Πανεπιστημιο Θεσσαλιας

Διπλωματικη Εργασια

Εισαγωγή σφαλμάτων για ανάλυση αξιόπιστης λειτουργίας σε επεξεργαστές πολλαπλών πυρήνων

Συγγραφέας:
Γεώργιος Τζιαντζιουλης

Επιβλέποντες:
Αναπληρωτής καθηγητής
Νικόλαος Μπελλας

Επίκουρος καθηγητής
Χρήστος Αντωνοπουλος

6 Ιουλίου 2011

Περίληψη

Καθώς οι τεχνολογίες των Ηλεκτρονικών Υπολογιστών εξελίσσονται, καινούργια προβλήματα εμφανίζονται μαζί τους. Ένα απο τα αναδυόμενα προβλήματα είναι η μείωση της αξιοπιστίας των ηλεκτρονικών κυκλωμάτων. Η αύξηση της ευαισθησίας των τρανζίστορ στη κοσμική ακτινοβολία, η διακυμάνσεις των ηλεκτρικών τους χαρακτηριστικών λόγο αστοχιών στη διαδικασία παραγωγής καθώς και η μείωση της αποδοτικότητας της παραγωγής ηλεκτρονικών κυκλωμάτων αποτελούν καίρια ζητήματα προς επίλυση ώστε να διατηρήσουμε την αξιοπιστία των μελλοντικών συστημάτων στα επιθυμητά επίπεδα. Σε αυτή τη κατεύθυνση δημιουργήσαμε ένα περιβάλλον εισαγωγής σφαλμάτων για την αξιολόγηση αξιόπιστης λειτουργίας υπολογιστικών συστημάτων. Παράλληλα, πειραματιστήκαμε με δύο εφαρμογές και παραθέτουμε μια εισαγωγή στο πώς επηρεάστηκε η συμπεριφορά τους απο σφάλματα υλικού.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Τις τελευταίες δεκαετίες γίναμε μάρτυρες της ραγδαίας ανάπτυξης των επιδόσεων και δυνατοτήτων των ηλεκτρονικών υπολογιστών αλλά και γενικότερα όλων των ψηφιακών συστημάτων. Οι συνεχείς εξελίξεις στην τεχνολογία της λιθογραφίας εξακολουθούν, τα τελευταία τριάντα χρόνια, να μειώνουν το μέγεθος των τρανζίστορ κατα το μισό κάθε 18 μήνες δημιουργώντας ένα πλεόνασμα σε κάθε νέα γενιά. Το πλεόνασμα αυτό χρησιμοποιείται για την υλοποίηση πιο περίπλοκων και αποδοτικών αρχιτεκτονικών, που ταυτόχρονα λειτουργούν σε υψηλότερες συχνότητες και μειωμένη τροφοδοσία.

Αυτή όμως η πρόοδος δεν γίνεται χωρίς την εμφάνιση προβλημάτων. Καθώς το μέγεθος των τρανζίστορ μίκρυνε δευτερεύοντα φαινόμενα που δεν επηρέαζαν τις προηγούμενες τεχνολογίες μετατράπηκαν στα κυρίως εμπόδια για περαιτέρω βελτίωση των ψηφιακών συστημάτων. Η δυσανάλογη επιτάχυνση των τρανζίστορ σε σχέση με τις δυναμικές μνήμες τυχαίας προσπέλασης δημιούργησε το επονομαζόμενο “memory wall”, ενώ η συνεχής συσσώρευση τους ανέβασε την ενεργειακή πυκνότητα σε δυσβάσταχτα επίπεδα.

Πέρα των παραπάνω προβλημάτων, ένα ακόμη —ιδιαίτερα σημαντικό— πρόβλημα είναι η μείωση της αξιοπιστίας των τρανζίστορ. Η μείωση των διαστάσεων τους έκανε τα ηλεκτρικά στοιχεία πιο ευάλωτα σε λάθη λόγω κοσμικής ακτινοβολίας. Παράλληλα, οι διαφορές λόγω ανακρίβειας στη διαδικασίας εκτύπωσης τους έχει επιπτώσεις στα ηλεκτρικά χαρακτηριστικά τους με αποτέλεσμα να συμπεριφέρονται πλέον περισσότερο ως τυχαίες μεταβλητές, αχρηστεύοντας την τεχνική της ανάλυσης χρονισμού χειρότερης περίπτωσης. Ακόμη, η αποδοτικότητα της διαδικασίας παραγωγής, δηλαδή πόσα απο τα παραγόμενα συστήματα τελικά θα βγουν στην αγορά, γίνεται πιο περίπλοκη. Συγκεκριμένα, καθώς περισσότερα στοιχεία συνθέτουν το τελικό μας σύστημα, μεγαλώνει και η πιθανότητα να εμφανιστεί αστοχία ή ανακρίβεια σε κάποιο απο αυτά. Η τάση αυτή μπορεί να οδηγήσει στο μέλλον, για λόγους κόστους, να δίνονται στη κατανάλωση συστήματα που μόνο ένα μέρος αυτών θα έχουν αξιόπιστη λειτουργία. Διαβλέποντας ότι τα προβλήματα αυτά θα γίνουν πιο έντονα με τις επόμενες γενιές υπολογιστών και η προοπτική της αύξησης των σφαλμάτων σε τέτοιο βαθμό ώστε ακόμη και συστήματα στα οποία δεχόμαστε να παρουσιάζονται λάθη (π.χ. προσωπικός υπολογιστής) να γίνουν υπερβολικά αναξιόπιστα, η ερευνητική κοινότητα στράφηκε τα τελευταία χρόνια στην αναζήτηση βιώσιμων λύσεων.

Προϋπόθεση για τη δημιουργία και παραγωγή λύσεων είναι η ανάλυση της επίδρασης των σφαλμάτων υλικού στις εφαρμογές που θα εκτελεστούν σε αυτό. Με τη σειρά του αυτό, προϋποθέτει την ύπαρξη κατάλληλων μετρικών για να ποσοτικοποιήσουμε την αξιοπιστία ενός συστήματος, εργαλεία τα οποία θα μας επιτρέψουν τον πειραματισμό με μη-αξιόπιστα περιβάλλοντα εκτέλεσης, αλλά και πρωτότυπα μηχανισμών επανάκαμψης από/αποφυγής σφαλμάτων. Σε αυτό το σημείο είναι και η συνεισφορά αυτής της διπλωματικής εργασίας.

Το προϊόν αυτής της εργασίας είναι η επέκταση ενός σύγχρονου, ευρέως διαδεδομένου προσομοιωτή πλήρων-συστημάτων (ακριβείας-κύκλου) με ένα σύνολο δομών και συναρτήσεων που επιτρέπουν την εισαγωγή σφαλμάτων στις δομές του. Ο επιλεγμένος προσομοιωτής (M5) παρέχει τη δυνατότητα προσομοίωσης μονοπύρηνων/πολυπύρηνων συστημάτων, καθώς και συστήματος-συστημάτων. Η νέες δυνατότητες εισαγωγής σφαλμάτων στις δομές του δεν υπόκεινται σε κανέναν περιορισμό στον αριθμό, το χώρο ή το χρόνο. Υποστηρίζονται η εισαγωγή σφαλμάτων σε 2 μοντέλα (λειτουργικό και αναλυτικό) της Μονάδας Επεξεργασίας Δεδομένων — συγκεκριμένα στο φάκελο καταχωρητών, το μετρητή προγράμματος και τα στάδια προσκόμισης, αποκωδικοποίησης και εκτέλεσης της εντολής — και στα περιεχόμενα της φυσικής μνήμης.

Όσον αφορά το τρόπο που θα αλλοιωθεί η τιμή του επιλεγμένου στοιχείου, το εργαλείο μας προσφέρει την επιλογή μεταξύ άμεσης ανάθεσης κάποιας τιμής, εναλλαγής ενός bit στη συμπληρωματική του τιμή, λογική διάζευξη της τιμής του στοιχείου με κάποια σταθερά καθώς και την ανάθεση σε όλα τα bit της δομής την τιμή 1 ή 0. Τα σφάλματα αυτά μπορεί να οριστούν ως παροδικά, μόνιμα ή στιγμιαία.

Μετά το πέρας της υλοποίησης της επέκτασης του προσομοιωτή συνεχίσαμε με πειραματισμό χρησιμοποιώντας το καινούργιο εργαλείο. Πειραματιστήκαμε με 2 εφαρμογές: μια εφαρμογή πολλαπλασιασμού πινάκων 64x64 και τον αποκωδικοποιητή βίντεο AVS.

Τα συμπεράσματα αυτών των πειραμάτων ήταν, αφενός ότι δεν φανερώνονται όλα τα λάθη στο επίπεδο της διεπαφής του χρήστη· δηλαδή αρκετά σφάλματα επικαλύπτονται/χάνονται και αυτό διότι υπάρχει επαναχρησιμοποίηση των καταχωρητών και 'επιθετική' προσκόμιση και εκτέλεση εντολών με αποτέλεσμα να μη χρησιμοποιούνται όλα τα αποτελέσματα που παράγονται/επηρεάζονται απο σφάλματα. Αφετέρου, ότι υπάρχουν τμήματα σε προγράμματα τα οποία μπορούν να υποστούν σφάλματα χωρίς να διακόπτεται η εκτέλεση ή η αλλοίωση των δεδομένων να μην τα καθιστά άχρηστα (π.χ. σφάλματα στο κάτω-δεξί κελί του πινάκα iDCT στον AVSA.2). Η τελευταία παρατήρηση έχει ιδιαίτερη σημασία καθώς μπορεί να μας βοηθήσει στην επίλυση του ενεργειακού προβλήματος, "power wall". Συγκεκριμένα, σε συνδυασμό με μια αρχιτεκτονική που θα προσφέρει ανοχή σε σφάλματα χρονισμού θα μας έδινε τη δυνατότητα να εκτελούμε τέτοιους κώδικες σε Μονάδες Επεξεργασίας Δεδομένων που λειτουργούν σε τιμές τροφοδοσίας κάτω των προδιαγραφών, το οποίο δημιουργεί σφάλματα χρονισμού στο κύκλωμα αλλά παράλληλα εξοικονομεί ενέργεια (βλέπε συνάρτηση 1.1).

$$Power = Capacitance * Frequency * Voltage^2 \qquad (1.1)$$

Chapter 2

Introduction

During the last four decades we witnessed an enormous growth in the performance and functionality of processors. Advances in fabrication technology conveyed us from 10μm features width in 1971 to today's 32nm, a trend that led to the doubling of processors' transistor count every two years — colloquially called Moore's law. This surplus of transistors enabled computer architects to design more sophisticated circuits which, at the same time, functioned at a higher frequency and lower voltage.

On the other hand, secondary phenomena that had little or no effect on previous technologies were exacerbated as transistor features shrank creating new obstacles that halt performance [9]. The disproportional speed-up of transistors, compared to that of DRAMs, created the so called "memory wall" and their continuous accumulation increased circuit power density to an unbearable degree, a phenomenon called "power wall".

Apart from the above problems, as transistors approached the nanometer threshold they became more vulnerable to radiation-induced faults (cosmic particles that enter the atmosphere). Furthermore, as their features size decreases, the problems from process variations increase. The electric characteristics of transistors now behave more like random variables making a deterministic worst case timing analysis ineffective. As the impact of physical variables becomes more intense, researchers are working on building reliable systems with unreliable components to guarantee the correct function of modern electronic devices under such constraints. Another important aspect, though not strictly technical, is the decrease of the process yield. Accumulation of more components for a single system means that there is a greater chance part of it to contain corruptions, which will affect its output's reliability. Currently, this problem is solved by disabling such modules, however, this is inefficient in both economic and performance terms. Except from the direct benefits from technological advancements in this area other areas may be benefit too; for example, fault-tolerant architectures would enable the use of processors in sub-threshold voltage levels to improve power efficiency.

Reliable execution on unreliable components is the next "wall" electronic system designers should overcome to further improve computers performance. To effectively attack this problem we need to perform a thorough analysis of the way hardware faults manifest to errors in the different abstraction layers of computer systems. The analysis and categorization of the behavior of faults will enable the creation of a hierarchy of targets/modules that need to be enhanced to achieve robustness.

In order to expand our understanding on the aforementioned problems new tools need to be created or the existing ones should be augmented to support the new constrains and physical variables. The main contribution of this Thesis is in this direction. In order to study the behavior of applications in the presence of transient faults, we developed a fault injection framework on top of M5 (a full system, cycle-accurate simulator) as we could not find an existing tool to cover our needs.

We choose to work with a software simulator for the perspective and attributes it provides. By using a simulator we are able to study architectures and configurations not yet implemented and at the same time exercise complete control over the nature of a fault; that is, the location and timing of the manifestation of the fault, as well as its effect on the affected module's value. Also, the provided controllability in experimentation enables the partial examination of a program's execution for a more thorough analysis of its behavior on an unreliable environment.

The other contribution of this Thesis is the experimental evaluation of two applications on an unreliable environment: an 64x64 matrix multiplication kernel and the AVS decoder.

This document is structured in two basic blocks: the first one introduces the theoretical background and the second describes the framework and outlines the experimental evaluation.

In the first part, a brief introduction in fault-tolerance and fault injection is given in Chapters 3 and 4. In Chapter 5 we discuss the benefits of full system simulation.

In the second part, starting at Chapter 6, we give an overview of the fault model in which this work was based upon. Chapters 7 and 8 discuss the implementation details of the framework and an analysis of the results obtained using it, in respect.

Finally, Chapter 9 presents the conclusion of this work and directions for future work.

Chapter 3

Fault Tolerant Systems

In the previous section we discussed the need for fault-tolerant design of future systems. A system is fault-tolerant if it is capable of performing its specified tasks in the presence of faults, either at the hardware or at the software level.

For example, software bugs need to be overcomed for continuing correct execution and hardware module failures should not inhibit the system's ability for correct execution. By correct we are not refering only to error free results, but also on results that maintain a small, user/application defined, margin of error.

Fault tolerance is an important feature of a system as it ensures its dependability; it is, as stated in [6], the ability to deliver service that can justifiably be trusted.

3.1 Dependability Attributes

Dependability can be considered as the integrating concept of the following attributes [6]:

Reliability : continuity of correct service.

Availability : readiness for correct service.

Safety : absence of catastrophic consequences on the user's environment.

Maintainability : ability to undergo modifications and repairs.

Following, we will see in more details the two main attributes of dependability [41], **reliability** and **availability**.

We can define **reliability** (R) as the probability that a component will not experience an error, visible to the defined abstraction's outer scope, in the time interval $(0, t]$,

$$R(t) = P(T > t), \qquad (3.1)$$

where T is a random variable expressing the component's lifetime.

The reliability of a system (i.e. an accumulation of similar components) of N modules in a period of time $(0, t]$ is given by the fraction of the unfaulted components until time t to the total amount of components.

$$R_N(t) = \frac{N_{\text{unfaulted}}(t)}{N_{\text{total}}} = \frac{N_{\text{total}} - N_{\text{faulted}}(t)}{N_{\text{total}}} = 1 - \frac{N_{\text{faulted}}(t)}{N_{\text{total}}}, \quad (3.2)$$

where N_{total} is the total number of components, $N_{\text{unfaulted}}(t)$ is the total number of components that *have not* experienced an error in the time interval $(0, t]$ and $N_{\text{faulted}}(t)$ is the total number of components that *have* experienced an error in the time interval $(0, t]$.

As **hazard rate** $H_N(t)$ we define the probability that a system of N components, that have not presented a fault till time t, will present an error in the time interval Δt.

$$H_N(t) = \frac{\frac{d}{dt}\left(N_{\text{faulted}}(t)\right)}{N_{\text{unfaulted}}} = \frac{\frac{\frac{d}{dt}\left(N_{\text{faulted}}(t)\right)}{N_{\text{total}}}}{\frac{N_{\text{unfaulted}}}{N_{\text{total}}}} \quad (3.3)$$

By combining equation 3.3 and the derivative of equation 3.2,

$$\frac{d}{dt}\left(R_N(t)\right) = -\frac{\frac{d}{dt}\left(N_{\text{faulted}}(t)\right)}{N_{\text{total}}}, \quad (3.4)$$

we get,

$$H_N(t) = \frac{-\frac{d}{dt}\left(R_N(t)\right)}{R_N(t)}, \quad (3.5)$$

which we can transform to equation 3.6,

$$R_N(t) = e^{-\int H_N(t)\, dt}, \quad (3.6)$$

so as to represent **reliability** in terms of the **hazard rate**.

The exponential relationship between a system's reliability and time, as shown in equation 3.6, is known as the *exponential failure law* and enables us to compute the overall reliability of a system with N components by the following equation.

$$R_N(t) = \prod_{i=1}^{N} R_i(t) = e^{-\left(\sum_{i=1}^{N}\left(\int H_i(t)\, dt\right)\right)t} \quad (3.7)$$

Finaly, **availability** can be defined as the probability that the system will function correctly at a given time.

$$\text{Availability} = \frac{\text{System up-time}}{\text{System overall time}} \quad (3.8)$$

3.2 Dependability Threats

When the dependability of a system is compromised, we say a failure has occurred; that is, the desired service of the system deviates from correct function. A failure occurs when an error is presented in a system component. An error is a divergence of a component to an incorrect state. The cause of an error is called a fault and with that we have formed the chain of dependability threats.

The connection between the dependability threats is depicted in figure 3.1.

$$\begin{tikzcd}\cdots \arrow[r] & \text{Fault} \arrow[r, "\text{activation}"] & \text{Error} \arrow[r, "\text{causation}"] & \text{Failure}\end{tikzcd}$$

Figure 3.1: Chain of dependability threats [6].

Faults can be categorized according to their nature in three basic categories[41]:

Permanent: Permanent faults are present indefinitely in the system until corrective measures are taken.

Intermittent: Intermittent faults appear, disappear and then reappear in the system's life time and can be early indications of permanent faults.

Transient: Transient faults appear and disappear without repetition.

A brief correspondence of different hardware faults to fault types can be seen in table 3.1.

		Fault Types	
		Permanent	Transient
Hardware faults	Electro-migration	✓	
	Metal Stress Voiding	✓	
	Gate Oxide Wear-out	✓	
	Hot Carrier Injection	✓	
	Negative Bias Temperature Instability	✓	
	Radiation		✓
	Transistor Variability		✓
	Thermal Cycling		✓
	Erratic Fluctuation in minimum voltage		✓

Table 3.1: Hardware faults and their type

3.3 Faults Impact on a System - An Architectural Perspective

Whether a fault manifests to a failure depends on whether the error will be masked/corrected or not. Figure 3.2 shows all the possible outcomes of an error occurrence by an architectural perspective.

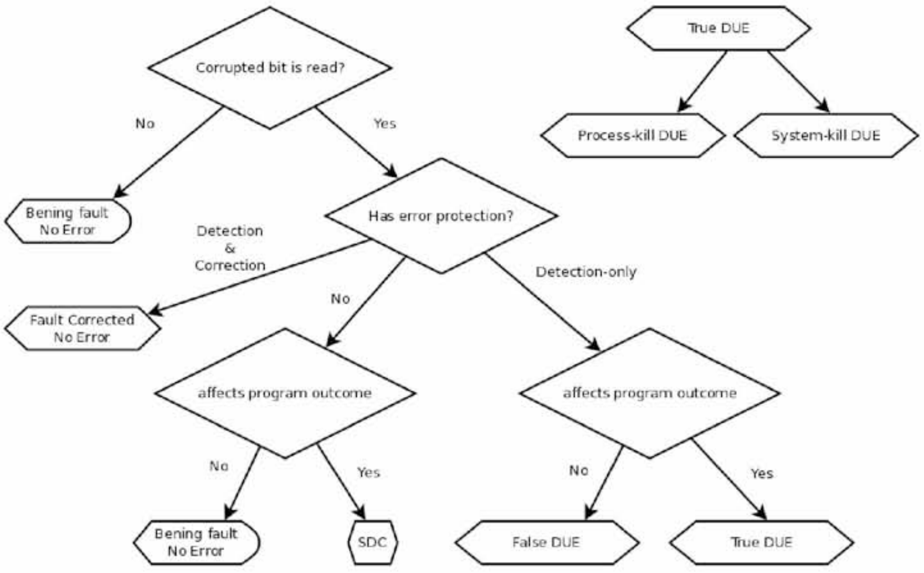

Figure 3.2: Faults impact [42, 43, 41]

To find the possible outcomes and the impact of a fault on a system's behavior we must first consider if the faulty part (bit) is used by the system for a particular service. If it is not used, then the fault is called benign and produces no error, otherwise we have to further consider if it is covered by any error-correction mechanism.

If the faulty part is covered by a detection and correction mechanism then the fault is corrected and produces no error. In case of no protection mechanism or a detection-only mechanism then we need to evaluate if the error influences the output of the particular service.

If no error protection covers the erroneous part but it does not influence the system's output then the fault is benign and no error occurs, but in the case that the output is influenced then we have a Silent Data Corruption (SDC).

On the other hand, if the fault is covered by detection-only mechanism then again we observe the influence that it has on the system's output. If the output is not altered we have a False Detected Unrecoverable Error (False DUE), else, if it affects the system's output we have a True Detected Unrecoverable Error (True DUE).

True Detected Unrecoverable Error can be further divided into System-kill and Process-kill faults based on whether the error can be isolated in a single process or it requires a system wide restart.

3.4 Quantitative Analysis of μarch Reliability

As the reliability assessment of a system is crucial and influences decisions in the design phase of a project, research has been made in producing quantitative measures of it.

The most well known measures of system's reliability are Mean Time to Failure (MTTF), Failure in Time (FIT) and Mean Time between Failures (MTBF)

[41].
MTTF expresses the mean time between two faults (i.e. if a module presents a fault every 2 years its MTTF is 2 years).

$$MTTF_N = \frac{1}{\sum_{i=0}^{N} \frac{1}{MTTF_i}} \tag{3.9}$$

Due to the complexity of computing the MTTF for a number of components many choose to use FIT which is additive. One (1) FIT represents a fault in a billion (10^9) hours, so if we have two components and the first one has a FIT of 10 and the other a FIT of 20 then their overall FIT is 30.

$$FIT_N = \sum_{i=0}^{N} FIT_i \tag{3.10}$$

the relation between MTTF and FIT is:

$$MTTF_{in\,years} = \frac{10^9}{FIT \times 24\,hours \times 365\,days} \tag{3.11}$$

MTBF expresses the mean time between the occurrence of two faults. To compute MTBF we need to know the Mean Time to Repair (MTTR).

$$MTBF = MTTF + MTTR \tag{3.12}$$

In the recent literature additional metrics have been proposed for use in evaluating a structure's reliability. In [43] and [42] the Architectural Vulnerability Factor (AVF) was introduced. AVF is defined as "the fraction of time an upset in a cell can cause a visible error in the final output of the program". AVF integrates both the application and μarchitecture vulnerability factors and does not provide any detail on the degree that a particular μarchitecture or application affects the vulnerability factor. An effort to separate these two factors into distinct metrics was made in [53] and [52] by introducing the Program Vulnerability Factor (PVF) and Hardware Vulnerability Factor (HVF).

Chapter 4

Fault Injection

After introducing the general concept of dependability/fault-tolerance of a system in this chapter we will present fault injection (FI) a technique to evaluate it.

Fault injection can be defined as the process of introducing and instrumenting faults/disruptions in a system for study of its behavior in the presence of faults. FI is an essential complement to other techniques used for validation of systems' dependability.

To enhance our confidence in the validation process of a system, it is necessary to use a variety of different methods to thoroughly examine and ensure a good level of dependability. Fault injection allows to confirm the structure and calibrate the parameters of existing fault-tolerance models or to develop new fault-tolerance models and validate them: for example x% of errors of type y are detected [28].

Based on the phase of a project, fault injection is employed using different techniques and layers of abstraction. During the design phase, simulation is mainly used for assessing the dependability of the system, whereas at the prototype phase physical fault injection is preferred.

4.1 A Fault Injection Taxonomy

This section serves as a brief taxonomy of fault injection techniques. The basic techniques will be presented along with related work and the advantages and disadvantages of each approach [60]. The first part classifies techniques based on the method used for the injection and the second enumerates the layers of abstraction at which the fault can be injected.

4.1.1 Physical Fault Injection

Physical fault injection methods are using the actual hardware of the System Under Test (SUT) and the injection is done by augmenting the system with special hardware that enables the introduction of faults. Such techniques can be further divided to those that have direct contact with the SUT and those that don't.

Fault injection *with contact* is performed by producing voltage and current disturbances to the system's external pins. The pins can be manipulated by either using *active probes* or using *sockets*. According to the first method, pin-level active probes are attached in the SUT pins in order to directly create stack-at faults or to short-circuit them to create bridging faults. According to the second method, injection is performed by inserting a socket between the targeted module and its circuit board. The socket enables the creation of stack-at, open or more complex logical faults like inverted, XORed, ANDed pins or previous value of them.

For fault injection *without contact* special equipment is needed and it mainly targets transient fault injection. To create hardware faults without contact one can employ radioactive isotopes, neutron beams, proton beams or other means for creating electromagnetic interferences in the SUT. Radioactive isotopes such us americium-241, uranium-238, californium-252 or thorium-232 that emit alpha particles can be used for measuring the impact of alpha particle induced errors, whereas for neutron induced soft errors one can use energetic proton or neutron beams[36]. A list of locations where one can find the equipment for such experiments can be found at the JEDEC standard No.89[5].

The main advantage of physical fault injection is that it uses the original hardware without changes and experiments are executed at real-time enabling the experimentation with large workloads and unaltered applications. Moreover, it can inject faults in places that are hard to reach by other means without changing the SUT's hardware, thus it is one of the less-intrusive FI methods.

On the other hand, these methods have low portability over systems, requiring new design and installation for each platform. The use of special purpose hardware that requires large funding and can be found in limited places makes difficult the broad adaptation of such techniques by the research community. In addition, physical fault injection lacks in terms of controllability and observability; especially in methods without contact it is extremely hard to create targeted and precise injection.

Physical fault injection was one of the first methods used to assess the dependability of a system. A lot of tools and techniques have been developed during the past decades. Some of them are:

RIFLE[39] A pin-level fault injection system that employs the socket method and was developed at the University of Coimbra for dependability evaluation.

MESSALINE[4] A pin-level fault injection system developed at LAAS-CNRS that uses both socket and probe methods.

FIST[34] Fault injection system developed at the Chalmers University of Technology for study of transient faults using heavy-ion radiation.

FTMP[26] A fault-tolerant processor created by NASA which has also been evaluated using fault injection techniques.

4.1.2 Fault Emulation

In Fault Emulation techniques (a.k.a SWIFI, Software Implemented Fault Injection) faults are introduced through changes in the software, so as to emulate corruptions from hardware faults, while executing in the actual SUT. Fault emulation techniques can be divided into compile-time techniques and run-time techniques, based on the time the injection is performed.

Compile time fault injection modifies the source code or assembly instructions of the software under evaluation during the compilation process in order to emulate hardware and software (bugs) faults.

On the other hand, in run time fault injection the application's execution is halted through timers, traps/exceptions or code is injected to perform fault injection in the hardware structures that are accessible through software. When using timers (hardware or software) a timeout occurs at a specific time and control is given to system software that performs the fault injection. Similarly, traps or exceptions are used to specify the timing of fault injection based on events (e.g. specific instruction execution or memory address access). Finally, the last approach, code injection, inserts instructions in parts of the source code at run-time to emulate hardware faults.

Fault emulation fault injection has the advantage of been fast (execution at real time) and use the actual SUT. Also, it enables targeting parts of applications or operating systems, which is hard to do with other techniques. Another advantage of software based techniques is the high controllability that they provide and the repeatability of the experiments.

However, fault emulation is an intrusive method both in modifying the original source code and in indirectly affecting the applications' timing. The execution of the fault injection and control framework on the same system can disrupt the statistics and observations of the experiment. In addition, software-based methods can only access structures that are available through the ISA, thus restricting the injection capabilities.

Fault emulation was the prevaling choice for fault injection during the 90's. Some of the most widely accepted frameworks are:

FERRARI[31] A framework for fault emulation using traps and systems calls developed at the University of Texas, Austin.

FINE[32] A run-time fault injection and monitoring environment for the Unix OS developed at the University of Illinois, Urbana Champaign.

FTAPE[55] A fault injection framework for assessing a system's fault tolerance developed at the University of Illinois, Urbana Champaign.

JIFI[51] An application-level fault injection framework developed for use on a fault tolerant parallel processing cluster at the California Institute of Technology.

XCEPTION[13] A software fault injection and monitoring environment that employs the debugging and performance monitoring capabilities of microprocessors for fault injection; developed at the University of Coimbra, Polo II.

DOCTOR[49] An integrated software fault injection environment developed at the University of Michigan with emphasis in portability.

EXFI[10] A SWIFI environment for fault coverage evaluation on embedded microprocessor-based boards that relies on the "Trace Exception Mode" of multiprocessors; developed at Politecnico di Torino.

NFTAPE[54] A configurable environment for automating fault injection experiments. It was developed in a modular way to enable the usage of different tools for fault injection, triggering and monitoring. It was developed at the University of Illinois, Urbana Champaign.

GOOFI[1] A fault injection tool with support for pre-execution fault injection and injection through scan-chain capabilities of the processor. It was developed at the Chalmers University of Technology with emphasis on portability.

FAIL-FCI[25] A high-level fault injection framework for evaluation of distributed applications in the presence of faults (crashes) developed at INRIA, France.

4.1.3 Software Simulation

Simulation-based fault injection requires the design and development of a simulation model of the SUT (i.e. processor, devices) that is simulated on another system. The simulation model of a system can be described either using a common programming language like C and C++ or a Hardware Description Language (VHDL, Verilog). Traditionally, the system under test is called target and the system in which the simulator is executed is called host.

Due to the extensive use of Hardware Description Languages, like VHDL, in the design process of modern ICs, several techniques have been proposed in simulation based fault injection with the use of HDL models. We can divide these techniques in two main categories: those that require modification of the original HDL source code and those that use built in capabilities of the simulator.

The first approach, that relies on code modification, augments the system's source code with dedicated fault injection modules called saboteurs or modify -mutate- the original code of a module to enable precise injection and tracing of faults.

The second approach uses modified simulators that support the injection and inspection of faults through built-in instructions/commands. The instructions target module signals and variables that can be directly accessed and manipulated through the simulator interface at run-time.

Another approach in simulation-based fault injection is the system modeling in a high-level programming language. The system's description in a functional layer enables full system simulation or system of systems simulation. Recent research in fault injection and reliability favors the usage of full system simulators that can be used, not only for assessment of the robustness of an architecture, but also for characterizing behavior of large workloads in the presence of faults.

Simulation-based fault injection frameworks can support any system abstraction level, electrical, logical, functional, architectural or hybrids, providing a plethora of choices. In addition, the use of software enables full control over fault modeling (transient, permanent, intermittent) and injection. Simulation provides maximum observability without being intrusive, as any element or module can be accessed at any time without influencing the behavior of the system. Simulation-based fault injection tools do not require the modification of application source code, enabling the validation of applications whose source code is not available. Another advantage of such tools is the low cost of reproduction of the system for multiple parallel simulations.

The main disadvantage of simulation based fault injection techniques is that experiments are consuming (compared to other methods). The more detailed the model, the more time it will take to have it simulated. This is in fact the main reason the research community started favoring simulators with higher level of abstraction, mainly developed using C and C++. Also, the implementation of an accurate model is expensive in time and effort and crucial for the simulation to be realistic and representative. Finally, as many existing simulators are commercial products, their source code is not available for modification.

Nowadays, simulation-based fault injection has become the most widely accepted methodology. Some of the past and currently used frameworks are:

SIMICS[40][8] A commercial full-system simulator that has been augmented with fault injection capabilities.

COTSon[3][23] A commercial full-system simulator that has been augmented with fault injection capabilities

SWAT-sim [38] A framework for hierarchical simulation for studying the system-level manifestation of gate-level faults.

VERIFY[50] A simulation based fault injection framework that augmented the VHDL language with fault injection signals and their occurrence enabling the description of components behavior after faults and instrumentation of fault injection campaigns.

DEPEND[22] A functional simulation framework focusing in providing a design and fault injection framework for system level dependability analysis.

MEFISTO[29] An integrated environment for applying fault injection with support of different levels of abstraction.

HEARTLESS[47] A hierarchical Register-Transfer-Level fault simulator.

GSTF[7] A simulation based fault injection tool with capabilities of evaluating medium-complexity system models.

4.1.4 Hardware Emulation

To cope with the time overhead of simulating the SUT, the usage of FPGAs has been proposed. The circuit under examination is implemented using the standard synthesis, placement and routing design flows and then it is downloaded to an FPGA. The fault injection process is instrumented through a computer that is connected with a high speed communication link to the FPGA board through its I/O pins. There are two approaches for enabling fault injection in FPGAs: the first is to alter the source code of the SUT and augment it with modules and structures for fault injection, whereas the alternative is to use the run-time reconfiguration. Run-time reconfiguration (RTR) exploits the ability of FPGAs to be reprogrammed on-the-fly. That way, the hardware can be alter directly in its low level structures to inject faults in it.

The benefits of hardware emulation based fault injection is the speed-up of experimentation compared to simulation-based methods and also the comparable controllability of these methods. Furthermore, FPGAs can be used for dependability assessment of reusable circuit components (IP blocks).

On the other hand, hardware emulation methods experience the same problems, thought not in such extend, with pin-level fault injection techniques. The injection capabilities are constrained by the number of free I/O pins of the programmable hardware. Moreover, the link used for the communication between the FPGA and the fault injection controller can influence the experimentation speed.

This approach for fault injection is new and has yet to receive wide acceptance from the research communication. Example publicatios on related research are:

FOCUS[14] A design automation environment developed at the University of Illinois at Urbana-Champaign that fashioned a scan-chain technique for fault injection in an FPGA board.

Using RTR for FI Applications [2] A proposal for exploiting the run-time reconfiguration capabilities of modern FPGA's for speeding-up the fault injection process.

4.2 Abstraction Based Categorization

Apart from the previous, method-oriented, categorization of fault injection techniques/frameworks, we can approach fault injection techniques based on the level of abstraction at which the faults are injected.

Experimentation in system's dependability, fault-detection and fault recovery mechanisms may be verified by injecting faults at any level of abstraction and allowing the errors to propagate to higher levels. Researchers prefer to work in the lowest level of abstraction to assure accuracy of the experiments, however, problems may arise from such a choice as it was previously described. Consequently, there has been extensive research in how physical faults manifest in higher levels (rtl, functional), to enable the usage of high-level simulators [59, 18, 12].

The following taxonomy is provided as given in [59] and presents a list of increasing levels of abstraction from the device level to the network level. It can be generalized into two broad categories: circuit-level abstractions at the lower end and functional-level abstractions at the higher end.

Circuit: In this category, the physical makeup of the processor is considered.

- **Device:** Its main focus is the transistor and other circuit elements. For simulated fault injection an analog simulator is need, whereas for physical injection radiation or other physical stress may be used.
- **Gate:** Its main focus are logic gates (AND, OR, NAND, XOR, multiplexers, etc). The stack-at or fixed-at model is employed although some implementations use more accurate models, modeling signals and storage cells coupling.
- **Basic Block:** Its main focus are functional units of the system like adders and registers and the fault models are high-level abstraction of physical fault models.
- **Chip:** Its main focus is the chip's boundary I/O.

Functional: In this category, the circuit description itself is no longer considered, but instead a functional description of it is used.

- **Micro-operation:** Its main focus in micro-instructions and faults are injected at data transfers and micro-sequencing.
- **Macro-operation:** Its main focus are ISA instructions and faults are injected by flipping bits in the instruction word.
- **System:** Its main focus is Memory and processor I/O in which faults are injected.
- **Network:** Its main focus are messages and other ways of communication where faults can be injected (e.g. message corruption, dropped package).

4.3 Our Choice

After studying past implementations and publication in the area of fault injection, we decided that simulation based fault injection and specifically full system simulation is what best serves our needs and objectives. Using a full system simulator we will be able to evaluate the impact of faults in large workloads, full OSs and in general real world applications which is our main target.

In brief, simulation provides maximum controllability over the injection procedure in both spatial and temporal manner, also, it enables the studying of independent or concurrent execution of applications. Another key feature that pushed toward this decision was the observability over the full trace of the behavior of the system, before and after an injection,that a simulator provides; needed for understanding the way different faults affect instructions and workload patterns.

As mention in section 4.1.3 software simulation is a non-intrusive method to the application that is executed and can provide statistics and logs of arbitrary detail. In addition, as our choosen simulation framework is open source we can alter the modules structure to match desired accuracy for each experiment.

Finally, we try to mitigate the time overhead of the simulation by running multiple simulation in parallel, something that can only be done with simulation-based fault injection techniques as the installations can be easily and with no cost reproduced.

In the next section we provide an overview of full system simulation and an introduction to the M5, the simulator we augmented with fault injection capabilities and used in for experiments.

Chapter 5

Full System Simulation

5.1 Introduction

Full system simulation of a computer system is the process of modeling its physical components in such a high detail that any software for the targeted system can run unmodified on the virtual hardware. Simulated components can be anything from CPUs, cache memories, network connections or any other peripherals/modules that compose or interact with the SUT.

Due to these characteristics full-system simulators are able to run operating systems without the need of modifying them or their device drivers, thous expanding the capabilities in hardware and system software design.

We should note here that the models' fidelity can be of arbitrary level, however, the more detailed the model the smaller the system we can test (model detail $\Uparrow \rightarrow$ simulation time $\Uparrow$).

5.2 Simulation Attributes

Full system simulation is becoming more and more appealing for use in research. The advancements in simulation and testing theory have boosted the efficiency of simulators and the computational power explosion enabled us to simulate large systems on cheap personal computers.

Because of those two factors an increasing number of researchers (Table 5.1) choose to use simulators on their experiments.

However, these are not the only reasons behind the increase in the number of publications that use simulators. Since simulation is done using software it offers many advantages compared to the real machine environment (see section 4.1.3).

Jakob Engblom lists the attributes that make simulation appealing in [19]. We set forth this list here for completeness.

Configurability. Technological barriers or lack of physical/hardware resources do not place constrains on system configuration options.

Extendability. The simulator can be easily augmented with additional components without any limitations (i.e. GPU/DRAM slots).

Year	Total papers	Simulation
2009	43	39
2004	31	27
2001	25	22
1997	30	24
1993	32	23
1985	43	12
1973	28	2

Table 5.1: Performance evaluation methodologies in papers appearing in the Proceedings of the International Symposium on Computer Architecture. Adapted from [56]

Controllability. The execution of the simulation can be arbitrarily controlled, stopped, and restarted.

Determinism. A simulation is completely deterministic (assuming correct programming).

Checkpointing. The state of the simulated components can be saved and restored.

Availability. Creating a new machine is just a matter of copying the setup. There is no need to produce hardware prototypes or development boards.

Inspectability. The complete state of the simulator can be investigated and monitored without disturbing the execution.

Sandboxing. The simulation environment is completely isolated. No external variable can influence it and no code or data can escape unless explicitly allowed.

These attributes create an ideal environment for a system designer.

The coexistence of Configurability and Extendability expand the possible configurations for simulation beyond technological barriers. We can design software for hardware that is not yet in production (e.g. Linux was compatible with the AMD 64bit architecture before it became available [35]).

The attributes of Controlability, Determinism, Checkpointing and Inspectability create an ideal environment for debugging. Each flaw can be spotted, isolated and recreated until it is solved.

Finally, sandboxing erases any environmental causes that can influence a physical system and prevent disasters from failed experiments (i.e. aircraft computer systems, nuclear factory sensors).

A major feature for both the industrial and academic community is the availability of the simulated system. After an initial cost for creating the setup, duplication comes with zero cost in time or budget. This can also be used to mitigate the time overhead of simulation by running multiple experiments in parallel. In addition, the development of a software description of the system is much faster than an actual physical implementation, thus the whole design process is boosted.

5.3 An overview of the M5 simulator

For the purposes of our research we used and extended the M5 simulator [16, 11]. M5 is described from its main site as *"a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture"*.

M5 is a full-system simulator that initiated at the University of Michigan. Until now it consists of around 180K *(C++ & Python)* lines of code, is freely distributed under a BSD-style license and has no dependency to any commercial or restrictive license software. These features make the M5 simulator an ideal tool for research on an academic environment and a great contribution to free and open source software.

As a framework, M5 is an event-driven simulator centered around a main event queue where all modules schedule their events. The implementation is heavily object oriented; all simulated components (i.e. CPUs, Caches) as well as their internal structures are instantiated as objects. This is the key to M5's modularity and flexibility.

M5 uses *Python* to initialize the simulation and configure the scheme of the SUT. For this to be feasible all simulated objects developed in *C++* are wrapped using *SWIG*[1] and exported as *Python* interfaces for use on the system configuration process. The configuration scripts are semi declarative, hiding all the unnecessary information about how the modules connect and communicate. The simulators capabilities scale from uni-/multi- processor system to multi-system systems without much overhead for creating the configuration.

Along with the scalability of configurations we can also choose from multiple object implementations based on our need for accuracy. For example, we can choose from 3 different CPU models, an 1 CPI CPU with atomic memory accesses, an 1 CPI CPU with detailed memory accesses and a full OoO Superscalar CPU based on the ALPHA 21264 processor.

Apart from the level of abstraction in the implementation, M5's uniform (method-based) API across object types enables us to interchange similar simulated objects. The simulation can start with a simple, functional CPU (so as to speed up the initialization process of the experiment) and then change into a more detailed model from which we will acquire our statistics and results. Moreover, M5 features a detailed event-driven memory system able to model complex multi-level cache hierarchies coupled with a coherence protocol implementation. Together, all these modules give a vast amount of possible composition of custom/complex systems for simulation.

In order to enable the easy expansion of its capabilities the simulator has been developed from the start so that it would not be coupled with any ISA increasing the number of applications and OS that could be used with it. Currently, the Alpha, SPARC, MIPS, ARM and x86 ISAs are supported.

Finally, one of M5's most appealing features is its full system simulation capabilities (M5 successfully simulates systems running an unmodified version of the Linux kernel). With full system simulation we are able to perform realistic simulation of the concurrent execution of applications on top of an operating system; in our case, to get a more detailed insight of specific parts of applications, the OS and their behavior under the influence of faults.

[1]Simplified Wrapper and Interface Generator: A software development tool that connects programs written in C and C++ with a variety of high-level programming languages

All the above features conveyed us in choosing M5 as the most suitable simulator for experimenting and implementing our fault injection framework. Even though other full system simulators support fault injection, our framework is unique in that it covers a substantial amount of errors and provides high injection precision. Also, it is provided under a free-software license enabling modification and experimentation by anyone.

Chapter 6

Generic Processor Fault Model

After explaining the reasons for choosing M5 as the base for our fault injection framework, in this section we provided an overview of the fault model on which we based our implementation. In general, proving the sufficiency of a fault model is very difficult. It is more realistic to assume that a fault model is sufficient and justify this assumption to the greatest extent possible with experimental and historical data or results published in literature.

To this end, Yount and Siewiorek [58, 57] developed a very generic fault model for the register file within a processor which Johnson, Cutringht and DeLong [18] have augmented through simulations [58, 57] so as to fully explain the results of their test.

The augmented generic behavioral-level fault model describe the faulty behavior of a general-purpose, implementation-independent processor like the one shown in Figure 6.1

The full model consists 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 only 4 of those locations for fault injection (1, 2, 3, 5). Following, a description of the attributes - where, when, what - of each fault model that we implemented will be given. Excerpts from the following text are taken from [18] where a detailed presentation of the model can be found.

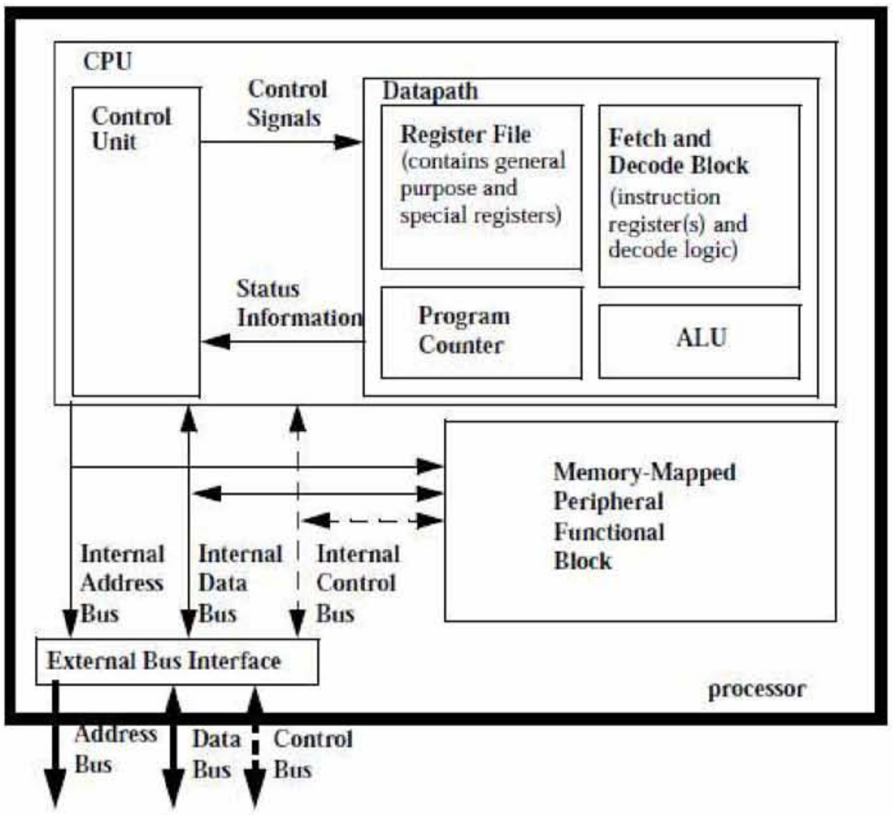

Figure 6.1: General processor diagram [18]

6.1 Register File Fault Model

6.1.1 Register Fault Model

The register fault model covers corruptions in the registers of the CPU. These can be general or special purpose registers.

Where

One or more registers of the CPU is/are potential candidate for corruption.

When

Corruption can occur at an instruction boundary.

What

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

1. **Missed load** (6.1) all or part of a register is not loaded when it should be.

$$R_k \leftarrow expr \Rightarrow R_k \leftarrow (expr \bigoplus (mask\langle(w-1)\ldots 0\rangle)) \tag{6.1}$$

2. **Extraneous load** (6.2) all or part of a register is loaded when it should not be.
$$R_k \leftarrow expr \Rightarrow R_j \leftarrow expr \quad \exists(j \neq k) \tag{6.2}$$
3. **Level change in storage** (6.3) the value of one or more bits in the register is complemented.
$$R_k \Rightarrow R_k \bigoplus (mask\langle (w\text{–}1) \ldots 0 \rangle) \tag{6.3}$$
4. **All 0/1** (6.4) Assign the value of all zeros and all ones to the register.
$$\begin{gathered} R_k \Rightarrow R_k \bigoplus R_k \\ R_k \Rightarrow R_k \bigoplus \overline{R_k} \end{gathered} \tag{6.4}$$

6.1.2 Read/Write Register Selection Fault Model

The read/write register selection fault model covers faults within the decoding stage of the pipeline where an error can result in selecting the wrong register to be used as an input/output operand to the current operation (6.5)(6.6).

$$\begin{gathered} (R_k \leftarrow R_i op R_j) \Rightarrow (R_k \leftarrow R_x op R_j) \quad \exists(x \neq i) \\ (R_k \leftarrow R_i op R_j) \Rightarrow (R_k \leftarrow R_i op R_x) \quad \exists(x \neq j) \end{gathered} \tag{6.5}$$

$$R_k \leftarrow R_i op R_j \Rightarrow (R_x \leftarrow R_i op R_j) \quad \exists(x \neq k) \tag{6.6}$$

Where

Instructions during the fetch and decoding stage.

When

Corruption can occur at an instruction boundary.

What

The corruption that will result to a read or write register selection error are shown in equations (6.7) and (6.8) respectively.

$$\begin{aligned} instr\_fetch(addr) \Rightarrow & \\ & instr\_fetch(addr) \\ & \bigoplus \\ & (((x - a - i)@0)\#mask\langle (i + a - 1) \ldots i \rangle \#(i@0)), \end{aligned} \tag{6.7}$$

$$\begin{aligned} instr\_fetch(addr) \Rightarrow & \\ & instr\_fetch(addr) \\ & \bigoplus \\ & (((x - b - j)@0)\#mask\langle (i + b - 1) \ldots i \rangle \#(j@0)), \end{aligned} \tag{6.8}$$

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.

6.2 Program Counter Fault Model

The Program Counter (PC) fault model covers corruptions of the Program Counter of a 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** (6.9) All or part of a register is not loaded when it should be.
$$PC \leftarrow expr \Rightarrow PC \leftarrow (expr \bigoplus (mask\langle (w-1)\dots 0\rangle)) \tag{6.9}$$

2. **Level change in storage** (6.10) The value of one or more bits in the register is complemented.
$$PC \Rightarrow PC \bigoplus (mask\langle (w-1)\dots 0\rangle) \tag{6.10}$$

3. **All 0/1** (6.11) Assign the value of all zeros and all ones to the register.
$$\begin{aligned} PC &\Rightarrow PC \bigoplus PC \\ PC &\Rightarrow PC \bigoplus \overline{PC} \end{aligned} \tag{6.11}$$

6.3 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 equations (6.7)

$$
\begin{aligned}
instr\_fetch(addr) \Rightarrow & \\
& instr\_fetch(addr) \\
& \bigoplus \\
& (((x-c-k)@0)\#mask\langle(k+c-1)\ldots k\rangle\#(k@0))
\end{aligned} \tag{6.12}
$$

where k is the starting position of the operation code field, and c is the width of the operation code. Note that values for k and c may vary, depending on the format of a given instruction.

6.4 ALU Fault Model

The ALU fault model covers corruptions in the ALU module of the processor, based on a general arithmetic instruction format such as (6.13)

$$D \leftarrow S_1 op S_2, \tag{6.13}$$

where D is the destination for the result of the operation (i.e. a register or a memory location) and S_1, S_2 are the sources for the operation again of the same possible types as the destination.

We define as possible locations for corruptions D , S_1 , and S_2 , as well as the operation itself, which could be corrupted to another valid instruction or an invalid instruction.

The behavior of the ALU fault model can be defined as a subset of the other fault models behavior and thus expressed through them.

Chapter 7

Implementation

In the introductory section we gave a brief overview of the reliability problems that arise with aggressive feature, voltage and frequency scaling and highlighted the importance of understanding how the function of each abstraction layer in computer systems is affected by such faults. The characterization and modeling of the errors will enable us to design and implement feasible solutions to preserve the current levels of system's robustness.

In order to assess the impact of faults to each layer, new tools need to be developed or the existing ones should be enhanced. An example of such a tool is M5, the full system, cycle-accurate simulator with broad acceptance in the area of performance analysis that we described in Section 5.3. To avoid re-inventing the wheel we augmented its existing infrastructure with fault injection capabilities, following the general processor fault model descibed in Chapter 6. The end-result is a modular and configurable framework for studying the effect of transient faults in reliability of applications. In addition, the developed framework can be used for the evaluation of new fault tolerance techniques or to calculate the robustness of a system through fault injection campaigns which can be automated using the provided tools and API.

The framework was developed using *C++* and employed the *SWIG* library for exporting the instrumentation API to the configuration interface of M5, thus creating a uniform development environment for the simulator and the fault injection framework. We should note here that fault injection is currently only supported in the ALPHA ISA. However, porting it to other ISAs does not require extensive modification as the only ISA-depended portion of the framework is the functions used to distinguishing processes/threads and user/privileged mode execution.

In the following Sections, we will attempt to give a general description of fault injection frameworks and present implementation details and choices that were made for the development of our tool.

7.1 Introduction: Where, When, What

Fault injection's instrumentation variables can be divided into three basic categories[23]:

1. Where: The location of the injected fault.

2. When: The time when the fault should manifest.
3. What: The nature of the fault; the way that the faulted structure will be influenced.

7.1.1 Where

The first aspect that we need to clarify when creating a fault injection scenario is the location of the faults, namely the modules/structures that will be targeted. A good method for selecting injection targets is the top-down approach; that is, we first select the high-level unit (CPU, DRAM module), then we proceed with internal modules and finally pick a single bit that will be affected. Common locations for fault injection are the levels of memory hierarchy, special and general purpose registers, control logic and pipeline stages of a processor (fetch, decoding and execution).

The location where a fault will be injected is crucial, as it partially defines/bounds the possible errors that can be created[24]. As different modules have different functions and process different types of data, we can create groups that contain all the possible errors for each structure, however, we should note that some interleaving does exist between errors in different structures.

A fault occurring in a register (integer, floating point or special purpose) will affect its internal storage and in the case of miscellaneous register the processor state (e.g. corruption of the processor's execution mode register).

A fault at a memory location will affect its internal storage and possibly the execution and output of a process that uses its content.

A fault at the fetch and decoding stage will affect the decoding of the instruction (i.e. register selection, operation selection (e.g. Table 7.1), immediate value).

A fault at the execution stage will affect the result of the ALU operation and the content of the destination register or memory location, or -in the case of program control instruction- a wrong path can be taken.

Mnemonic	Format	Opcode (hex)	Function Code (hex)	Description
BEQ	Bra	39	–	Branch if = zero
BGE	Bra	3E	–	Branch if >= zero
BGT	Bra	3F	–	Branch if > zero
BIC	Opr	11	08	Bit clear
BIS	Opr	11	20	Logical sum
BLBC	Bra	38	–	Branch if low bit clear

Table 7.1: Qualifiers for operate instructions, excerpt from Table A-2[17]

In general, the location of an injection is described deterministically and the spatial distribution of faults in the simulated modules is produced by a distribution function. The distribution function can be derived from low level characterization of the hardware's behavior under different internal and external conditions (i.e. radiation, thermal distribution, utilization). Alternatively, statistical fault injection can be used as shown in [37]. Note that most fault injection frameworks are evaluated through statistical fault injection and not by strict mathematical proof of their correctness.

7.1.2 When

Another essential variable in a fault injection campaign is the timing of the injections; the time that a fault will occur (manifest).

Faults can be set to occur based on the value of a system variable. Commonly used variables are overall simulation time, processor cycles, executed/fetched instructions or occurrence of a specific addresses in the Program Counter (PC).

The flexibility in setting the manifestation time of a fault is of great importance when one is interested in acquiring targeted results; for example, when we are interested in studying specific regions of an application.

However, fault manifestation based on simulation time, clock cycles and fetched instructions is not deterministic especially when done on a full system multi-core simulation environment, where multiple applications are running and no control over their scheduling in time and hardware is possible. As a more precise timing method the PC value can be used.

Similarly to the location variable, the timing variable of the injection can be generated from temporal distribution models of hardware faults (module error rate) or statistical methods (e.g. uniform, logarithmic distribution function).

7.1.3 What

The final fault injection instrumentation variable that we will describe is the nature of the fault; 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[30] where a signal is permanently set (stuck) to one (1) or zero (0) (Figure 7.1). Transient faults are modeled using the bit-flip model[41] where a bit's value is flipped to its complement (Figure 7.2).

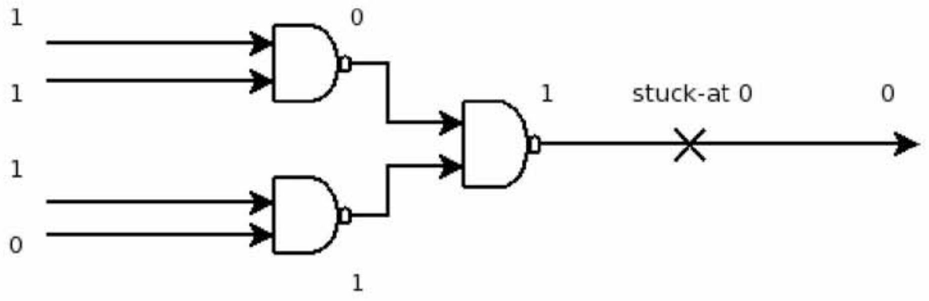

Figure 7.1: Stuck-at model illustration

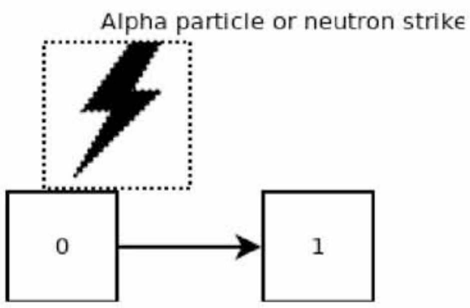

Figure 7.2: Bit-flip model illustration [41]

Except from these two methods, more accurate models can be used for modeling hardware faults. Recent research results show an increase in multi-bit [21] and circuit timing errors that are not covered by the above fault models.

7.2 Our Implementation

Having explained the aspects of fault injection instrumentation, we will try to place our implementation details in the general picture. For the development of our tool we abhered to the object oriented approach of the M5 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.

The hierarchy of fault types that our framework currently supports is depicted in Figure 7.3. All fault objects derive from the *InjectedFault* class that contains the basic variables for fault injection and the generic attributes of a fault. The *InjectedFault* class derives from the *SimObject/MemObject* classes so that we can use the fault injection objects at the configuration interface. More details about each fault type will be given later.

A core component of the fault injection framework is the queue structure. All faults described at the configuration file are inserted in 4 queues, based on their characteristics; execution, fetch, decode and other. The queues are constructed at the beginning of the simulation and are global to the framework and the simulator. The queue class provides public functions for inserting, removing and searching a fault (based on injection time and location). In addition, to improve performance the faults are kept in a descending order to decrease the average search time.

At the initialization of the simulation the configuration file is parsed and simulation objects are created for all described components. All faults that can be scheduled at that point are send in the global event queue of the simulator or a processor's instruction event queue. Faults that manifest on a specific cycle are pushed into the ***mainEventQueue*** and faults that manifest based on a CPU's fetched instruction count are pushed into the CPU's ***comInstEventQueue***. All other instances are kept in the queues that are scanned in every cycle to find if any fault instance is ready to manifest. The scanning of the queues is done independently for each pipeline stage in order to support injection in-between them. This is the main reason we altered the source code of the CPU models, even though are initial approach was to avoid any changes to existing M5 code.

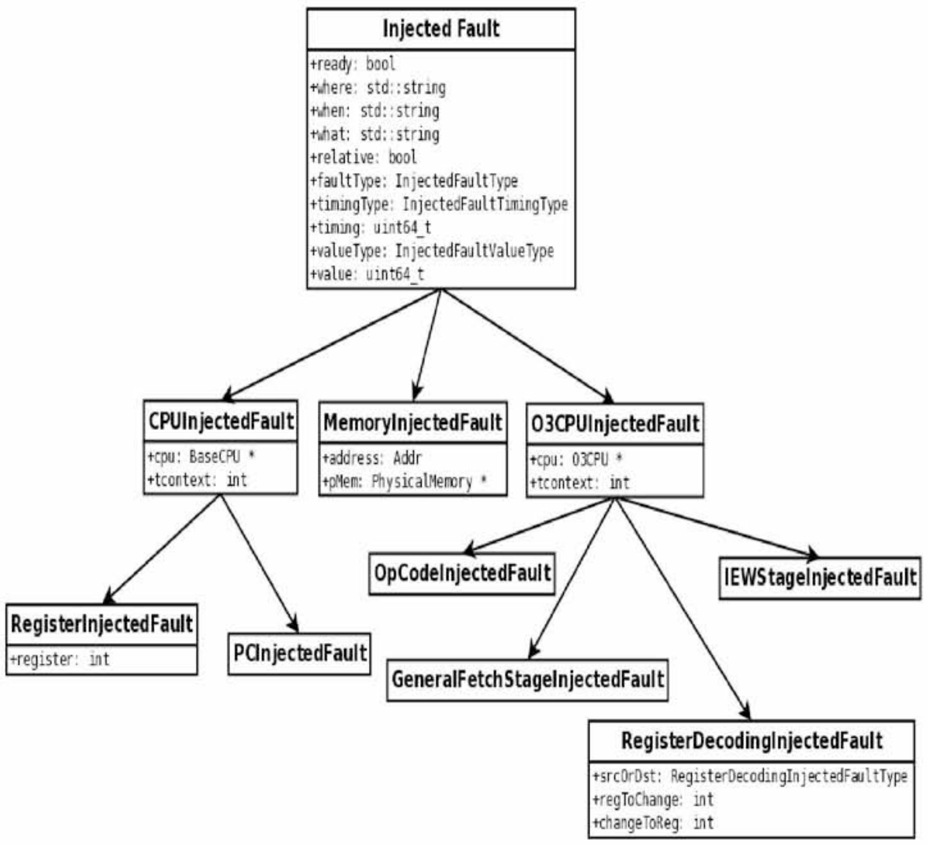

Figure 7.3: Fault classes

Another feature of our framework is "Relative Fault Injection", that is, faults can be set to manifest relatively to the value of a processor's variable. To enable relative fault injection an extra instruction has been added to the ISA, namely ***fi_vaddr_inst***[1].

The ***fi_vaddr_inst*** instruction is used to set a relative point for fault injection. When executed by the processor it stores the PC address, Tick cycles and fetched instructions number so that these values can be used as reference points in the injection of relative faults.

At the section where we described the timing of an injection (When) we mentioned the importance/need of bounding faults' occurrence. In our implementation this is supported through the use of another instruction that was introduced to the ISA, ***fi_activate_inst***[2], and a function of the M5 simulator namely ***inUserMode()***.

The ***fi_activate_inst*** is used to enable and disable the manifestation of faults. More specifically, when inserted in a code its first occurrence enables the manifestation of faults for the specific process/thread and the second one disables it. To achieve such behavior we exploit the Process Control Block

[1] ***fi_vaddr_inst*** → asm(" .long 0x04XXXX67")[3]

[2] ***fi_activate_inst*** → asm(".long 0x04XXXX66")[3]

[3] The Xs can be of any value.

(PCB) address [3], which is unique for each process/thread. Whenever a fault event tries to manifest, we search using the PCB address as a key on a hash table that contains the processes/threads for which fault injection has been enabled.

For the needs of our experiments we used an already implemented function of the M5 simulator, the ***inUserMode()*** function. ***inUserMode()*** returns true if the processor is currently in user mode, thus enabling us to restrict faults in user mode application code; in the same way we are able to inject faults only to operating system code. As we were not interested in injecting faults in the OS, by-default our framework isolates fault injection to the user mode of the processor. However, this feature can easily became a run-time argument of the configuration file.

7.2.1 Where

As described previously (Chapter 6), our implementation is based on the generic processor fault model described in [18]. It supports fault injection in memory locations, registers and the fetch, decode and execution stages of an instruction.

All the possible locations for fault injection, according to [18], are shown in table 7.2; the supported locations are marked with a $\surd$.

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

Table 7.2: Modules where fault injection is supported

A fault location in our framework is described in a deterministic way, however, statistical fault injection is easily implemented at a higher level through use of python scripts and the configuration file. As a proof of concept, we have automated statistical fault injection campaigns through python scripts for evaluating our framework with real world applications; the statistical model we used is described in [37].

7.2.2 When

Our 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 time can be either absolute (i.e. global from the beginning of the simulation) or relative to a simulation milestone.

[3]In the case of ALPHA ISA the pointer to the PCB is stored on the IPR_PALtemp32 register

Another supported timing feature is the enabling and disabling of the manifestation of fault using a "magic instruction", ***fi_activate_inst***, mentioned previously.

Except from the timing of the injection, we can also specify the duration of the fault, enabling the emulation of intermittent and permanent faults. However, even though a fault's first occurrence can be given with any of the three available timing methods, faults that use the PC address as a trigger can only be used when experimenting with repeated corruption when a specific instruction is fetched. This limitation is necessary due to branches. In the presence of a branch instruction we do not know its result so as to set the correct consequent PC value in which the fault should manifest. Nevertheless, this does not reduce the capabilities of the framework as the PC Address timing option is ideal in studying repeated corruptions of a specific data/computation; for example, the upper-left cell of the iDCT matrix.

It is also worth noting here that memory fault injection can only be used with absolute timing. Otherwise it should have been coupled with a specific CPU.

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

Table 7.3: Trigger Mechanisms for each Fault Type

7.2.3 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.

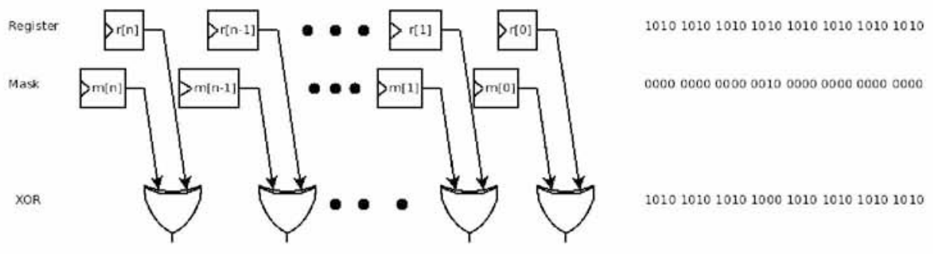

Figure 7.4: XOR example

Bit-Flip Change the specified bit to its complementary value.

All0 Set all bits to zero (0).

All1 Set all bits to one (1).

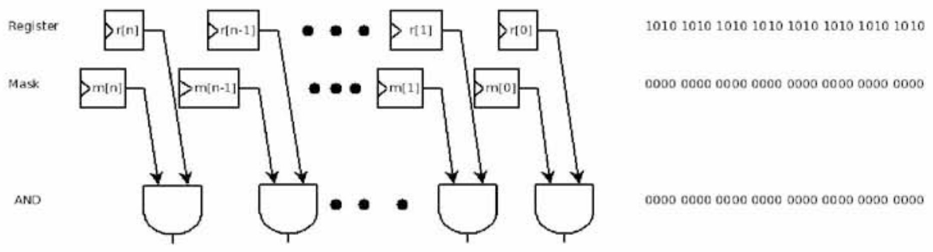

Figure 7.5: AND example

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.

7.3 Usage/Export to M5 configuration file

As we have previously mentioned, the instrumentation API of the fault injection framework is exported at the M5 configuration interface using SWIG. A fault can be considered as another simulated object that thus we describe in the configuration file. A reference, example, configuration file can be found in the tarball provided with this Thesis.

Following, we give examples of different fault types and how they would have been defined in the configuration script.

The mandatory fields for all faults are:

Where: In which CPU/Memory to inject the fault (format: ¡module's name at the configuration script¿).

When: When to inject the fault (format: ¡timingType:timingValue¿).

What: What value should be injected and how (format: ¡valueType:value¿).

and the optional fields are:

Relative: Is the fault injection timing relative to a "magic instruction"? By default false.

Occurrence: How many times should the fault manifest? By default one (1) - transient fault.

7.3.1 Register Injected Fault Configuration

The additional required fields for a register fault are:

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

Register: which register should be injected.

Examples:

1. Inject a permanent fault at the first (1) integer register of CPU "system.cpu" when the PC is 8 + (PC @ magic instruction). After the fault the register should contain the value 57005.

```
RegisterInjectedFault(RegType = "int",
                      Register = 1,
                      where = "system.cpu",
                      when = "Addr:8",
                      what = "Immd:57005",
                      relative = True,
                      occurrence = 0)
```

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

```
RegisterInjectedFault(RegType = "float",
                      Register = 1,
                      where = "system.cpu",
                      when = "Tick:50000",
                      what = "Mask:57005")
```

3. Inject an intermittent fault at the first (1) miscellaneous register of CPU "system.cpu" when the total fetched instructions are 1984 + (fetched instructions @ magic instruction). After the fault the register should contain the value 0.

```
RegisterInjectedFault(RegType = "misc",
                      Register = 1,
                      where = "system.cpu",
                      when = "Inst:1984",
                      what = "Immd:0",
                      relative = True,
                      occurrence = 3)
```

7.3.2 PC Injected Fault Configuration

PC faults do not require any additional field.
Examples:

1. Inject a fault at the PC register of CPU "system.cpu" 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(where = "system.cpu",
                when = "Addr:4831838348",
                what = "Mask:2")
```

7.3.3 Memory Injected Fault Configuration

Memory faults do not require any additional field.
Examples:

1. Inject a fault at address 512 of memory module "system.physmem" when the total amount of simulation ticks is 2000. After the fault the address should contain the value 3.

```
MemoryInjectedFault(address = 512,
                    where = "system.physmem",
                    when = "Tick:2000",
                    what = "Immd:3")
```

7.3.4 Fetch Stage Injected Fault Configuration

Fetch stage injected faults can either be "general", in the whole bitwidth of a fetched instruction, or targeted at the Opcode.
Examples:

1. Inject a fault at the fetched instruction of CPU "system.cpu" in the 45.000 tick. After the fault the instruction should be 540999681 — hex(203F0001).

```
GeneralFetchInjectedFault(where = "system.cpu",
                          when = "Tick:45000",
                          what = "Immd:540999681")
```

2. Inject a fault at the Opcode of the fourth (4) fetched instruction of CPU "system.cpu". After the fault the instruction's Opcode should be 32 — hex(20).

```
OpCodeInjectedFault( where = "system.cpu",
                     when = "Inst:4",
                     what = "Immd:32")
```

7.3.5 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¿.

Example:

1. Change the destination register zero (0), of the instruction that will be decoded in the 45.000 simulation tick at CPU "system.cpu", to one (1).

```
RegisterDecodingInjectedFault(regDec = "Dst:0:1",
                              where = "system.cpu",
                              when = "Tick:45000",
                              what = "Immd:0")
```

7.3.6 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 relative (to the magic instruction) instruction of the CPU "system.cpu", the result should be 10.

```
IEWStageInjectedFault(where = "system.cpu",
                      when = "Inst:10",
                      what = "Immd:10",
                      relative = True)
```

7.3.7 Monitoring

When running fault injection campaigns it is essential to have the ability to observe how a fault influences the function of the system under test. In M5 simulation can be monitored through enabling various trace/debug flags. The simulator by-default provides an excessive amount of options on what simulation events should it print as output throughout the simulation. In this way fetching, decoding, execution, memory and register accesses can be set to be printed out whenever they happen, creating a full trace of the application's execution.

We have expanded the monitoring capabilities of the simulator by introducing a new trace/debug flag called ***FaultInjection***, used for creating output for the fault injection framework. This consists of printing the creation of a fault and the manifestation of it, together with the attributes of the fault and the way that it influenced the targeted structure.

Chapter 8

Fault Injection Campaigns/ Experiments

Even though the main target of this Thesis was the creation of a fault injection framework, our final goal is an analysis of the behavior of applications in the presence of faults. Therefore, we conducted a series of fault injection campaigns in order to validate the fault injection functionality and at the same time to perform a preliminary evaluation of application behavior in the presence of faults. This Section presents the results of these experiments that targeted two (2) applications: a simple matrix multiplication kernel and the AVS decoder.

8.1 Experimentation Methodology

In our experiments we used two (2) different system configurations. Their main difference lies in the CPU model's abstraction level/detail.

Both applications have been tested using statistical fault injection in order to narrow the possible injection configurations, yet maintain a good confidence interval on our results. Still, the volume of the needed experiments required the automation of the experimentation and results analysis process; for that, additional python and bash scripts were created.

In addition to the statistical fault injection, in the case of AVS we targeted specific functions to assess their inherent reliability and to test the existence of variations in fault-tolerance between portions of the application.

To acquire the instrumentation variables for the statistical fault injection campaigns we assumed that faults in all structures and bits are equally possible. In addition, we assumed that fault occurrences in time follow a uniform distribution [37]. Finally, in each simulation a single fault was injected.

All programs were compiled using a cross-compiler, version 4.3.2 of gcc for the Alpha ISA, without any optimization options enabled. The compiler can be found at Alpha ISA:gcc-4.3.2, glibc-2.6.1 (NPTL,x86/64) or a new one can be created following the instructions at crosstool.

8.2 Statistical Fault Injection in Matrix Multiplication

This Section presents the results from our first fault injection campaign. The faults were injected in the matrix multiplication kernel, which was executed in the detailed system configuration.

The fault injection campaign was composed of 2450 experiments. Each experiment contained a single fault injection, whose instrumentation variables (i.e. Where, When, What) were provided by a uniform distribution function (Python implementation of Mersenne Twister). In Table 8.1 we see the accumulated outcome of the experiments. Table 8.2 provides a more detailed view of the same results. The results are also given as pie-charts in Figure 8.1.

No Diff(Activated)	1912	78.04%
Diff	105	4.29%
Panic	345	14.08%
Infinite	1	0.04%
No Output	4	0.16%
Stack Overflow	13	0.53%
Unimplemented	2	0.08%
Not Activated	23	0.94%
No Diff(NotActivated)	45	1.84%
Total	**2450**	**100%**

Table 8.1: (MM)Minimal: Fault Injection Outcomes.

	Fetch	Execute	PC	Register	**Total**	%
No Diff(Activated)	8	17	6	1881	**1912**	78.04%
Diff	17	22	2	64	**105**	4.29%
Panic	13	33	60	239	**345**	14.08%
Infinite	0	0	0	1	**1**	0.04%
No Output	0	0	2	2	**4**	0.16%
Stack Overflow	0	0	0	13	**13**	0.53%
Unimplemented	0	0	2	0	**2**	0.08%
Not Activated	0	0	2	21	**23**	0.94%
No Diff(NotActivated)	38	7	0	0	**45**	1.84%
Total	**76**	**79**	**74**	**2221**	**2450**	100%
%	3.1%	3.22%	3.02%	90.65%	100%	

Table 8.2: (MM)Detailed: Fault Injection Outcomes Per Structure.

Description of row data for Tables 8.2 and 8.1.

No Diff(Activated) The fault manifested but did not create any user-visible error.

Diff The fault created a user-visible error but was not detected (SDC).

Panic The fault created an error that was caught by the system's error detection mechanisms and resulted in a crash (DUE).

Infinite The fault created an error that had as a result the infinite execution of the application.

No Output The fault created an error that had as a result the absence of an output (the program terminated properly).

Stack Overflow The fault created an error that had as a result the overflow of the stack structure.

Unimplemented The fault created an error due to an unimplemented function of the simulator.

No Diff(NotActivated) The error did not manifest. Either because the fault injection framework was disabled or the CPU was not in User Mode.

From Table 8.1 and Figure 8.1 (a) we can observe that almost 80% of the injected faults were masked; that is, they did not produce any visible errors in the user abstraction layer. The rest 18% (2% of fault were not activated[1]) of injected faults create user-visible errors that can be split into two basic categories: Detected Unrecoverable Errors (DUE) and Silent Data Corruption (SDC). Their percentiles are 14% and 4% respectively; for more details on the definition of the fault types please see Section 3.3 and [41].

The results create a strong impression that the Matrix Multiplication is acceptably fault-tolerant, considering the absence of a reliability mechanism. However, if we separate the results based on the structure that was corrupted, we can see that the robustness of the structures exhibit great variance. Table 8.2 and Figures 8.1 (b), (c), (d) and (e) provide the per structure outcomes.

We observe that the register file is the most fault-tolerant of all structures. 85% of faults produce no user-visible error. This is a result of fault masking through register rewriting, unused registers, registers that were injected after their last use and logical masking (e.g. an AND operation that will keep only a part of the original value of the register). Another important aspect that improves the robustness of the register file is the bit-width that has been selected during coding and compilation for the application's variables. The program manipulates 32bit integer variables, which makes part of faults injected between the 64th and 33rd bit ineffective.

On the other hand, the rest of the injected structures — led by the PC register with 89% of injections resulting in user-visible errors and followed by the Fetch and Execution stage of the pipeline — are less fault-tolerant. If we disregard the experiments in which the faults did not manifest (notActivated), the percentage of PC register, Fetch and Execution stage experiments that resulted in erroneous execution equals 80% on average .

The PC register is the most likely structure to create an unrecoverable error as all of its bits are important and its value is needed on every cycle. A corruption of the PC can create an error by moving the execution flow before (re-executing part of the instructions[2]) or after the correct execution address (skipping part of the application, e.g. the computational part) or even point at an unmapped address space.

[1]Not Activated faults are faults that either were set to manifest when the CPU was operating in non User Mode(not allowed by default to our framework) or out of our region of interest

[2]However, this does not always result in a user-visible error

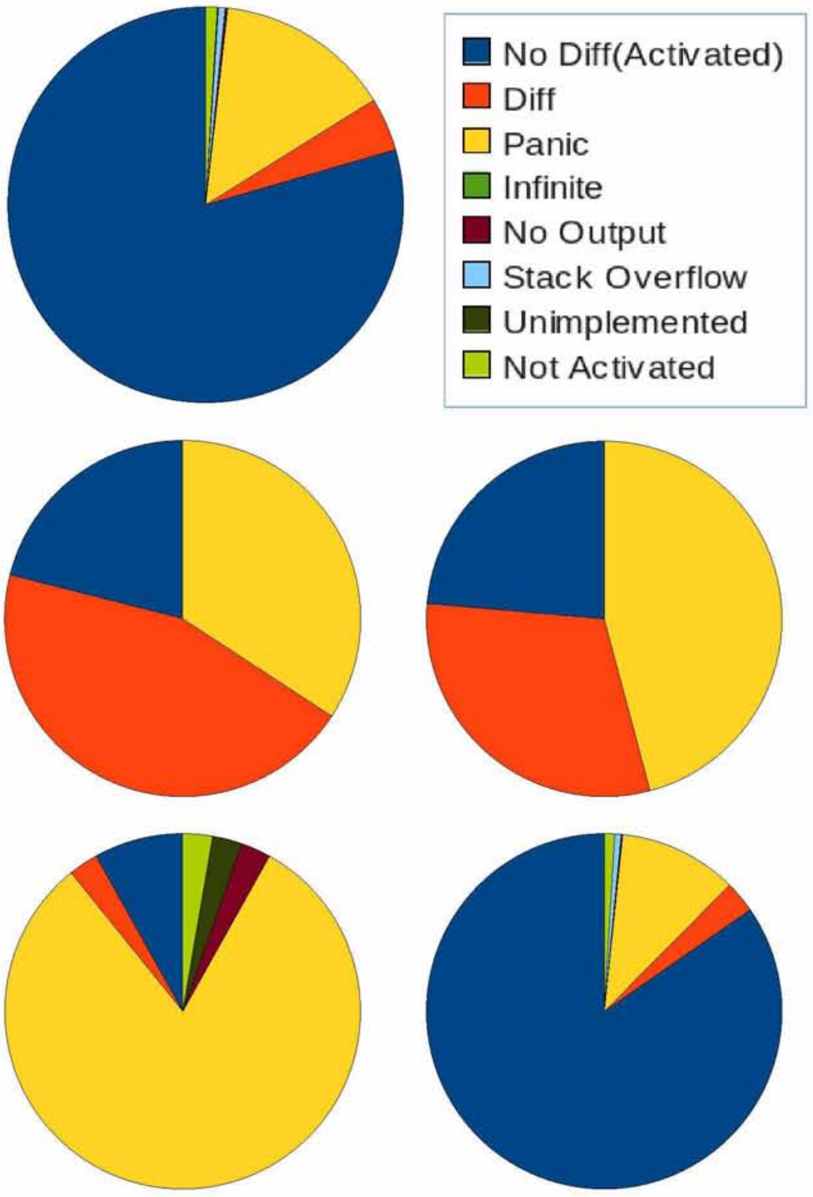

Figure 8.1: (MM) Behavior under fault injection (w/out notActivated). (a) Total, (b) Fetch, (c) Execute, (d) PC, (e) Int Registers

In the faults that were injected in the Execution and Fetch stage we can see a similar behavior to that observed for the PC. On average, 75% of the injected faults create a SDC or DUE error. In regard to the fetch stage, possible corruptions can either change the immediate value, registers that would be used (source/destination) and the function or operation code, altering the instruction's behavior or creating an unrecognizable value. As for the execution stage, we observe that the faults mainly produce SDC on the output of the program or panic due to accessing unmapped addresses. This can be coupled with the nature of the code that it is used, as it contains a lot of computations that directly influence the output and manipulates all values through load/stores in the memory.

An outcome that we did not mention earlier is the case of infinite execution. In this case, an injected fault corrupts a value in such a way that the application never reaches termination. This is an example of how a fault can influence the run-time of an application. Other possible ways in which the run-time or performance can be effected is in the presence of fault-tolerance techniques; fault-tolerance techniques are known to create extra overhead, something that can be quantified through the simulator performance statistics.

You can see more graphs and data related to the results of the statistical fault injection in the MM kernel in Appendix A.1.

8.3 Statistical Fault Injection In The AVS Decoder

The second application with which we experimented was the AVS Decoder[27]; an H.264/ACC competitor video compression standard. Our first approach was similar to the MM. We created a random fault injection campaign, using the previously mentioned uniform distribution function, to get a general picture of the application's behavior in an unreliable environment. As a second step, we experimented with targeted fault injection to assess the fault-tolerance of specific functions and modules. All the injections were done in integer registers, as the AVS Decoder lacks floating point calculations.

In the configuration of the statistical fault injection campaign a detailed out-of-order CPU was used, combined with a 2 level cache hierarchy and a physical memory module.

No Diff(Activated)	1135	67%
Diff	163	9.62%
Panic	396	23.38%
Infinite	0	0%
No Output	0	0%
Stack Overflow	0	0%
Unimplemented	0	0%
Not Activated	0	0%
Total	**1694**	**100%**

Table 8.3: (AVS)Fault Injection Outcome.

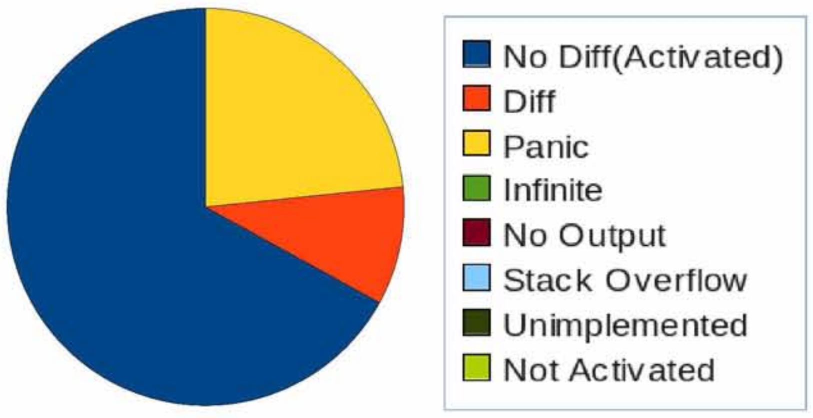

Figure 8.2: (AVS)Total Outcomes Chart

In Table 8.3 and Figure 8.2 we summarize the outcome of the random fault injection campaign for the AVS decoder. We can see that compared to the MM application AVS is less fault-tolerant as the percentage of incorrect executions is almost double of MM, reaching 33% of the overall outcomes. This can be justified by the fact that AVS is a more complex application, facilitating all of the available registers and containing a lot of control instructions and memory accesses that can create a DUE when corrupted. We can observe the influence of more memory accesses in the reliability of the application by observing that the number of errors due to attempts of accessing unmapped addresses is double (compared to the MM number) even though we conducted less experiments with the AVS decoder.

8.4 Targeted Fault Injection In The AVS Decoder

Except from the random fault injection in the AVS decoder, we also executed targeted fault injection campaigns to assess the fault-tolerance of different code modules.

Through these experiments we wanted to assess the correctness of our assumption that there exist code segments that do not require reliable execution or at least not on their entirety. This assumption has also been presented and evaluated by other research groups [33, 48].

For the execution of these experiments we used a "simple" CPU model coupled with a physical memory module; no cache memory was simulated.

Due to timing constrains (each experiment required 4 hours even when using the simple timing simulation objects) we only targeted reading of the inverse transformation matrix, motion vector decoding, header segments of I,P and B frames, the sequence header and the chroma and luma coefficients.

The fault environment we wanted to simulate was a heavily unreliable one, where faults would be continuously injected on the targeted functions.

In the experiments targeting the inverse transformation we injected faults in the upper-left and lower-right cells of the matrices in order to prove that data or computational parts can be categorized based on their impact in the quality of the application's output. As expected, the injections only affected the quality of the produced video (SDC errors) and verified our assumption. The videos in which the upper-left cell (containing the most significant part of information) was corrupted exhibited worse quality compared to the same corruption in the lower-right cell. For the quality comparison two metrics were used: *Mean Square Error (MSE)* and *Peak Signal to Noise Ratio (PSNR)* (see Appendix A.5).

Even though the quality of the video is affected in a more negligible way when corrupting less significant data and computations, we observed that it can still make the output unacceptable for use. A better approach would be the use of a default value (acquired through offline statistical sampling). In the case of inverse transformation, when pervasive errors are present in the computation of the lower-right cell of the matrix, it may be better to skip the computation and use as a default value zero (0). This would affect the produced output in a more favorable way.

Having experimented with a fault-tolerant[3] function, we proceeded to the assessment of a code segment that we expected to be non-fault-tolerant: the sequence and frame header data. In these experiments we corrupt a bit of the read value to emulate faults during the processing of data. Even though the header data were generally non-fault-tolerant this behavior did not cover 100% of the cases. Due to reserved bits and/or unused field, we recorded cases where faults did not manifest to the user visible output. Together with the header data we tested the fault-tolerance of the chroma and luma coefficients reading functions. These also prooved unable to sustain multiple faults.

The last segment we selected to test in our experiments is the motion vector (MV) values. The injections on the motion vector values did not create any DUE error, however, SDCs were produced and the quality of the image degraded due to missing/blank macroblocks. We should note here that the periodic occurrence of I frames in the video encoding had a positive influence on the video's quality, as MV values are computed using the last I frame.

For a graphical representation of the experimental outcome and additional data, we refer you to Section A.4 of this chapter's appendices.

[3]in terms that the program did not crash

Appendix A

A

A.1 Statistical Fault Injection in Matrix Multiplication — Additional Data

This section contains percentile data for the inter-fault-type outcomes.

	Fetch	Execute	PC	Register
No Diff(Activated)	10.53%	21.52%	8.11%	84.69%
SDC	22.37%	27.85%	2.7%	2.88%
Panic	17.11%	41.77%	81.08%	10.76%
Infinite	0%	0%	0%	0.05%
No Output	0%	0%	2.7%	0.09%
Stack Overflow	0%	0%	0%	0.59%
Unimplemented	0%	0%	2.7%	0%
Not Activated	0%	0%	2.7%	0.95%
No Diff(NotActivated)	50%	8.86%	0%	0%

Table A.1: (MM)Detailed: Inter-Fault-Type Outcomes (Percentiles)

A.2 Statistical Fault Injection in Matrix Multiplication — Overview of DUE errors

This section provides a more detailed overview of DUE errors and their causes for the statistical fault injection campaign in the Matrix Multiplication program.

The possible causes for a panic (sudden termination of the program) are:

1. Attempt to access unmapped address
2. Attempt to execute unknown instruction
3. dfault (Data stream fault or sign check error on virtual address)
4. Attempt to execute unmapped address
5. unalign (Data stream unaligned reference)

6. iaccvio (Instruction stream access violation or sign check error on PC)

	Fetch	Execute	PC	Register	**Total**
Tried to access unmapped address	3	24	5	127	**159**
Attempt to execute unknown instruction	4	0	3	0	**7**
dfault	0	8	1	88	**97**
Tried to execute unmapped address	2	0	27	12	**41**
unalign	4	1	4	12	**21**
iaccvio	0	0	20	0	**20**
Total	**13**	**33**	**60**	**239**	**345**

Table A.2: (MM)Detailed: Panic Causes

A.3 Statistical Fault Injection in AVS — Overview of DUE errors

This section provides a more detailed overview of DUE errors and their causes for the statistical fault injection campaign in the AVS decoder.

Tried to access unmapped address	215
Attempt to execute unknown instruction	2
dfault	98
Tried to execute unmapped address	27
unalign	30
iaccvio	24
Total	**396**

Table A.3: (AVS)Detailed: Panic Causes

A.4 Targeted Fault Injection in AVS — Detailed Results

	1bit	2bit	4bit	8bit	16bit	32bit	64bit
header	F	F	F	F	C	C	C
picture_data	F	F	F	F	C	C	C
I_Picture_Header	F	X	X	X	X	X	C
PB_Picture_Header	F	F	F	F	F	F	C
SequenceHeader	X	X	X	F	F	F	C
readLumaCoeff	F	F	F	F	F	F	C
readCromaCoeff	F	F	F	F	F	F	C
I_Picture_Header – bbv_delay	C	C	C	C	C	C	C
I_Picture_Header – time_code_flag	F	F	F	F	F	F	C

I_Picture_Header – marker_bit	C	C	C	C	C	C	C
I_Picture_Header – picture_distance	F	F	X	X	X	C	C
I_Picture_Header – progressive_frame	F	F	F	F	F	F	C
I_Picture_Header – top_field_first	C	C	C	C	C	C	C
I_Picture_Header – repeat_first_field	C	C	C	C	C	C	C
I_Picture_Header – fixed_picture_qp	F	C	C	C	C	C	C
I_Picture_Header – picture_qp	X	X	X	C	C	X	C
I_Picture_Header – reserved_bits	C	C	C	C	C	C	C
I_Picture_Header – loop_filter_disable	X	X	X	X	X	X	C
PB_Picture_Header – bbv_delay	C	C	C	C	C	C	C
PB_Picture_Header – picture_coding_type	F	F	F	F	F	F	C
PB_Picture_Header – picture_distance	F	F	X	X	X	X	C
PB_Picture_Header – progressive_frame	F	F	F	F	F	F	C
PB_Picture_Header – top_field_first	C	C	C	C	C	C	C
PB_Picture_Header – repeat_first_field	C	C	C	C	C	C	C
PB_Picture_Header – fixed_picture_qp	F	C	C	C	C	C	C
PB_Picture_Header – picture_qp	X	X	X	C	F	F	C
PB_Picture_Header – no_forward_reference_flag	C	C	C	C	C	C	C
PB_Picture_Header – reserved_bits	C	C	C	C	C	C	C
PB_Picture_Header – skip_mode_flag	F	C	C	C	C	C	C
PB_Picture_Header – loop_filter_disable	X	X	X	X	X	X	C
SequenceHeader – profile_id	C	C	C	C	C	C	C
SequenceHeader – level_id	C	C	C	C	C	C	C
SequenceHeader – progressive_sequence	C	C	C	C	C	C	C
SequenceHeader – horizontal_size	X	X	X	F	F	F	C
SequenceHeader – vertical_size	X	X	X	F	F	F	C
SequenceHeader – chroma_format	C	C	C	C	C	C	C
SequenceHeader – sample_precision	C	C	C	C	C	C	C
SequenceHeader – aspect_ratio_information	C	C	C	C	C	C	C
SequenceHeader – frame_rate_code	C	C	C	C	C	C	C
SequenceHeader – bit_rate_lower	C	C	C	C	C	C	C
SequenceHeader – marker_bit	C	C	C	C	C	C	C
SequenceHeader – bit_rate_upper	C	C	C	C	C	C	C
SequenceHeader – low_delay	F	F	F	F	F	F	C
SequenceHeader – marker_bit	C	C	C	C	C	C	C
SequenceHeader – bbv_buffer_size	C	C	C	C	C	C	C
SequenceHeader – reseved_bits	C	C	C	C	C	C	C
motion_vectors_1	X	X	X	X	X	X	C
motion_vectors_2	X	X	X	X	X	X	C
motion_vectors_3	X	X	X	X	X	X	C

Table A.4: Targeted fault injection campaign in AVS results per section and faulted bit. X :SDC, F :Panic, C :Correct,

The above table presents the outcome for targeted fault injection in various functions of the AVS Decoder. The first rows contain the outcome for injection in all fields, if more than one, of the named function. The following rows present the outcome for injection only on the named field of the function. Finally, the last three rows contain the outcome for fault injection in the motion vectors.

A.5 Example Frames with SDC

This section contains images from experiments that produced Silent Data Corruptions (SDC) on the output of the AVS decoder.

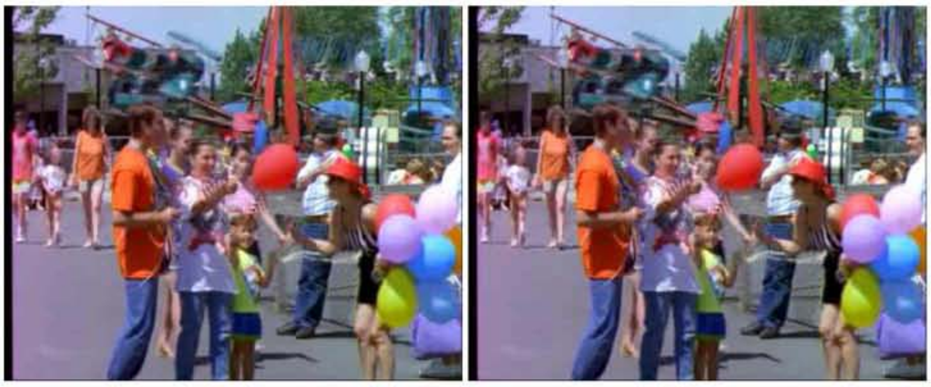

Figure A.1: Example outputs for fault injection in the AVS inverse transformation function. Left: top-left cell, right: bottom-right cell. 32nd least significant bit flipped. No error produced as the idct matrix manipulates short(16bit) values.

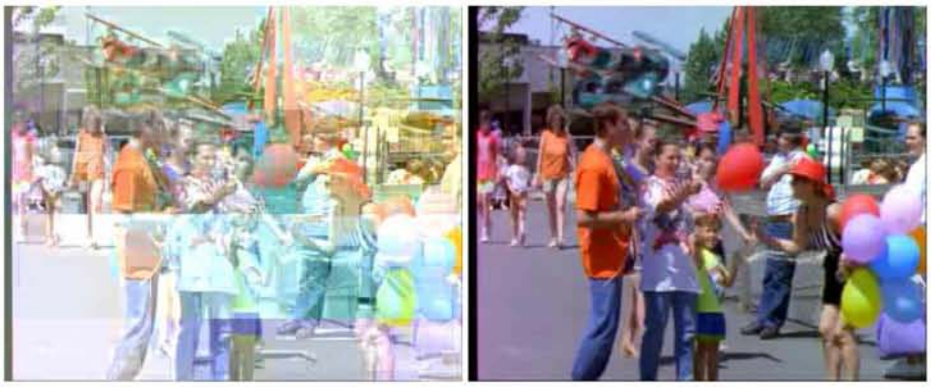

Figure A.2: Example output for fault injection in the AVS inverse transformation function. Left: top-left cell, right: bottom-right cell. All bits set to 0.

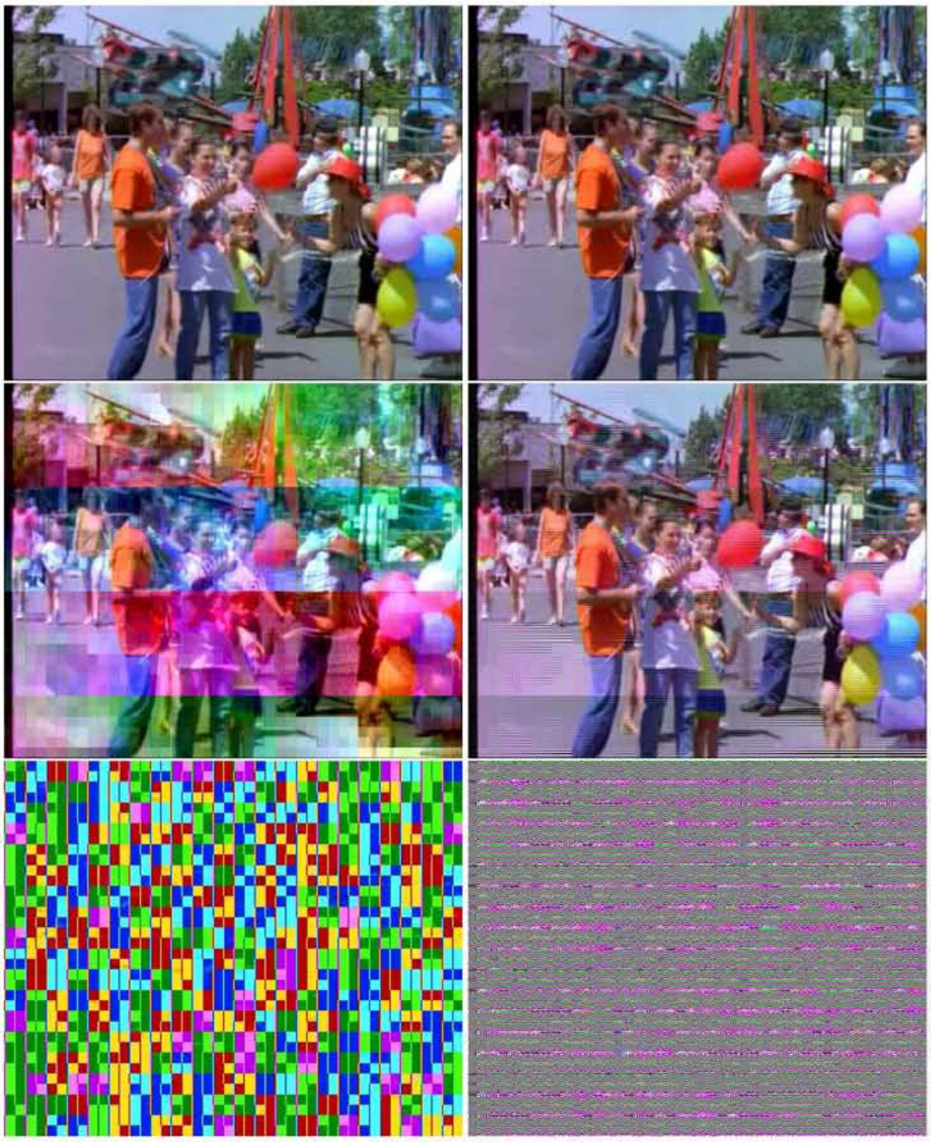

Figure A.3: Example outputs for fault injection in the AVS inverse transformation function. Left: top-left cell, right: bottom-right cell. Row 1, least significant bit (LSB) flipped. Row 2, 8th LSB flipped. Row 3, 16th LSB flipped.

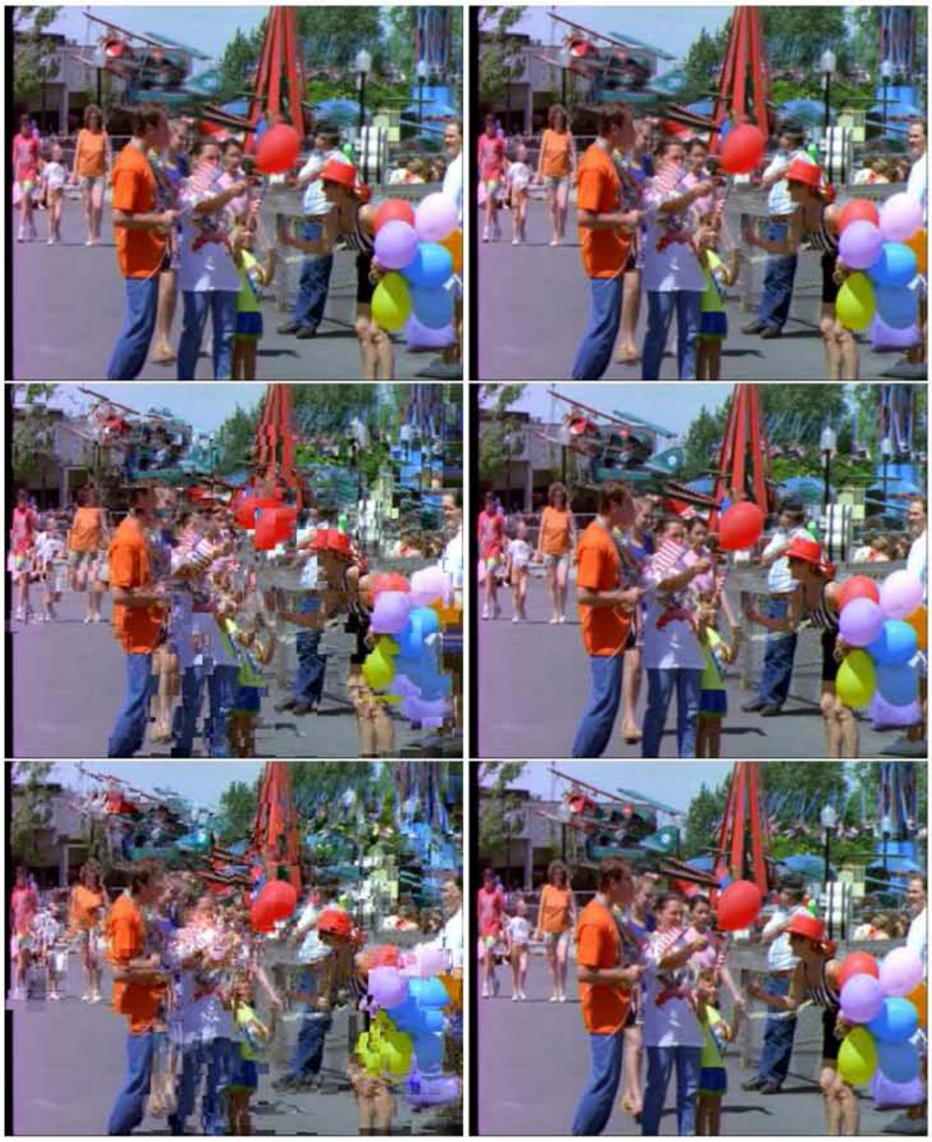

Figure A.4: First 3 frames from fault injection in the AVS motion vector values. Left faulted, right original

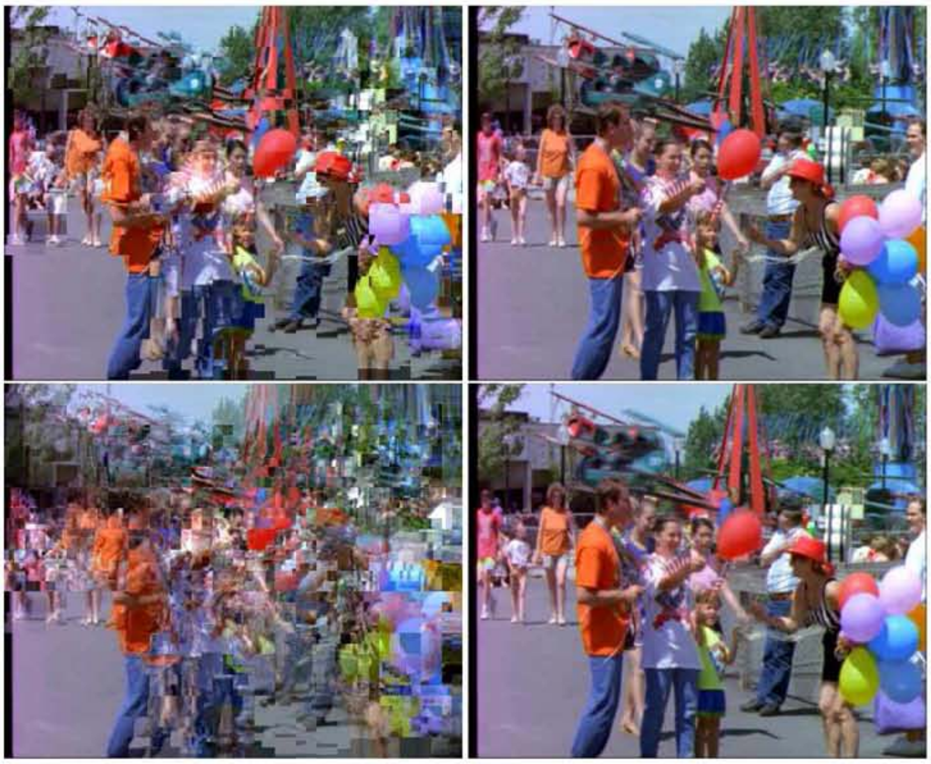

Figure A.5: Next 2 frames from the fault injection in the AVS motion vector values. Left faulted, right original

Chapter 9

Conclusions

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

9.1 Conclusion

As we mentioned in Chapter 2, the effort to enhance the performance of digital systems through the shrinking of the transistor's size had a negative effect in the reliability of the ICs. The increased susceptibility of transistors to cosmic radiation along with the incompetence of deterministic worst-case timing analysis of modern digital circuits raise a new "Wall" in the progress of electronic systems. To overcome this obstacle we praised the need to assess the impact of the newly introduced faults in the behavior of applications in order to create 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 a modern, widely adapted, full system simulator with fault injection capabilities. This new framework enables the injection of transient, intermittent and permanent faults, in order to simulate an unreliable environment. Furthermore, it is not limited to models covering radiation or timing induced faults, but also facilitates an easily extensible architecture to support the adaptation of future fault models. Through experiments our framework proved its effectiveness and ability to work with large workloads, in whole or partially, and to produce accurate injections. Moreover, the automation of the fault injection campaign and the ability to run multiple simulations on parallel create a great environment for experiments mitigating the effect of large simulation time per experiment. However, this tool does not try to invalidate other techniques but to serve as a complement to them in the design process.

An additional contribution of this Thesis is the experiment analysis we presented in Chapter 8 which validated previous research on this area [48, 23] and gave us a better insight on the behavior of applications in an unreliable environment. We observed the difference between the fault-tolerance of CPU components (integer registers v.s. PC register, fetch and execution stage) and the effect of manipulating 32bit data on a 64bit architecture. Another important observation is the variance of fault-tolerance in modules of the same application. Specifically, the targeted fault injection proved the existence of code segments that even in the presence of a substantial amount of faults can produce results in the margin of acceptable error. This inherent reliability can be exploited in order to improve other metrics, for example power consumption. As proposed in [33] reliability tags can be placed in the application code to characterize the reliability requirements of its each segment. In that way, we can use CPUs or computational units that function in subthreshold voltage, to execute portions with high fault-tolerance with lower power consumption.

9.2 Future Work

As an enhancement of this work we are interested in experimenting with more applications and creating a model of how faults affect computational patterns or groups of similar applications. Moreover, as a next step we plan to implement the mechanism for scheduling different code parts in processing units/CPUs based on their reliability needs proposed at [33, 48] and to evaluate its effectiveness in terms of performance and power consumption.

Bibliography

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: Generic object-oriented fault injection tool, 2001.

[2] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection in hardware prototypes. *Defect and Fault-Tolerance in VLSI Systems, IEEE International Symposium on*, 0:245, 2002.

[3] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43:52–61, January 2009.

[4] J Arlat, M. Aguera, L. Amat, Y. Crouzet, JC. Fabre, JC. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16:166–182, February 1990.

[5] Jedec Solid State Technology Association. Measurement and reporting of alpha particles and terrestrial cosmic ray-induced soft errors in semiconductor devices.

[6] A. Avizienis, JC. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[7] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil. A prototype of a vhdl-based fault injection tool: description and application. *J. Syst. Archit.*, 47:847–867, April 2002.

[8] B. Bastien. A technique for performing fault injection in system level simulations for dependability assesment. Master's thesis, University of Virginia.

[9] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3):258 – 266, may-june 2005.

[10] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. Exfi: a low-cost fault injection system for embedded microprocessor-based boards. *ACM Trans. Des. Autom. Electron. Syst.*, 3:626–634, October 1998.

[11] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[12] D. Brah and J. A. Abraham. Functional testing of microprocessors. *IEEE Trans. Comput.*, 33:475–485, June 1984.

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

[14] P. Civera, L. Macchiarulo, M. Rebaudengo, M.S. Reorda, and A. Violante. Exploiting fpga for accelerating fault injection experiments. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, 2001.

[15] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Exploiting fpga for accelerating fault injection experiments. *On-Line Testing Workshop, IEEE International*, 0:0009, 2001.

[16] M5 Community. M5 simulator, http://www.m5sim.org.

[17] Compaq. *Alpha 21264 microprocessor data sheet*, revision 1.0 edition, 1999. EC-R4CFA-TE.

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

[19] J. Engblom. Full-system simulation technology, 2003.

[20] L. Entrena, C. Lopez, and E. Olias. Automatic generation of fault-tolerant vhdl designs in rtl. *Forum for Design Languages*.

[21] G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer. Investigation of increased multi-bit failure rate due to neutron induced seu in advanced embedded srams. In *VLSI Circuits, 2007 IEEE Symposium on*, pages 80 –81, 2007.

[22] K. Goswami, R. Iyer, and L. Young. Depend: A simulation-based environment for system level dependability analysis. *IEEE Transactions on Computers*, 46:60–74, 1997.

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

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

[25] W. Hoarau, S. Tixeuil, and F. Vauchelles. Fail-fci: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913 – 919, 2007.

[26] Jr. Hopkins, A.L., III Smith, T.B., and J.H. Lala. Ftmp a highly reliable fault-tolerant multiprocess for aircraft. *Proceedings of the IEEE*, 66(10):1221 – 1239, oct. 1978.

[27] http://www.avs.org.cn/en/. Audio video standard.

[28] Arlat J. Validation de la surete de fonctionnement par injection de fautes. Master's thesis, INP Toulouse.

[29] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66 –75, June 1994.

[30] B. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems.*

[31] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44:248–260, 1995.

[32] WL. Kao, Iyer R., and D. Tang. Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Trans. Software Eng.*, 19(11):1105–1118, 1993.

[33] G. Karakonstantis, N. Bellas, C. Antonopoulos, G. Tziantzioulis, V. Gupta, and K. Roy. Significance-driven computation on next-generation unreliable platforms. In *IEEE Design Automation Conference (DAC)*, 2011.

[34] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14:8–11, 13–23, 1994.

[35] A. Kleen. Porting linux to x86-64. Technical report, SUSE Labs, 2001.

[36] P. Kudva, J. Kellington, Pia N. S., R. Mcbeth, J. Schumann, and R. Kalla. Fault injection verification of ibm power6 soft error resilience.

[37] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 502–506, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[38] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Architecture-level soft error analysis: Examining the limits of common assumptions. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 266–275, Washington, DC, USA, 2007. IEEE Computer Society.

[39] H. Madeira, F. Moreira, and J. Gabriel Silva. Rifle: A general purpose pin-level fault injector, 1994.

[40] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.

[41] S. Mukherjee. *Architecture Design for Soft Errors.*

[42] S. Mukherjee, J. Emer, and S. Reinhardt. The soft error problem: An architectural perspective. *High-Performance Computer Architecture, International Symposium on*, 0:243–247, 2005.

[43] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *In International Symposium on Microarchitecture*, pages 29–42, 2003.

[44] J. H. Patel and L. Y. Fung. Concurrent error detection in alu's by recomputing with shifted operands. *IEEE Transactions on Computers*, 31:589–595, 1982.

[45] J. H. Patel and L. Y. Fung. Concurrent error detection in multiply and divide arrays. *IEEE Transactions on Computers*, 32:417–422, 1983.

[46] D. Priore. Circuit implementation of a 600mhz superscalar risc microprocessor. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 104, Washington, DC, USA, 1998. IEEE Computer Society.

[47] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. *Design, Automation and Test in Europe Conference and Exhibition*, 0:0811, 2001.

[48] S. Roy, T. Clemons, S. M. Faisal, K. Liu, N. Hardavellas, and P. Srinivasan. Elastic fidelity: Trading-off computational accuracy for energy reduction. Technical report, Northwestern University, 2011.

[49] H. Seungjae, K.G. Shin, and H.A. Rosenberg.

[50] V. Sieh and O. Tschache. Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions, 1997.

[51] R.R. Some, W.S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J.J. Beahan. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 501 –506, 2001.

[52] V. Sridharan and D. Kaeli. Using hardware vulnerability factors to enhance avf analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture.*

[53] V. Sridharan and D.R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 117 –128, 2009.

[54] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, 2000.

[55] T. Tsai and R. Iyer. Ftape: A fault injection tool to measure fault tolerance,. In *Computing in aerospace, San Antonio, TX.*

[56] J.J. Yi, L. Eeckhout, D.J. Lilja, B. Calder, L.K. John, and J.E. Smith. The future of simulation: A field of dreams. *Computer*, 39(11):22 –29, nov. 2006.

[57] C. Yount. *The automatic generation of instruction-level error manifestations of hardware faults: a new fault-injection model.* PhD thesis, Pittsburgh, PA, USA, 1993.

[58] C. Yount and D. Siewiorek. A methodology for the rapid injection of transient hardware errors. *IEEE Trans. Comput.*, 45(8):881–891, 1996.

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

[60] H. Ziade, R. Ayoubi, and R. Velazco. A survey on fault injection techniques, 2003.