Albert Kahira DESIGNING RELIABLE MICROARCHITECTURES ACCORDING TO APPLICATION REQUIREMENTS AGU 2017

DESIGNING RELIABLE MICROARCHITCTRURES ACCORDING TO APPLICATION REQUIREMENTS

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE OF ABDULLAH GUL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

By
Albert Kahira
July 2017

DESIGNING RELIABLE MICROARCHITECTURES ACCORDING TO APPLICATION REQUIREMENTS

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE OF ABDULLAH GUL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

By
Albert Kahira
July 2017

SCIENTIFIC ETHICS COMPLIANCE

I hereby declare that all information in this document has been obtained in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Albert Kahira:

REGULATORY COMPLIANCE

M.Sc. thesis titled *Designing Reliable Microarchitectures according to Application Requirements* has been prepared in accordance with the Thesis Writing Guidelines of the Abdullah Gül University, Graduate School of Engineering & Science.

Prepared By | Advisor

Albert Kahira | Dr. Gulay Yalcin

Head of the Electrical and Computer Engineering Program

Dr. V. Cagri Gungor

ACCEPTANCE AND APPROVAL

M.Sc. thesis titled *Designing Reliable Microarchitectures according to Application Requirements* and prepared by *Albert Kahira* has been accepted by the jury in the *Electrical and Computer Engineering* Graduate Program at Abdullah Gül University, Graduate School of Engineering & Science.

……….. /……….. / ………..

(Thesis Defense Exam Date)

JURY:

Advisor :……………………………….signature

Member :……………………………….signature

Member :……………………………….signature

Member :……………………………….signature

Member :……………………………….signature

APPROVAL:

The acceptance of this M.Sc thesis has been approved by the decision of the Abdullah Gül University, Graduate School of Engineering & Science, Executive Board dated ….. /….. / ……….. and numbered …………..……. .

……….. /……….. / ………..

(Date)

Graduate School Dean

Dr. İrfan Alan

ABSTRACT

DESIGNING RELIABLE MICROARCHITECTURES ACCORDING TO APPLICATION REQUIREMENTS

Albert Kahira
MSc. in Electrical and Computer Engineering
Supervisor: Dr. Gulay Yalcin

July 2017

One of the most important factors to consider when designing a new computer architecture besides cost, energy consumption and performance is reliability. Reliability looks into how often the computer produces the correct results and when it's expected to fail (Mean time to failure). Reliability heavily affects all the other factors such as cost, area and performance and therefore a careful tradeoff has to be made between reliability and the other factors.

One factor that has come into play recently is application requirement. The need for more computing power by applications has been increasing. Because of this, designers have designed much more powerful and sophisticated architectures putting millions of transistors into a single chip and more recently increasing the number of chips. However, this has increased the likelihood of failures occurring. A study of these failures and the reliability of this microarchitectures is therefore required.

In this study, we investigate the reliability of current micro architectures for different applications and further propose reliable microarchitectures for those applications or mechanisms to adjust reliability parameters based on the application. We mostly focus on fault tolerance as a reliability parameter.

Keywords: Reliability, fault tolerance, hardware transactional memory.

ÖZET

UYGULAMA İHTİYAÇLARI DOĞRULTUSUNDA GÜVENİLİR İŞLEMCİLER TASARLANMASI

Albert Kahira
Elektrik ve Bilgisayar Mühendisliği Ana Bilim Dalı Yüksek Lisans
Tez Yöneticisi: Dr. Gulay Yalcin
Temmuz -2017

Bir bilgisayar mimarisi tasarlanırken; maliyet, enerji tüketimi ve başarımın yanı sıra düşünülmesi gereken en önemli etkenlerden birisi de güvenilirliktir. Güvenilirlik, bir bilgisayarın ne kadar süre doğru sonuç ürettiğininin ve ne sıklıkla çöktüğünün ya da yanlış sonuç ürettiğinin ölçümüdür (MTTF: İki çöküş arasında geçen süre). Güvenilirlik, diğer tüm faktörleri yani bilgisayarın kapladığı alanı, maliyetini ve başarımını önemli ölçüde etkilediği için, bilgisayar tasarımı sırasında güvenilirlik ve diğer faktörler arasında doğru bir denge kurmak gerekmektedir.

Güvenilirlik konusunda son zamanlarda kullanılmaya başlanan etmenlerden bir tanesi de uygulama gereksinimleridir. Her geçen gün uygulamaların ihtiyaç duyduğu bilgisayar hesap yapma gücü artmaktadır. Bu sebeple de tasarımcılar her seferinde daha güçlü ve karmaşık bilgisayarlar tasarlayıp; önce tek bir çipe milyonlarca transistör yerleştirmiş ardından da bir çipte yer alana çekirdek sayısını artırmaya başlamışlardır. Ancak bu durum, bilgisayar sistemlerinde hata oluşması ihtimalini artırmıştır. Bu sebeple de oluşan bu hataların ve mikroişlmecilerin güvenilirliğinin incelendiği çalışmalara ihtiyaç duyulmaktadır.

Bu çalışmada, günümüz mikroişlemcilerinin farklı uygulamar açısından güvenilirlik ihtiyaçları incelenmiş, bunun devamında bu uygulamalar için güvenilir mikroişlemci tasarımları ve uygulama ihtiyaçları doğrultusunda güvenilirlik parametrelerini ayarlayacak mekanizmalar önerilmiş. Çalışma kapsamında güvenilirlik ölçütü olarak hata müsamaha değeri kullanılmıştır.

Anahtar kelimeler: Güvenilirlik, hata müsamaha oranı, donanımsal geçişli bellek.

Acknowledgement

First and foremost I would like to express my sincerest gratitude and appreciation to my mentor and adviser Dr. Gulay Yalcin. Thanks for making me fall in love with computer architecture, introducing me to research and for the endless effort you put to make me a person able to complete a thesis. The uncountable hours you spent mentoring me and your selflessness will forever be appreciated.

My appreciation also goes to members of the Jury Dr. Cagri Gungor and Dr. Selcuk Okdem for reading through this thesis and participating in my defense. The information and suggestions they provided during my defense have helped to make this even better and more refined.

I would also like to thank Dr Osman Unsal, Dr Leornardo Bautista and Dr Adrian Christal of Barcelona Supercomputing Center(BSC). The 3 months I spent in barcelona was the beginning of this thesis and their mentorship, help and suggestions for all the experiments conducted during my time at BSC have all shaped this thesis.

My appreciation also goes to Dr. Pascal Feber of University of Nauchatel for granting us the permission to use POWER 8 machine and all the help for the experiments that were conducted with their lab especially on hardware transaction memory.

I would also like to thank Dr. Zenmei Ohkubo for the invaluable tips and help with Linux experiments. For the late night calls I made and hours spent in your office trying to fix my Linux scripts.

My appreciation also goes to fellow grad students in the grad students office. The time we spent together and your support during my thesis can never be forgotten.

Finally I would like to express my deepest gratitude to my family and friends especially my mother. For their understanding and endless support during the entire thesis period. I am forever grateful to have you and call you family.

Table of Contents

1 Introduction **1**

1.1 Problem Statement and Contributions of this Thesis 5

1.2 Thesis Organization 5

2 Background **7**

2.1 Reliability 7

2.2 Faults in Computer Systems 7

2.2.1 Transient Faults 9

2.2.2 Intermittent Faults 9

2.2.3 Permanent Faults 9

2.2.4 Effects of Hardware Faults on Applications 10

2.3 Reliability Schemes 11

2.3.1 Error Detection Schemes 11

2.3.2 Error Correction Schemes 12

3 Resilience Characterization of Artificial Neural Networks **13**

3.1 Introduction 13

3.2 Background and Related Work 15

3.2.1 HPC Systems Reliability 15

3.2.2 Fault Tolerance in Artificial Neural Networks 16

3.2.3 Fault Injection Characterization on other Applications 17

3.3 Fault Tolerance Assessment 17

3.3.1 Benchmark Applications and Simulation Environment 17

3.3.2 Fault Injection 19

3.4 Evaluation 21

3.4.1 Vulnerability of Different Neural Network Layers and phases . . 22

3.4.2 Vulnerability of Processor Registers 23

3.4.3 Characterization of Errors for Different Input Datasets 25

3.4.4 Vulnerability of ImageNet DNN 26

3.5 Discussion and Suggestions 26

3.6 Summary 28

4 Hardware Transactional Memory for Tolerating Transient Hardware Errors 29

4.1 Introduction 29

4.2 Previous Work 31

4.3 Hardware Transactional Memory Implementations 33

4.3.1 IBM POWER8 33

4.3.2 Intel TSX 34

4.4 Proposal 36

4.4.1 Preliminary Work 36

4.4.2 Design Methodology 38

4.4.3 Challenges of the Design 41

4.5 Proof of Concept 42

4.6 Summary 43

5 Conclusion 44

5.1 Future Work 44

A Example 52

B Bubble Sort Example 55

List of Figures

1.1 Von Neuman Architecture 2
1.2 Software Size 3
1.3 Hardware Trends 3

2.1 Effect of faults 8
2.2 Bathtub curve representing permanent faults 10

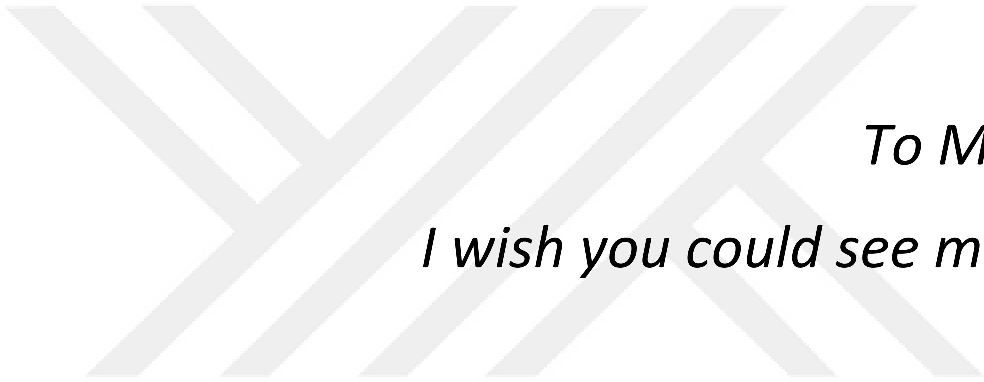
3.1 A simple artificial neural network 18
3.2 A deep neural network (DNN) 19
3.3 Fault Injection Process Using PIN 20
3.4 Injecting in the weight matrix between the input layer and the hidden layer. 22
3.5 Injecting in the weight matrix between the hidden layer and the output layer. 23
3.6 Faults during training 24
3.7 Effect of faults on general purpose registers. 25
3.8 Effect of faults on different images 26
3.9 Average error in prediction. 27
3.10 Average error in prediction. 27

4.1 POWER8 Processor 35
4.2 Performance of STAMP applications implemented with TinySTM 37

List of Tables

3.1 General Purpose Registers and their Functions 21

4.1 PowerPC HTM High Level Inline Functions 33



To My Dad

I wish you could see me now

Chapter 1

Introduction

The history of computers is a rather interesting one. Early computers were intended for military and scientific community and required high level of expertise to use. They occupied huge spaces and cost thousands of dollars. Over the last decades, computers have continued to shrink in size, finding their way to our offices, homes and most recently even to our pockets. This improvement can be attributed to significant breakthroughs made in hardware such as silicon technology and up until recently Moore's law.

But its not only computer hardware that has seen such a drastic change. Software, especially application software has gone from a few lines of code running on a simple hardware to today's millions of code optimized to run in multiple cores in a coordinated way, drastically cutting down execution time. High level programming languages have enabled pretty much everyone to create their own programs to computerize their tasks. Recently we have seen the emergence of machine learning applications that will revolutionize the way we use and interact with computers [20].

An application is a program or group of programs that is designed for the end user. Applications enable the user to perform a group of coordinated functions, tasks, or activities. They range from word processors to complicated computer vision applications. However, they run the the same basic principles and design of Von Neumann architecture (Shown in 1.1) . As such, where basic hardware design has remained relatively same over the years, applications and application design has changed. This is evident by looking at the code that took man to the moon for the first time in 1969 which was recently published on Github.

The number of lines, a general measure of application size, has continued to increase. Facebook for instance has over 60 million lines of code and a modern self driven car can have upto 100 million lines of code. The number of lines of code in a Boeing exceed all other parts including bolts [1] . Figure 1.2 shows this trend and compares the size of different applications. Sophisticated applications that run on multiple processors utilizing major breakthroughs in hardware such as multithreading and parallelism. This

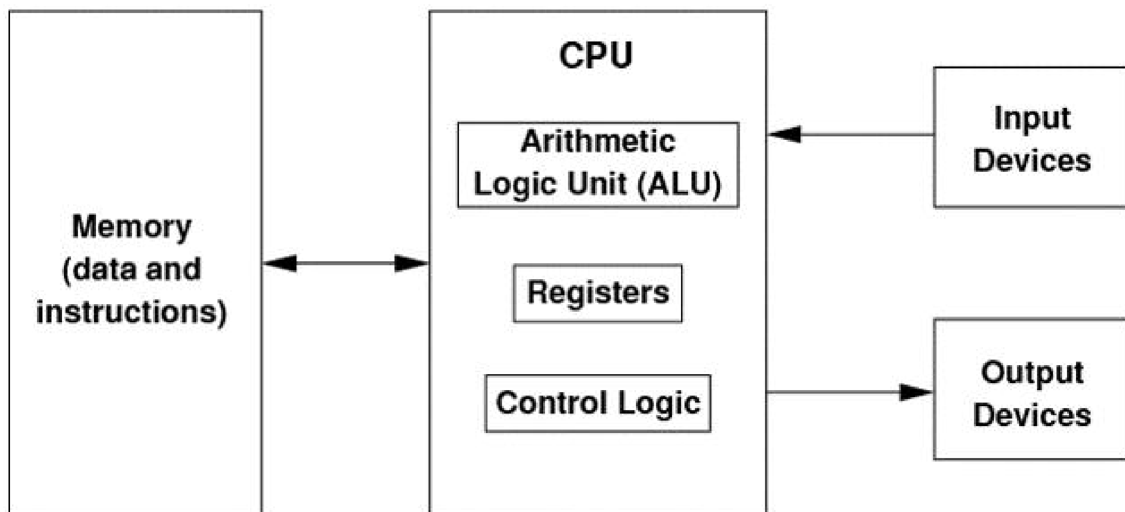


Figure 1.1: Von Neuman Architecture

of course has led to the emergence of better applications, capable of performing tasks and great speeds and extremely high precision. It has also led to an increase in the number of tasks performed by computers.

This development in software can be accredited to more superior and advanced hardware. The Apollo Guidance Computer (AGC) that took man to the moon had approximately 64KB of memory and operated at 0.043MHz. Today a simple phone has over 2GB of memory and clocks at speeds hundreds of folds than AGC. Figure 1.3 shows trend in hardware in the last several years. These developments have made computers part of our daily lives. We have integrated computers so much into our lives that its hard or even impossible at times to live without them. Mobile phones, washing machines, cars, microwaves, airplanes and medical equipment, all rely on some kind of computers.

Superior hardware has come at a cost. Hardware designers and computer architects have to make careful trade-offs between different design constraints such as cost, energy consumption and area. For instance, high clock speeds would mean more heat dissipated and the shrinking size of a single core would require lower voltage which may in turn lead to low performance. Another factor that computer architects have to consider is reliability. Reliability looks into how often a computer produces the correct results and when its expected to fail (Mean time to failure). Reliability heavily affects all the other factors such as cost, area and performance and therefore a careful trade off has to be made between reliability and the other factors [44] .

An unreliable system can be catastrophic as shown by numerous case studies. For

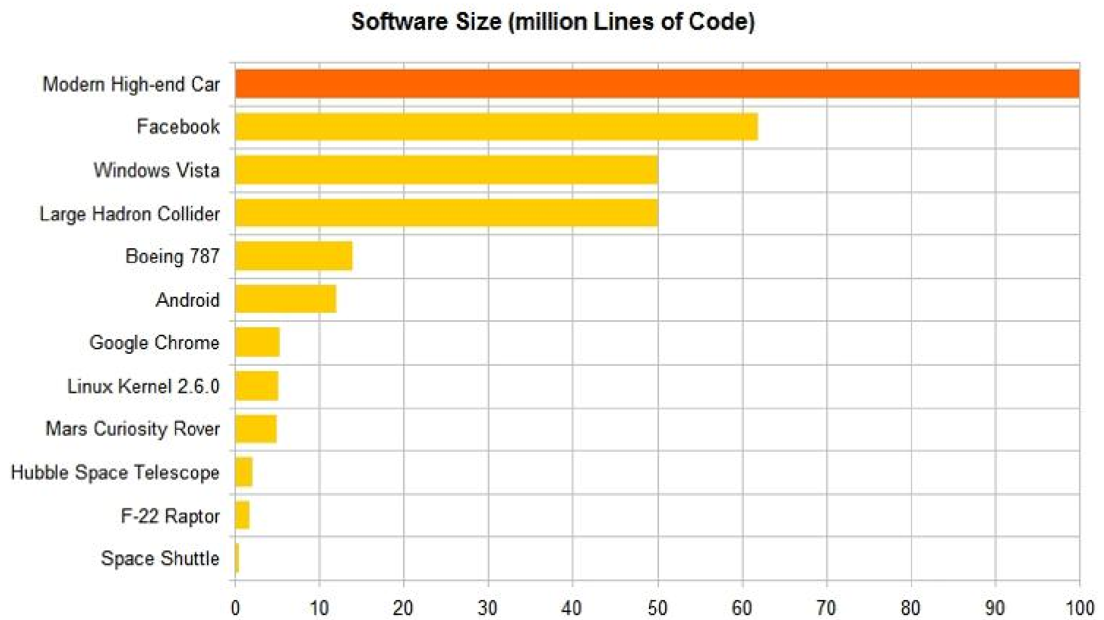


Figure 1.2: Software Size

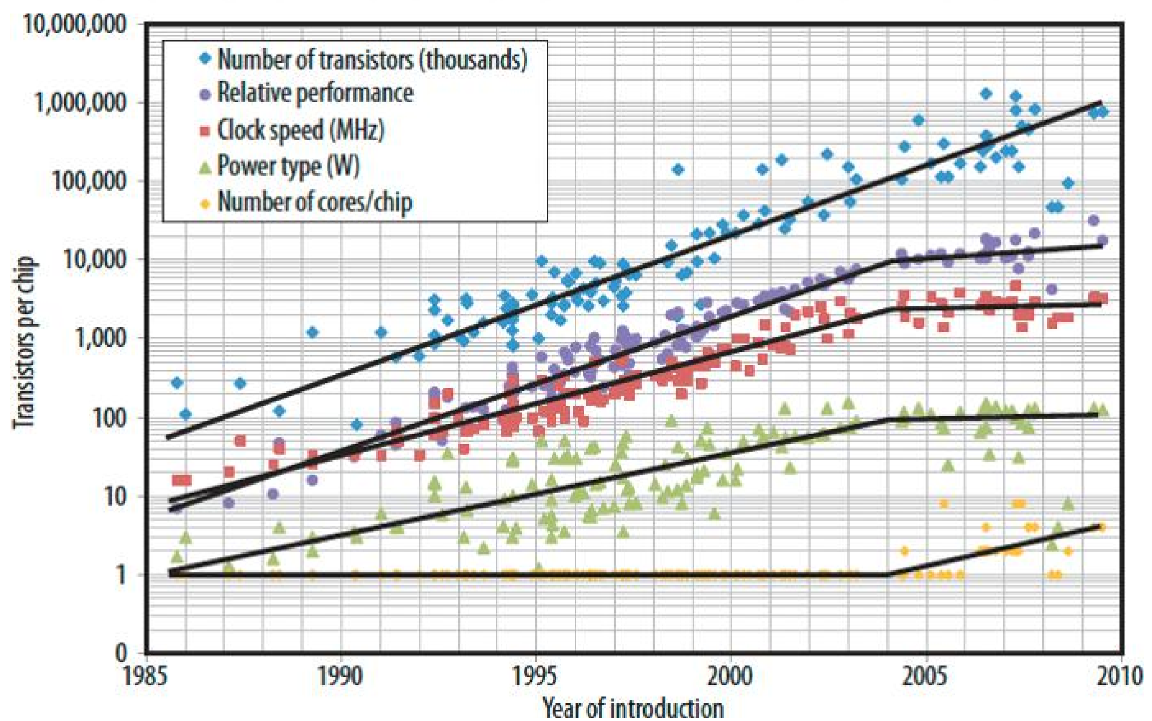


Figure 1.3: Hardware Trends

example, on July 20th 2008, a single corrupt bit completely collapsed Amazons S3 cloud computing Service for almost 8 hours [5]. Such a failure could have been very fatal if it involves very sensitive data. Nevertheless, it damaged the company reputation. On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana[6]. Billions of dollars and tons of scientific information were lost because a computer system was not reliable. There are many more examples where computers have proved not so reliable. Recently a fatal crash occurred with Tesla model S where autopilot was activated [37]. This accident has raised numerous questions about the reliability of machine learning and artificial intelligence applications. Coming at a time when there is a significant increase in AI applications, a reliability study of these applications could prevent future disasters.

There are several error correction mechanisms implemented in the literature [27]. These error correction mechanisms increase reliability but come at a cost. They mean an increase in cost, power consumption at a hardware level [4] and more code at a software level. It is because, in order to detect and correct errors, some degree of redundancy must be introduced in the architecture [8] which significantly increases the cost. Furthermore, most of these methods are applied homogenously in a one fits it all manner as shown by Yixin et all [29] .

Applications however don't respond the same to hardware faults [29] . Wheres a faults may cause one application to crash or give faulty results, it may go unnoticed in another application. For instance, when a computer error occurs in a general-purpose computer such as laptops used in our daily lives, the effect is not big and may even go unnoticed. However, a fault in a computer monitoring the heart of a patient could be catastrophic. This means a homogeneous approach to reliability such as ECC and redundancy is not wise economically and there is a need for application specific architectures. It is,therefore, important to understand applications robustness in order to design reliable application specific architectures. This will in turn reduce cost.

If the entire application can be understood and somehow broken down into parts based on the sensitivity of each part, then tremendous gains can be made in terms of performance, energy requirements and cost. For example, parts of an application that are not so prone to errors can be executed at higher speeds than others increasing performance while delicate parts can be executed at lower speeds saving energy and increasing relia-

bility. This will bring an end to the idea of setting a general reliability measurement an introduce a form of dynamic reliability management as suggested in [29] .

1.1 Problem Statement and Contributions of this Thesis

As explained earlier, there is need for application based reliability designs. However, in order to come up with such designs, a proper analysis of applications is required. This thesis is an initial attempt to solve the problem of application based reliability. It shows and proves the need for such designs by analyzing several well known applications. It further goes ahead to propose a design that utilizes existing hardware to improve reliability of applications. This thesis studies reliability requirements of different applications and to further proposes reliable microarchitectures for those applications or mechanisms to adjust reliability parameters based on the applications.
The aims of this thesis can therefore be summarized as follows:

- Determining the reliability requirement of applications
- Showing the need for heterogeneous microarchitecture designs which can allocate components according to the reliability requirements of the application or the data used by the application
- Reducing the overhead of reliability so that reducing the energy consumption as well as increasing the lifetime of computers.

In the end, this thesis aims to achieve the following:

- A method classifying applications or application sections according to their reliability requirements
- A prototype of a heterogeneous microarchitecture in terms of reliability requirements

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives a background of reliability. It looks into things such as fault tolerance, hardware faults and existing reliability methods. Chapter 3 gives resilience characterization of artificial neural networks, application

we select to evaluate our hypothesis. Chapter 4 proposes using Hardware Transactional Memory as a means to increase reliability of some applications. Finally chapter 5 gives a conclusion.

Chapter 2

Background

In this chapter, we give a detailed background of faults in computer systems and different reliability schemes proposed to tolerate these faults. We start by giving a general overview of reliability and its importance. Then we cover different faults in computer systems and categorization. Finally we cover the existing reliability schemes.

2.1 Reliability

Reliability is the measure of how often the computer produces the expected results. In other words, its how much reliable a computer system is. This is very important because, as we shall show later, computer systems experience all sorts of defects and don't guarantee to always give the correct results. Reliability is therefore a key design constraints among others such as cost, area and power.

However, as stated earlier, reliability comes with a cost and careful trade off has to be made between reliability and other design constraints. The cost of reliability besides actual economic cost includes the following.

- Energy overhead
- Hardware redundancy
- Performance compromise
- Design overhead

2.2 Faults in Computer Systems

In computer systems, a fault is said to occur when there is a defect in one or several components of the system. Faults can occur in both software and hardware, however, this thesis focuses on faults in hardware components. Faults in hardware are hardware defects that can manifest themselves as errors. However, not all faults cause errors since

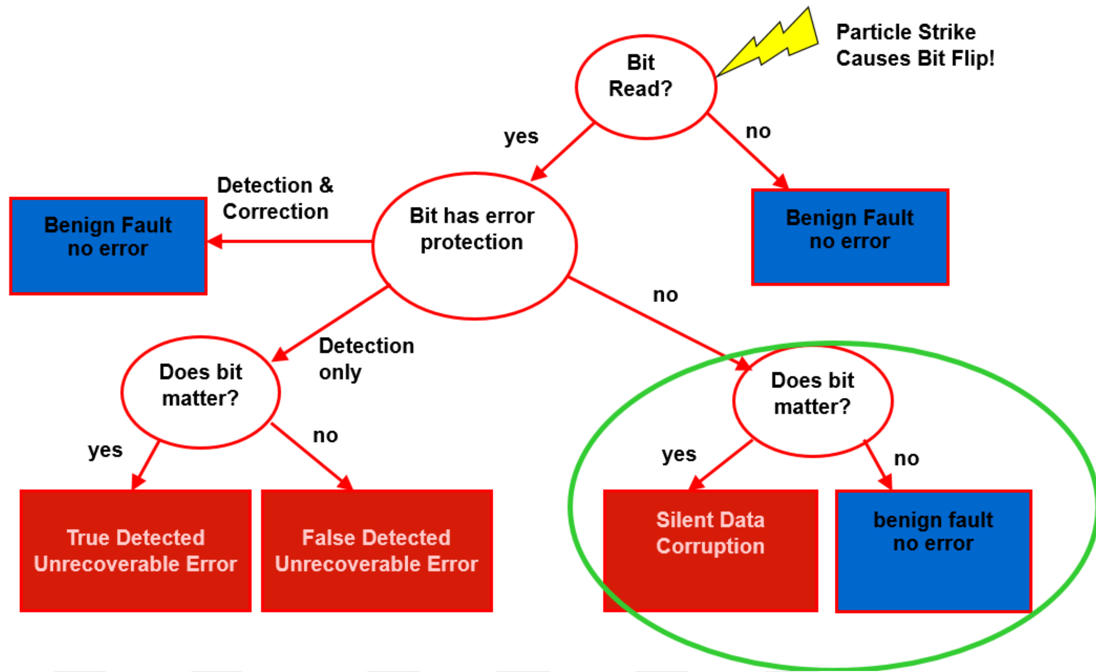


Figure 2.1: Effect of faults

some faults are corrected by existing fault tolerance mechanisms while others are not of enough magnitude to manifest themselves as visible errors and therefore go unnoticed. Several causes of faults such as particle strike, manufacturing defects, and age have been identified as causes faults.

Recently hardware faults have been on the rise and this trend is expected to continue due to shrinking size of micro processors, increasing number of transistors and low voltage designs [7]. As stated earlier, when a fault is not corrected, it may lead to an error or it might just go unnoticed.

When a particle strike or any other factor causes a bit flip, the bit will be either read by the next instruction or not. If its not read, this a benign fault and causes no error. However, if the bit is read, the outcome will be determined by whether the bit is protected by error correction and detection scheme or not. If the bit is protected, then this is also a benign fault and causes no error since the error correction mechanism can handle the fault. If there is no correction mechanism and the bit matters, that means its important for the next instruction then silent data corruption occurs. If the bit doesn't matter then its just another benign fault. If the bit has only error detection mechanism and no form of correction, then it can lead to a unrecoverable error. Figure 2.1 summarizes everything explained above.

Faults in hardware components, our key are of interest in this thesis, can be categorized as follows:

- Transient
- Intermittent
- Permanent

We look at each of the fault categories.

2.2.1 Transient Faults

Transient faults, commonly known as soft errors, are faults changes in bit values(bit flips) that occur due to particle strike or radiations. This faults are temporary and the corrupted bit can be corrected when it is overwritten by another instruction data. Even though transient faults are temporary, they could have drastic effects on an application. As such several measures have been implemented to detect and recover from transient faults.

2.2.2 Intermittent Faults

Intermittent faults on the other hand are faults that can be loosely defined as several bits getting stuck at a certain point for a while, i.e couple of cycles or even milliseconds. They occur due to unstable hardware and can be activated by temperature or voltage changes. Traditionally, intermittent faults have been considered as the beginning of permanent faults. Intermittent faults and their effects have been studied comprehensively previously [30]

2.2.3 Permanent Faults

Finally, permanent faults are irreversible changes that occur in semiconductor either during manufacturing or later on in their life due to aging [43]. Figure 2.2 shows a bathtub curve that is generally used to represent permanent faults and their occurrence during the life time of a chip. At the beginning, there is a high chance of permanent faults in a chip due to manufacturing and packaging defects. However, once the chip is operational, in continues operating with minimal permanent faults until it ages out. After a certain age, a chip then begins to experience more permanent faults due to age.

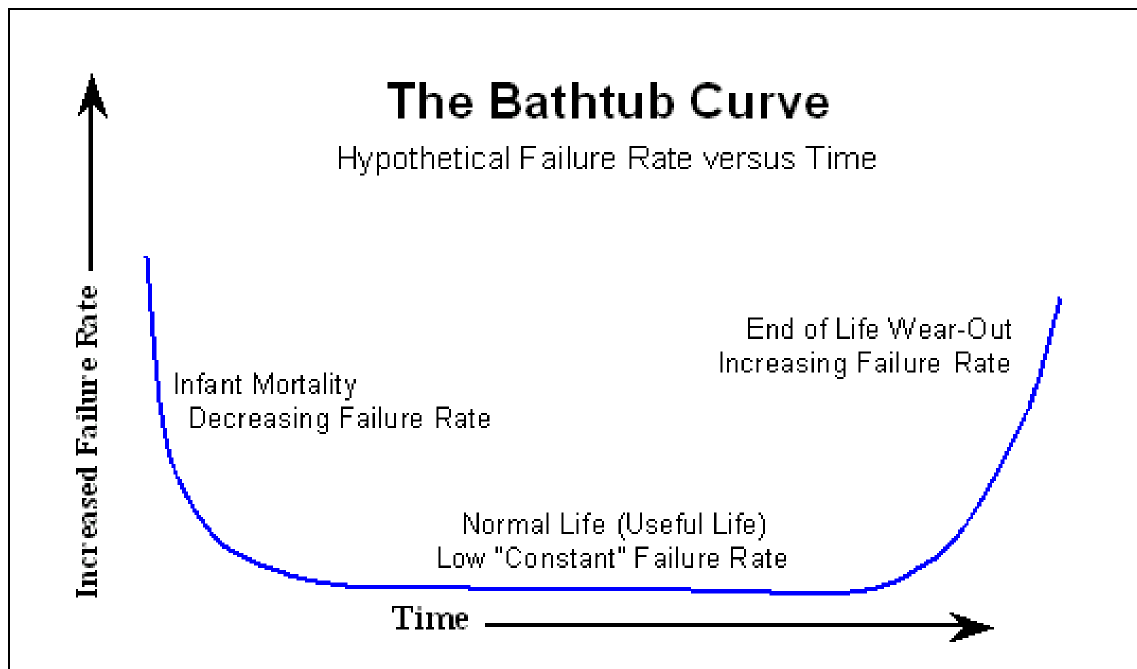


Figure 2.2: Bathtub curve representing permanent faults

2.2.4 Effects of Hardware Faults on Applications

When one or more of the above mentioned hardware faults occurs in a system, several things are bound to happen. This could have varying effects on an application as listed below.

- Application produces incorrect results
- Application slows down
- Application crashes
- Error is masked and nothing happens(benign faults)

Depending on the application, incorrect results could be catastrophic such as the case of mission critical applications. An application taking longer than usual to complete could hold up significant resources which is extremely costly especially in cases of high performance computing (HPC) systems.

Gracia-Morn et al. have studied the effects of intermittent faults on Reduced Intruction Set Computing (RISC) [13]. They underline the fact that intermittent faults are gaining importance with the continued reduction in transistor size. They also compare the

effects of both transient and permanent faults and intermittent faults. Finally they stress the importance of having several techniques to detect and correct intermittent faults.

Wei et al also look into the the effects of both intermittent and transient hardware faults on programs [41]. They attempt to answer the question whether there exists considerable differences in how intermittent faults affect applications as compared transient faults. Such a study would therefore address the need or lack thereof of a novel techniques to address intermittent faults different from those currently implemented to address transient faults. They concluded that the impact of intermittent faults on a program is different from that of transient faults and therefore techniques that increase resilience to transient faults are not exactly effective for intermittent faults.

Since hardware faults affect also the memory, the effects of memory faults on applications has also been studied. Luo et al characterize the errors in memory in attempt to reduce the cost of a data center [29]. They show that the traditional one fit all approach is not viable as faults in memory have varying effects on different applications. They therefore propose and develop a methodology to classify fault tolerance of applications to memory errors. They show that using heterogeneous reliability techniques can greatly reduce the cost of data centers.

2.3 Reliability Schemes

2.3.1 Error Detection Schemes

Error detection is the process of identifying or becoming aware that an error has occurred. Even though the occurrence of an error can be easily identified by a program crash or an obvious false result, its not always the case especially with transient faults or silent errors. Several schemes have therefore been proposed in the past to detect both intermittent and transient faults, the most famous one being error correction codes (ECC). Most of these schemes fall into three categories :

- Redundant execution
- Encoded Processing
- Monitoring Error Symptoms

Redundant execution is the most common error detection scheme. In its simplest form, redundant execution involves executing an operation two or more times and then comparing the results. Lack of uniformity in the output generated by the operation means an error has occurred. This technique is common in mission critical applications. Encoded processing involves using Error Correction Codes (ECC) to detect and correct soft errors. Finally the system can also be monitored for errors. This can be manifested through false prediction in high confidence branches or increased activity in the operating system

2.3.2 Error Correction Schemes

Error correction or error recovery is an attempt to restore normalcy after an error has occurred. Error correction schemes can be divided into two major categories

- Forward Error Recovery (FER)
- Backward Error Recovery (BER)

FER is based on replicating the execution in order to use the correct results if the actual execution fails. BER (also called checkpoint/rollback mechanism) stores an error-free state of the system (checkpoint) and reverts the system state upon error detection (rollback).

Chapter 3

Resilience Characterization of Artificial Neural Networks

3.1 Introduction

Artificial Intelligence (AI) is now playing a big part in our day to day life and this is likely to increase tremendously in the near future. Artificial Neural Networks (ANN), which are utilized in many contemporary AI applications, are computational models inspired by neurons to solve many computationally complex problems such as character recognition, image compression, speech recognition and recently computer vision. ANN is attractive for those sort of problems due to its continuously learning behavior from given data. An ANN consists of interconnected neurons that process information. While a basic ANN may consist of an input layer,hidden layer and an output layer, recently, deep neural networks(DNN) with many hidden layers have been proposed and implemented. ANNs have also been directly implemented on hardware, a concept knowns as neurocomputers [33], however, such implementations are beyond the scope of this work.

Due to their nature, ANNs are computationally very intensive applications. When very large data sets are involved, training ANNs could take days. Because of this, ANN are now deployed in high-performance computing (HPC) systems where they utilize high computational power and massive parallelism offered by HPC systems. It is widely agreed that future implementations of ANN and AI applications will continue to be deployed in HPC systems and large data-centers.

However, resilience is the one of the major challenges in HPC systems, since the rate of the hardware failures such as transient and permanent faults is significantly high in HPC systems. A transient fault is a bit flip occurring mainly in processor structures (i.e. register files, caches etc.) due to radiation events, thermal conditions or power supply noise. Permanent faults, on the other hand, are irreversible physical changes in the semiconductor devices.Unfortunately, transient and permanent faults will continue to be

a major concern in HPC systems [5], due to processor technology node shrinking thus making transistors more susceptible to faults and due to the fact that the number of cores in HPC systems is expected to rise up to several millions.

Hardware faults may lead to different consequences in the result of the running application. While it is possible that a single bit flip on a critical register can cause an entire application to crash, it is also possible that an undetected bit flip causes the application to produce a considerably different result than the correct output; this is called silent data corruption (SDC). For instance, in the case of an ANN application trained to recognize certain patterns such as characters or images, a single bit flip could lead to predicting the character or image falsely (e.g. recognizing an handwritten A character as B). It can also lead to the ANN taking longer than usual to complete or being stuck at a certain point. This holds the much needed resources which is very costly in the case of HPC.

In order to tolerate hardware faults, a HPC system requires several robust resilience mechanisms for detecting errors and preventing them. However, providing resilience to any system causes significant overheads in performance, energy consumption and hardware cost. Hence, it is impractical to keep the entire HPC system completely fault-free due to those overheads. Therefore, it is essential to evaluate the most vulnerable system components in order to trade off between resilience and its overheads. For instance, understanding the extent to which faults affect the application and knowing the locations where faults cause the highest impact is important for reducing the reliability cost since such information will be useful when designing error detection and prevention mechanisms.

In this chapter, our goal is to look into effects of hardware faults on ANN. We analyze the overall robustness of both classical ANN and DNN. We are motivated by the fact that ANN applications, an important subset of AI applications, will be exposed to many of the reliability issues experienced in HPC systems and a deep understanding of these faults is required. Thus, we perform a complete fault injection analysis on two standard ANN benchmarks, first a simple ANN that represents the basic principles in ANN and then a more sophisticated deep neural network that represents current and emerging trends in ANNs.Fault injection is a widely used experiment-based resilience evaluation approach in which faults are introduced to the system by changing the value of randomly selected bits [12].We injected faults to (1)different phases of the ANN applications (i.e

training and testing phases), (2) different architectural registers and (3) different layers of the ANN.

We have several take-away points from our ANN resilience analysis. Our results show that in a neural network with three layers (i.e., input, hidden and output layer) hardware faults occurring in the weight matrix between hidden layer and output layer affect the outcome more than faults between input and hidden layer. Our results also show that a fault on distinct X86 registers (X86 being the most often deployed processor in the HPC data-centers) have very different vulnerability profiles for the ANN application: faults on some registers (e.g. the RSP) almost always manifest themselves as a crash, while faults on other registers (e.g. the RDX and RBP) causes SDC. We also find that some faults in architectural registers cause the program to take longer than usual and even in some cases not to complete at all. Finally, we find that the likelihood of a fault in the input causing a misclassification - in this case an SDC - depends highly on which digit is being recognized. The contributions of this chapter are summarized as follows.

- We show the effects of faults on different architectural registers
- We show the vulnerability of different layers of ANN
- We show how errors affect a DNN
- We discuss possible design choices for the reliability of ANNs.

The rest of this chapter is organized as follows. Section 3.2 gives a background of several fault tolerance studies of ANN and reliability on HPC systems. Section 3.3 explains in detail our fault injection environment and the experiments conducted. Section **??** presents and discusses the results obtained. Finally section 3.6 concludes this paper.

3.2 Background and Related Work

In this section we provide background information and related work on fault tolerance of ANNs and faults in HPC systems.

3.2.1 HPC Systems Reliability

HPC systems consist of tens of thousands of nodes with hundreds of thousands cores working together on solving a specific scientific model. These systems offer the engi-

neering and scientific community the much needed computing power. The number of processors continues to rise and the feature size of those computing devices continues to shrink. For instance, it is expected that the number of cores reach several million cores running up to a billion threads [5] in the coming years. Although the expected lifetime of each one of those processing units is measured in tens of years, the mean time between failure (MTBF) of a system with tens of thousands of such devices is measured in hours.Moreover, power constrains is forcing system architects to design systems that work near the power thresholds, which increases the likelihood of errors [2].

3.2.2 Fault Tolerance in Artificial Neural Networks

In the ANN domain, fault tolerance is the ability of a system to continue to perform to specification in the presence of faults, such as broken connections, connections with erroneous weight or neuron with inaccurate outputs [23] . ANNs are generally considered fault tolerant due to their parallel structures [33]. As such, they will perform as expected in the presence of errors as their parallel nature offer some sort of redundancy. This argument has been the basis and justification for the usage of ANN in mission critical applications or applications where no repair mechanism is available such as space explorations [40]. However, as it has been pointed out [34], neural networks cannot be considered completely fault tolerant. An artificial neural network with no built in fault tolerance could be disastrously affected by faults [6]. Most of the work done on fault tolerance of ANN has focused on faults that occur within the neural network at the application level. Also, there has been studies about faults in hardware implementations of ANN applications. Other works [33] identify the following as faults that can occur independently of hardware.

1. Fault in connection/weight or multiplier
2. Fault in an input
3. Fault in activation function

The above mentioned faults have been the focus of other studies [22]. Even though extensive work has been done on the fault tolerance of ANN, with many researchers concluding that ANN are inherently fault tolerant, not much has been done on the impact of

hardware faults on ANNs especially on HPC systems. In this paper we analyse hardware errors that manifest themselves as bit flips in architectural registers. These errors stem from faults occurring in the processor pipeline and are more difficult to detect and correct compared to memory errors which can be detected and corrected through ECC hardware.

3.2.3 Fault Injection Characterization on other Applications

Previous work on characterization of vulnerability of applications has been vital in designing fault tolerance techniques. Weining et al. studied the behavior of Linux kernel under errors and concluded that a large number of errors result in a total crash of the operating system and require file system reformatting [14].

Fault injection campaigns have been conducted to study the resilience of various class of applications ranging from supercomputing [3] , desktop [42] [35] , multi-threaded [24] to GPGPU [17]. Only few of the above papers studied the impact of faults on hardware structures such as registers [35] [42] . In particular, Yalcin et al. presented a fault injection tool in the microarchitecture level simulator [42]. They evaluated the vulnerability of SPEC2006 applications in the presence of faults in different micro-architectural hardware structures. Rashid et al. characterised the impact of intermittent faults on SPEC2006 benchmark by injecting faults in micro-architectural simulators [35] . The study mostly focuses on errors that lead to the program crashing and the intention is to diagnose those errors.

The motivation of such studies has been to detect , diagnose and offer recovery mechanisms. In this study, we extend this analysis to the application domain of ANN and we study the impact of faults on registers as well.

3.3 Fault Tolerance Assessment

This section explains the fault simulation environment, fault injection process and the assessment of these faults.

3.3.1 Benchmark Applications and Simulation Environment

Two well known benchmark applications are used in this study. MNIST, which represents classical ANN applications and ImageNet which represents emerging trends in ANN such

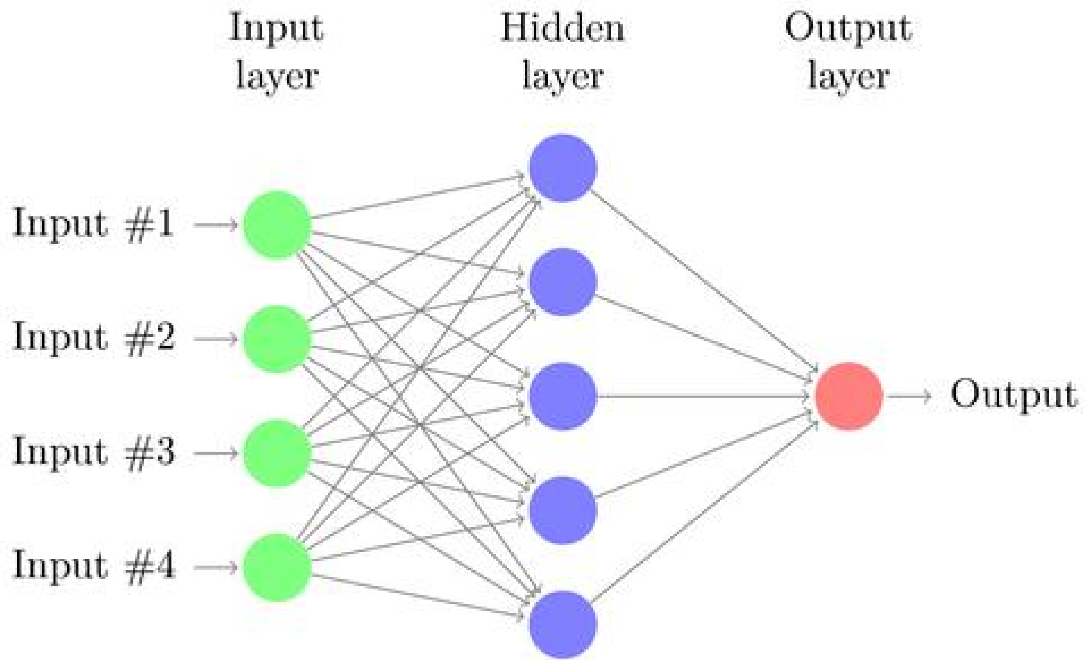


Figure 3.1: A simple artificial neural network

as deep neural networks (DNN) and convolutional neural networks (CNN)

Classical ANN - MNIST

A classical ANN consists of an input layer, a hidden layer and an output layer. Figure 3.1 shows the architecture of classic ANN. MNIST is a neural network application trained to recognize handwritten digits. The application is implemented in C++. MNIST uses a database of handwritten digits [26] from 0 to 9. The neural network has three layers (i.e input, hidden and output layers). Two weight matrices are generated as a result of training process. The first weight matrix is between the input layer and the hidden layer while the second weight matrix is between the hidden layer and the output layer. This weight matrices are then used in the testing(operation) phase. We train the neural network with 60,000 samples. We then test the ANN with 10,000 samples. With everything running without errors, we are able to attain a 94.40 percent accuracy in predicting the 10,000 images.

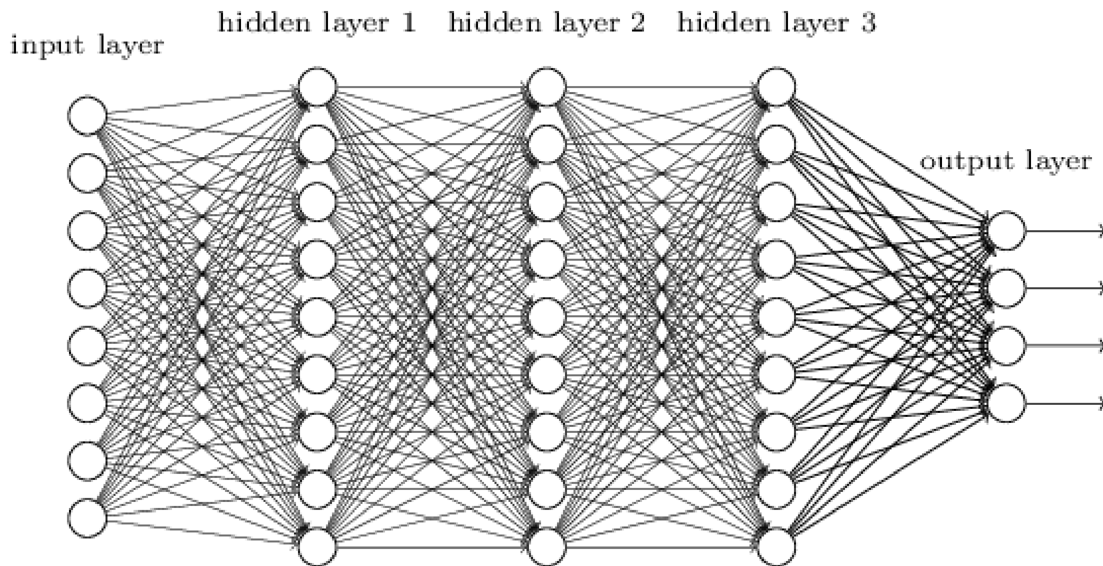


Figure 3.2: A deep neural network (DNN)

Deep NN - ImageNet

A Deep neural networks is essentially an artificial neural network with multiple hidden layers. Figure 3.2 shows the architecture of a deep neural network. In cases of extreme complexity, deep neural networks provide better results and are able to learn complex patterns as shown in the case of ImageNet. ImageNet is a data-set of hundreds of thousands of high resolution images belonging of multiple categories [36]

3.3.2 Fault Injection

This subsection shows the different methods used to inject faults. Experiments are conducted in a Dell Poweredge R720 machine with the following specifications: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz ,16 cores, 128 GB of DRAM memory, Linux 4.4.0-57 and GCC version 5.4.0.

Architectural Register Faults

When a transient fault occurs in a registers or memory unit , it changes the information stored in that register. These are called single event upsets (SEU).Multiple bit upsets (MBU), where several bits are affected can also occur. In order to asses such faults, we flip the bits of architectural registers as the application is running. We assess both SEU and MBU cases. We inject no more than 5 faults for each time an application runs and

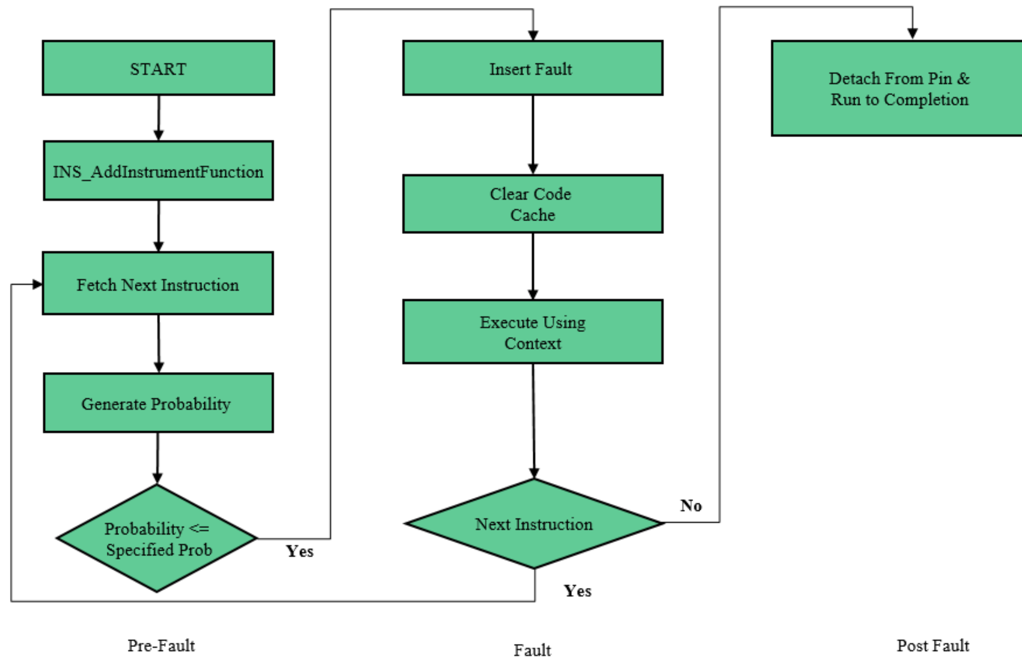


Figure 3.3: Fault Injection Process Using PIN

randomly distribute the faults across the entire time line of the application. The bit to be flipped is also randomly selected. We repeat each fault injection experiment 50 times per register for each sample which means in total 800,000 fault injection experiments are conducted.

We inject faults into architectural registers using PIN [28], a binary instrumentation tool provided by Intel.First, fault injection code is added before every instruction. This code is called analysis code. Since we don't inject a fault for every instruction, we randomly select an instruction to inject the fault. If an instruction is selected, this triggers fault injection and a fault is injected in one of the 16 general purpose registers. Table 3.1. shows the 16 general purpose registers and their functions. The fault injected is a bit flip in one of the bit of a general purpose registers. After injecting the fault, the next instruction is executed and the run continues.We assess both single event upsets (SEU) and Multiple bit upsets (MBU). Figure 3.3 summurises the entire fault injection process.

We inject no more than 5 faults for each time an application runs and randomly distribute the faults across the entire time line of the application. The bit to be flipped is also randomly selected. We repeat each fault injection experiment 50 times per register for each sample which means in total 800,000 fault injection experiments are conducted.

Table 3.1: General Purpose Registers and their Functions

Register	Function
RAX (Accumulator)	Used in arithmetic operations
RBX (Base)	Used as a pointer to data
RCX (Counter)	Used in state/rotate instructions and loops
RDX (Data)	Used in arithmetic operations and I/O operations
RBP (Stack Base pointer)	Used to point to the base of the stack.
RSP (Stack Pinter)	Pointer to the top of the stack.
RSI (Source)	Used as a pointer to a source in stream operations.
RDI (Destination)	Used as a pointer to a destination in stream operations
R8-R15	Used in 64 bit mode

Faults in Neural Network Layers and phases

An ANN has many components such as neurons, connectors, activation function and weights. A fault can affect any of these components. During training phase, a training model is generated which is then used in the testing (operation). This training model is a result of learning, the main feature of ANN, and involves continuously adjusting different parameters as the ANN is fed with new data. We simulate the occurrence of faults during the training of a neural network by directly changing some number of bits in the weight matrices. For the case of DNN, we inject permanent errors to Imagenet DNN. First, we select 10000 well-predicted images from the 50000 validation set. In each layer, we inject between 1 to 8 permanent errors. Once the layer was injected with the errors, the neural network predicts the output, and it is informed. We inject the errors in the bits 31 (sign bit), 30 (most significant bit of the exponent) and 29 (a second most significant bit of the exponent).

3.4 Evaluation

We divide our evaluation in three parts. First, the impact of injecting errors in different parts of a neural network, more precisely in different layers for both AN and DNN.Then faults in different processor registers Finally, errors in different inputs, which might be representative of corruption occurring in hard drives or in the transfer from storage to

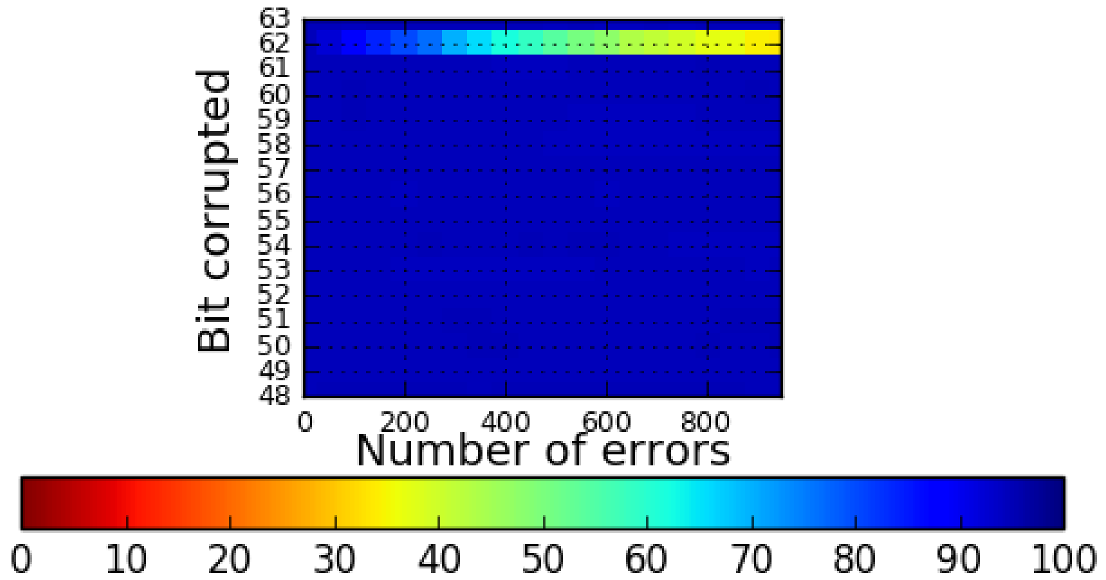


Figure 3.4: Injecting in the weight matrix between the input layer and the hidden layer.

computing nodes.

3.4.1 Vulnerability of Different Neural Network Layers and phases

In this experiment we construct a neural network with three layers, the input layer composed of 784 neurons, the hidden layer composed of 128 neurons and finally the output layer composed of 10 neurons. Then, we inject errors in the weight matrices between the different layers. For each layer, we inject errors in different bit positions, starting from bit position 48 up to bit position 63 which represents the sign in the IEEE floating point representation. Then, for each bit position, we increment the number of errors by 100 from 0 up to a $1,000$ errors.

Figure 3.4 shows the accuracy of the NNA when errors are injected into the the first weight matrix, i.e between the input layer and the hidden layer. The results show a significant accuracy drop as we inject errors in the strongest bit of the exponent (i.e., bit 62), and the accuracy decreases linearly with the number of errors, going all the way down to about 30% of accuracy. While this drop in accuracy is significant, it is interesting to notice that all the other bit position are pretty much insensitive to corruption, keeping for all of them, accuracy over 90%. This goes in pair with previous research showing that ANNs are very robust applications.

Figure 3.5 shows result when we inject errors in the second weight matrix (i.e.,

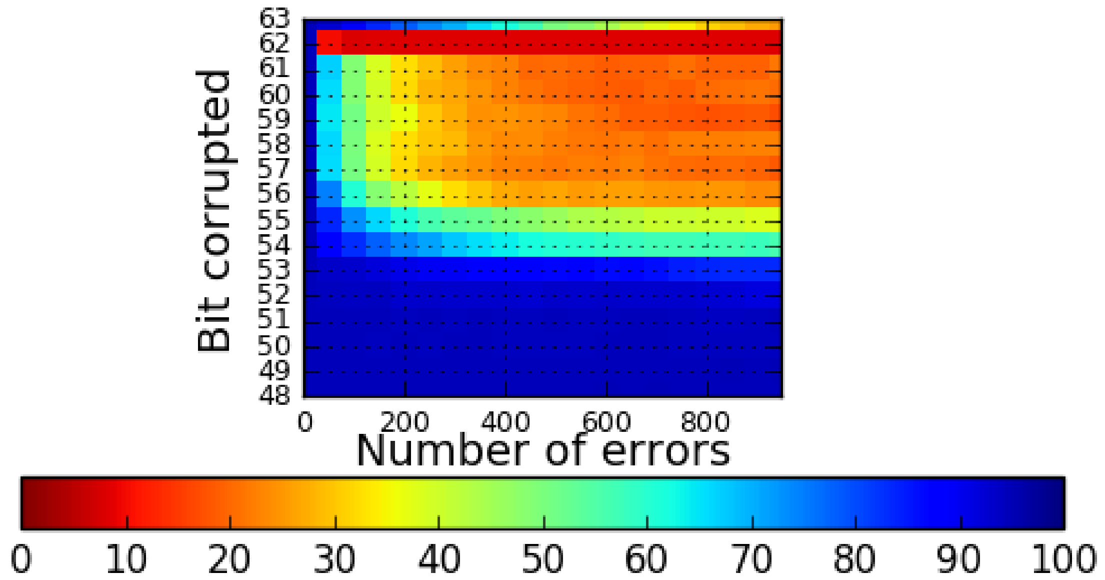


Figure 3.5: Injecting in the weight matrix between the hidden layer and the output layer.

from hidden to output), as opposed to the first one. The faults in this level have a much higher effect. As we can see, for a fixed bit position, the accuracy drops as we increase the number of corruption. But interestingly, the accuracy drops even faster when we change the bit position, for the same number of errors. In fact, for bit position 62 the accuracy drops dramatically to under 10%, showing that such corruption could make the ANN completely useless. We notice that for errors in lower bit positions (i.e., mantissa) the accuracy remains over 90%.

Finally Figure 3.6 shows how the errors affect the training phase. With no errors during the training phase, we attain an accuracy of over 94%. However as the number of errors continue to increase between 0 and 1000, a signifant drop in accuracy is observed. It is also interesting to observe that even a poorly trained model, ie a corrupted model, still has some degree of accuracy.

3.4.2 Vulnerability of Processor Registers

In this experiment, we inject faults in the processor registers during the execution of the ANN application. Note that these are considered faults (as opposed to errors) because not all of them propagate to affect the application. This is because the value on a register can be written after a corruption occurs, making it a bening fault. This masking effect is what we try to measure in this experiment. In addition, if the fault propagates to affect the

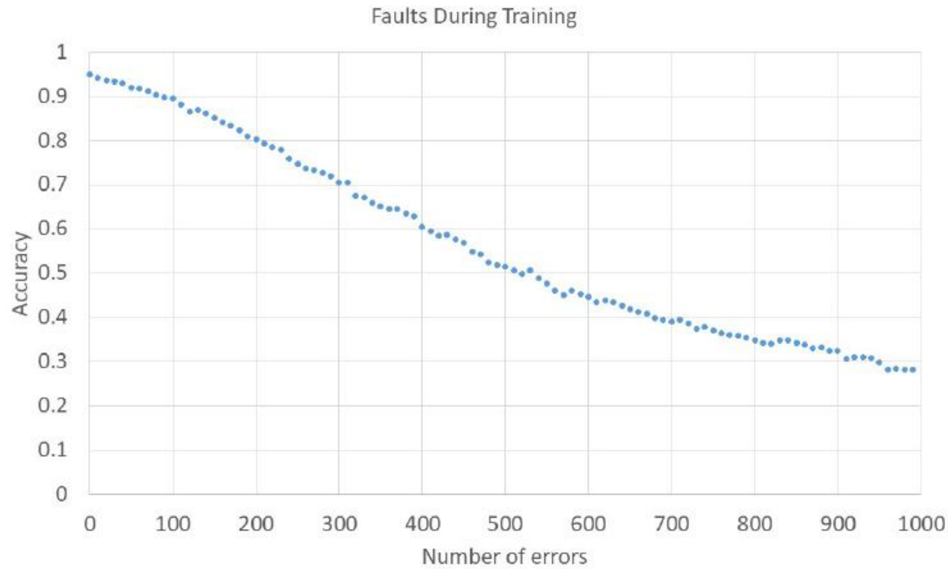


Figure 3.6: Faults during training

application, depending on how the application is affected, the fault could cause a crash, or a SDC. Thus, we divide the possible run outcomes in those three categories.

Figure 3.7 shows the results of injecting faults to different registers during the testing phase. There are three main observations that can be made from this figure. The number of crashes is high in RSP and RBP registers. These two registers, also called special purpose registers are the most critical registers because they are highly involved in function calls. RBP points to the base current stack while RSP points to the top of the stack. Our results show that the ANN failed nearly every time any of those registers were affected. Second, SDC is high on RDX register. This register is a data register thus depending on the data it is holding in a specific instruction, a fault on it can cause SDC or not. This is evident from the high number of false predictions seen when faults are injected into RDX. We note that when a fault occurring in any of these registers leads to SDC, such corruption could affect the weight values of the matrices, which in turn could have catastrophic effects (i.e, dramatically low accuracy) depending on where the SDC occurs. Finally , contrary to some previous work, these results indicate that ANN are quite vulnerable since we see that every fault occuring in registers may lead to SDC or crash.

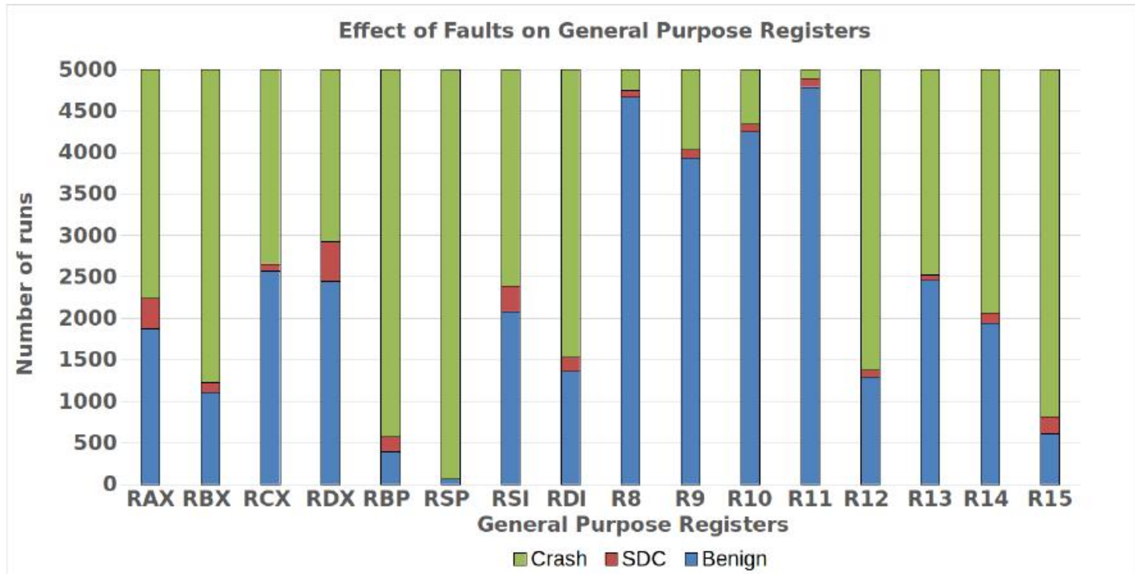


Figure 3.7: Effect of faults on general purpose registers.

3.4.3 Characterization of Errors for Different Input Datasets

Now, we want to measure the impact of corruptions occurring on different types of input datasets. More precisely, we want to measure whether some images can be more prone to prediction errors. We classify each of the samples according to the digits they represent (i.e digit 1 as label 1 , digit 2 as label 2). We then inject register faults targeting different labels. Figure 3.8 shows the effect of faults when predicting different labels. From these results, we notice that some images are more affected than others. For instance, the label 0 is almost not affected by SDC. One explanation for this could be the nature of the image itself and its *distance* to other characters. It is highly likely that a neural network will predict it correctly since it does not resemble any other image. Independently on how it is written, the label 0 is easily identifiable. However, label 4 is more affected by SDC. This seems to be because it is quite easy to confuse the number 4 with another number such as 9 when it is handwritten.

In the ANN that we are testing to predict a digit (e.g., the handwritten digit is predicted to be 9), an error score is generated. If the error score value is below a certain value, then the label is predicted correctly. However, if the error value is above that threshold, the prediction is incorrect. Figure 3.9 shows how the error score changes as faults are injected to different registers. Injecting faults on processors registers has a significant effect on the error score for all registers. Furthermore, the error value changes with respect

Figure 3.8: Effect of faults on different images

to the register. Some registers such as RDX, RBP and R15 cause this value to increase significantly.

3.4.4 Vulnerability of ImageNet DNN

Permanent faults are injected to bits 29, 30 and 31. We observe that the bit that significantly affects the results is bit 30.Figure 3.10 shows the results of bit 30 Important differences can be observed in prediction accuracy with a drop in accuracy as the number of faults increases to 8. We also observe that the convolutional layers are more affected than the fully connected.

3.5 Discussion and Suggestions

Our results show that some registers play an important role in ANNs while others have almost no impact on their accuracy. We also show that different parts of ANN application are affected differently by errors. As such, ANNs applications are good candidates for application-specific reliability designs. One such design is applying error correction differently to different registers. Important and vulnerable registers such as RSP and RSP require ECC while registers such as R8, R9, and R12 that are rarely used might not necessarily require ECC. For instance, one can implement two groups of registers with different levels of protection. Such difference can be made known to compiler and appli-

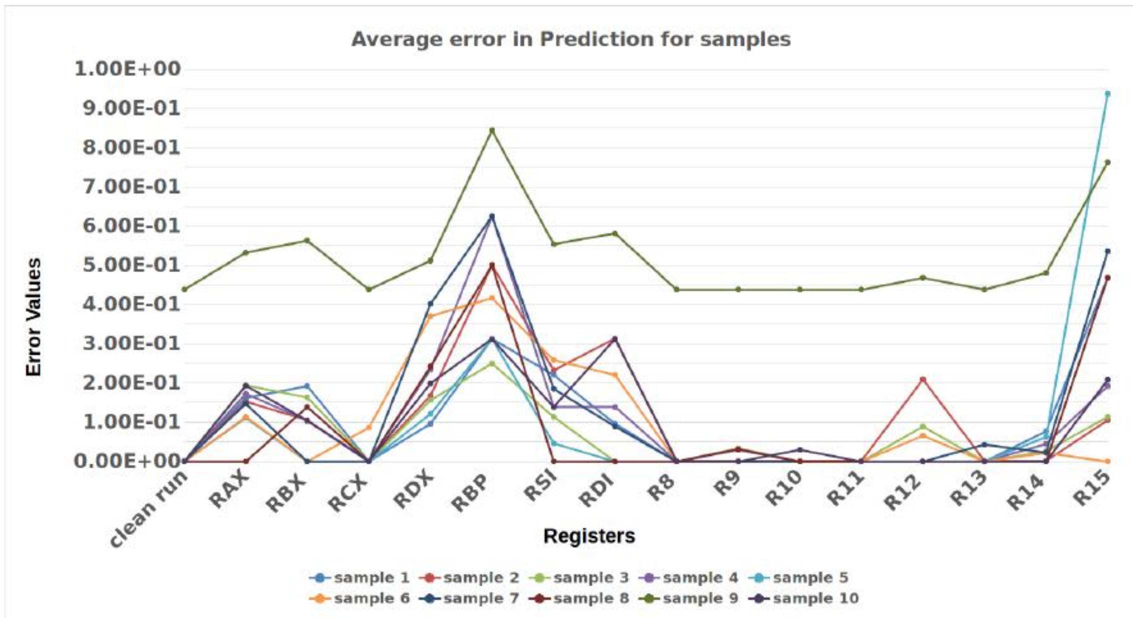


Figure 3.9: Average error in prediction.

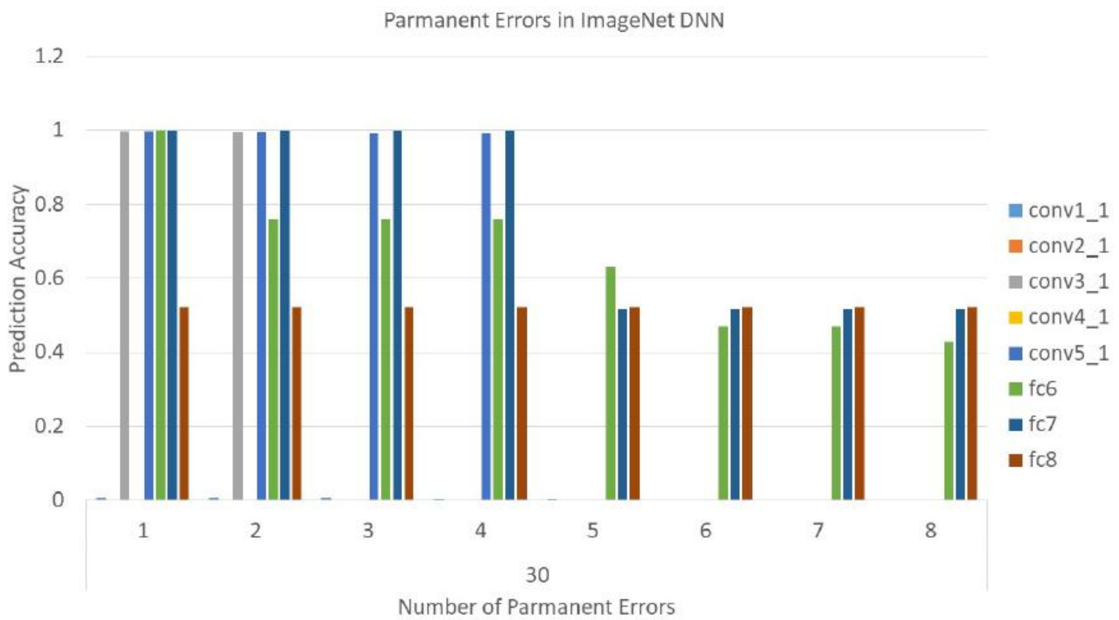


Figure 3.10: Average error in prediction.

cation designers to make proper use of those registers depending on the vulnerability of the data.

Our results also show that at an application level, the training phase is the most important phase. This is because the results of training phase are fed to the testing phase. Data corruption during the training phase means that the NN will be poorly trained which will, in turn, affect the testing phase. Therefore, extra protection and/or verification mechanisms must be enforced to the training phase especially for the data between the hidden layer and output layer.

Finally our results show that different labels are affected differently by faults. We suggest that for specific labels which are more vulnerable to faults (i.e label 4) execution can be done twice and results can be compared for more reliable output.

3.6 Summary

Its clear that ANN will continue to gain popularity as a computational model especially in artificial intelligence. ANN applications will be executed in HPC systems and as we approach exascale systems, the reliability of such application pauses serious questions due to the high number of faults expected in exascale systems. In this paper we have shown that the widely held belief that ANN are fault tolerant is not exactly true. We have shown that different parts of a NN behave differently in the presense of faults. We have also shown the effects of silent data corruption on an ANN and the effects of hardware faults. Finally we have pointed out some possible designs to cut down reliability costs of ANN applications. This could significantly cut down the reliabilty cost.

Chapter 4

Hardware Transactional Memory for Tolerating Transient Hardware Errors

4.1 Introduction

Occurrence of faults is inevitable with the current trend in processors. Attaining a 100% reliable system would come at great costs, both in terms of price due to extreme redundancy, and power. Designers have to compromise and make reliability variations based on different applications. For instance, for safety critical applications, reliability has to be a top priority but for other applications such as desktop computers, existing fault tolerance mechanisms have proven enough. There is still need, however, for improved reliability and cheaper costs and researchers have proposed numerous tools and methods to improve reliability. One such method is leveraging transactional memory for reliability.

Transactional memory (TM) [18] is an attempt by researchers and engineers to simplify parallel programming by executing transactions atomically and in isolation. Transactinal memory attempts to solve the challenge of data synchronization. Traditionallly, parallel programmers have to address this problem using lock/unlock methods where a thread acquires a lock to the data its currently working on and only releases it when done. This means that other threads cannot access that data which leads to long waits and performance depredations. A worst case scenario can occur when a thread experiences a deadlock [9], where one thread locks some data indefinitely. TM solves this problem by executing each thread as a transaction, checking at the end of the transaction for mismatches and then committing or aborting the transaction. Even though it is not a total replacement of lock/unlock methods, TM promises to make parallel programming easier by taking away the burden of data synchronization.

TM can be implemented either in software as Software Transactional Memory (STM) or hardware as Hardware Transactional Memory (HTM). A combination of both has also been proposed [8]. Wheres STM is flexible and easy to implement, it is slow.

HTM on the other hand offers better performance but it is hard to implement due to the fact that its hardware based.

For reliability engineers however, TM offers an opportunity to detect and recover from transient hardware faults without the need for extra hardware.This is due to the fact that transactional memory executes an operation as a single atomic operation, independent of other operations. If we can duplicate the critical parts of an application, then we can detect differences in write sets of each thread, or lack thereof, at the end of each transaction. A difference in the write sets could mean a possible fault in one of the threads.

The idea of utilizing HTM for reliability first proposed by Yalcin et al. as FaulTM [45]. The idea argues that if we can duplicate the critical parts of an application, then execute them as transactions concurrently and in isolation, we can detect differences in write sets of each thread at the end of each transaction. A difference in the write sets could signal a possible fault in one of the threads. Although FaulTM presents low performance overhead compared to other HW-based reliability schemes, it still requires hardware extensions on top of an HTM implementation. Moreover, during the proposal of FaulTM, there was lack of HTM support by hardware was a major challenge to the scheme and therefore the authors proposed hardware changes to the then existing processors to support HTM. Even then, their HTM based scheme showed relatively low performance overhead and a wide error coverage.

Recently, significant gains have been made in hardware design and now several hardware vendors such as Intel [19], IBM [25] and AMD [] provide off the shelf hardware that implements HTM for concurrency control. As multi-core systems become mainstream, concurrency control will continue to be an important factor. TM will therefore continue to play an important role and more hardware vendors are going to produce processors with HTM support. HTM, though build primarily for concurrency control, offers benefits and features that could be leveraged to build a fast and easy to implement reliability scheme.

Roll-back only transactions(ROT), available in some HTM implementations, is another feature of HTM that could be leveraged. ROTs allow for speculative execution of instructions at minimal cost [25] and is built on the hardware. Unlike other transactions ROT tracks only memory writes of a transaction and does not undergo implicit conflict detection.This is particularly important because of two reasons. First, in most applications,

writes are fewer than reads and there is a limited buffer size for transactions. Tracking only writes allows us to have longer transactions. Second, since ROT writes are not visible to other instructions till commit, we can conceal the duplicate thread as a ROT, hiding all its memory writes till commit. This way, we protect our transaction from implicit conflict detection and implement our own conflict detection scheme to detect any data inconsistencies between the main transaction and ROT.

HTM is particularly attractive for reliability because of following reasons as shown by [45]

- TM has a well defined conflict detection mechanism.
- TM has the ability to abort a transaction in case of a conflict
- TM executes transactions atomically and in isolation hence isolating failures
- Error detection only during commit hence reducing performance overhead.

However, HTM comes with some challenges such as limited cache, interrupts that may cause aborts, lack of a guarantee that transactions will start or finish and forbidden instructions within a transaction.

In this chapter, our main goal is to show how HTM and ROT can be used to build robust and fault resilient applications with minimal performance overhead and without requiring any hardware change. To this end, we propose and application development methodology that leverages existing and easily available off the shelf hardware that supports HTM. As a proof of concept, we use IBM POWER8 processor that has both HTM and ROT to build and test our design. We also introduce novel techniques to overcome the challenges posed by HTM.

The remainder of this chapter is organized as follows; section 2 provides the previous work, section 3 covers the different HTM implementations that exist and finally section 4 provides our design for increasing reliability using HTM.

4.2 Previous Work

A reliable system should include 1) a mechanism to discover that an error has occurred, called error detection mechanism, and 2) a mechanism of restoring the system's integrity after the occurrence of the error, called error recovery mechanism.

Checkpoint/Recovery is the most well-known error recovery technique which stores an error-free state of the system (checkpointing) and reverts the system state upon error detection (recovery). TM provides mechanisms to abort transactions in case of a conflict, thus they discard or undo all the tentative memory updates and restart the execution from the beginning of the transaction. Thus, a transaction's start can be viewed as a locally checkpointed stable state which can be used for error recovery. Due to this benefit, TM systems are proposed to be used for reliability.

Using transactions to recover from application crashes have been first proposed in SymptomTM [46] by Yalcin et al. and disclosed in a patent filed by IBM [4]. In those schemes, unless any symptom of error (e.g. fatal trap) is raised in the reliability-purposed transaction, the write-sets of the transaction is committed to the shared memory. Otherwise, the transaction aborts and restarts the execution from the beginning of it. Obviously this scheme has limited error coverage since it cannot detect silent data corruptions. Due to this limitation, FaulTM [45] proposed using redundant transactions in order to increase error coverage and provide high reliability for mission critical systems. Later on, it is also presented how to extend FaulTM mechanism for parallel applications including transactional memory applications by Yalcin et. al [47]. FaulTM leverages a HTM that features lazy conflict detection and lazy data versioning which provides the benefit of reducing the comparison overhead of error detection (ie. comparing two transaction to check whether they produce the same result or not). However, FaulTM stalls the execution at the commit stage of the transactions. Thus, Sanchez et. al proposed LBRA [38] which utilizes HTM with eager data versioning and presents an alternative design option with higher comparison but lower synchronization overhead. Ferreira et al [11] and Gurumurthi et al. [15] proposes the usage of TM for the reliability of GPU architectures.

In all these mentioned studies, several hardware changes are required. Therefore, their cost of implementation is high and they cannot be used unless extensions are included in the hardware. In order to reduce the hardware cost, Haas et. al presented how to use the existing transactional memory hardware in Intel TSX for reliability purpose [16]. However, since rollback-only transactions are not supported by Intel, it presents high performance overhead. Another reliability scheme using existing TM hardware is proposed by Shalev et al. [39]. In that study, only critical kernel code is surrounded by reliability-purposed transactions and the failures in the kernel is recovered by transactional aborts.

To our knowledge, this is the first study implementing reliability on a real Transactional Memory hardware by using rollback-only transactions and it is the first study using IBM Power8 machine as HTM implementation.

4.3 Hardware Transactional Memory Implementations

In this section we discus existing hardware memory transactions from major vendors and how they are implemented. We further discus key features in each of the implementations.

4.3.1 IBM POWER8

Even though TM has been around for quite sometime, POWER8 is the first implementation of transactional memory that is directly supported by the power ISA (Instruction set architecture) [25] . Power ISA is an instruction set architecture designed to expose and exploit parallelism in a wide range of applications, from embedded computing to high-end scientific computing to traditional transaction processing [21] . Power ISA has extensions that enable a programmer to access transactional memory. According to Le et al, " The transactional memory extensions of the Power ISA architecture consist of a set of instructions for implementing a strongly isolated, best effort hardware transactional memory and appropriate hardware state information to control the execution of transactions. " [25]. TM instructions such as begin, suspend, resume, commit or abort enable the programmer to deal with transactions. Table 4.1 provides a list of such functions.

long __TM_simple_begin (void)
long __TM_begin (void* const TM_buff)
long __TM_end (void)
void __TM_abort (void)
void __TM_named_abort (unsigned char const code)
void __TM_resume (void)
void __TM_suspend (void)

Table 4.1: PowerPC HTM High Level Inline Functions

Besides HTM support, IBM Power8 and PowerISA offers unique features that we leverage for our design.

Rollback-only Transactions

One key and unique feature of IBM Power8 is Rollback-only Transactions (ROT). ROT allows the creation of transactions whose purpose is support single-thread speculation of instructions and allow those instructions to be rolled back under software control [25] . A roll back transaction tracks only memory writes and does not undergo implicit conflict detection, making it ideal for our reliability design.

Suspend and Resume Instructions

PowerISA provides us with instructions to suspend and resume transactions. This is particularly important because it gives us full control over the transactions. During a transaction execution we can suspend a transaction and execute some instructions that would otherwise be forbidden within a transactions. Rather than abort the entire transaction, suspending a transactions allows the transaction to resume and continue after the execution of the forbidden instruction.

Figure 4.1 shows the architecture of POWEER8 processor. Key features include 12 cores with 8 way simultaneous multithreading (SMT) and upto 230GB/s memory bandwidth. The Power8 Processor also features a 16 Execution Pipeline followed with 64K data cache per-core and 32K instruction cache.

4.3.2 Intel TSX

IBM is not the only vendor to implement HTM in its processor, Intel Haswell TSX [19] is another commercial processor that implements HTM. Intel implements Reduced Transactional Memory (RTM) which provides an interface to start, end and abort transactions. Atomic regions are defined by surrounding them with xbegin() and xend() functions. All writes are kept internally within a transaction until the transaction commits. Similar to POWER8, a conflict occurs if two threads attempt to write on the same data, in which case the transaction is aborted.

It is important to note that both Intel TSX and POWER8 are best effort HTM implementations and non guarantees that a transaction will successfully start and complete. When a transaction fails, it restarts, however there is no guarantee that it will be successful the next time, as such HTM does not guarantee forward progress and a mechanism for

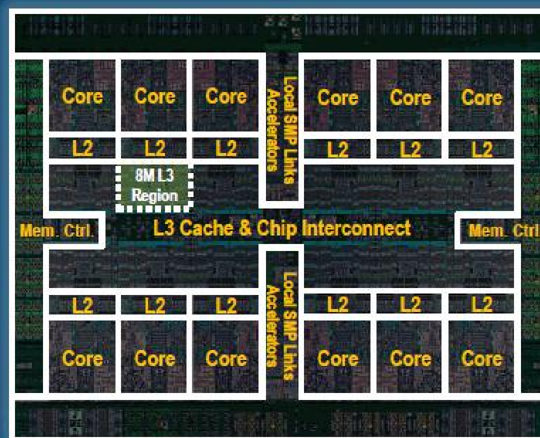


Figure 4.1: POWER8 Processor

this is therefore required. One such mechanism could be a manual implementation by the programmer where the application resorts to lock/unlock method after a specific number of transaction failures.

4.4 Proposal

We propose a new HTM based reliability scheme. Our proposal is motivated by the fact that HTM is now commercially available in processors and this trend is likely to continue. As such, for applications that need enhanced reliability, a few modifications in the application could drastically increase its error detection and correction ability. Furthermore, it would do so at little or no cost in terms of hardware and it would take very little effort to a programmer familiar with parallel programming to turn an unreliable application to a reliable one. With modifications, HTM could further be improved to provide reliability for safety critical applications as part of embeded systems such as the one proposed in [31] .

In this section we provide a method to design applications that are robust and fault resilient. We argue that a serial or parallel application can be made resilient to faults that occur in commodity hardware. To do this, we leverage HTM that is now available in commercial hardware. We also leverage Rollback only transactions, a unique feature of IBM Power8. In this section we provide details of the design methodology. First by discussing different implementations of HTM that are commercially available then discussing how to build robust applications using such implementations.

4.4.1 Preliminary Work

The preliminary work involved getting familiar with parallel programming and parallel computing. We use STAMP [32], a benchmark that cosists of different parralle applications implemented with TM . The results are shown in Figure 4.2 show the speed up of different STAMP applications using software transactional memory (STM). For this we used tinySTM [10], a lightweight word-based STM implementation.

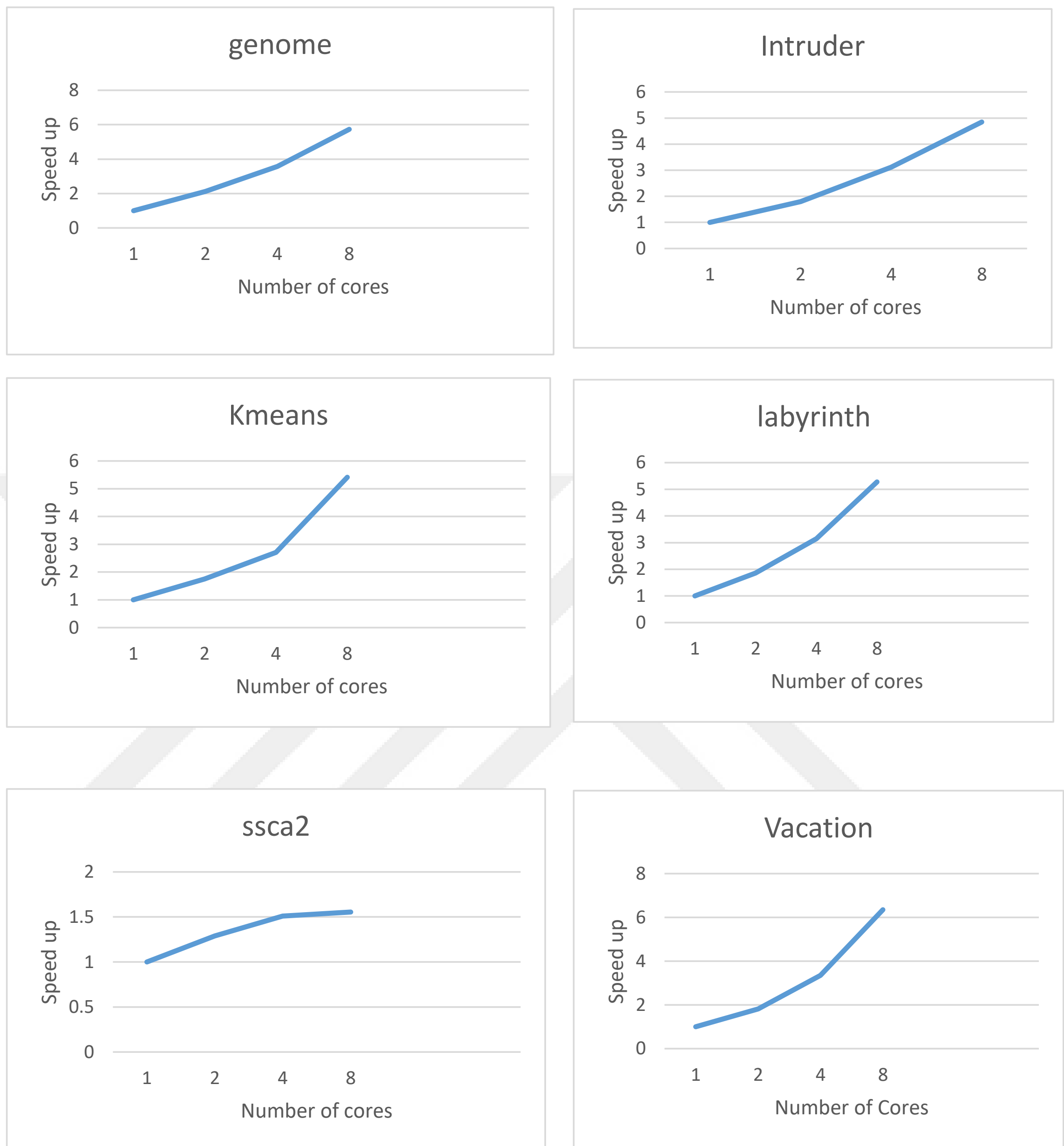


Figure 4.2: Performance of STAMP applications implemented with TinySTM

4.4.2 Design Methodology

The underlying idea in our design methodology is to execute critical parts of an application as transactions in parallel then compare the write sets of the two transactions before commit. In so doing, we achieve two main objectives 1. We can detect irregularities and any difference in write sets in case a fault occurs and 2. Using HTM mechanisms such as check-pointing and roll back, we can achieve error correction by aborting the transaction and rolling back the transactions to a previous state. Hence our design is a complete reliability scheme offering both error detection and error correction.

1. Identifying critical sections
2. Creating threads
3. Executing both threads at the same time as transactions
4. Comparing the results of the threads (transactions)

The details of each section are discussed below.

Identifying critical sections

As stated earlier in the motivation, not all applications require the same level of reliability measures. Furthermore, in the same application some parts are more vulnerable than others. This means that when a fault occurs, its likely to affect some parts of the application more than others. For instance, during multiplication, a fault in a critical register (such as the one holding the multiplicand or multiplier) could lead to a faulty result. In the Artificial Neural Networks experiments performed in chapter 3, we identified that the second layer calculations for example were more critical than the first layer. A programmer could therefore identify this critical parts and mark them

Creating threads

Once critical sections are identified, the next step is to parallelize them. To achieve this, we create two threads and call them *reliableThread* and *originalThread* .Thread based parallel programming paradigms such as pthreads and c++ threads involve passing a function and its parameters to a thread. We pass the function and its parameters containing the

critical sections to each of this threads. The code listing 4.1 below shows a main function showing how the parameters are passed to the functions.

```
#include <htmintrin.h>
#include <iostream>
#include <thread>
using namespace std;
int aborts;
int main()
{
int counter = 0;
int &ref = counter;
std::thread t1(increment, std::ref(counter));
std::thread t2(reliableIncrement, std::ref(ref));
t1.join();
t2.join();
cout << "After all transactions have ended" << endl;
cout << counter << endl;
cout << aborts << endl;
return 0;
}
```

Listing 4.1: Creating threads example

Executing the threads as transactions

The next step is place the critical sections into transactions. This is done by including the begin transaction and end transactions instructions provided by the ISA. We include a HTM pre-processor that enables the use of built in functions to manage transactions. GCC provides two interfaces to access HTM in POWER8 processors.

- PowerPC HTM Low Level Built-in Functions
- PowerPC HTM High Level Inline Functions

In this work, we use PowerPC HTM High Level Inline Functions by including the pre-processor *"-mhtm or -mcpu=CPU "* where CPU is "power8" in our case and include the

header *htmxlintrin.h* . A simple example of counter increment is shown listing 4.2

```
#include <htmintrin.h>
while (1)
   {
     if (__builtin_tbegin(0))
       {
          for (int i = 0; i < 1024 * 1024; i++)
          {
             counter++;
          }
          __builtin_tend(0);
          break;
       }
   }
```

Listing 4.2: Counter Example

Comparing the results of the transactions

When the two transactions complete, the final step is to compare the results of the two transactions before committing. This is where we utilize HMT's unique feature of conflict detection. In the absence of errors, these two transactions should have the exact same write sets since they execute the same instructions. Since HTM is primarily built for concurrency, a conflict is detected here as the instructions accessed the same data. HTM will therefore abort both transactions. This is a major challenge of our design and a clever maneuver is required to overcome it. One such method would be as shown in our previous example, passing data to the functions by reference and value respectively. As such the two transactions will appear to be working on two different data sets, hence avoiding aborts. Another method would be to use Rollback-only transactions(ROT). These transactions track only write sets and don't undergo conflict detection. The reliable thread could be implemented as a ROT.

Before exiting the transaction state, we compare the write sets of the two threads. If the write sets are the same, then it means that no fault occurred. We abort the reliability thread and commit the main thread. If the write sets are different, then a bit corruption or any other error might have occurred during one the threads execution, we abort both

transactions and restart. A consistent conflict in the two write sets could mean a permanent fault occurred while a simple bit corruption could be resolved by restarting.

Comparing the results is not a straight forward as it involves two threads communicating. By principal of atomicity, this is not acceptable and would cause the transaction to abort. To overcome this, we utilize Power8's unique suspend and resume transaction features. We suspend the transaction, make the comparison and then resume.

poletnjeri@gmail.com

4.4.3 Challenges of the Design

This designs raises several challenges that must be addressed. These are mainly due to the fact that HTM is built for concurrency and not reliability. However, with a few modifications, this challenges can be overcome.

Conflict Detection

As stated earlier, HTM is built for conflict detection. As such when there is a conflict, it tends to abort both transactions. Because both threads are executing the same set instructions, HTM will detect a conflict and in most cases abort the transactions. To overcome, we utilize the special Roll-back Only Transactions (ROT). These track only the write sets and do not cause aborts.

HTM is best effort

Power8 implementation of HTM is a best effort implementation and does not guarantee that transactions will start or transactions will successfully commit. For this, we retry the transactions a defined number of times (maximum retries). However if after maximum retries the transaction still fails, we have to ensure continuity of the program by running an unreliable version(without transaction protection). From our experiments however, the possibility of this is very low.

Interrupts

User interrupts such as writing on the screen or keyboard input cannot be rolled back. As such, such instructios are not allowed within a transaction and they are enclosed in a transaction, they cause the transaction abort. System interrupts also behave in a similar

manner but Power ISA provides a mechanism that can handle system interupts [25]. To handle user interrupts, a programmer has to avoid using them within a transaction. But an ideal program has several interrupts within its execution and therefore avoiding interrupts is practically impossible.

An alternative therefore is to use suspend and resume instructions. Whenever we want to use an instruction such as print, we suspend the transaction, execute the interrupt then resume the transaction. This way the transaction wont abort because whenever the compiler encounters a suspend command, it stops atomicity,

4.5 Proof of Concept

In this section we present a proof of concept and evaluate our design methodology. The main purpose of creating a proof of concept is to see if indeed our design methodology works and if it can be used to implement robust and reliable application from unreliable ones. For a proof of concept, we change a well known sorting algorithm application, bubble sort, into a reliable one using the design methodology presented earlier. The sorting function takes an array of numbers and returns an array of the same numbers sorted in ascending order. We identify the critical sections of this application as the sorting function and redesign the application with this in mind. The application is written in C++ and compiled with G++. We executed the application on an IBM Power8 machine with the following specifications.

- 10 cores running at 3.425GHz
- 80 hardware threads
- 128B cache lines
- 80MB 8-way shared L3 cache
- 64KB per-core 8-way L1 caches

During numerous tests, we were successfully able to detect any conflicts that we introduced into the transactions. Further we were successfully able to compare the results of two threads. The code implementation for the proof of concept is provided in Appendix A.

4.6 Summary

In this chapter, we have provided a design methodology to design robust and resilient applications using off the shelf hardware transactional memory implementations. We have shown that this scheme is not only efficient but its also easier to implement with minimum design overhead. We have further implemented a proof of concept using the proposed design methodology. Our proof of concept shows that its possible to leverage hardware transactional memory and build robust and fault tolerant applications.

Chapter 5

Conclusion

The spectrum of applications is going to keep growing as more and more applications are created with each single day. This applications will continue to have different levels of reliability requirements. Furthermore, with the ever advancing programming and software engineering methodologies, the number of lines of code in an application continues to rise. If the current trend of one fits it all reliability continues, its going to be extremely costly , both to the programmer and economically, to provide homogeneous reliability with applications that have such huge code bases. As the number of both transient and permanent faults is expected to rise, there is need for a cheaper and convenient method to provide reliability. Apart from showing that there is need for application specific reliability measures, this thesis has two major contributions.

First , in chapter 3 we did a detailed resilience characterization of Artificial Neural Networks applications. Our results show that this application behave differently in the presence of different errors. Our results also showed that different parts of the application are not affected the same way by errors. Furthermore, some register faults almost have no effect on an ANN application while other register faults completely break down the ANN.

Finally in chapter 4, we presented a novel idea of increasing reliability by using hardware transacational memory that already exists in some commercial hardware. Even though HTM is primarily designed for concurrency, we have shown that with a few modifications, we could utilize it to create reliable applications.

5.1 Future Work

As a future work, we intend to run further resilience characterization for different applications in the AI and ML domain. We believe that this application will be more prone to transient errors as they are mostly executed in high performance computing centers and large data centers where this faults are likely to occur.

We also intend to further develop our HTM solution so that we move most of the design to the compiler. As such, any programmer will have an easy time creating reliable applications since it will involve just calling a few commands and a programmer doesn't have to understand the underlying mechanism of HTM. As such they can focus on making reliable applications.

Bibliography

[1] 2017.

[2] Bilge Acun, Akhil Langer, Esteban Meneses, Harshitha Menon, Osman Sarood, Ehsan Totoni, and Laxmikant V Kalé. Power, reliability, and performance: One system to rule them all. *Computer*, 49(10):30–37, 2016.

[3] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2015.

[4] M.A. Blumrich, D. Chen, A. Gara, M.E. Giampapa, P. Heidelberger, M. Ohmacht, B. Steinmacher-Burow, and K. Sugavanam. Local rollback for fault-tolerance in parallel computing systems, January 24 2012. US Patent 8,103,910.

[5] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, nov 2009.

[6] C. T. Chiu, K. Mehrotra, C. K. Mohan, and S. Ranka. Robustness of feedforward neural networks. In *IEEE International Conference on Neural Networks*, pages 783–788 vol.2, 1993.

[7] Cristian Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23(4):14–19, July 2003.

[8] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*, volume 41, pages 336–346. ACM, 2006.

[9] A. O. Abd El-Gwad, A. I. Saleh, and M. M. Abd-ElRazik. A novel scheduling strategy for an efficient deadlock detection. In *2009 International Conference on Computer Engineering Systems*, pages 579–583, Dec 2009.

[10] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[11] Ronaldo R. Ferreira, Jean da Rolt, Gabriel L. Nazar, Álvaro F. Moreira, and Luigi Carro. Adaptive low-power architecture for high-performance and reliable embedded computing. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 538–549, 2014.

[12] Leonardo Arturo Bautista Gomez and Franck Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 595–602. IEEE, 2015.

[13] J. Gracia-Morn, J. C. Baraza-Calvo, D. Gil-Toms, L. J. Saiz-Adalid, and P. J. Gil-Vicente. Effects of intermittent faults on the reliability of a reduced instruction set computing (risc) microprocessor. *IEEE Transactions on Reliability*, 63(1):144–153, March 2014.

[14] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 459–468, June 2003.

[15] S. Gurumurthi and V. Sridharan. Using redundant transactions to verify the correctness of program code execution, September 20 2016. US Patent 9,448,933.

[16] Florian Haas, Sebastian Weis, Stefan Metzlaff, and Theo Ungerer. Exploiting intel TSX for fault-tolerant execution in safety-critical systems. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2014, Amsterdam, The Netherlands, October 1-3, 2014*, pages 197–202, 2014.

[17] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: Evaluating resilience of gpu applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2015.

[18] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[20] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[21] Tejas S. Karkhanis and José E. Moreira. *IBM Power Architecture*, pages 900–907. Springer US, Boston, MA, 2011.

[22] F. Kausar and P. Aishwarya. Artificial neural network: Framework for fault tolerance and future. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 648–651, March 2016.

[23] Anton Kulakov, Mark Zwolinski, and Jeffrey S. Reeve. Fault tolerance in distributed neural computing. *CoRR*, abs/1509.09199, 2015.

[24] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 25. ACM, 2016.

[25] Hung Q Le, GL Guthrie, DE Williams, Maged M Michael, BG Frey, William J Starke, Cathy May, Rei Odaira, and Takuya Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[26] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits.

[27] Ikhwan Lee, Mehmet Basoglu, Michael Sullivan, D Hyun Yoon, Larry Kaplan, and Mattan Erez. Survey of error and fault detection mechanisms. *University of Texas at Austin, Tech. Rep*, 11:12, 2011.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[29] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, June 2014.

[30] Yashwant K Malaiya and STEPHEN YH Su. A survey of methods for intermittent fault analysis.

[31] Stefan Metzlaff, Sebastian Weis, and Theo Ungerer. Towards transactional memory for safety-critical embedded systems. 2013.

[32] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[33] F. Morgado Dias and A. Antunes. Fault tolerance of artificial neural networks: an open discussion for a global model. *International Journal of Circuits, Systems and Signal Processing*, 4:9–16, 2010.

[34] Vincenzo Piuri. Analysis of fault tolerance in artificial neural networks. *J. Parallel Distrib. Comput.*, 61(1):18–48, January 2001.

[35] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Characterizing the impact of intermittent hardware faults on programs. *IEEE Transactions on Reliability*, 64(1):297–310, March 2015.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[37] Anjali Russell. Inside the self-driving tesla fatal accident, 2017.

[38] Daniel Sánchez, Juan M. Cebrián, José M. García, and Juan L. Aragón. Soft-error mitigation by means of decoupled transactional memory threads. *Distributed Computing*, 28(2):75–90, April 2015.

[39] Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, and Yaron Weinsberg. Csr: Core surprise removal in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 50(2):773–787, 2016.

[40] Elko B. Tchernev, Rory G. Mulvaney, and Dhananjay S. Phatak. Investigating the fault tolerance of neural networks. *Neural Comput.*, 17(7):1646–1664, July 2005.

[41] J. Wei, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Comparing the effects of intermittent and transient hardware faults on programs. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 53–58, June 2011.

[42] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero. Fimsim: A fault injection infrastructure for microarchitectural simulators. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 431–432, Oct 2011.

[43] Gulay Yalcin and Osman Unsal. Transactional memory for reliability. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 268–282. Springer, 2015.

[44] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, and Mateo Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In Wei Liu, Scott Mahlke, and Tin fook Ngai, editors, *Pespma 2010 - Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Saint Malo, France, June 2010.

[45] Gulay Yalcin, Osman S. Unsal, and Adrián Cristal. Faultm: error detection and recovery using hardware transactional memory. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 220–225, 2013.

[46] Gulay Yalcin, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. Symptomtm: Symptom-based error detection and recovery using hardware transactional memory. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 199–200, 2011.

[47] Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 4:1–4:9, 2013.

Appendix A

Example

```
#include <htmintrin.h>
#include <iostream>
#include <thread>

extern __inline long
    __attribute__((__gnu_inline__, __always_inline__, __artificial__))
    __TM_begin_rot(void *const TM_buff)
{
    *_TEXASRL_PTR(TM_buff) = 0;
    if (__builtin_expect(__builtin_tbegin(1), 1))
    {
   return _HTM_TBEGIN_STARTED;
    }
#ifdef __powerpc64__
    *_TEXASR_PTR(TM_buff) = __builtin_get_texasr();
#else
    *_TEXASRU_PTR(TM_buff) = __builtin_get_texasru();
    *_TEXASRL_PTR(TM_buff) = __builtin_get_texasr();
#endif
    *_TFIAR_PTR(TM_buff) = __builtin_get_tfiar();
    return 0;
}

using namespace std;

int aborts;
void reliableIncrement(int &reliableCounter)
{
```

```
    while (1)
    {
   aborts++;
   if (__builtin_tbegin(0))
   {
       for (int i = 0; i < 10000000; i++)
       {
      reliableCounter++;
       }
       __builtin_tend(0);
       break;
   }
    }

}

void increment(int &counter)
{
    while (1)
    {
   aborts++;
   if (__builtin_tbegin(0))
   {
       for (int i = 0; i < 10000000; i++)
       {
      counter++;
       }
       __builtin_tend(0);
       break;
   }
    }

}
```

```
int main()
{
    int counter = 0;
    int &ref = counter;
    std::thread t1(increment, std::ref(counter));
    std::thread t2(reliableIncrement, std::ref(ref));
    t1.join();
    t2.join();
    cout << "After all transactions have ended" << endl;
    cout << counter << endl;
    cout << aborts << endl;
    return 0;
}
```

Appendix B

Bubble Sort Example

```
#include <htmintrin.h>
#include <htmxlintrin.h>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <thread>

using namespace std;
TM_buff_type TM_buff;
extern __inline long
__attribute__ ((__gnu_inline__, __always_inline__, __artificial__))
__TM_begin_rot (void* const TM_buff)
{
  *_TEXASRL_PTR (TM_buff) = 0;
  if (__builtin_expect (__builtin_tbegin (1), 1)){
    return _HTM_TBEGIN_STARTED;
}
#ifdef __powerpc64__
*_TEXASR_PTR (TM_buff) = __builtin_get_texasr ();
#else
*_TEXASRU_PTR (TM_buff) = __builtin_get_texasru ();
*_TEXASRL_PTR (TM_buff) = __builtin_get_texasr ();
#endif
*_TFIAR_PTR (TM_buff) = __builtin_get_tfiar ();
return 0;
}
```

```
void reliableBubbleSort(int a[], int n)
{
 while (1){
    cout << "Trying to ROT " << endl;
    if (__TM_begin_rot (TM_buff) == _HTM_TBEGIN_STARTED){


        for (int i = 1; i < n; ++i)
        {
            for (int j = 0; j < (n - i); ++j)
            {

                if (a[j] > a[j + 1]){

                    int temp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = temp;

                }
            }

        }

    }
    __TM_end ();
    break;
}

}

void bubbleSort(int a[], int n)
{
    while (1){
        cout << "Trying to start Transaction " << endl;
```

```
        if (__TM_begin (TM_buff) == _HTM_TBEGIN_STARTED){

            for (int i = 1; i < n; ++i)
            {
                for (int j = 0; j < (n - i); ++j)
                {

                    if (a[j] > a[j + 1]){

                        int temp = a[j];
                        a[j] = a[j + 1];
                        a[j + 1] = temp;

                    }

                }

            }

        }
        __TM_end ();
        cout    << "Ended Transaction" << endl;
        break;
    }

}

int main()
{
    const int size = 10;
    int array[size];
    srand(time(NULL));

    for (int i = 0; i < size; ++i)
```

```
    {
        array[i] = rand() % 1000;
    }
    std::thread t1(bubbleSort, std::ref(array), std::ref(size) );
    std::thread t2(reliableBubbleSort, std::ref(array), std::ref(size) );
    t1.join();
    t2.join();
    return 0;
}
```