Transient hardware faults simulation in GEM5 - Study of the behavior of multithreaded applications under faults

Εξομοίωση παροδικών σφαλμάτων υλικού στον "GEM5" - Μελέτη συμπεριφοράς πολυνηματικών εφαρμογών υπό την παρουσία σφαλμάτων

by

Konstantinos Parasyris

Submitted to the Department of Computer & Communication Engineering
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer & Communication Engineering

at the

University Of Thessaly

February 2013

© University Of Thessaly 2013. All rights reserved.

Certified by. . . . .
Nikos Bellas
Associate Professor
Thesis Supervisor

Certified by. . . . .
Christos Antonopoulos
Assistant Professor
Thesis Supervisor

Transient hardware faults simulation in GEM5 - Study of the behavior of multithreaded applications under faults
Εξομοίωση παροδικών σφαλμάτων υλικού στον "GEM5" - Μελέτη συμπεριφοράς πολυνηματικών εφαρμογών υπό την παρουσία σφαλμάτων

by

Konstantinos Parasyris

Submitted to the Department of Computer & Communication Engineering
on Febryary 14, 2013, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer & Communication Engineering

Abstract

Reliable computing under unreliable circumstances is the next challenge the computing community must overcome. To achieve such a difficult task we need to perform a thorough analysis of the way hardware faults manifest errors to architectural components and how such errors affect the applications behavior. In this direction the first contribution of my diploma thesis is the enhancement of new concepts in an already existed fault injection tool which was created by another thesis and improved by mine. The new framework utilized the Gem5 full cycle accurate simulator in order to enable fault injection. The current tool provides a variety of fault injection methods while it is not limited to models covering radiation or timing induced faults, but also facilitates an easily extensible tool to support future effective fault models. Extensive experimentation showed that our GEM5-based fault injection mechanism was very effective in emulating the behavior of faults in modern high-performance processors running complex workloads. An additional contribution of my thesis is the experimental analysis on two different applications: *blackscholes* and *fluidanimate*. We observed that tolerance to injected faults was highly dependent on the spatial location of the faults (e.g. registers, program counter, IF unit, etc.) and on the specific portion of the code affected. To accelerate data gathering and increase simulation speed, we made extensive use of a checkpoint mechanism , called *DMTCP (Distributed MultiThreaded CheckPointing)*, while the whole procedure was automatized to execute on a distributed

Thesis Supervisor: Nikos Bellas
Title: Associate Professor

Thesis Supervisor: Christos Antonopoulos
Title: Assistant Professor

Contents

List of Figures

6-1 Fault Injection Techniques 23

9-1 ThreadEnabledFault - cpuExecutedTicks 42
9-2 Fault Classes 43
9-3 General View of Framework 44
9-4 Application with 4 threads, each thread activates the fault injection framework 52

10-1 Topology of the Laboratory 54
10-2 mainscript.sh responsible for initializing each PC 55
10-3 run.sh responsible for running experiments in each core 56
10-4 (a) Initially all experiments followed this procedure.(b) Only first experiment boots machine and reads input,(c) all remaining experiments execute from the saved checkpoint 57

11-1 Last column represents the solution of one equation,the remaining columns represent the stages until the computation is finished.The red node is a when an error causes a user visible fault.The gray are correct computations 61
11-2 Blackscholes: Fault Injection results 1 thread Unoptimized 62
11-3 Blackscholes: Fault Injection results 4 thread Unoptimized 64
11-4 Blackscholes: Fault Injection results 1 thread Optimized 65
11-5 Blackscholes: Fault Injection results 4 thread Optimized 66
11-6 Fluidanimate: Fault Injection results 1 thread No-Optimizations . . . 67
11-7 Fluidanimate: Fault Injection results 4 threads Unoptimized 68
11-8 Fluidanimate: Fault Injection results 1 thread Optimized 68
11-9 Fluidanimate: Fault Injection results 4 thread Optimized 69

List of Tables

4.1 Fault Classes for Sequential Circuits 17

9.1 Modules where fault injection is supported 45
9.2 Trigger Mechanisms for each Fault Type 45
9.3 Trigger Mechanisms for each Fault Type depending on which-where . 46

11.1 Experimental test Series 58
11.2 Parsec Benchmarks 59
11.3 Blackscholes: Fault Injection results 1 thread Unoptimized 62
11.4 Blackscholes: Fault Injection results 4 thread Unoptimized 63
11.5 Blackscholes: Fault Injection results 1 thread Optimized 64
11.6 Blackscholes: Fault Injection results 4 thread Optimized 65
11.7 Fluidanimate: Fault Injection results 1 thread (Unoptimized) 67
11.8 Fluidanimate: Fault Injection results 4 thread (Unoptimized) 67
11.9 Fluidanimate: Fault Injection results 1 thread Optimized 69
11.10 Fluidanimate: Fault Injection results 4 thread Optimized 69

1 Περίληψη

Κατά τη διάρκεια των τελευταίων δεκαετιών είμαστε μάρτυρες της τρομακτικής ανάπτυξης που εμφανίζουν η επίδοση και η λειτουργικότητα των επεξεργαστών. Το μέγεθος των τρανζίστορ μειώθηκε δραματικά από τα 10μμ στα 32νμ. Αυτό το φαινόμενο είναι συμβατό με την παρατήρηση του Moore· ο αριθμός των τρανζίστορ που μπορούν να τοποθετηθούν σε ένα συγκεκριμένο εμβαδόν διπλασιάζεται ανά 18 μήνες. Αυτή η ανάπτυξη έχει οδηγήσει τους μηχανικούς υπολογιστών στο να σχεδιάσουν πιο περίπλοκες αρχιτεκτονικές οι οποίες όχι μόνο παρείχαν ευρεία λειτουργικότητα άλλα λειτουργούσαν σε υψηλότερες συχνότητες και χαμηλότερη παροχή ενέργειας.

Παρόλα αυτά αυτή οι συνεχόμενη πρόοδος χτύπησέ σε έναν 'τοίχο'. Αυξάνοντας την συχνότητα λειτουργίας και χρησιμοποιώντας βαθύτερα στάδια παροχέτευσης βελτιώναμε την επίδοση του συστήματος, αλλά αυτή η διαδικασία γίνεται όλο και πιο απαιτητική Τα βαθύτερα στάδια παροχέτευσης αποτελούν έναν βασικό τρόπο για να βελτιώσεις την απόδοση του συστήματος. Αλλά αυτή η βελτίωση εξαφανίζεται όταν η καθυστέρηση των Flip-Flop γίνει συγκρίσιμη με την καθυστέρηση των λογικών πυλών Επιπλέον βαθύτερα στάδια παροχέτευσης αυξάνουν το CPI και έχουν αρνητικό αντίκτυπο στην επίδοση του συστήματος. Ο άλλος τρόπος βελτίωσης της επίδοσης του συστήματος ήταν η μείωσή του μεγέθους του τρανζίστορ για να αυξηθεί η συχνότητα λειτουργίας. Αυτή η μέθοδος έφτασε στα όριά της καθώς είτε το μέγεθος του τρανζίστορ γίνεται συγκρίσιμο με το μέγεθος των ατόμων και η συμπεριφορά του δεν είναι ντετερμινιστική είτε το κόστος παραγωγής ενός τέτοιου τρανζίστορ είναι απαγορευτικό.

Τα προηγούμενά οδήγησαν την τεχνολογική κοινότητά να μεταβεί από μονοπύρηνα συστήματα σε πολυπύρηνα. Άλλα φαινόμενα που παλαιότερα δεν είχαν καμία επίδραση στα κυκλώματα πλέον καθυστερούν την εξέλιξη της τεχνολογίας. Η δυσανάλογη επιτάχυνση των τρανζίστορ σε σύγκριση με την ταχύτητα της μνήμης δημιούργησαν το λεγόμενο Power Wall. Ένα φαινόμενο που έγινε ακόμα πιο έντονο με τα πολυπύρηνα συστήματα λόγω της επικοινωνίας μεταξύ του κάθε πυρήνα. Η συσσώρευση των τρανζίστορ αύξησαν την παροχή ενέργειας και δημιουργήθηκε ένα άλλο φαινόμενο ονομαζόμενο Power Wall.

Εκτός από τα προηγούμενα η σμίκρυνσή του τρανζίστορ αύξησε την πιθανότητα εμφάνισης παροδικών λαθών. Αυτά τα λάθη έχουν μη προβλεπομένη συμπεριφορά και απειλούν την λειτουργικότητα των συστημάτων της επόμενης γενιάς. Οι τωρινές τε-

χνολογίες χρησιμοποιούν επιπλέον υλικό για να εγγυηθούν την σωστή λειτουργία του συστήματος. Αυτό το υλικό είναι όχι μόνο ακριβό αλλά καταλαμβάνει και αρκετό χώρο μέσα στο chip. Καθώς το προαναφερθέν πρόβλημα γίνεται όλο και πιο έντονο και οι τωρινές λύσεις δεν είναι ικανοποιητικές οι ερευνητές προσπαθούν να δημιουργήσουν συστήματα τα οποία λειτουργούν αξιόπιστα κάτω από μη αξιόπιστες συνθήκες.

Επομένως αξιόπιστοι υπολογισμοί κάτω από αναξιόπιστες συνθήκες είναι η επόμενη πρόκληση για την επιστημονική κοινότητα. Για να επιτευχθεί ένας τέτοιος στόχος νέα εργαλεία ανάλυσης της συμπεριφοράς των λαθών στο επίπεδο του υλικού καθώς και του λογισμικού πρέπει να δημιουργηθούν. Αυτή η ανάλυση θα μας βοηθήσει να δημιουργήσουμε μια ιεραρχία λαθών και να την συσχετίσουμε με το υλικό έτσι ώστε να μπορέσουμε ενισχύσουμε τα κατάλληλα σημεία του συστήματος.

Ένα τέτοιο εργαλείο είχε δημιουργήσει ο Γιώργος Τζιαντζιούλης κατά την διάρκεια της πτυχιακής το οποίο και επέκτεινα κατά την διάρκεια της δικής μου πτυχιακής. Το εργαλείο που δημιουργήσαμε είναι βασισμένο στον προσομοιωτή Gem5 και μας επιτρέπει την εισαγωγή λαθών στο σύστημα έτσι ώστε να μπορέσουμε να μελετήσουμε την συμπεριφορά εφαρμογών όταν αυτές εκτελούνται με την παρουσία λαθών.

Η άλλη συμβολή της πτυχιακής μου είναι η αυτοματοποίηση της διεξαγωγής πειραμάτων σε 30 διαφορετικούς υπολογιστές έτσι ώστε να ελαχιστοποιηθεί η διάρκεια εκτέλεσης των πειραμάτων. Τέλος εκτιμήθηκε η συμπεριφορά 2 διαφορετικών εφαρμογών υπό την παρουσία λαθών.

Το πρώτο κομμάτι της πτυχιακής μου αναφέρεται στην αξιοπιστία, το δεύτερο κομμάτι παρουσιάζει το εργαλείο μας και στο τρίτο παρέχονται τα αποτελέσματα των πειραμάτων.

2 Introduction

During the past decades we witnessed a terrifying development in the performance and functionality of processors. The consecutive decrease of the transistor size conveying us from 10μm feature size to today's 32nm . A trend speculated by Moore's Law, doubling the processors transistors every 18 months. This multitude led computer engineers to design more complex processor architectures which not only gave great functionality but also functioned at higher frequency and lower voltage.

However this constant progress has hit a wall. Increasing clock frequencies and using wide issue architectures has worked well to improve processor performance, but recently has become more challenging. Deeper pipeline is one of the key techniques to increase the clock frequency and performance, but the benefit of the deeper pipeline is eventually diminished when the inserted Flip-Flop's delay is comparable to the combinational logic delay. Moreover deeper pipeline stages increase Cycles Per Instruction (CPI) and negatively impact the system's performance. The other key technology technique - shrinking the transistor size to increase the clock frequency and integration ability has eventually reached it's limit. Either because of the physical limit when the size of transistors approaches the size of atoms, or because of the fabrication cost prohibiting further progress.

The current solution for the addressing problems was to shift from uni - core to multi-core processors. However secondary phenomena that on previously generations had little to no effect halt the system performance halted the evolution. The disproportional speed-up of transistors, compared to that of DRAMs, created the so called "memory wall". A phenomenon that has become greater on multi-core processors due to the communication between the cores. Their continuous accumulation increased circuit power density to an unbearable degree, a phenomenon called "power wall".

Besides the previously mentioned scaling of the CMOS feature size increase the probability of transient faults (radiation-induced faults which are caused by cosmic particles that enter the atmosphere). These faults have an unpredictable impact on gate delay and consequently lead to delay failures threatening the functionality and operational reliability of next generation systems. Current technologies use extra hardware for error detection and correction which not only is expensive but also posses a great proportion of the total area of the chip. These techniques are very strict

and detect - correct all faults without taking into consideration the effect of the fault on the primary output. As the previously mentioned obstacle becomes more intense and current solutions are not satisfactory researchers are working on building reliable systems which operate under unreliable circumstances (dysfunctional hardware components or dysfunctional operation due to transient faults) . Another important aspect is the decrease of process yield. Accumulation of more components for a single system means that there is a greater chance that part may contain corruptions, hence lead to a corrupted output. This problem is solved inefficiently by disabling such modules,however this has an disadvantageous impact on the systems cost and area utilization. Besides all these , power efficiency can be achieved by enabling processors usage in the sub threshold levels.

Reliable computing under unreliable circumstances is the next challenge the computing community must solve. To achieve such a difficult task we need to perform a thorough analysis of the way hardware faults manifest errors to architectural components and how on their turn affect the applications behavior. The analysis of the faults may help us construct an hierarchy of target-modules that need to be enhanced in order to achieve robustness.

To achieve a high grasp of the previously mentioned phenomenon new tools had to be constructed and the existing ones to be extended by adding new functionalities. The main contribution of this thesis is extending such a tool. In order to study the effects of transient faults on various applications G. Tziantzioulis has developed such a framework on top of M5. This framework was extended by this thesis in order to cover our needs.

The other contribution of this thesis is to automatize the experimental procedure on a distributed system in order to speed up the process of experimenting. Finally evaluation of 2 applications (blackscholes, fluidanimate) included in the Parsec Benchmark Suite on an unreliable environment.

This document is constructed in three blocks. The first one introduces a theoretical background, the second describes our framework and the laboratory setup and the third part outlines the experimental evaluation.

In the first part we discuss the concept of dependability (Chapter 2) , various fault models which are used widely for research purposes (Chapter 3). Chapter 4 describes some characteristics of application which demonstrates tolerance towards faults. Chapter 5 describes fault injection methods and chapter 6 describes the benefits of a simulator.

In the second part starting at chapter 7 we discuss the fault model which this work is based on, On chapter 8 the tool is demonstrated and on chapter 9 methods which accelerated the experimental procedure are discussed.

The third part chapter 10 is an analysis of the experimental results for each application separately.

Finally, chapter 11 presents the conclusion of this work and directions for future work.

3 Introduction To the Reliability Theory

In the previous section we discussed the need for fault tolerant design in the near future systems. In this section we will introduce some simple aspects of Reliability theory in order to obtain a better grasp on how faults are categorized and how they are connected to Power consumption.

3.1 Dependability

Dependability is an 'umbrella' term that comprises core attributes that describe particular dependability-related aspects of relevant performance attributes.

3.1.1 The Concept of Dependability

Depending on the applications different emphasis may be put on different aspects of dependability. According to the application behavior dependability may be viewed by different viewpoints which enables the attributes of dependability[5]. The attributes are the following:

Availability Describes the extent to which an item is operational and able to perform any required function or set of functions if a demand is placed on it. It is derived from reliability and maintainability (where hardware failure is concerned)

Reliability Measures the component continuity, hence how long does the component function as it is constructed to function

Safety The non occurrence of catastrophic and unexpected consequences.

Confidentiality The non occurrence of unauthorized disclosure of information.

Integrity The non-occurrence of improper alteration of Information.

Maintainability Measure the ability to perform maintenance under given conditions and indicates the ease with which an item can be repaired. A high degree of maintainability means that repairs consume little time and effort, on average. Maintenance support describes various aspects responsible for maintenance, e.g., skill of repair personnel, location of repair facilities, traveling time, time taken to procure spare parts, etc.

An association of integrity and availability with a combination of authorized actions together with confidentiality lead to **Security**.

Reliability

The reliability of a system is the probability function $R(t)$, defined on the in the internal $[0, \infty]$, that a system will operate correctly with no repair up to time t. The reliability is defined as a function o failure rate $\lambda(t)$ .Another commonly variable used to describe the systems reliability is the Mean Time To Failure

$$R(t) = e^{-\int_0^t \lambda(t)dt} \tag{3.1}$$

$$MTTF = \int_o^\infty R(t)dt \tag{3.2}$$

3.1.2 Factors of Dependability

A system may not always perform the function it is intended for. The causes and consequences of deviations from the expected function of a system are called the factors to dependability [32] :

Fault is a physical defect, imperfection, or flaw that occurs within some hardware or software

Error is a deviation from accuracy or correctness and is the manifestation of a fault.

Failure is the non-performance of some action that is due or expected

When a fault causes an incorrect change in a machine stage, an error occurs. Although the fault may remain localized in the affected code and alter the functionality of a certain circuit, multiple errors can originate one fault site and propagate throughout the system.

3.1.3 Fault categorization

A fault as a deviation in a hardware or software component from its intended function can arise during all stages in a computer system design process[32]: Specification, design, development, manufacturing, assembly, and installation throughout it's operational life. Most faults that occur before full system deployment are discovered

and eliminated through testing techniques. Faults which are not removed on this stage may reduce the dependability of the system. Hardware- Physical faults are best classified based on their duration:

Permanent faults Caused by irreversible component damage, such as a semiconductor junction that has shorted out because of thermal aging, improper manufacture, or misuse. Recovery can only be accomplished by replacing or repairing the damaged component or subsystem.

Transient faults triggered by environmental conditions such as power-line fluctuation, electromagnetic interference, or radiation. These faults rarely do any lasting damage to the affected component, however the may lead the system to an erroneous state. Transient faults occur far more often that permanent ones, and are far harder to detect.

Intermittent faults Caused by unstable hardware or varying hardware states. They can be repaired by replacement or redesign.

3.1.4 Methods for dependable Computing

In order to achieve dependable computing there is a need for a combined utilization of a set of methods-mechanisms [18] :

Fault prevention How to prevent a fault to occur.

Fault tolerance How to ensure that under presence of faults a service will achieve the system's functionality.

Fault removal How to reduce the number as well as the seriousness of faults

Fault forecasting How to forecast the upcoming faults and the consequences of them

3.2 Power Consumption - Reliability

Although this thesis does not concentrate on power consumption aspects a brief introduction will be done in order to understand how the consumption of the system affects its reliability.

3.2.1 Methods of Decreasing Power Consumption

The following power model help us in the abstraction of the problem and keep an convenient distance from a more detailed model which would add more information about hardware components and would be difficult to comprehend. The model is taken from [12]

$$P = P_s + h(P_{ind} + P_d) \tag{3.3}$$
$$P_d = C_{ef} V^2 f \tag{3.4}$$

- P_s is the static power (Includes the power to maintain basic circuits active, keep clock running and memory in power saving sleep mode)
- $h = 1$ if circuit is active or 0 if circuit inactive.
- P_{ind} is the frequency independent active power.(consists of part of memory and processors power as well as any power used that can removed by putting the system into system sleep state and is independent of system supply voltages and processing frequencies)
- P_d is the frequency dependent active power.
- C_{ef} is the switch capacitance
- V is supply voltage
- f is operating frequency

Dynamic Frequency Scaling

Dynamic Frequency Scaling (DFS) the processor frequency is lowered and thus the power consumption is decreased since frequency dependent power is related with the system frequency, however as a major disadvantage the cycle period is increased and this means worse performance. Moreover since the execution time is increased the probability of a fault also increases.

Dynamic Voltage Scaling

Dynamic Voltage Scaling (DVS) the supply Voltage is decreased. Due to the square relationship between supply Voltage and frequency dependent active Power the overall gain is significant. However as the voltage supply is decreased the energy of a circuit decreases and becomes more vulnerable to smaller energy particles and so the probability of a fault is greater

Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) as the name implies decreases both frequency and Voltage supply however due to the previously stated reasons the probability that a fault is going to manifest is greatly increased.

3.2.2 Power Consumption - Faulting Probability

The relationship between fault probability and the frequency is the following [28]:

$$\lambda(t) = \lambda_o 10^{\frac{(b(1-f)}{(1-f_{min})}} \tag{3.5}$$

- λ_o is the nominal failure rate per time unit.
- $b > 0$ is a constant
- f is the frequency scaling factor
- fmin is the lowest operating frequency
- The failure rate is maximal at $fmin/Vmin : \lambda_m ax = \lambda_o 10^b$

4 Fault Modeling

As mentioned before faults are sorted into 3 main categories. Permanent, Transient , Intermittent faults. In this chapter we will demonstrate some mathematical models which correspond to the behavior of each fault.

4.1 Permanent Faults

Permanent faults usually occur because of a deflecting hardware component. Many models have been used in order to create tests for such faults. There will be demonstrated the most commonly used Single Stuck at faults and bridging faults.

4.1.1 Single Stuck at Faults

The single stuck at fault model is one of the most widely used fault models in practice. There are 2 types of stuck at faults:

1. stuck at 1 (s-a-1) for which he faulty net takes permanently the value 1.
2. stuck at 0 (s-a-0) for which the faulty net takes permanently the value 0.

The model has some basic assumptions :

1. The fault only affects the interconnection between gates.
2. Only one line in the circuit is faulty
3. The fault is permanent set to 0 or 1
4. The fault can be in the either in the input or in the output of the gates
5. The fault does not affect the functionality of the gates in the circuit

Because of its simplicity this model offers many advantages in fault modeling. This is the reason why this model is the most used in the industry. Some of the advantages are the following:

- The model covers a large portion of manufacturing defects.
- It comparably easy to develop algorithms to generate test patterns for stuck at faults detection. Current already developed algorithms are very efficient
- It results in a reasonable fault number. At most $2n$. This number can be furthered reduced with fault collapsing techniques.
- Some other fault models can be mapped into a series of stuck at faults.

However the stuck at faults does not cover all permanent fault types.

4.1.2 Bridging Fault Model

A bridging fault model corresponds to a shortcut between a group of signals. This is usually modeled at the gate or transistor level. Bridging faults are usually found in lines which are placed physically close by. A bridging fault can be 1-dominant,0-dominant. They can also be classified as non-feed back bridging faults and feedback bridging faults. Non feedback faults are usually combinational and stuck at fault testing may detect them. Feedback faults produce memory states.

4.2 Transient Faults

As stated before as systems are scaled down to the very deep sub micron range they are increasingly sensitive to radiation strikes or other similar single event upsets (SEU), which are capable of producing transient (soft) errors. Their impact ranges from o minor glitch to a major crash. The behavior depends mainly on interacting physical factors, such as :

- Where The SEU occurs
- How much energy has the strike circuit have at the current state
- The strike time relevant to the system clock cycle.

4.2.1 Single Transient Fault Model

This model is targeting Synchronous digital circuits composed of logic gates, Flip Flops , Register transfer level (RTL). $C = (I, O, S, \delta, \lambda, So)$ is a sequential circuit with k logic lines. A single transient fault(STF) is given by $f(l/p, x, s)$ and defines the next properties [27]

1. it causes line l to be stacked at the opposite of the current value (flip) for one cycle.
2. The new state of C is x,s where $x \in I$ and $s \in S$

The total explicit number of STFs in C is $2k|l||S|$. Although the method gives a numerous of faults the problem is not intractable. There is no relationship between an STF and the associate state of C. The STF is a fault that occurs in state x by chance.

Although there is a huge similarity between STF and SAF (stuck at faults) the main difference is that an SAF remains after it is manifested and is not associated with specific states.

Combinational Circuits

In such cases STF is similar to an SAF and is reduced to a more simple form $f(1/p, x)$. Assuming that C has one output z, k lines and n inputs. Assuming that all STFs are equiprobable. The error probability $perr(z)$ is the total number of possible errors seen at output z divided by the total number of STFs.

Sequential Circuits

In such cases STF becomes a bit more complicated. Besides the output of the sequential circuit there is an internal state which controls the functionality of the circuit. Suppose that z is the output of the circuit and c is a wire that controls the next stage. the next table lists a set of classes which will help us understand the model. After

STF class	Class definition	Number of STFs in class
FaultType1	No effect neither on z nor on c	Calculate STFs of Circuit under examination that do not effect the circuit at all
FaultType2	Error on z no effect on next stage(c)	Calculate STFs of Circuit under examination that effect only z
FaultType3	Error on c but no effect on current result z	Calculate number of STFs of Circuit under examination that effect only c
FaultType4	Error on both z and c	Calculate number of STFs of Circuit under examination that effect both c and z

Table 4.1: Fault Classes for Sequential Circuits

calculating all the number of STFs for each class each easy to find the probability for each fault class by dividing the faults of the class with the total number of STF in all classes. After Calculating the probabilities a new FSM must be constructed by taking into consideration all the new faulty transitions.

4.3 Intermittent Faults

Intermittent faults is a malfunction of a device or a system that occurs at intervals, usually irregular in a device that normally functions correctly [19]. An intermittent fault is caused by several contributing factors, some of which may be random , which occur at the same time. The more complex the faulting mechanism is involved the greater the probability of an intermittent fault.

A simple example of intermittent fault is a borderline electrical connection in a wiring component (bridging fault) because of 2 conductors may be touching each other leading to a increase in temperature. An application may fail to initialize a variable which is required to be initially 0 if the program is executed under circumstances that memory is cleared before executing, it will malfunction on the rare occasions that the memory is not stored before loaded.

This faults seemingly to occur randomly for a period of time $[t1, t2]$ and the occurrence is strongly related to the state of the device and the state of the application. These factors make intermittent faults difficult to model. Most model use SAF for a period of time in order to model the behavior of Intermittent faults.

5 Self-Tolerant Applications

Usually, a program executes properly only when the produced architectural state is correct on a cycle-by-cycle basis. A looser (though still fairly strict) notion of program correctness commonly adopted by reliability researchers is that the visible memory state after program completion is correct in its entirety. Such strict notions of program correctness are appropriate for traditional workloads that are numerically oriented. However, a growing number of important workloads produce results that have a higher (often qualitative) user-level interpretation. These computations are been referred to as soft computations. An example of a soft computation is the processing of human sensory information common in multimedia workloads. Another example is cognitive information processing, an emerging application domain that applies artificial intelligence algorithms for reasoning, inference, and learning to commercial workloads. While data corruptions can change the numerical result of soft computations, they often do not change the user's interpretation of those numerical results. Consequently, faults that would otherwise be deemed unacceptable from a numerical standpoint may in fact be tolerable (or even imperceptible) from the user's standpoint. Systems that can identify and exploit such error resiliency at the user level offer new opportunities for fault tolerance optimization.

5.1 Soft Computing Characteristics

In the past, researchers have observed soft computing characteristics and proposed exploiting them for reduced energy consumption [24, 2, 3, 25] as well as for fault tolerance in ASIC design [1, 23]. Three important characteristics of soft computations make them resilient to error: redundancy, adaptivity, and reduced precision [22].

Redundancy Soft computations that are iterative or that exhibit reduced precision often contain some degree of redundancy. These redundant computations contribute to the application result,but may not improve answer quality appreciably. Programs with redundant computations are more error resilient because the redundancy can mask faults.

Adaptivity The possibility of error has been taken into account during the design of many soft computing algorithms . This is particularly common in those that compute on noisy or probabilistic data. Such soft computations include code to detect certain forms of error noisy or probabilistic data. Such soft computations include code to detect certain forms of error, and adapt the computation accordingly. Due to their self-healing nature adaptive codes are naturally error resilient.

Reduced Precision Reduced Precision: Soft computations often have precision requirements that are lower than the data types supported by the programming environment / hardware architecture. These soft computations are resilient to errors that modify data values within the precision tolerance, as described earlier. Furthermore, they are tolerant to errors whose magnitude decay as the errors propagate through the computation.

5.2 Define Program correctness

These attributes offer a less strict definition of program correctness because of the fact that soft computations in comparison with traditional numerical-oriented computations exhibit an increased resilience to faults. Five definitions of program correctness are listed below in increasing strictness as stated in [22]:

1. Architectural state is numerically correct on a per-cycle (or per multiple-cycle) basis.
2. Output state (i.e., computation results visible at program completion or during system calls) is numerically correct.
3. Output state is numerically correct within some tolerance.
4. Output state is qualitatively correct based on higher-level interpretation.
5. Output state is qualitatively correct based on higher-level interpretation within some tolerance.

Definitions I,II are widely used for evaluating program correctness in existing fault tolerance researches. The remaining definitions are less strict and are appropriate for soft computations. Even though definition III is numerical, it allows for a slight error resiliency. That may be possible because results are computed in a greater precision than necessary. Definition IV applies to applications which have a higher level of interpretation than the numerical stand point. Finally definition V is quiet similar to definition IV but allows for some error even at the interpretation level.

One major disadvantage of the previous definitions is that all of them underestimate other factors of reliable systems. Many faults may not have an impact to the output of the program but may in fact propagate to other application causing them to have different results . Moreover if these faults are propagated on important system applications (OS) the results may prove to be destructive. In addition, faults could

possibly prolong the execution time of an application without altering the output, for example more iterations on an iterative method. Increasing the execution time of a real time application can lead to a fatal error while increasing and the possibility of other errors to manifest.

6
Fault Injection

After introducing the aspects of dependability, fault-tolerance, correctness of a system in this chapter we will represent Fault Injection a technique to evaluate it. Fault Injection Is defined as the dependability validation technique that is based on realization of the controlled experiments where the observation behavior of the system is present of faults, in explicit induced by the deliberate introduction of faults into the system [32, 6]. Fault tolerance is measured in both electronic hardware and software systems by using fault Injection. Hardware may be injected with faults into the simulations of the system as well as into implementation. Software faults can be injected in the simulation of a system.

Moreover, fault injection is split into execution and simulation based. The first the system is functioning and some fault creation mechanism are used. The execution then is observed to determine the behavior of the system. The second model a model of the system is created and faults are introduced into that model. The model is then simulated to find the effects of the fault. These methods are usually slower however easier to change.

Furthermore from another viewpoint fault injection technique's can be grouped in another two categories, invasive and non-invasive. The former one contains mechanism which leave a footprint after a fault is injected. This footprint may be fatal for real time applications because the footprint may delay some executions. . Non invasive mechanisms are those which do not leave a footprint and the fault injected system does not realize the corruption.

6.1 Fault Injection Categories

6.1.1 Hardware-Based Fault Injection

Special designed hardware allows fault injection in the targeted system. Usually the faults are injected into the Integrated Circuit (IC) pin level. The system traditionally is injected with stuck at , bridging, transient faults, and after the injection the overall behavior of the system is studied. The system under examination is subjected into

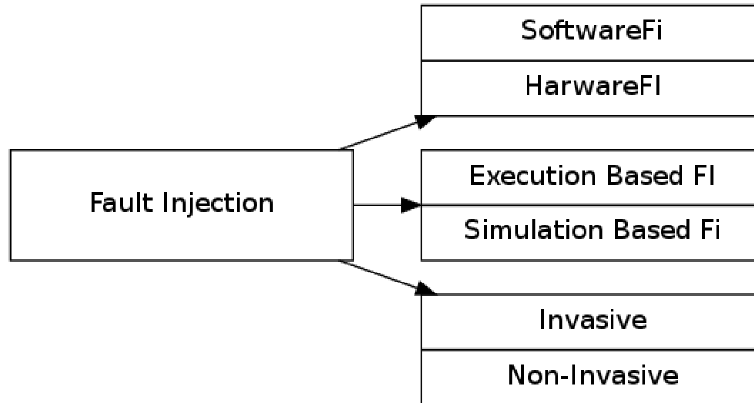


Figure 6-1: Fault Injection techniques

some kind of interference to produce the faults [4]. The hardware based fault injection has two types of fault injecting techniques.

Forcing technique: The fault is injected directly into the circuit without any disconnection of the parts .

Insertion technique: A special device replaces a part of the circuit which injects faults to the system.

Advantages

1. Hardware fault injection can access locations which are difficult to access by other means.
2. The technique is great for systems which need high time resolution
3. The experiments are fast. Runned in near real time giving the opportunity to run many experiments and having statistically reliable evaluation of the system.

Disadvantages

1. Hardware fault injection introduces the system in damage risk.
2. Many devices are used limiting the accessibility of fault injection
3. Special purpose hardware is required in order to inject faults.
4. Expensive in terms of cost

Tools

- AFIT[6]: Pin level fault injection system produced by the Polyethinc University Of Valencia (Spain)

- RIFLE[6]: Pin level fault injection system made at University of Coimbra, Portugal

- FOCUS[6]: A chip level fault injection system developed at the University of Illinois at Urbana-Champaing (USA)

- FIST[6]: A heavy ion radiation fault injection system developed at Chalmers University of Technology (Sweden)

- MESSALINE [6]: A pin level fault forcing system developed at LAAS-CNRS, France.

- MARS [6]: A time triggered fault tolerant distributed system made in Technical University of Vienna (Austria)

6.1.2 Software-Based Fault Injection

Nowadays software faults (bugs) are probably the major cause of system failures. Software based fault injection is a method of assessing the consequences of hidden bugs. In most cased this technique involves the modification of the software executing on the system. All kind of faults may be injected from the memory to registers.

Software fault injection has a deep connection the implementation details, and may address the program states as well the communication and the interactions. The system is runned with faults in order to examine the behavior of the system

Software injection may be non intrusive if the timing of the application under test is not time relevant. On the other hand if the timing is involved in the application then the required time of the injection may disrupt the result of the application.

Advantages

1. This technique can target applications and operating systems which are difficult to be targeted by using hardware fault injection.

2. Experiments are running in almost real time, giving the opportunity to run a big number of experiments.

3. Special Purpose hardware is not required.

Disadvantages

1. Limited fault injection places. The lowest level of injecting the fault is at the assembly instruction.

2. Modification of the source code in order to support the fault injection, so the executing code is not the same code that runs in the field.

3. It is very difficult to model Permanent Faults.

Tools

- BOND[6]: A fault Injection tools for COTS applications developed at Politecnico di Torino (Italy).
- XCEPTION[10]: A software fault Injection tool for dependability analysis implemented at University of Coimbra (Portugal).
- MAFALDA[14]: A fault injection tool for real time COTS developed at LAAS-CNRS, Toulouse (France).
- DOCTOR[15]: An integrated tool developed at University of Michigan (USA)

6.1.3 Simulation-Based Fault Injection

The simulation Based Fault Injection tools involve the creation of simulation model of the system under analysis including a detailed model of the processor in use.The simulator is usually developed in hardware description language such as Verilog - VHDL.

Advantages

1. Simulation fault Injection can support all level of abstraction.
2. No intrusion is conducted in the simulated System
3. Great support of fault models and fault injection mechanisms.
4. Does not require special purpose hardware.
5. Can model all kind of faults (permanent,transient, intermittent).

Disadvantages

1. Implementation needs a lot effort and time.
2. Time consuming.
3. The accuracy of the results depends on the goodness of the developed model and of the abstraction level of the simulator.
4. Models may not include all the possible design faults that can be present in a real system

Tools

- VFIT: A fault injection toolset, develop at University of Valencia (Spain).
- MEFISTO [21]: A tools using VHDL models to conduct fault injection developed at University of Technology (Sweden).
- ALIEN [26]: A tool based on mutation techniques developed at LCIS-ESISAR, valence (France)
- VERIFY [29]: VHDL fault injector developed at University of Erlangen-Nurnberg (Germany)

6.1.4 Hybrid Fault Injection

The hybrid approach combines two or more fault injection techniques in order to gain more controllability - accessibility or other attributes.For example a fault injection technique combined with the Simulation based Fault injection gains in terms of accessibility an observability. On the other hand a combination of Hardware and software fault injection combines the accuracy of the hardware injection with the versatility of the software injection.

Tools

- LIVE [6]: Hybrid hardware/software fault injection tool developed at Ansaldo-Cris (Italy)
- A software/simulation-based fault Injection developed at Chalmers University of Technology (Sweden)

6.2 Categorization From Another Viewpoint

Fault injection methods may be categorized based on their abstraction level. To be more precise faults may be injected to any abstraction level of the system. The abstraction levels are considered as "black boxes". The level of abstraction is directly analogous to the volume of a black box; the more complicate the functionality of a black box, the higher its abstraction level.

Experimentation in system's dependability, fault detection and fault recovery mechanisms may be verified by injecting faults in any level of abstraction. Researchers prefer to work on low levels in order to achieve optimal accuracy. However as mentioned before (Hardware based fault Injection) problems may arise from such a choice. Consequently there is a great interest to investigate how faults propagate to higher level mechanisms in order to create methodologies for fault injection to these levels.

The following taxonomy as given in [31] presents a categorization of faults based on their abstraction level. It can be categorized into 2 main categories circuit level and functionality level. As the name imply circuit level describe a low level abstraction and the functionality category describe a higher level of abstraction.

Circuit: A physical viewpoint of the processor is considered.

- Device : Focuses on transistor and other circuit elements. Hardware fault injection on devices is achieved with radiation or other physical stress methods. Simulated fault injection needs an analog Simulator.
- Gate : Stuck at fault model is used in order to inject gates (AND,NOR,NOT,etc). Sometimes more accurate models are used.
- Basic Block: The abstraction level of this method is high. Faults are injected in entire "units" such as register file, adders etc.
- Chip Focuses in the chip's I/O boundary

Functional: The circuit description in this category is not used however the functionality of a unit is used.

- Micro-operation: Focuses in micro-instruction and faults are injected at data transfers and micro sequencing.
- Macro-operation: The ISA is targeted and the instruction word is hit with faults.
- System: Faults are inserted in memory and in Processors I/O.
- Network: Communication methods are targeted.

6.3 Our Choice

After carefully studying past implementations and publications in the are of fault injection we concluded that the hybrid model, simulation based fault injection combined with software based fault injection suits our purposes the most. To be more precise we adopted full system simulations which serve our purposes. Using a full system simulator we can evaluate the impact of faults in large widely used workloads. The software-based fault injection serves the purposes of which thread/application will be inserted with faults. However we tried to keep minimized the alteration of the source code. More information will be given later in the document.

Simulation provides the maximum controllability of all previous mentioned methods in both spatial and temporal manner. Moreover it enables to study applications while running under real circumstances in the presence of OS and other applications. Another key factor for pushing us towards this choice is the observability that simulations offer. By tracing the behavior of the system before and after a fault injection can provide many useful information.

Simulation based fault injection is a non intrusive method and can provide statistics and logs of great detail. In addition our choice is open source so we can alter the modules on each experiment to match our desires.

Finally we try to nullify the time overhead of the simulation by running many experiments in parallel and using smart checkpointing. Only simulation based techniques offers such an attribute since the installation cost is none , hence many experiments may be runned concurrently.

The simulator we chose to extend with fault injection was Gem5. More information about the simulator will be given in the next section.

7 Full System Simulations

7.1 Introduction

A full system simulator is a computer architecture simulator that simulates an electronic in such level of detail that complete software stacks for real time systems can run on the simulator without any modification. A full system simulation provides virtual hardware that is independent from the hardware of the host computer. The full system simulator model typically provides virtual processor cores, memory, interconnection networks, etc.

Due to these characteristics full system simulators can run operating systems without the need to modify them. This provides a great opportunity to test innovative ideas on a simulator and after provide them on real computers.

One of the ”disadvantages” of full system simulators is that the more detailed information we include in the simulator the more time it executes.

7.2 Simulation Attributes

The past decades simulators have shown an increasing amount of interest. This is reasonable because simulator offer the ”ability of testing” with almost no cost. New cache coherence protocols may be tested in a simulator to see the performance of the protocol and if the results are acceptable the protocol may be implemented in real hardware.

As Jakob Engblom states a simulator is ”just software” and offers many advantages compared to a real machine[13].

1. **Configurability** All possible ”hardware” configuration may be used. Hardware resources do not place a constraint on a simulator.

2. **Extendability** The simulator can be extended with any ”new” modules we wish (DRAMS,MEMORY)

3. **Controllability** The execution of the simulator can be stopped - restarted, thus complete control of the simulation must be offered.

4. **Determinism** A simulator is completely deterministic. (Behavior must be the same as if it was a real computer)

5. **Globally synchronous** Multiple processors, multiple devices, multiple networks all can be stopped at the same time and get a global snapshot of the system

6. **Checkpointing** The state of the system can be written in the disk and restored at any time.

7. **Availability** Creating a new machine is a matter of copying the setup. There is no need to produce hardware prototypes.

8. **Inspectability** The state of the simulator can be viewed and monitored without affecting the simulator.

9. **Sandboxing** The simulator environment is completely isolated. No external variable can affect the execution of the code.

The previous mentioned attributes create a perfect environment for a system design. The coexistence of Configurability with Extendability give the opportunity for many designers to surpass the current technological limits and hardware barriers. Controllability,Inspectability,Globally synchronous give the opportunity to have full knowledge of the simulating system during its execution. Checkpointing and Determinism give produce a perfect system for debugging. Create a checkpoint before the "bug" restore from there until the bug is fixed.

A major feature for both the industrial and academic community is the availability of the simulated system. After the initial cost of producing the simulator the duplication comes with no cost nor in budget nor in time. This is feature is used in thesis in the fullest for mitigating the time overhead of the simulation by running multiple experiments in parallel . More information about the setup of the experiments is going to be given later in the document.

7.3 Gem5

For the purposes of our research we used and extended the Gem5 simulator[9, 20]. Gem5 is described from its main site as "a modular platform for computer system architecture research, encompassing system-level architecture as well as processor micro-architecture"

Gem5 is a full system simulator which is the merge of the best aspects of the M5 and the GEMS simulators. M5 provides a highly configurable simulation framework, multiple ISA and diverse CPU models. GEMS extends these feature with flexible memory system, cache coherence protocol and interconnect models. Until now it consists of about 180k lines of code (C++,Python) and is freely distributed under the BSD style license and has no dependency to any commercial or restrictive license software. These features make Gem5 ideal tool for research purposes.

Object Oriented design encapsulates Gem5's flexibility. The ability to construct configurations from independent objects leads to advanced capabilities such as multi-core and multi-system design. All basic simulation components in the gem5 simulator are SimObjects and share some common behaviors for configuration, initialization,statistics and serialization(Checkpoint). SimObjects represent concrete hardware components (processor cores,interconnect elements, etc).

Every SimObject is represented by two classes. One in Python and one on C++ which derive from SimObject base classes present in each language. The python class definition specifies the SimObject parameters and is used in the script based configuration. The configuration provides mechanisms for instansiation, naming and setting parameter values. The C++ classes handles the SimObject state and behavior. All this are feasible by using SWIG which exports information from the C++ class to the Python class and thus to the configuration script.

Along with the configurability of Gem5, four different CPU models are provided each of them lie to a specific point in the speed vs accuracy spectrum. Atomic Simple is a Single IPC CPU model, Timing Simple is similar but also simulates the timing of memory references, InOrder is a pipelined in order CPU, and O3 is a pipelined out-of-order CPU model.

Besides the four CPU models Gem5 includes two different memory system models. Classic and ruby. The classic provided by M5 is fast and easily configurable memory system while the RUBY model from GEMS provides a flexible infrastructure capable of accuracy simulating a wide variety of cache coherence memory systems. Moreover gem5 can operate in two modes

System Call Emulation (SE) In this mode gem5 emulates most common system calls. Whenever the program executes a system call, gem5 traps and emulates the call usually by passing it to the host OS. Currently there is no thread scheduler in SE mode so threads must be statically mapped to a core limiting its use with multi-threaded applications

Full System (FS) In this mode gem5 support an environment for running an OS. This includes support from interrupts exceptions I/O devices. FS improves the simulation accuracy and variety of workloads that gem5 supports. Running benchmarks in FS mode produces more realistic results.

Apart from the previous because of Gem5's uniform API across object types allows to interchange similar simulated objects. The simulation can start with a Simple functional CPU (in order to speed up the initialization process of the experiment , even boot the machine (FS)) and then change to a more detailed model from which statistics and results.

Many ISAs are supported by Gem5, including Alpha, MIPS,. ARM, Power, SPARC and x86. The simulator's modularity allows these different ISAs to be plugged into the generic CPU models and the memory system without having extra overhead. However some combinations of ISAs and other components are not working.

All the above features leaded us to chose Gem5 as the most suitable simulator for implementing the fault injection framework. Although other full system simulators

supported fault injection our framework is unique because no only it covers a large amount of errors but also provides high timing accuracy. Finally it is provided by free-software license enabling modification and experimentation by anyone.

8 Generic Processor Fault Model

After explaining the reasons why we chose Gem5 as the base of our fault injection Tool in this section we provided an overview of the fault model we based our implementation. In general providing a sufficient fault model is almost impossible. The sufficiency of the model is measured after it's creation with experimental and historical data which are published in literature.

Yount and Siewiorek developed a generic fault model [31, 30] for the register file within a processor. Johnson, Cutringht and DeLong [11] measured the sufficiency of the model through simulations. The generic behavioral-level fault model describes the faulty behavior of a general purpose, implementation independent processor .

The model consist of seven locations where faults can manifest.

1. Register File
2. Program Counter
3. Control Unit/Instruction Decode
4. Bus
5. ALU
6. Fetch and execute
7. Memory Mapped Peripheral Functional Block

The framework that we developed covers 4 of this places (1,2,3,5). A more detailed description for each fault place will be given. excerpts from the following text are taken from [11] where a detailed presentation of the model can be found.

8.1 Register File Fault Model

8.1.1 Register Fault Model

The Register fault model represents faults that propagate to the registers of a processor defined by the programmers model or even special purpose registers.

where

Any register visible to the programmer's space or any special purpose register of CPU may be a possible candidate for a corruption

when

Corruption can occur at an instruction boundary.

what

Regarding the value that is going to take the faulty place there are 4 possible scenarios:

1. Missed load all or part of a register is not loaded when it should be

$$R_k \leftarrow expr \Rightarrow R_k \leftarrow (expr \oplus (mask\langle(w-1)...0\rangle)) \quad (8.1)$$

2. Extraneous load (6.2) Part of the register or even the whole register is loaded when it should not be.

$$R_k \leftarrow expr \Rightarrow R_j \leftarrow expr \quad \exists(j \neq k) \quad (8.2)$$

3. Level change in storage. The value of the register is $\oplus$ with a mask value and stored to the register (often referred as the bit flip model)

$$R_k \Rightarrow R_k \oplus (mask\langle(w-1)...0\rangle) \quad (8.3)$$

4. All 0/1 (6.4) Assign all bits of a register to 0 or to 1

$$R_k \Rightarrow R_k \oplus R_k \quad (8.4)$$

$$R_k \Rightarrow R_k \oplus \bar{R}_k \quad (8.5)$$

8.1.2 Read/Write Register Selection Fault Model

The read/write Selection fault model represents an error that propagates to the decoding stage. The decoding stage erroneously selects a given register other than the correct one.

$$(R_k \leftarrow R_i \; op \; Rj) \Rightarrow R_k \leftarrow (R_x \; op \; R_j) \quad x \neq i \quad (8.6)$$

$$(R_k \leftarrow R_i \; op \; Rj) \Rightarrow R_k \leftarrow (R_i \; op \; R_x) \quad x \neq j \quad (8.7)$$

$$(R_k \leftarrow R_i \; op \; Rj) \Rightarrow R_x \leftarrow (R_i \; op \; R_j) \quad x \neq k \quad (8.8)$$

Where

Instructions during the fetch and decoding stage.

When

Corruption can occur at an instruction boundary.

What

The corruption that will propagate to read/write error are shown in the following equations:

$$instr_fetch(addr) \Rightarrow \\ instr_fetch(addr) \\ \oplus \\ (((x - a - i)\#mask\langle(i + a - 1)...i\rangle)\#(i@0)) \quad (8.9)$$

$$instr_fetch(addr) \Rightarrow \\ instr_fetch(addr) \\ \oplus \\ (((x - b \ - j)\#mask\langle(j + b - 1)...j\rangle)\#(j@0)) \quad (8.10)$$

where x is the instruction width, i/j is the starting position of input/output register selection filed, a/b is the register selection field width, v@0 stands for repeating zero (0) v times and stands for concatenate.

8.2 Program Counter Fault Model

The Program counter (PC) fault model covers errors which propagate to the program counter of the CPU

Where

The processor's Program Counter register.

When

Corruption can occur at an instruction boundary.

What

There are three scenarios on how the structure's value can be corrupted:

1. Missed Load : All or part of a register is not loaded when it should be.

$$PC \leftarrow expr \Rightarrow PC \leftarrow (expr \oplus (mask\langle(w-1)...0\rangle)) \tag{8.11}$$

2. Level Change: in storage The value of one or more bits in the register is complemented.

$$PC \Rightarrow PC \oplus (mask\langle(w-1)...0\rangle) \tag{8.12}$$

3. All 0/1 :Assign the value of all zeros and all ones to the register

$$PC \Rightarrow PC \oplus PC \tag{8.13}$$
$$PC \Rightarrow PC \oplus \bar{PC} \tag{8.14}$$

8.3 Control Unit/Instruction Decode Fault Model

The Control Unit/Instruction Decode fault model covers corruptions of similar type to the read/write register fault model and mainly refers to corruption of the opcode field.

Where

Any location where an instruction may reside (i.e. memory, instruction register).

When

Corruption can manifest at an instruction boundary or on a memory reference.

What

The corruption that will result in a fetch/decode error is described in equation :

$$instr_fetch(addr) \Rightarrow instr_fetch(addr) \oplus (((x-c-k)@0)\#mask\langle(k+c-1)...k\rangle)\#(k@0)) \tag{8.15}$$

Where k is the starting position of the operation code field and c is the width of the operation code. The values of k, c are not constant and depend of the given instruction.

8.4 ALU Fault Model

The ALU fault model covers corruptions in the ALU module of the processor, based on a instruction formats such as :

$$D \leftarrow S_1 op S_2 \tag{8.16}$$

$$D \leftarrow mem_read(Addr) \tag{8.17}$$

$$mew_write(Addr, Data) \tag{8.18}$$

$$S_1, S_2, (Label) \tag{8.19}$$

Where 8.16 is a general format for arithmetic instructions such as *add* $\$R_d$ $\$R_{s1}$ $\$R_{s_2}$, 8.17/8.18 demonstrate general formats for memory reads/write and 8.19 demo state general branch instructions.

Where

At the Execution state of the processor.

When

Corruption can manifest at an instruction boundary.

What

Depending on the executed instruction the fault manifest in a different way.

1. Arithmetic instruction:

   - Part of the operation is not completed:

$$D \leftarrow expr \Rightarrow D \leftarrow (expr \oplus (mask\langle (w-1)...0 \rangle)) \tag{8.20}$$

   - Level change in storage. The value of the result (D) is $\oplus$ with a mask value and stored to the result (often referred as the bit flip model)

$$D \Rightarrow D \oplus (mask\langle (w-1)...0 \rangle) \tag{8.21}$$

   - All 0/1 (6.4) Assign all bits of the result to 0 or to 1

$$D \Rightarrow D \oplus D \tag{8.22}$$
$$D \Rightarrow D \oplus \bar{D} \tag{8.23}$$

2. Write/Load instruction : The fault propagates to the calculated address of the read/write:
   - A part of the source/destination address is not computed.

$$Addr \leftarrow expr \Rightarrow Addr \leftarrow (expr \oplus (mask\langle (w-1)...0\rangle)) \tag{8.24}$$

   - The value of the address is $\oplus$ with a mask value and stored to the address (often referred as the bit flip model)

$$Addr \Rightarrow Addr \oplus (mask\langle (w-1)...0\rangle) \tag{8.25}$$

   - All 0/1 Assign all bits of the source/destination address to 0 or to 1

$$Addr \Rightarrow Addr \oplus Addr \tag{8.26}$$
$$Addr \Rightarrow Addr \oplus A\bar{d}dr \tag{8.27}$$

3. Branch instructions. In this case the condition of the branch is flipped.

9 Our Implementation.

In the introduction we gave a brief overview of the problem that arise with new technologies and how the impact the reliability of the system. Furthermore we discussed the fault types and some fault models and how the faults propagate to outer abstraction levels. Creating realistic error models and characterizing errors will enable us to design solutions in order to preserve the current levels of system's robustness.

In order to asses the impact of faults to each abstraction level new tools had to be reinvented or existing tools had to be extended. Such a case is Gem5 a full system ,cycle accurate simulator with broad usage across the world for performance analysis that we presented in section 6.3. To avoid recreating a simulator we extended Gem5 with fault injection capabilities following the General Processor fault model described in chapter 7. The result is a configurable framework for studying the effect of faults in a processor.

The framework was developed mainly using C++ and in some places Python. The configuration interfaces is exported to the user by employing the SWIG library. So a uniform model is created for configuring and the fault injection framework and the rest of the simulator capabilities. It must be mentioned that the framework only supports the ALPHA isa in full system mode although an older implementation exists which supports bots SE and FS mode. The current fault injection framework targets multi-threaded applications runned on realistic circumstances (full OS support), since SE does not fully support multi-threaded applications because of the luck of a thread local storage implementation (TLS) and a scheduler we implemented the current framework only for FS. Porting the code of the framework for other ISA's does not require extensive modification as the ISA depended portions of the framework is used for disguising the processes/threads , the user/privileged mode execution and the parser of the Supported ISA for fault injection during execution stage.

In the following sectors we will describe the fault injection framework and present implementation details.

9.1 Introduction on the idea of WWW(W)

Fault injection attributes can be divided into 3 basic categories [17].

- **Where:** *The location of the injected fault.*
- **When :** *The time of the fault manifestation.*
- **What :** *The nature of the fault, as well as its effects on the fault structure.*

Besides these three categories we created one more category in order to gain more controllability over the fault injection framework and be able to target specific portions of an application with different kind of faults. This attribute is the software based fault injection of our hybrid model. We tried to minimize the extend of this part of the model so that the applications under investigation will have minimized changes.

- **Which :** *Which Thread/application is going to manifest the injected fault.*

9.1.1 Where

A very important aspect that we need to clarify when creating a fault injection scenario is the location of the fault e.g which modules are going to be targeted. A good method is a top down approach. We begin by selecting the high level unit, in other words specifying which core, then we proceed with internal modules and in the end select a specific bit which is going to be modified. Common locations are general purpose registers, pipeline stages (fetch, decode , execute).

The location of the fault s of crucial importance since it "bounds" the possible errors that can propagate throughout the system. An error in a register (integer,floating) will affect the internal storage of the processor and in the case of hitting a special purpose register the fault may alter the state of the processor. On the other hand an error in a memory location will affect the internal storage of the system and may affect the execution and the output of the process that uses this content.

9.1.2 When

Another important aspect of the fault injection infrastructure is the timing of the injection thus the time when the fault will be manifested. Faults can be set to occur on the value of a system variable, such as a specific address (PC address) or can be scheduled to a) a more software approach like :

1. the number of executed instruction of a specific tread/application
2. the number of executed instructions of a specific core
3. the cycles that has thread/application executed on a specific core
4. the total instructions/cycles executed by all cores since the start of the framework

The timing of the fault is closely related with the values of "Which","Where". More information are going to be given later on the document.

9.1.3 What

These fault injection instrumentation variable will describe the nature of the fault; more specifically how the structure's value will be corrupted. As faults manifest in different ways , based on their cause, we need to use different models for each type of hardware fault. The most common way to model permanent faults is the stuck-at model where a signal is permanently set (stuck) to one (1) or zero (0) . Transient faults are modeled using the bit-flip model where a bit's value is flipped to its complement .

9.1.4 Which

A software aspect of our fault injection framework. Which threads/applications is going to be injected with a fault. Each thread/application which is going to be injected with a fault executes an instruction and is inserted into a group of faults. This is achieved by extending the ALPHA ISA with one more instruction. This instruction is executed by the application/thread.

Moreover the value of the attribute which describes which instructions/ticks are going to be taken under consideration for defining the behavior of "When"

9.2 Implementation

Having explained the aspects of fault injection instrumentation, we will now present the general form of the implementation details. For the development of our tool we adhered to the object oriented approach of the Gem5 simulator. The framework is composed of classes that define different fault types and fault queues where the objects are stored for quick access and easy manipulation. Moreover due to the fact that faults are scheduled for a specific thread/application or a group of threads and applications we needed a class that stores information for all threads/applications that have enabled fault injection called ThreadEnabledFault. This class, with the aid of the cpuExecutedTicks class (9-1), keeps track of all the needed information for each core and each thread that has enabled fault injection. It is worth mentioning that the alpha architecture cannot distinguish between thread and applications and treats them with the same way.

The hierarchy of faults types that our framework currently supports is depicted in Figure 9-2 . All faults objects derive from the InjectedFault class which contains the basic variables for fault injection as well as the generic attributes of a fault. A core component of fault injection framework is the queue structure. All faults described at the input file are inserted in 4 queues, based on their attributes; execution, fetch ,decode and other. The queue class provides public functions for inserting, removing

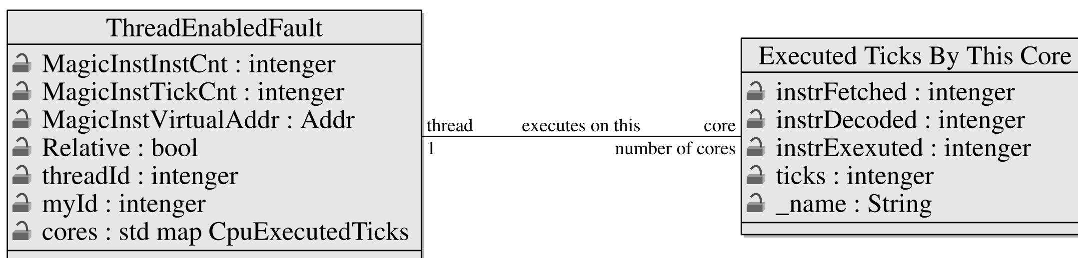


Figure 9-1: ThreadEnabledFault - cpuExecutedTicks

and searching for a fault. In order to improve performance the faults are kept in a descending order thereby decreasing the average search time.

Finally all the previously mentioned information are encapsulated in a wrapper class called Fi_System. The Fi_System class derives from the SimObject/MemObject class. Fi_system is initialized at the configuration file and is responsible for initializing all information necessary for the framework after reading the fault injection input file. The initialization of the framework is done on the beginning of the simulation and after restoring from a checkpoint. More information about the checkpoint and the input file are going to be given later in the document. A general hierarchy scheme of our framework is presented in 9-3

To enable fault injection in specific threads a new instruction was added into the Alpha ISA called ***fi_activate_inst(int id)***.The usage of the instruction is to enable/disable the manifestation of faults for each thread. For this to be achieved we had to exploit the Process Control Block (PCB) address which is unique to each application/thread. When a thread executes ***fi_activate_inst(int id)*** for first time the instruction enables fault injection for this thread by creating an object ThreadEnabledFault and attaches it to the last index of the array. The index is stored using a hash table which uses as a key the PCB address. Quick access is provided to the thread information by refraining from searching an entire array each time. When the simulator is running on each cycle the PCB address is read and the hash table returns index corresponding to this thread if -1 is returned then the current thread has not activated the framework and so the the fault injection code is not executed. Moreover with the help of an already implemented function ,the inUserMode() function which returns true if the processor is currently in user mode, the framework is restricted to be used only in user mode code.

Another feature of our framework is "Relative Fault Injection", that is, faults can be set to manifest relatively to the value of a processor variable. To enable relative fault injection an extra instruction has been added to the ISA, namely ***get_Pc_address()***,. The ***get_Pc_address()*** instruction is used to set a relative point for a fault injection. When executed by the processor it updates the information of this thread by storing the new information to the correct index of the array.

Finally a third instruction is created ***init_fi_system()*** which initializes our framework. When the instruction is executed all stored information of our framework is reset, the input file which contains a description of the fault is read from the beginning. This helps for injecting different faults in different places and if the instruction

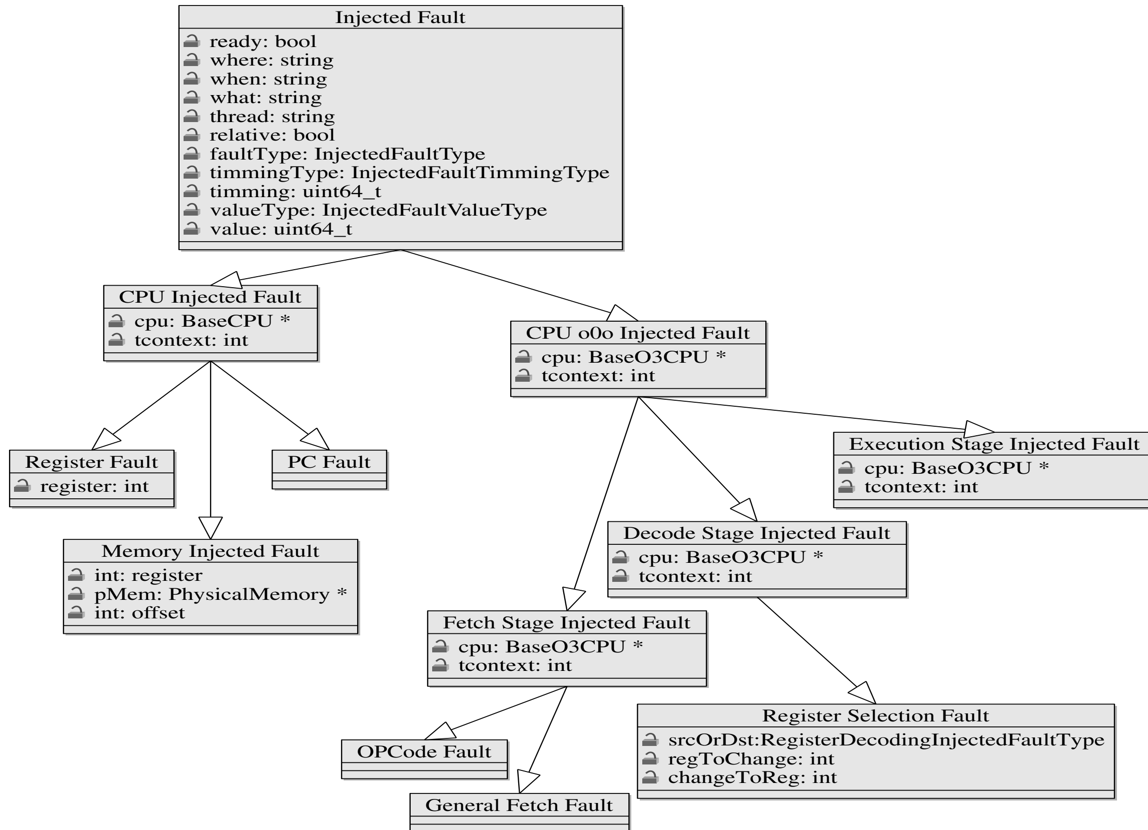


Figure 9-2: Fault Classes

is coupled with the right configuration options a checkpoint is created right before the initialization of the framework.

9.2.1 Where

Our implementation supports fault injection on registers, fetch, decode, execute stages on each core separately. All supported places of our implementation are shown 9.1. All faults in the implementation are described in an a deterministic way, however statistical injection is easily implemented with the usage of python scripts and the configuration file.

Memory faults are relative to the value of the register. To be more exact the stored value of a register is read and the offset is added afterwards the value is transformed from a Virtual Address to an Physical Address.Finally the block which is referenced by this Physical address is injected with a fault

9.2.2 When

The implementation provides three different options for the timing of the manifestation simulation ticks, fetched instructions or the value of the PC in a CPU. The value of the manifestation can be absolute (i.e. distance from the first appearance

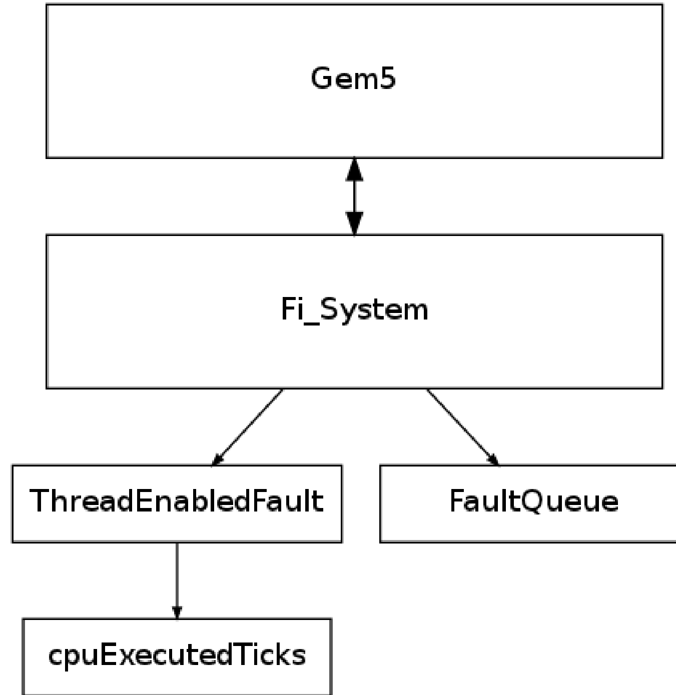


Figure 9-3: General View of Framework

of the instruction ***fi_activate_inst(int id)*** or relative to a simulation milestone which is different for each thread. The milestone is achieved with a new instruction ***get_Pc_addr()***. When a thread executes this instruction all the counters of the thread are set to zero and the PC address of the current instruction is stored. The when attribute depends on the values of which and where, more information on how instructions are counted is explained on table 9.3.

9.2.3 What

In our implementation a module's/structure's value can be corrupted in a variety of ways. The supported methods of affecting the value of the structure that is injected are:

- **Immediate Value Assign:** the provided constant to the structure. XOR: XOR the current value with the given constant.
- **Bit-Flip:** Change the specified bit to its complementary value.
- **All0:** Set all bits to zero (0).
- **All1:** Set all bits to one (1).

Taking into account the possible changes and enhancements of fault models in the future, our implementation for the corruption of the targeted structures was design to be as modular as possible.

Faults	Status
Register File	√
Program Counter	√
Memory Unit	√
Control Unit	√
Fetch & Decode Logic Block	×
Internal Data Bus, Internal Address Bus, and Internal Control Bus, as well as the External Bus Interface	×
ALU	√
Memory-Mapped Peripheral Devices	×

Table 9.1: Modules where fault injection is supported

Fault Type \ Timing Methods	Inst	Tick	Addr	relative
Memory	√	√	√	√
PC	√	√	√	√
Register	√	√	√	√
Fetch	√	√	√	√
Operation Code	√	√	√	√
Register Decoding	√	√	√	√
Execution	√	√	√	√

Table 9.2: Trigger Mechanisms for each Fault Type

9.2.4 Which

The final attribute of our implementation is ”which” and may be the most difficult to comprehend. Each fault described in the fault injection input file has an id. The id is **not** unique for each fault so there is the option to create group of faults (faults with the same id). When a thread executes the ***fi_activate_inst(int id)*** instruction the id is stored, again one or more threads may execute the instruction with the same id (group of threads with the same id). In this way we achieve to destine a group of faults to a group of threads.

The which attribute gives the ability to test many software relaxed reliability schemes. To be more precise faults may be destined to hit a thread which executes insignificant code compared to other threads and errors may not affect the output of the application.

9.2.5 Checkpointing

As referred to a previous section simulations offer many advantages. One of those was the ability to retrieve a snapshot of the state of the system stored to the hard drive and retrieve from that point on another time. Although Gem5 already provided

Where Option \ Which Option	"ID"	"ALL"
ID	Only the cycles spent in this core by the thread are taken into consideration in order to count if the fault must manifest. This applies for the instructions.	All cycles that this core has spent executing any thread that has enabled fault injection are taken into consideration in order to count if the fault must manifest. This applies for the instructions
ALL	All cycles that this thread has spent in any core are taken into consideration in order to count if the fault must manifest, the same applies for the instructions	All cycles spend in any core for any thread that has enabled fault injection are taken into consideration in order to count if the fault must manifest. This applies for the instructions

Table 9.3: Trigger Mechanisms for each Fault Type depending on which-where

checkpointing under some limitations which did not suited our purposes. Checkpoint on full system detailed simulation was achieved with two ways. For the first method CPUS were switched from detailed to atomic mode create the checkpoint and afterwards switching again from atomic to detailed mode in order to continue the simulation. For this to be done the pipeline stages on the detailed mode where flushed prior taking the checkpoint and thus there could be a potential accuracy loss in our fault injection framework. The other method was achieved by simulating MOESI_hammer cache coherency protocol. This method did not switched between detailed and atomic modes however the simulation time increased dramatically and after the checkpoint was created the system exists and must be restarted from the checkpoint in order to continue the simulation.

Due to the previous limitations we had to turn to a Linux based checkpoint package which checkpointed the simulator's state from an outer scope. After carefully studying many available Linux based checkpointing packages we concluded to the DMTCP checkpointing package for various reasons. DMTCP is distributed under the terms of Lesser GNU Public License (LGPL) and supports checkpointing the state of multiple of multiple simultaneous applications, including multi-threaded and distributed applications.Among the applications supported by DMTCP are OpenMPI, MATLAB, Python, Perl, and many programming languages and shell scripting languages.

The main reason for choosing DMTCP was based on the ability to take checkpoints not only inside of the simulator by calling functions given by the API of DMTCP but also outside of the simulator. The ability to invoke DMTCP inside of the sim-

ulator gave us the opportunity to keep the already existed checkpointing front-end of the Gem5 simulator (special instruction added on ISA so that applications may call a checkpoint internally) and alter the back-end of the checkpointing method (use the DMTCP API to checkpoint instead). When an application internally created a checkpoint the simulator exited and when the simulator resumes the fault injection framework is reset. By doing this we achieve to have a stable point (checkpoint) prior the portion of code that is going to be injected with faults and every time we restore from that checkpoint we test another faulting scenario since our framework after the restoration of that checkpoint resets all counters and reads the input file. By changing the contents of the input file before restoring the checkpoint we test different faulting scenarios without re executing the simulator. More information are going to be given in the next chapter.

Besides the internal checkpointing DMTCP offers the ability to take checkpoints periodically. After a specified by the user time has passed a checkpoint is created. In cases of unexpected events, such as power failures, a recent state of the experiment is kept in order to restore after the failure is fixed. Moreover a checkpoint can be created manually outside of the simulator by sending a special message to the DMTCP coordinator.

9.3 Usage-Exporting Into configuration file

Our framework may be consideration as another simulation option and thus can be described in the configuration file prior execution of the simulation. As an option to the configuration file the path to an input file is given. This file describes all the faults that are going to be injected in our simulation. Each line of the file represents a specific fault and defined the attributes of the fault.

- **When :** When to inject the fault (format: ¡timingType:timingValue¿).
- **What :** What value should be injected and how (format: ¡valueType:value¿)
- **Which :** Which thread or threads will execute the fault (format: ¡ID¿)
- **Where :** In which CPU to inject the fault (format: ¡module's name at the configuration script¿)

and the optional fields are:

- **Occurrence :** How many times should the fault manifest? By default one (1) - transient fault.
- **tcontext :** Which Hardware thread is going to execute the fault (Currently not implemented always 0).

9.3.1 Register Injected Fault Configuration

The additional required fields for a register fault are:

RegType: what type of register should be fault-injected (format: "int", "float", "misc").

RegNumber: which register should be injected.

Thus a fault which is going to be injected into a register is described in the input file as a line with 10 fields:

$$RegisterInjectedFault\ When\ What \\ Which\ Where\ Occ \\ tcontext\ RegType\ RegNumber"endofline" \tag{9.1}$$

The examples are related on a system with 4 cores which executes an abstract application (9-4) with 4 threads. Each thread and the main thread of the application enable fault injection. All threads have an id. Thread1 and Thread3 share the same id (1) so each fault with which = 1 will be manifested either from thread 1 or thread3.

1. Inject a permanent fault at the first (1) integer register of the first cpu "system.cpu1" when the PC is 8 + (PC @ magic instruction). After the fault the register should contain the value 57005. Since the fault is permanent we want all threads that have enabled the fault injection framework to be able to manifest the fault so which = "ALL".

```
"RegisterInjectedFault Addr:8 Immd:57005
                        ALL system.cpu1  1 0 int 1"
```

2. Inject a transient fault at the second thread (thread2) at the first (1) floating point register of the cpu2 "system.cpu2" when the total cycles of the second thread executed on the cpu2 are 50000. After the fault the register should contain the result of the XOR product of 57005 and the initial value.

```
"RegisterInjectedFault Tick:50000  Mask:57005
                        2 system.cpu2  1 0 float 1"
```

3. Inject a transient fault on any core at the second thread (thread2) at the first (1) floating point register when the total sum of cycles on any core of the second thread are 50000. After the fault the register should contain the result of the XOR product of 57005 and the initial value.

```
"RegisterInjectedFault Tick:50000 Mask:57005
                      2 ALL  1 0 float 1"
```

4. Inject an intermittent fault ($occ = 3$) at the first (1) miscellaneous register of CPU4 "system.cpu4" when the total fetched instructions on core 4 of thread 4 are 1984 + (fetched instructions @ magic instruction). After the fault the register should contain the value 0.

```
"RegisterInjectedFault Inst:1984 Immd:0
                      5 system.cpu4 3 0 misc 1"
```

PC Injected Fault Configuration

PC faults do not require any additional field so a line in the input file describing such a fault has 8 fields

$$PCInjectedFault\ When\ What \\ Which\ Where\ Occ \\ tcontext"endofline" \tag{9.2}$$

Examples:

1. Inject a fault into the main thread (id = 0) at the PC register of CPU2 "system.cpu2" when the PC of the CPU becomes 4831838348. After the fault the register should contain the result of the XOR of 2 and the register's previous value.

```
"PCInjectedFault Addr:4831838348
      Mask:2 0 system.cpu2 1 0"
```

Memory Injected Fault Configuration

Memory faults require 2 additional fields

RegNumber: From which register will the framework read the Virtual Address.

Offset: The offset from the previously read address

Examples:

1. Inject a fault to any thread at virtual address $readRegister(10)+512$ of memory module "system.physmem" when the amount of simulation ticks of cpu1 is 2000. After the fault the address should contain the value 3.

```
"MemoryInjectedFault Tick:2000 Immd:3 ,
                    ALL system.cpu1 1 0 512 10"
```

It is wise to use registers which we know that contain addresses to the memory (global pointer,stack pointer etc..)

Fetch Stage Injected Fault Configuration

Fetch stage injected faults can either be “general”, targeting the whole bit width of a fetched instruction, or affecting just the Opcode.
Examples:

1. Inject a fault at the fetched instruction of any cpu“ALL” when the first or the third thread (id = 1) have been executed for 45.000 ticks. After the fault the instruction should be 540999681 — hex(203F0001).

   ```
   "GeneralFetchInjectedFault Tick:45000 Immd:540999681
                             1 ALL 1 0"
   ```

2. Inject a fault at the main thread at the Opcode of the fourth (4) fetched instruction of CPU1 “system.cpu1”. After the fault the instruction’s Opcode should be 32 — hex(20).

   ```
   "OpCodeInjectedFault Inst:4 Immd:32
                     0 system.cpu1 1 0"
   ```

Decode Stage Injected Fault Configuration

Decode stage faults are targeted at the decoding of source and destination registers and require an additional field:

regDec: format: whether it should inject the ¡destination or source registers¿:¡which register to change¿:¡in which register to change¿.

$$RegisterDecodingInjectedFault\ When\ What \\ Which\ Where\ Occ \\ tcontextregDec\text{"}endofline\text{"} \tag{9.3}$$

Example:

1. Change the destination register zero (0), of the instruction that will be decoded by any thread on any core after 45.000 ticks have passed since the first occurrence of ***fi_activate_inst(int id)***, to one (1).

   ```
   "RegisterDecodingInjectedFault Tick:45000 Immd:0
                                 ALL ALL 1 0 Dst:0:1"
   ```

Execution Stage Injected Fault Configuration

Execution stage faults do not require any additional configuration field and target the output of the ALU, if any.
Example:

1. Inject a fault at the execution result of the 10th instruction of the thread2 on “system.cpu3”, the result should be 10.

```
"IEWStageInjectedFault Inst:10 Immd:10
                        2 system.cpu3 1 0"
```

   Note that if the 10th executed instruction is a branch the condition will flip despite the value of $What$

Figure 9-4: Application with 4 threads, each thread activates the fault injection framework

10 Setup of the Workstation Optimization

As mentioned on previous chapters Simulation based fault injection has a main disadvantage ,the huge time consumption. In order to overcome this obstacle we runned many simulations simultaneously on a laboratory .

10.1 Setup of The Laboratory

Our laboratory is consisted of 30 PC. Each PC has an Intel processor (Xeon CPU E5606) with four cores clocked at 2.13 Ghz and 6Gb of memory (RAM). All PC's have access to a network File system and a local file system(10-1). Each PC executes simultaneously 4 experiments one for each core. So in the best scenario totally 120 experiments are executed simultaneously. Bash script language and other linux tools are used to control the experiments. To be more exact one script called mainscript.sh (Figure 10-2)is responsible for initializing the file system of the computer. Copy scripts ,executable, image disks from the NFS to the Local FS. After everything is copied and ready mainscript.sh launches 4 child's which execute a script run.sh (Figure 10-3).

This script is responsible for all experiments runned by this core on this PC. To be more exact after the script is launched it checks if the maincheckpoint exists if not it launches a simulation from the start. The maincheckpoint is a checkpoint which holds the system state just before fault injection is begun. After that is checks if a previously experiment has begun but not finished (someone closed the PC) if this is the case the experiment is finished by launching a simulation from the latest checkpoint. Finally at this point the script knows that the maincheckpoint exists and that no unfinished experiment exists so it checks a file stored in the NFS called experiments.txt. Each line of this file represents an experiment. If the file is not empty the script locks the file (flock) reads the last line of the file store it to a file in the Local FS and delete the line from the experiments.txt after that it unlocks the file (rm -f lock). After all these the script restores the simulator from maincheckpoint and waits until the experiment is finished. When this is done the script copies the results to a folder in the NFS.

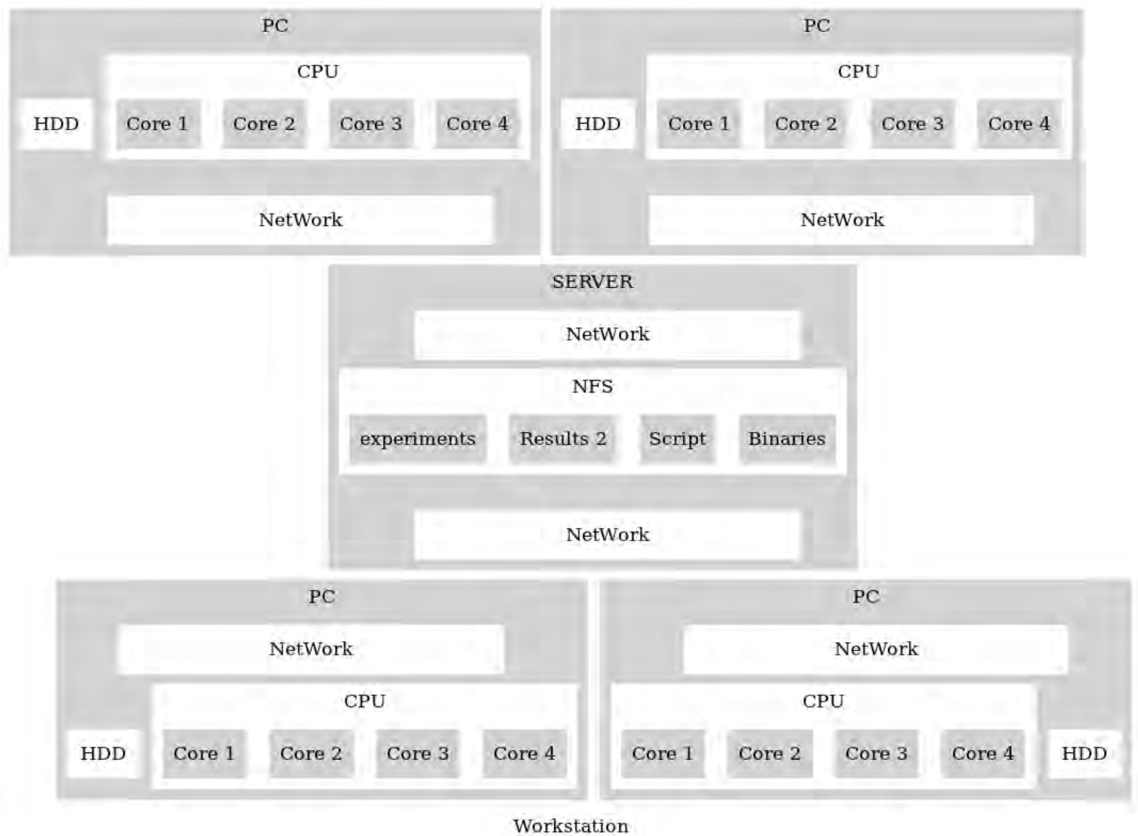


Figure 10-1: Topology of the Laboratory

It is worth mentioning that a third script is runned on the background which checks periodically if an experiment is taking to long (stuck in an infinite loop), if the output of the experiments exceeds some limits or if another user is logged in the machine; on both first cases the experiment is killed a message is printed on the output. On the last case a checkpoint is created for each experiment and after the experiments are killed in order to respect the other user.

Besides the scripts the following linux tools where used to help the workstation setup:

- WON (Wake On Lan) : Remotely boot each PC by sending a special Packet to the Network card.
- SSH (Secure Shell) : Remotely start the experiments (start mainscript.sh to each PC)
- SFTP (Secure file Transfer) : Used for putting the experiments on the NFS and for getting the results after they are finished.

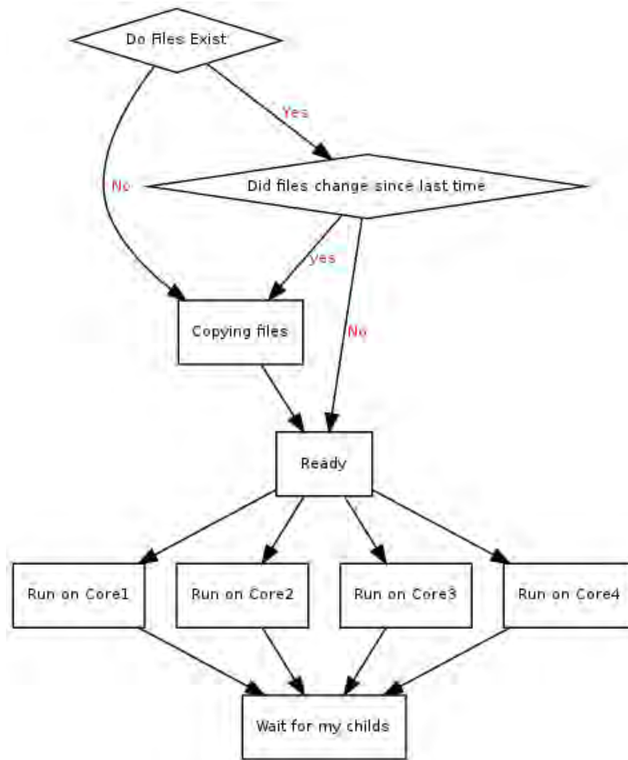


Figure 10-2: mainscript.sh responsible for initializing each PC

10.2 Optimization

Apart of running experiments in parallel we tried to reduce the duration of each experiment individually. This was achieved by two mayor Optimization (Figure 10-4).

Main Checkpoint : Create a checkpoint just before the fault injection framework starts. After the checkpoint is created for each new experiment we do not have to reach that same state (boot, Read Inputs , etc) but we only have to restore from the previous mentioned checkpoint. Since our framework every time an internal checkpoint is created it resets everything and reads again the input file new experiments can be executed.

Switch CPU: Gem5 has developed a special instruction which allows to switch between CPU models. When the instruction is executed the simulator pauses and a Python script initializes the next CPU model and then restored the execution with the new CPU model. In some cases except the of the CPU model the memory model is also switched (atomic - simple). In our experiments after our framework is stopped (usually after computation are finished and the applications results are going to be printed) we switch from detailed mode to atomic. By doing this all computation which do not concern our experiments are been done faster (atomic offers offers less accuracy so less time).

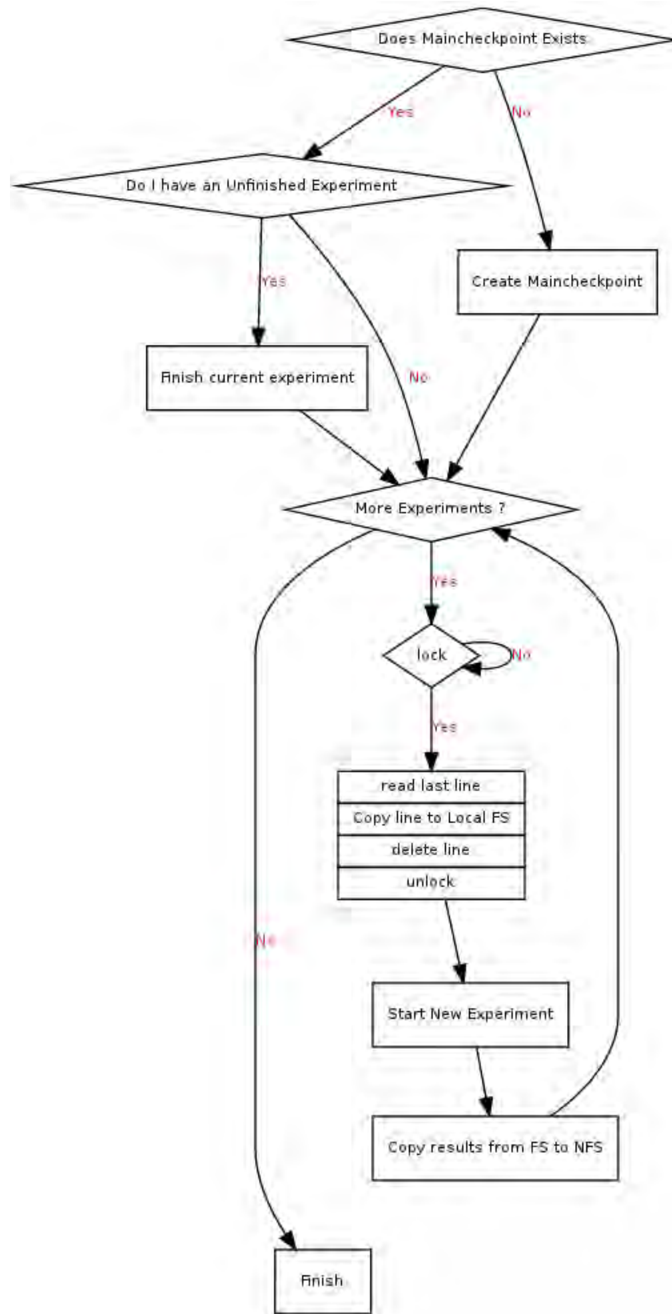
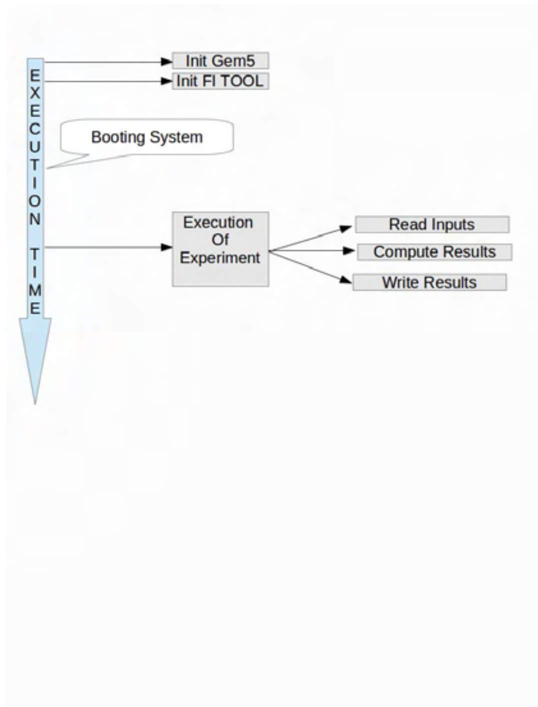
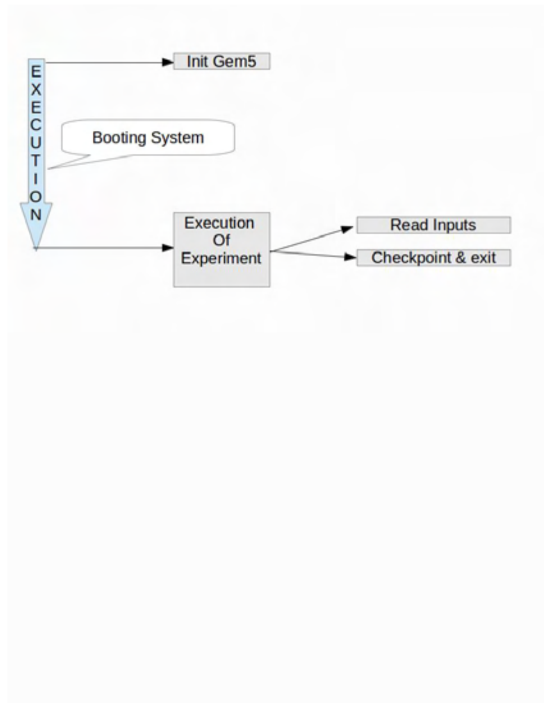


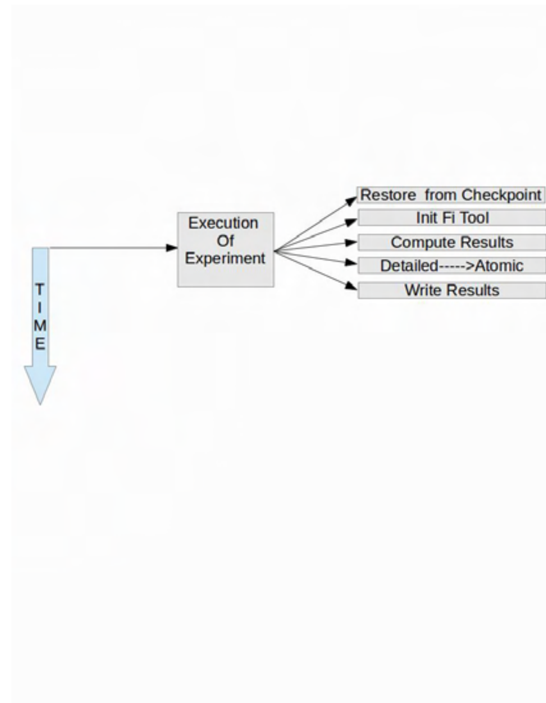
Figure 10-3: run.sh responsible for running experiments in each core



(a)



(b)



(c)

Figure 10-4: (a) Initially all experiments followed this procedure.(b) Only first experiment boots machine and reads input,(c) all remaining experiments execute from the saved checkpoint

11 Fault Injection Campaigns Experiments

After setting up of the workstation this Thesis conducted a series of fault injection campaigns in order to test the workstations functionality and the functionality of our framework. We targeted the broad used benchmark suite PARSEC which offers high parallel applications.

11.1 Experimental Methodology

In our experiments we conducted 4 series of experiments on each application (11.1). All applications have been tested using statistical fault injection in order to evaluate their behavior under unreliable circumstances. Due to the volume of the experimental results additional python and bash scripts were implemented in order to process the results.

Compilation \ Execution	1 Thread	4 Threads
No Optimization	A set of experiments for each applications is executed with no Optimization runned with 1 thread	A set of experiments for each applications is executed with no Optimization runned with 4 thread
Full Optimization	A set of experiments for each applications is executed with full Optimization runned with 1 thread	A set of experiments for each applications is executed with full Optimization runned with 4 threads

Table 11.1: Experimental test Series

In order to obtain the instrumentation variables for statistical fault injection campaigns we assumed that faults in all structures and bits are equally possible. In addition we assumed that time occurrences in time follow a uniform distribution. Finally each simulation was injected with a single fault.

Compilation of the parsec benchmark for the alpha ISA was achieved by following the instructions from [16]. To be more exact all programs were compiled using a cross

compiler, version 4.3.2 (gcc/g++). The first campaign of experiments was with no Optimization while the second campaign was with full Optimization (given by parsec Makefiles/ Tutorial).

Finally each we executed 4 different campaigns of fault injection for each Benchmark. Each fault campaign was composed of 2450 experiments runned in detailed mode. Each experiment contained a single fault injection and hitted all the threads. The values of the instrumentation variables were provided by a uniform distribution function (Python implementation of Mersenne Twister).

11.2 Quick Parsec Overview

Parsec is an open-source parallel benchmark suite of emerging applications for evaluating multi core and multi processor systems[8, 7]. It covers a broad range of application domains such as financial , computer vision, physical modeling, future media, content based search, deduplication. Table 11.2 list all benchmarks and their characteristics.

Program	Application Domain	Parallel Model	Working Set	Communication
blackscholes	Financial Analysis	data parallel	small	low
bodytrack	Computer Vision	data parallel	medium	medium
canneal	Engineering	Unstructured	unbounded	high
dedub	Enterprise Storage	Pipeline	unbounded	high
facesim	Animation	data parallel	large	medium
ferret	Similarity Search	pipeline	unbounded	high
fluidanimate	Animation	data parallel	large	low
freqmine	Data Mining	data parallel	unbounded	high
raytrace	Rendering	data parallel	unbounded	medium
streamcluster	Data Mining	data parallel	medium	low
swaptions	Financial Analysis	data-Parallel	medium	low
vips	Media Processing	data parallel	medium	low
X264	Media Processing	pipeline	medium	high

Table 11.2: Parsec Benchmarks

11.3 Result demonstration

For each campaign of experiments a table is going to be displayed demonstrating the results while they will be visualized in stacked percentage stats. A quick description of the row categories:

No difference The fault manifested but did not create any user visible error.

Crashed The fault created an error which leaded the application to abort (segmentation fault, Divide by zero etc.)

Detected A fault was manifested which create a user visible error and the application handle it. (Only possible if the application has error detection mechanism)

Not Detected The fault created a user-visible error

Not Manifested The error did not manifest either because the fault injection framework was disabled or the CPU was not in User Mode

11.4 Fault Injection In Blackscholes

This section presents the results from our first fault injection campaign. We chose blackscholes as the first application under test since it is the simplest of all parsec workloads and initially was used for debugging and testing our framework.

The blackscholes application solves the Black–Scholes formula (11.1) for a set of data (2-D array a line represents a different set of inputs) and stores the result on an other array. Finally the results are printed on the output.

$$C(S,t) = N(d_1)\ S - N(d_2)\ Ke^{-r(T-t)}, \tag{11.1}$$

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \tag{11.2}$$

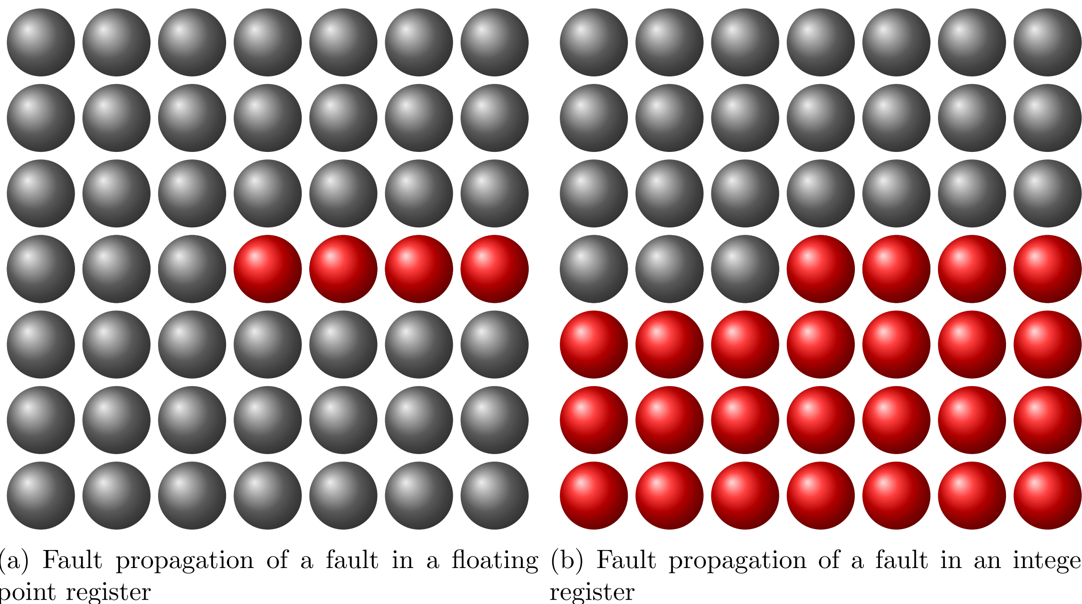
$$\begin{aligned} d_2 &= \frac{\ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \\ &= d_1 - \sigma\sqrt{T-t} \end{aligned} \tag{11.3}$$

$$\begin{aligned} P(S,t) &= Ke^{-r(T-t)} - S + C(S,t) \\ &= N(-d_2)\ Ke^{-r(T-t)} - N(-d_1)\ S \end{aligned} \tag{11.4}$$

- N(.) is the cumulative distribution function of the standard normal distribution
- T-t is the time to maturity
- S is the spot price of the underlying asset
- K is the strike price

- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset

Almost all computation are floating point and each solution is arithmetically independent from the previous ones. Having that in mind we suppose that fault injection in registers which hold data (floating point) will only affect the result of one set of inputs and will not propagate through the entire computation as long as the fault did not propagated to an exception. On the other hand we do not expect such a behavior from an injection which affects other software-related attributes. For example injecting errors to integers may affect the computation of other solutions, To be more specific hitting a register which holds the virtual address of the stored input data may shift the entire computations by some bytes which will alter all the results after the manifestation of the fault.



(a) Fault propagation of a fault in a floating point register (b) Fault propagation of a fault in an integer register

Figure 11-1: Last column represents the solution of one equation,the remaining columns represent the stages until the computation is finished.The red node is a when an error causes a user visible fault.The gray are correct computations

11.4.1 Unoptimized binary code

1 thread

Table 11.3 demonstrates the results of the first fault campaign while figure 11-2 visualizes the results in stacked percentage chart.

From table 11.3 we can observe that almost 87% percentage of faults where masked so they did not produce any visible error to the user space. The rest 10% (3% did

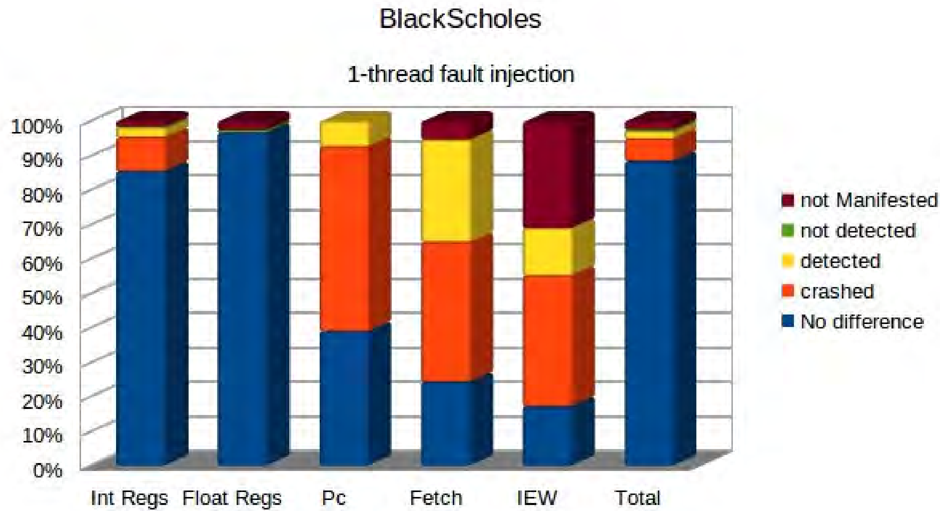


Figure 11-2: Blackscholes: Fault Injection results 1 thread Unoptimized

Fault Place \ Corruption	No difference		crashed		detected		not detected		not Manifested	
	#	%	#	%	#	%	#	%	#	%
Integer Register	993	86%	114	10%	31	2%	3	0%	20	2%
Floating Register	1144	97%	0	0%	0	0%	9	1%	30	2%
PC address	16	39%	22	54%	3	7%	0	0%	0	0%
Fetch	9	24%	15	41%	11	30%	0	0%	2	5%
IEW	5	17%	11	38%	4	13%	0	0%	9	31%
Total	2167	87%	162	8%	49	2%	12	0%	61	3%

Table 11.3: Blackscholes: Fault Injection results 1 thread Unoptimized

not manifest) are splitted into 3 main categories. The first one are those that crashed the program (8%). The second category are the faults where detected by the error detection mechanism of blackscholes and constitute the 2% of faults.The third and last category are the faults which produced an error to the output however the error detection mechanism did not detect those errors consist the remaining errors although on a insignificance percentage.

The results create the impression that the blackscholes when executed unoptimized is in general fault tolerant considering the simplicity of the error detection mechanism and the absence of other reliability mechanism. However if we separate the results based on a the corruption structure we can observe that each structure demonstrates a great variance.

We observe that the floating point register file is the most fault-tolerant of all structures. Less than 1% lead to an undetected error and none fault in the floating register file crashed the application. This is a result of fault masking through register rewriting, unused registers, register liveness. Moreover this low percentage of errors in the floating register file is a result of the applications behavior. Since the computation of each result is individual from the computation of other results a fault in a floating

register is difficult to propagate to other results. Finally floating point exceptions in tha ALPHA architecture are minimal so we did not expect to have crashed from this injection although it is possible (divide by zero,etc).

Furthermore the next most fault tolerant structure is the integer register file. Since almost 12% of injected faults resulted to an error visible to the user space. Although the same fault masking mechanism of the floating point register file apply to the integer file the increase of the visible errors is explained by the global usage of some registers (global pointer, stack pointer frame pointer). Moreover some registers are used as pointers to the data of the applications and their usage is wide during the execution of the benchmark, this is the main reason for many crashes..

On the other hand the rest of the injected structures led by the Fetch faults with 71% or user visible errors and followed by the PC address and the IEW faults with 61% and 52% visible errors respectively. These structures are characterized as less fault tolerant and the possibility for a crash is also high because they are used on each cycle.

4 thread

Table 11.4 demonstrates the results of the first fault campaign while figure 11-3 visualizes the results in stacked percentage chart.

Corruption / Fault Place	No difference		crashed		detected		not detected		not Manifested	
	#	%	#	%	#	%	#	%	#	%
Integer Register	975	86%	81	8%	26	2%	3	0%	50	4%
Floating Register	1148	97%	0	0%	0	0%	7	0%	39	3%
PC address	4	10%	28	68%	3	8%	2	4%	4	10%
Fetch	2	5%	23	63%	10	27%	0	0%	2	5%
IEW	16	36%	15	35%	5	11%	0	0%	8	18%
Total	2145	88%	147	6%	44	2%	12	0%	103	4%

Table 11.4: Blackscholes: Fault Injection results 4 thread Unoptimized

On this case the results are quit same in comparison with the results of the execution with 1 thread. A slight increase in shown in the number of faults that did not manifest at all. This is because gem5 serializes the parallel execution of the cores. Hence in the globally executed code appear portions of code with higher density of OS code. Since injected faults do not manifest when OS code is executing the probability for fault to not manifest has slightly increased.

This similarity is quiet reasonable since all computation are pure parallel. There is no communication between threads and a user visible error can not propagate to the computation of another thread as long as the application does not crashes.

11.4.2 Optimized binary code

We compiled this application with the Optimization given by the Parsec Benchmark Suite (-O3 -funroll-loops -fprefetch- loop-arrays -fpermissive -fno-exceptions) while we added some Optimization referred by [16] (-mcpu=ev67 -mtune=ev67).

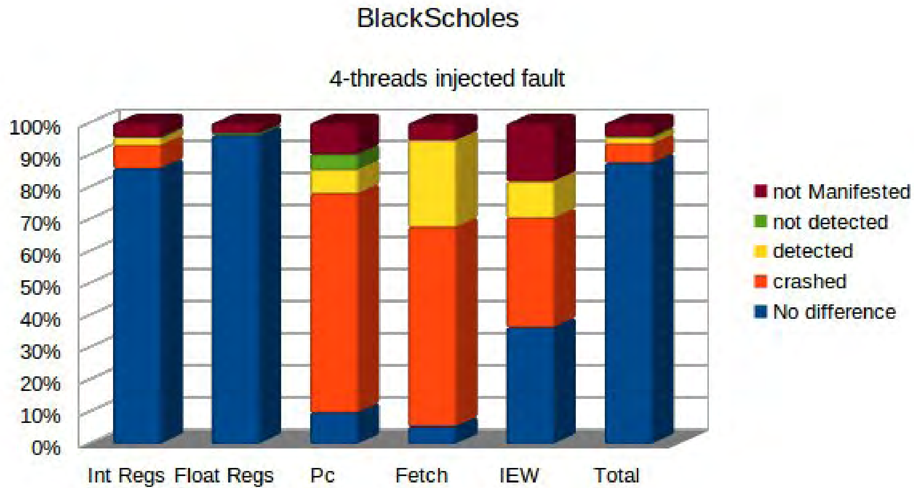


Figure 11-3: Blackscholes: Fault Injection results 4 thread Unoptimized

Corruption / Fault Place	No difference		crashed		detected		not detected		not Manifested	
	#	%	#	%	#	%	#	%	#	%
Integer Register	994	84%	114	9%	31	2%	2	0%	52	5%
Floating Register	1087	94%	0	0%	0	0%	4	0%	64	6%
PC address	14	38%	19	50%	2	5%	0	10%	3	7%
Fetch	9	27%	14	43%	8	24%	0	0%	2	6%
IEW	9	28%	8	26%	4	12%	0	0%	11	34%
Total	2113	86%	155	7%	45	2%	6	0%	132	5%

Table 11.5: Blackscholes: Fault Injection results 1 thread Optimized

In order to avoid repeating the results during my thesis I will explain only the interesting differences of these experiments. During these campaign of experiments we expected an increase in the experiments which had a different output since the purpose of a compiler is to utilize the hardware on the maximum and hide latency's between memory and cpu. The main trick behind these Optimization is to bring something from the memory to a register do as many computations with this value as possible and the return the value again to the memory. So a higher utilization of the number of registers is accomplished however this is disadvantageous for the reliability of the system since the probability for a fault to affect a register which is used increases thus the probability of a fault to affect the output is higher.

On the other hand the differences between the results are minimal on the optimized code for example the faults which did not manifest has increases this is reasonable because although the user code displays a decrease in the execution time the decrease in the execution time of the Operating System s stable and thus the probability for a fault to try to manifest during the execution of the Operating slightly increases.

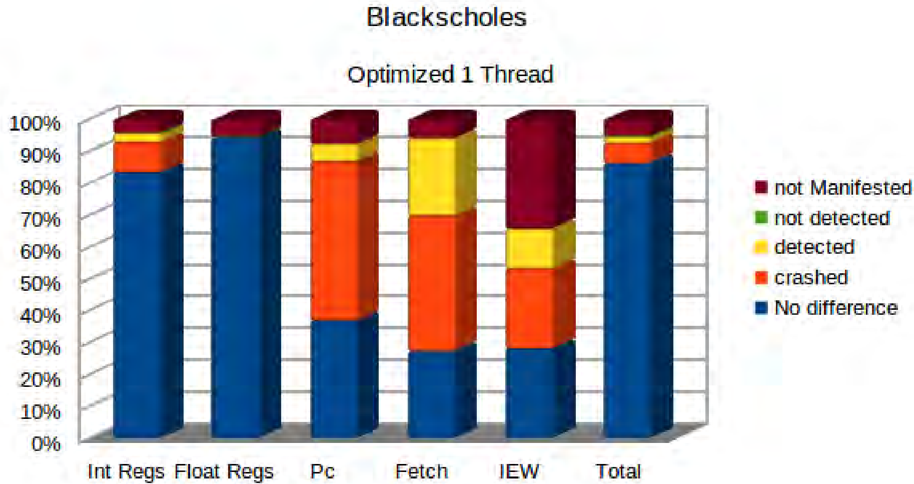


Figure 11-4: Blackscholes: Fault Injection results 1 thread Optimized

Corruption / Fault Place	No difference		crashed		detected		not detected		not Manifested	
	#	%	#	%	#	%	#	%	#	%
Integer Register	902	81%	74	7%	30	3%	1	0%	110	9%
Floating Register	1080	89%	7	1%	0	0%	6	0%	125	10%
PC address	5	13%	25	68%	3	8%	0	0%	4	11%
Fetch	8	27%	7	23%	10	33%	0	0%	5	17%
IEW	12	25%	9	18%	8	16%	1	2%	19	39%
Total	2007	82%	122	5%	51	2%	8	0%	263	11%

Table 11.6: Blackscholes: Fault Injection results 4 thread Optimized

11.5 Fault Injection In Fluidanimate

Smoothed Particle Hydrodynamics (SPH) is a computational method for simulating fluid flows. The benchmark application computes one time step of a liquid simulation which solves the Navier Strokes equation using SPH. THe liquid is represented by a set of of particles which interact with each other. These interactions are short ranged which enables the usage of a uniform grid to accelerate the determination of which particles interact.

After computing the density of the fluid at each particles position the acceleration is calculated. Next collision detection is performed and collision response is done with a penalty method. Finally the new position and velocity of each particle is calculated based on its acceleration.

The parallelization algorithm is based on spatial partitioning. As mentioned above particles are sorted spatially into a uniform grid which covers the entire simulation domain. The grid is evenly partitioned along cell boundaries in order to produce sub grids and assign them to different threads. Since particles which reside in adjacent cells interact with each other multiple threads may need to update cells which lie on sub-grid boundaries.

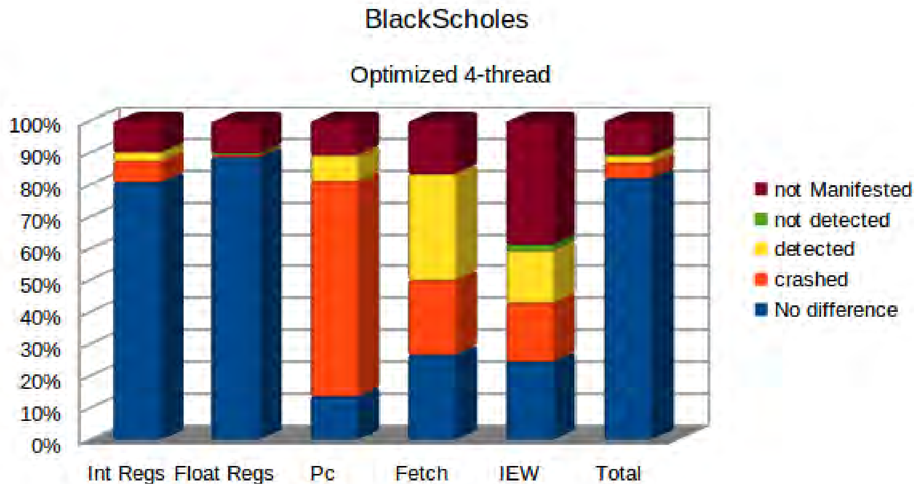


Figure 11-5: Blackscholes: Fault Injection results 4 thread Optimized

Each grid cell has a flag which indicates whether or not it lies on a sub grid boundary. When a particle in a boundary cells needs its info to be updated locks are used for that update. In other case which particles do not reside on boundaries locks are not used since updates will never occur concurrently.

Fault injection was performed on the main functions (ComputeDensities ComputeDensities2MT ComputeForces, ProcesscollisionsMT,AdvanceParticlesMt) which co respond to more than the 90% of the execution time of the benchmarks. Since a huge amount of floating point calculations are performed in this benchmark and each particle interact with other particles faults on a particle may propagate to the entire data space producing crashes or resulting to incorrect results.

On the other hand since the application visualizes the movement of liquids there is a slight fault tolerance in the results.Furthermore due to the order of floating point operations, results have slight differences between them if runned with different number of threads. In order to distinguish the results between valid and faulty we turned to Mean squared error (MSE) allowing an error to be less than 10^{-4}.

11.5.1 Unoptimized code

On table (11.7) the results of the statistical fault injection are demonstrated. In comparison with the results of the fault injection in the unoptimized blackscholes fluidanimate seems to be less tolerant. These can be explained by 2 major reasons.

- The utilization of the systems components is higher. If we observed the system on a per tick basis we would probably recognize that on each tick fluidanimate utilized more registers than those that blackscholes did. Hence the more registers the system uses the more the probability that the fault leads to an error increase.

Fault Place \ Corruption	No difference #	No difference %	crashed #	crashed %	detected #	detected %	not detected #	not detected %	not Manifested #	not Manifested %
Integer Register	865	77%	163	15%	0	0%	47	4%	52	4%
Floating Register	1069	88%	74	6%	0	0%	0	0%	76	6%
PC address	8	22%	23	64%	0	0%	1	2%	4	12%
Fetch	8	22%	21	58%	0	0%	3	8%	4	12%
IEW	9	27%	11	33%	0	0%	2	7%	11	33%
Total	1959	81%	292	11%	0	0%	53	3%	147	5%

Table 11.7: Fluidanimate: Fault Injection results 1 thread (Unoptimized)

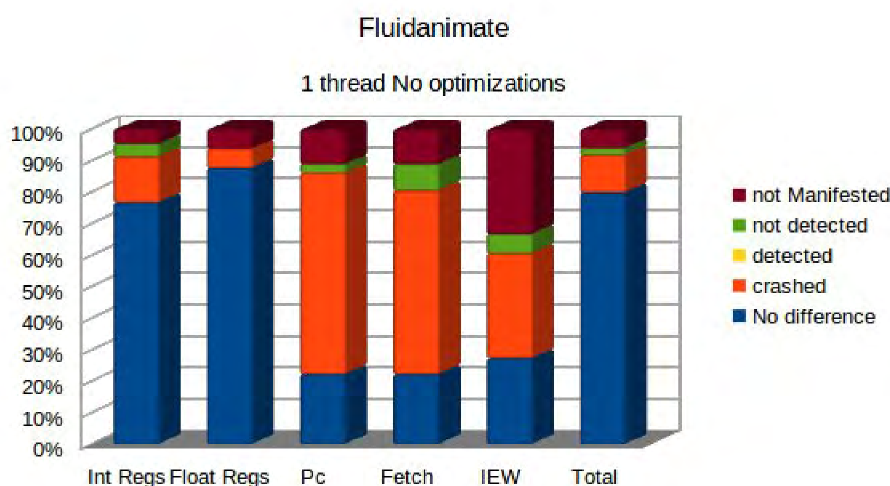


Figure 11-6: Fluidanimate: Fault Injection results 1 thread No-Optimizations

- Since many calculations are performed on each particle and a possible error on a particle propagates to the neighboring particles which on their turn compute wrong results. These effect travels through the entire application and increase the probability that a fault will be fatal. These explain why many injected faults on floating point registers crashed the system.

Fault Place \ Corruption	No difference #	No difference %	crashed #	crashed %	detected #	detected %	not detected #	not detected %	not Manifested #	not Manifested %
Integer Register	987	85%	125	11%	0	0%	45	4%	3	0%
Floating Register	1186	100%	0	0%	0	0%	0	0%	5	0%
PC address	10	28%	13	36%	0	0%	3	8%	10	28%
Fetch	4	12%	21	64%	0	0%	4	12%	4	12%
IEW	4	13%	14	45%	0	0%	5	16%	8	26%
Total	2191	90%	173	7%	0	0%	57	2%	30	1%

Table 11.8: Fluidanimate: Fault Injection results 4 thread (Unoptimized)

Besides the register faults the remaining places (PC,IEW,Fetch) have shown a similar behavior with the behavior of blackscholes. Most of fault injection crashed

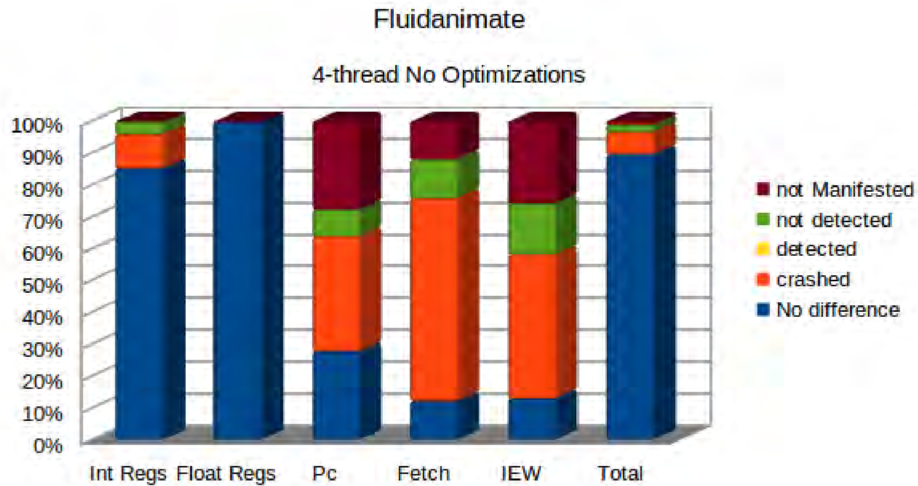


Figure 11-7: Fluidanimate: Fault Injection results 4 threads Unoptimized

the system which actually shows that the errors behavior of these components is similar regardless the application management.

11.5.2 Optimized Code

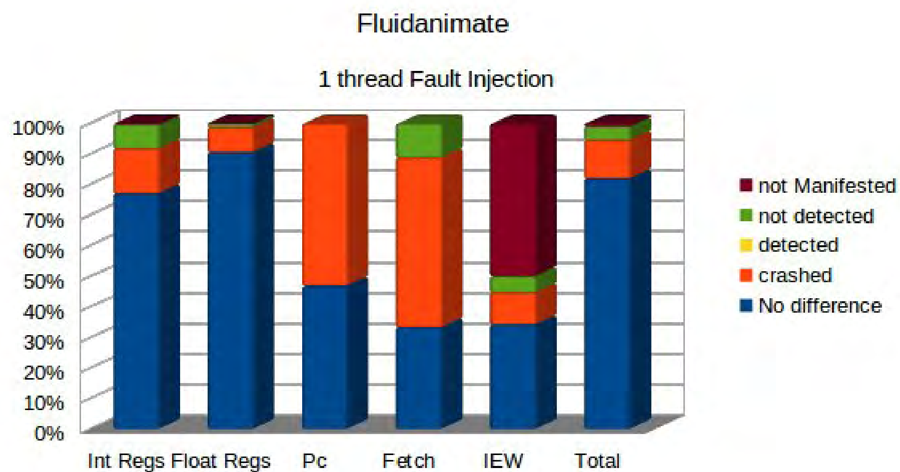


Figure 11-8: Fluidanimate: Fault Injection results 1 thread Optimized

For this campaign we employed the default optimazations of the Parsec benchmark suite (-O3 -funroll-loops -fprefetch- loop-arrays -fpermissive -fno-exceptions) while we added the optimazations which are referred in [X] (-mcpu=ev67 -mtune=ev67). All these optimazations demonstrate a huge reduce in the total number of executed instructions (almost 60 % reduction)

Fault Place \ Corruption	No difference #	No difference %	crashed #	crashed %	detected #	detected %	not detected #	not detected %	not Manifested #	not Manifested %
Integer Register	880	78%	167	14%	0	0%	90	8%	4	0%
Floating Register	1091	90%	99	9%	0	0%	12	1%	2	0%
PC address	15	47%	17	53%	0	0%	0	0%	0	0%
Fetch	12	34%	20	56%	0	0%	4	0%	0	0%
IEW	13	35%	4	10%	0	0%	2	5%	19	50%
Total	2011	82%	307	12%	0	0%	108	5%	25	1%

Table 11.9: Fluidanimate: Fault Injection results 1 thread Optimized

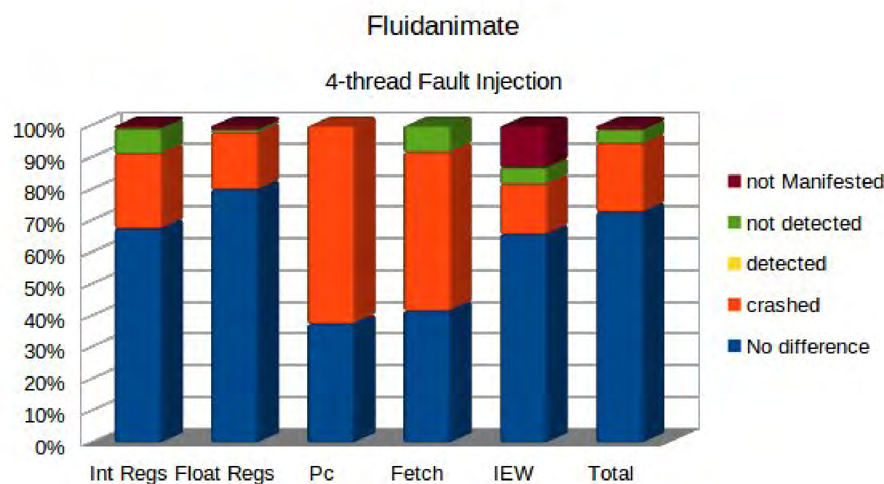


Figure 11-9: Fluidanimate: Fault Injection results 4 thread Optimized

The negative side effect of the optimization becomes more visible in these application since the application is larger and the nested loops on each function of the application gives the opportunity to the compiler to aggressively optimize the code by unlooping the loops. The unlooping of the loop has a higher demand in register utilization which is the main reason for the difference between the results.

Fault Place \ Corruption	No difference #	No difference %	crashed #	crashed %	detected #	detected %	not detected #	not detected %	not Manifested #	not Manifested %
Integer Register	772	67%	270	24%	0	0%	90	8%	10	1%
Floating Register	961	80%	216	18%	0	0%	10	1%	16	1%
PC address	12	37%	20	63%	0	0%	0	0%	0	0%
Fetch	15	41%	18	50%	0	0%	3	9%	0	0%
IEW	25	65%	6	17%	0	0%	2	5%	5	13%
Total	1785	73%	530	22%	0	0%	105	4%	31	1%

Table 11.10: Fluidanimate: Fault Injection results 4 thread Optimized

Moreover the relationship between neighboring particles do not help the application to barrier the faults into a single place but on the other hand a fault which affects the applications data can travel through the entire application. Thus the effect of

a fault is larger and it is almost impossible for a fault which affects the data of the application to be masked or stopped on a single place.

Finally the applications computation involve many “bad” floating point computations (multiplications,divisions) thus a fault which affects bits included in the exponent portion of a register lead to a great difference between the fault data and the correct data. These value if it is included in the following computation may lead to a floating point exception since diving with a great value produces a very small number which in the end may not be able to be represented by the architectures register (NAN,INF) the same applies for multiplication.

12 Conclusions

After the presentation of our framework and the analysis of the experiments, in the final chapter we restate our observations and discuss potential future work and conclude.

12.1 Conclusion

As we mentioned in the introduction the effort to enhance the performance of digital systems through the shrinking of the transistor size has a negative effect in the reliability of the circuit. The increase susceptibility of transistors to cosmic radiation along with the need of reducing the power consumption of a system raise a new "Wall" in the progress of electronic systems. We approached this barrier by observing the behavior of applications when introduced to faults in order to invent new fault tolerance techniques that will preserve the reliability of future systems in acceptable levels.

In this direction the first contribution of this Thesis is the enhancement of new concepts in an already existed fault injection tool which was created by another thesis and improved by this one. The new framework enables fault injection of transient intermittent and permanent faults in a full cycle accurate simulator in order to simulate an unreliable environment while allowing to inject different kind of faults on different applications/threads while executed on the same system. Moreover it is not limited to models covering radiation or timing induced faults, but also facilitates an easily extensible tool to support future effective fault models. Through many experiments our framework proved to be effective and demonstrated great ability to work with multiple workloads.

An additional contribution of this thesis is the automation of the fault injection campaign and the ability to run multiple simulations on parallel on a distributed system nullified the time consumption disadvantage while creating an ideal environment for experiments. Finally this thesis experimental analysis presented in chapter 10 helped to validate the correctness of the tool while giving a better insight on the behavior of applications in an unreliable environment. We observed the difference between the fault tolerance of CPU components and the effect of the faults on the

output. Specifically the architectural modules demonstrated diverse fault tolerance behavior. The inherent reliability can be exploited in order to improve other metrics, for example power consumption. As proposed in [33] portions of code can be characterized as insignificance . In that way we can use CPUs or computational units that function in sub threshold voltage to execute segments of code with lower power consumption thus higher fault probability.

12.2 Future Work

As an enhancement of this work we are interested in experimenting with more application and create a fault model which characterize computational portions of code as significant/insignificant. Moreover as a next step we plan to implement mechanism for scheduling different code parts in different units/CPUS based on their reliability or even adopt a software approach; scheduling work to different threads via working pools. Each thread or group of threads has some malfunctioning components. By doing all these we can evaluate the effectiveness of each method in terms of power consumption.

Bibliography

[1] High asic reliability by using fault-tolerant design techniques.

[2] A holistic design framework for energy efficient and reliable systems through quality-adaptive software.

[3] Significance-driven computation on next-generation unreliable platforms.

[4] Jean Arlat, Member, Johan Karlsson Yves Crouzet, Peter Folkesson, Emmerich Fuchs, and Gunther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE TRANSACTIONS ON COMPUTERS*, 52(9), September 2003.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 1, 2004.

[6] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation.* Kluwer Academic Publishers, 2003.

[7] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding parsec performance on contemporary cmps.

[8] Christian Bienia†, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. *Princeton University Technical Report*, January 2008.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, , and David A. Wood. The gem5 simulator.

[10] J. Carreira, H. Madeira, and J. Gabriel Silva. Xception: Software fault injection and monitoring in processor functional units. 1995.

[11] E. Cutringht, T. DeLong, and B. Johnson. Generic processor fault model. *Technical report, University Of Virginia*, 2003.

[12] R. Melhem D. Zhu and D. Moss. Analysis of an energy efficient optimistic tmr scheme international conference on parallel and distributed systems. *ICPAD'S*, 2004.

[13] J. Engblom. Full-system simulation technology.

[14] J.-C. Fabre, M. Rodriguez-Moreno F. Salles, and J. Arlat. Assessment of cots microkernels by fault injection.

[15] J.-C. Fabre, M. Rodriguez-Moreno F. Salles, and J. Arlat. Doctor: An integrated software fault injection environment.

[16] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler*. Running parsec 2.1 on m5.

[17] J. Gramacho. Analyzing the effects of transient faults into applications. *Master's thesis, Universitat Autonoma de Barcelona*, 2009.

[18] C. Fuhrman H. Ammar, B. Cukic and A. Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineerin.

[19] http://en.wikipedia.org/wiki/Intermittent_fault. Intermittent fault.

[20] http://www.m5sim.org/Introduction. The gem5 community.

[21] Eric Jenn, Jean Arlat, Marcus Rimén, Joakim Ohlsson, and Johan Karlsson. Fault injection into vhdl models: The mefisto tool.

[22] Xuanhua Li and Donald Yeung. Exploiting soft computing for increased fault tolerance. *Workshop on Architectural Support for Gigascale Integration*, June 2006.

[23] Jane W. S. Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. Use of imprecise computation to enhance dependability of real-time systems. *Foundations of Dependable Computing:Paradigms for Dependable Applications*, October 1994.

[24] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. Significance driven computation: A voltage-scalable,variation-aware, quality-tuning motion estimator.

[25] K. V. Palem. Energy aware algorithm design via probabilistic computing: From algorithms and models to moore's law and novel (semiconductor) devices. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, October 2003.

[26] B. D. Petrovic and G. S. Nikolic. Software tool for distributed elevator systems. *SCIENTIFIC PUBLICATIONS OF THE STATE UNIVERSITY OF NOVI PAZAR*.

[27] Ilia Polian, John P. Hayes, Sudhakar M. Reddy, and Bernd Becker. Modeling and mitigating transient errors in logic circuits. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 8(4), 2011.

[28] Paul Pop1, Kåre Harbo Poulsen1, Viacheslav Izosimov, and Petru Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems.

[29] F. Balbach V. Sieh, O. Tschäche. Vhdl-based fault injection with verify.

[30] C. Yount and D. Siewiorek. A methodology for the rapid injection of transient hardware errors. *IEEE Trans. Comput*, 1996.

[31] C. R. Yount and D. P. Siewiorek. Software-implemented fault injection of transient errors.

[32] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *The International Arab Journal of Information Technology,*, 1(2), July 2004.