modelling collaborative services for business and QoS compliance," in *2011 IEEE International Conference on Web Services (ICWS)*, pp. 299–306, IEEE Press, 2011.

[108] T. Ylonen and C. Lonvick, "The secure shell (SSH) protocol architecture," 2006.

[109] R. Hempel, "The mpi standard for message passing," in *High-Performance Computing and Networking*, pp. 247–252, Springer, 1994.

[110] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang, "PETSc Users Manual Revision 3.3," 2012.

[111] Indiana University, "OpenMPI." http://www.open-mpi.org/. Online: accessed in March, 2013.

[112] Xen project, "Xen hypervisor." http://www.xenproject.org/. Online: accessed in March, 2013.

[113] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[114] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[115] D. A. Reed, C. Lu, and C. L. Mendes, "Reliability challenges in large systems," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 293–302, 2006.

[116] W. G. Ireson, C. F. Coombs, and R. Y. Moss, *Handbook of reliability engineering and management*. IET, 1988.

[117] K.-B. Li, "ClustalW-MPI: ClustalW analysis using distributed and parallel computing," *Bioinformatics*, vol. 19, no. 12, pp. 1585–1586, 2003.

[118] ALTERA, "High-Performance Computing Applications." http://www.altera.com/end-markets/computer-storage/computer/hpc/applications/cmp-applications.html, 2013. Online; accessed August, 2013.

[119] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system,"

in *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID'09*, pp. 124–131, IEEE Press, 2009.

[120] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, pp. 4–16, IEEE Press, 2009.

[121] B. Yang, H. Hu, and S. Guo, "Cost-oriented task allocation and hardware redundancy policies in heterogeneous distributed computing systems considering software reliability," *Computers & Industrial Engineering*, vol. 56, no. 4, pp. 1687–1696, 2009.

[122] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 41–47, 1982.

[123] C.-C. Hsieh and Y.-C. Hsieh, "Reliability and cost optimization in distributed computing systems," *Computers & Operations Research*, vol. 30, no. 8, pp. 1103–1119, 2003.

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

[125] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, 1994.

[126] A. Kumar, L. Shang, L.-S. Peh, and N. K. Jha, "System-level dynamic thermal management for high-performance microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 96–108, 2008.

[127] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, pp. 23–32, ACM Press, 2007.

[128] Trac, "Lm-sensors." http://http://lm-sensors.org/. Online: accessed in June, 2013.

[129] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys (CSUR)*, vol. 42, no. 3, p. 10, 2010.

[130] 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*, pp. 426–435, ACM Press, 2003.

[131] R. S. Michalski, "A theory and methodology of inductive learning," *Artificial intelligence*, vol. 20, no. 2, pp. 111–161, 1983.

[132] J. Thompson, D. W. Dreisigmeyer, T. Jones, M. Kirby, and J. Ladd, "Accurate fault prediction of BlueGene/P RAS logs via geometric reduction," in *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pp. 8–14, IEEE, 2010.

[133] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma, "Making Scheduling" Cool": Temperature-Aware Workload Placement in Data Centers.," in *USENIX annual technical conference, General Track*, pp. 61–75, 2005.

[134] T. J. Jech, *Set theory*, vol. 79. Access Online via Elsevier, 1978.

[135] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," tech. rep., DTIC Document, 1981.

[136] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for HPC systems," in *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pp. 474–486, Springer, 2006.

[137] A. Alex and Y. Dmitry, "iSCSI." http://www.open-iscsi.org/. Online: accessed in March, 2013.

[138] A. Petitet, C. Whaley, J. Dongarra, A. Cleary, and P. Luszczek, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers." http://www.netlib.org/benchmark/hpl/. Online: accessed in March, 2013.

[139] The Trustees of Indiana University, "Transparent Checkpoint/Restart in Open MPI." http://osl.iu.edu/research/ft/ompi-cr/. Online: accessed in August, 2013.

[140] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *25th IEEE VLSI Test Symposium, 2007*, pp. 277–286, IEEE Press, 2007.

[141] B. Feng Chen, *On performance optimization and system design of flash memory based solid state drives in the storage hierarchy.* PhD thesis, The Ohio State University, 2010.

[142] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic acids research*, vol. 22, no. 22, pp. 4673–4680, 1994.

[143] A. G. Carlyle, S. L. Harrell, and P. M. Smith, "Cost-effective HPC: The community or the Cloud?," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 169–176, IEEE Press, 2010.

[144] J. Ekanayake and G. Fox, "High performance parallel computing with clouds and cloud technologies," in *Cloud Computing*, pp. 20–38, Springer, 2010.

[145] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Evaluating the performance impact of Xen on MPI and process execution for HPC systems," in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed computing*, p. 1, IEEE Computer Society, 2006.

[146] Y. Jinhui, N. Alex, C. Shiping, N. Surya, and F. Carsten, "A Performance Evaluation of Public Cloud Using TPC-C Benchmark," in *The 1st International Workshop on Analytics Services on the Cloud (ASC 2012), in conjunction with ICSOC*, pp. 1–7, ICSOC, 2012.

[147] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster:

evaluating amazon cluster compute instances for running mpi applications," in *State of the Practice Reports*, p. 11, ACM Press, 2011.

[148] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, 2011.

[149] E. Pinheiro, "EPCKPT-A Checkpoint Utility for Linux Kernel," *Online at: http://www.research.rutgers.edu/~edpin/epckpt/*, 2002. Online: accessed in June, 2014.

[150] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *The 10th International Parallel Processing Symposium, 1996., Proceedings of IPPS'96*, pp. 526–531, IEEE Press, 1996.

[151] V. Zandy, "Ckpt-a process checkpoint library," 2005.

[152] B. J. Overeinder, P. M. Sloot, R. N. Heederik, and L. Hertzberger, "A dynamic load balancing system for parallel cluster computing," *Future Generation Computer Systems*, vol. 12, no. 1, pp. 101–115, 1996.

[153] O. O. Sudakov, I. S. Meshcheriakov, and Y. V. Boyko, "CHPOX: transparent checkpointing system for Linux clusters," in *4th IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2007. IDAACS 2007*, pp. 159–164, IEEE Press, 2007.

[154] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogeneous

architectures," in *27th Annual International Symposium on Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers.*, pp. 58–67, IEEE Press, 1997.

[155] B. Blackham, "CryoPID: A Process Freezer for Linux." http://cryopid.berlios.de/. Online: accessed in March, 2013.

[156] W. R. Dieter and J. E. Lumpp Jr, "User-Level Checkpointing for LinuxThreads Programs," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 81–92, 2001.

[157] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa, "PM2: High performance communication middleware for heterogeneous network environments," in *Supercomputing, ACM/IEEE 2000 Conference*, pp. 16–16, IEEE Press, 2000.

[158] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent advances in parallel virtual machine and message passing interface*, pp. 346–353, Springer, 2000.

[159] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pp. 1–10, IEEE Press, 2007.

[160] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, *et al.*, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *ACM/IEEE 2002 Conference Supercomputing*, pp. 29–29, IEEE Press, 2002.