Thèse de doctorat

Pour obtenir le grade de

Docteur de l'Université Polytechnique Hauts de France

Discipline: **Informatique**

Présentée et soutenue par: Alexandre CHABOT.

Le 03/02/2020, à Gif-sur-Yvette

Ecole doctorale :

Sciences Pour l'Ingénieur (SPI)

Equipe de recherche, Laboratoire :

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH)

Laboratoire Sûreté des Logiciels (CEA/LIST/DILS/LSL)

Reliability-Guided Design Space Exploration for Safety-Critical Applications - Exploration des Architectures des Systèmes Embarqués Dirigée par la Fiabilité

Président de jury et Rapporteur:

- Alberto Bosio. Professor, Ecole Centrale de Lyon

Rapporteur

- Francois Pecheux. Professor, Sorbonne Université -LIP6

Examinateurs

- Karine Heydemman. Professor, Sorbonne Université -LIP6

Declaration of Authorship

I, Alexandre CHABOT, declare that this thesis titled, "Reliability-Guided Design Space Exploration for Safety-Critical Applications" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date: 03 February 2020

“Everybody is a genius. But if you judge a fish by its ability to climb a tree, it will live its whole life believing that it is stupid”

Albert Einstein

UNIVERSITÉ POLYTECHNIQUE DES HAUTS DE FRANCE

Abstract

Faculty Name

Department or School Name

Doctor in Computer Sciences

Reliability-Guided Design Space Exploration for Safety-Critical Applications

by Alexandre CHABOT

Technological advances allow the production of increasingly complex electronic systems. Nevertheless, technology and voltage scaling increased dramatically the susceptibility of new devices not only to Single Bit Upsets (SBU), but also to Multiple Bit Upsets (MBU). In safety critical applications, it is mandatory to provide fault-tolerant systems, providing high reliability while responding to applications requirements. The problem of reliability is particularly expressed within the memory which represents more than 80% of systems on chips.

To tackle this problem we propose a new memory reliability enhancement techniques called DPSR: Double Parity Single Redundancy, designed to answer SBU and MBU problematic. To evaluate our proposition we modify historically used simulation single fault injection model by adding memory monitoring and MBU injection and compare it to state of the art fault injection engines. Based on a thorough fault injection experiments, DPSR shows promising results. It detects and corrects more than 99.6% of encountered MBU and has a performance overhead of less than 3% in mean. Our fault injection methodology shows also promising results by discovering more than 11% of incorrect behavior after fault injection than the classical single bit random injection.

Keywords: Reliability, DPSR, SBU/MBU, Fault Injection, Memory

French Abstract :

Les avancées technologiques ont permis la production de systèmes électroniques de plus en plus complexes. A l'ombre de ces évolutions, la diminution de la taille des transistors ainsi que la diminution de tension ont dramatiquement impactés la sensibilité aux fautes de ces nouvelles plateformes électroniques. Leur susceptibilité aux fautes multiples a également été tout d'abord découverte puis augmentée. Dans les applications critiques, il est obligatoire de fournir des systèmes résistants aux fautes tout en conservant un comportement identique à celui attendu peu importe les conditions. Le problème de fiabilité est tout particulièrement exprimé dans la mémoire car elle représente aujourd'hui plus de 80% de la totalité de la surface des plateformes électroniques.

Pour adresser ce problème de fiabilité, nous avons proposé dans nos trois ans de recherche une nouvelle technique améliorant la fiabilité appellée DPSR (Double Parity Single Redundancy traduit littéralement par Double Parité associée à une Simple Redondance). Cette technique est désignée tout particulièrement pour contrer les soucis de fautes simples et multiples. Pour évaluer notre technique, nous avons modifié les modèles permettant de faire des injections simples en y ajoutant des paramètres supplémentaires comme la surveillance de l'utilisation mémoire. Ce modèle a également été enrichi de la possibilité d'injecter des fautes multiples. Nous avons au cours de ces années de recherche comparé notre modèle aux modèles existants. Basé sur de longues expérimentations, notre technique DPSR a montré des résultats prometteurs avec plus de 99.6% de fautes mutiples corrigées et détectées en introduisant une perte de seulement 3% de performance. Notre modèle d'injection a également montré des résultats prometteurs en découvrant plus de 11% de comportements non désirés que si on utilisait une méthode d'injection de l'état de l'art.

Mots Clés: Fiabilité, DPSR, Fautes Multiples, Injection de Fautes, Mémoire

Acknowledgements

La thèse étant un marathon plus qu'un sprint, il est indispensable de s'entourer des bonnes personnes et il est impossible de mener une thèse à bien sans un entourage personnel et professionnel de qualité. Cette partie de remerciements me permet notamment de remercier chaque personne qui a participé de près ou de loin, directement ou indirectement à ce que je présente aujourd'hui. Je voudrais commencer par remercier les membres du jury qui ont pris de leur temps . . .

Dans un second temps je voudrais remercier tout particulièrement mes encadrants de thèse. Smail Niar qui a été un guide avec ces précieux conseils. Il a su gérer mes errements administratifs et s'est toujours montré bienveillant vis à vis de moi. Ses remarques et son implication ont été précieux tout au long de cette thèse. Réda Nouacer qui avec sa bonne humeur et nos conversations autour de mon sujet mais aussi autour de la vie de chercheur et des conseils de vie qu'il a su me donner m'ont fait grandir tout au long de la thèse. Enfin, Ihsen Alouani a su venir me challenger scientifiquement avec ses questions, ses remarques et sa sympathie a su faire grandir et murir des idées que j'avais pour donner des articles. Il sortait tout juste de son doctorat quand on s'est rencontré et aujourd'hui il est maître de conférence, une reconnaissance méritée au vu de la qualité de son travail et de sa connaissance en différents domaines.

Même si je ne suis pas venu souvent à Valenciennes, j'ai établi des connexions avec différents doctorants du lamih. Un gros merci à Hannen, Ayoub et Ismat pour leur gentillesse, leurs sourires et leurs bon conseils. Un merci à tout les autres avec qui je n'ai pas eu la chance de partager autant mais qui m'ont toujours bien accueilli.

Basé à Nano-Innov j'ai évidement lié des liens tout particuliers avec tout les membres du laboratoire LSL. Merci à eux pour leur accessibilité et leur bienveillance. Je pense notamment à Florent Kirschner qui a toujours été franc et ravi de voir mes avancés pendant ma thèse. Un merci à Gilles et Yves qui ont toujour été disponibles pour répondre à mes différentes questions techniques et qui m'ont toujours accordé du temps quand j'en avais besoin. Merci à Zaynah qui était un peu ma tata pendant la thèse, elle, qui sait la difficulté que c'est de grandir en tant que chercheur et qui a toujours su venir partager des moments bons comme mauvais avec moi. Un merci évidement à Frédérique qui est particulierement compétente et qui est un organe essentiel du laboratoire. Merci également à Julien qui s'est montré à l'écoute de mes questions notamment académiques. Merci à Zakaria et David pour nos discussions et leur bonté tout au long de mes trois années. Hors du laboratoire j'ai pu notamment noué des liens avec Quentin, Agnès, Onder qui malgré les distances entre nos bureaux ont toujours montré de la curiosité pour mon travail

Je souhaite également remercier à part les non permanants du laboratoire qu'ils soient stagiaires, doctorants ou post doctorants. Je pense à mes cobureaux comme Quentin et Quentin B., Guillaume, Marc, Dongho, Thibault et Virgile avec qui la cohabitation a toujours été agréable et qui m'ont permis de décompresser parfois. Une grosse pensée pour Florent qui est toujours prêt à discuter de n'importe quel sujet avec cette pointe d'humour qui fait de la discussion un plaisir. Merci à Christophe qui est une force tranquille et dont la bonne humeur reste inchangée peu importe l'heure de la journée. Merci à Zinab et ne t'en fais pas tout ira bien pour toi j'en suis sûr. Evidement merci à Hugo, mon ami qui est dorénavant bien plus qu'un simple collègue de boulot pour nos discussions intellectuels autour du ballon rond mais aussi et surtout autour de tout et de rien.

Enfin, je ne serais pas là sans le soutien indéfectible de toute ma famille. Que ce soit mes grands-mères ou mon grand-père, ils ont toujours su s'investir dans mon projet et me soutenir. Merci à mes tontons et tatas pour tout. Merci à Gaêlle pour ta gentillesse, tes bons conseils et nos franches rigolades. Merci à Tom pour ta candeur due à ton âge et ton sourire à chaque fois qu'on se voit. Un énorme merci à Maman et Papa, vous êtes merveilleux et vous avez fait de ces trois ans un long fleuve tranquille. Vous étiez là pour les bons comme les mauvais moments, toujours derrière moi avec cette confiance et cet amour que vous me portez sans relâche. Finalement, je vais remercier Laura, ma compagne avec qui j'ai vécu pendant toute cette thèe et qui partage ma vie depuis 8 ans maintenant. Tu as accepté mes départs, mes voyages, mes horaires, mes travaux hors des horaires de bureaux et surtout tu m'as soutenu tout au long des trois ans. Je ne pouvais pas rêver mieux.

Contents

List of Figures

List of Tables

List of Abbreviations

MBU	**M**ultiple **B**it **U**pset
SBU	**S**ingle **B**it **U**pset
SEU	**S**ingle **E**vent **U**pset
SER	**S**oft **E**rror **R**ate
VP	**V**irtual **P**latform
SECDED	**S**ingle **E**rror **C**orrection **D**ouble **E**rror **D**etection
DECTED	**D**ouble **E**rror **C**orrection **T**riple **E**rror **D**etection
RTEG	**R**eliability **T**echnique **E**valuation **G**rade
DMR	**D**ouble **M**emory **R**edundancy
TMR	**T**ouble **M**emory **R**edundancy

Chapter 1

Introduction

Thanks to manufacturing process and integration improvements, modern mobile and embedded systems are now able to execute complex applications and can implement advanced functionalities, such driver assistant systems in autonomous automotive, drones etc. System-on-chip (SoC) manufacturers are expecting to produce in the next coming years chips with thousands of processing elements, very large cache memories and variety of dedicated accelerators using sub-micron node technologies.

Consequently, future SoC architectures will be more and more complex and the resulting hardware architectures will have a particular impact on the energy consumption, robustness and reliability. The obtained improvement in performance will go with a reliability downgrade due to the hardware integration rate increase. In [29] it has been proved that the error rate increases by a factor of $\sqrt{2}$, i.e. an increase of 40%, every 18 months due to node technology reductions. Transistor-node size shrinking combined with voltage reduction, create the need to tackle soft errors caused by transient faults. This type of faults is due to environment factors, such temperature and radiations, and corrupts data in memory and combinational units.

For this reason, next generation embedded systems have to be more resilient to transient faults than before. Robustness against transient faults, is for example, a standard requirement for safety-critical applications such as autonomous driving systems.

Consequently, a large number of works have been devoted to study the impact of transient faults caused by energy particles striking in systems running safety critical applications. A large set of software and hardware solutions have been proposed to detect and eventually correct the resulting faults. Space and time redundancy solutions, such as Triple Modular Redundancy (TMR) combined with a voting system, have been widely used to support Single Event Upset (SEU).

Critical systems are system that needs to be highly reliable even after evolution, a failure of a critical system may have dramatic consequences for environment, financial or human lifes. However, in most of the existing approaches real environmental factors are not take into account and they assume only single faults. The rise of Multiple-Bit Upset (MBU) in nanometer technologies-based SoC, creates the need of simulation tools to explore their effect on system reliability. Indeed, solutions proposed to fight against single faults may be not appropriate to fight againt MBU. Moreover, systems complexity is rising even for critical systems. Thus a modification of the system is more and more expensive as the system advances in its development lifetime.

In this thesis, we will present our memory reliability techniques. Our technique called DPSR for double parity single redundancy is designed to answer specifically MBU while keeping a good trade-off between different criteria: computation overhead, memory area consumption, correction and detection probability, and integration complexity. To ensure our technique to bring an improvement to the state of the art, we have compared our approach to existing memory reliability techniques regarding the same criteria. To realize this study we have used our methodology of fault injection using a virtual platform. Our fault injection methodology takes into account environmental conditions, MBU and application behavior to inject realisitic faults.

The plan of the thesis is the following. In Chapter 2 we expose a state of the art arround fault injection techniques and reliability enhancement techniques. To do so, we define fault types, we present a probabilistic model used in litterature and we explain fault consequences using a code example. We then define fault injection mechanisms and present some fault injection methodologies while focusing more onto simulation based fault injection. We then exposed different path of thinking about reliability enhancement techniques while focusing more onto memory reliability techniques.

In Chapter 3, we present and motive our memory reliability technique called DPSR. We then use a mathematical approach to validate our hopes about our proposition. This technique is proposed to answer a lack of reliability techniques designed to answer specifically MBU patterns and to fullfill a lack of memory reliability tehcniques designed to answer MBU. This section ends by a presentation of a proposed global criteria, the reliability evaluation technique grade (RETG) to evaluated reliability techniques.

In Chapter 4, we expose our fault injector. We explaine first our global way of thinking and then we detailled: how the application behavior is taken into account, how MBU are considered in our model, and how works our global algorithm. We end this Chapter by presenting the other injection mode that can be used while using our fault injection tool and why they are interesting.

Chapter 5 is the sum-up of our work, in this Chapter the goal is to use our fault injection methodology to evaluated reliability techniques. To do so, we firstly present our experimental setup. We then first evaluate our injection tool itself looking at efficiency, representativeness and overhead induced by the addition of the fault injector. In a third time, we ensure our mathematical approach to be validated by real experiments and we finally obtained the computation overhead induced by the memory reliability techniques. We end this Chapter by computing the global criteria RETG for our DPSR techniques and compare it to state of the art techniques.

Finally we conclude our work by resuming all our propositions to the state of art but also by proposing evolution of our fault injection methodology but also to DPSR.

Chapter 2

State of the Art

In this Chapter, we will first give basic definitions. We will then survey existing methods to model and simulate single and multiple bit upset in SoC. We will also present existing methods to improve system reliability. In the first part we expose and define what is a fault (Section 2.1), in the same part we talk also about Multiple Bit Upsets (MBU). Section 2.2 exposes fault injection techniques at different levels with a zoom made onto simulation based fault injection. We end this section with a presentation of different ways to protect the system against faults focusing onto memory reliability techniques in Section 2.3.

2.1 Fault Types

When functioning, embedded systems are subject to two kinds of faults. The distinction of those faults is made upon its duration. The two types are the following [50], [1]:

1. **Permanent Fault**. Permanent faults are caused by an undesired short or open circuit. When permanent faults appear, they are in place for the rest of the system life. For this reason, they are corrected by realizing a maintenance of the Hardware. This maintenance may be a component replacement or a soft/hard reset of the system. Due to functioning or fabrication issues, permanent fault occur mainly due to three different causes [7], [50], [47]:

   - Manufacturing and Design Time: these faults comes from error in the design or in the manufacturing process of the Hardware and manifest as stuck at one or zero and delay.
   - Wearout Mechanisms: these mechanisms are influenced by the aging of the system. Negative-Bias temperature instability, hot carrier injection, time-Dependent dielectric breakdown and electromigration are some of the mechanisms that produce this kind of faults. All cited mechanisms induce at the beginning intermittent faults that become permanent faults.

- Process Variations: The manufacturing induces a lot of process variability such as a non perfect doping for example. This randomness causes differences between transistors of the same chip. Differences of behavior for the same chip can be the reason for delays and thus alterate the behavior of the system.

2. **Transient Fault**. Transient faults are logical faults in circuit's operation and they occur at random mainly due to charged particle emissions [31]. Transient faults are non-permanent faults. The system is only perturbed during a small amount of time. The time of the perturbation is in general considered to be an instruction execution time at the application level. This reduction can be explained by the fact that transient faults are due to particle strike and thus happen for a very short moment even if the fault can stay for longer. The fault is materialized by one or more bit flips or a flip-flop modification. This change is called a single event and can cause a single or a multiple upset. The metric used to evaluate the sensitivity of the system to its environment is the soft error rate (SER) [30]. The SER is of course influenced by the type of particle encountered in the environment. At the ground level, there are three kinds of particles that are able to modify the state of a system. First, alpha particle is the most type of encountered particles. Second and third are the atmospheric neutrons that are usually separated in two categories based on their energy. More present, atmospheric neutrons with an energy inferior to 1 MeV and finally the atmospheric neutrons with an energy superior to 1 MeV. In the space environment, it exists different radiation sources such as: Van Allen radiations, solar activity and cosmic radiations [55]. Energies of these cosmic particles vary between some MeV and up to 10^{30}.

These two types of faults induce a failure rate for the entire system. Figure 2.1 extracted from [21] shows the evolution of the failure rate among three major steps of a system. In the first period the infant mortality, the system has a higher failure rate due to permanent fault high presence especially stuck at and process variations faults. Then the system enter in its maturity period where the system spends the main part of its lifetime. In this life period, the failure rate is almost constant and is mainly due to transient faults. Finally, the system enters in its wear out period, in this period the aging of the system makes the failure rate to increase. We add to transient errors permanent faults due to wearout mechanisms, this period is irreversible, only maintenance is able to reduce the failure rate rise.

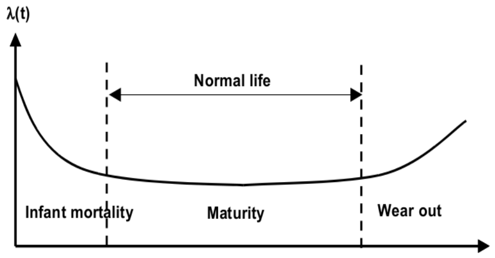


FIGURE 2.1: Failure Rate Evolution During Life Periods For An Electrical System [21]

The main focus of our study concerns transient faults and thus we make the assumption our system is in its maturity life period. SER determines the number of soft errors per unit time. SER unit is the FIT, one FIT rates the number of failures expected for a device during one billion functioning hours. FIT is used mainly into the semiconductor industry. An accurate SER is obtained by measuring number of failures in devices operated in real uses conditions. It necessitates a large number of devices and also specific installation. A large number of guidelines to measure the SER are provided in the JEITA soft error rate testing guideline [20]. SER can also be measured during accelerated conditions using raditation sources such as thermal neutron, high energy neutron or alpha particle exposure.

2.1.1 Multiple Bit Upsets

With the aim to maintain the Moore law prediction with the reality [43], transistor size has been reduced. This shrink has a direct impact onto the sensitivity of Hardware to single upset with the apparition and the raise of multiple faults observed for newest technologies. One of the first paper talking about

MBU in SRAMs is [53]. The phenomena of MBU has been noticed above certain thresholds and due to strong constants such as high resistance.
This phenomena of MBU is highlighted once again for SRAMs in [19] which shows that transistor miniaturization goes with the rise of single event multiple bit upset presence.

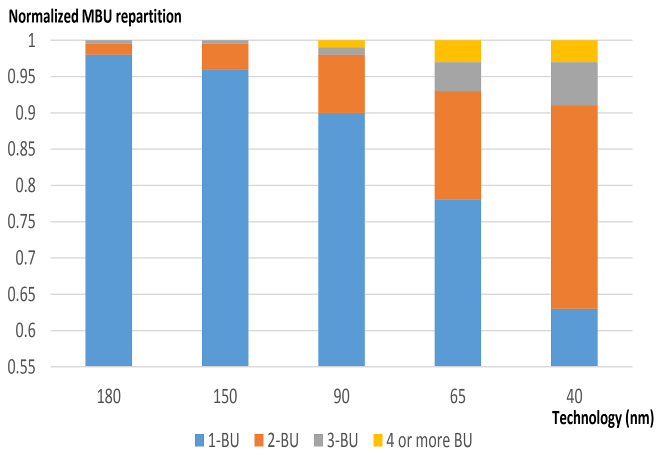


FIGURE 2.2: Single and Multi Bit Upset (BU) percentages by technology nodes in nm for SRAMs [19]

Figure 2.2 shows the growing presence of multiple bit upsets patterns. For example, in SRAMs under 40nm, more than 40% of particle strikes result in multiple bit upsets [19]. Usually, single event were linked to a single upset. With the rise of multiple upsets, Figure 2.2 shows that this hypothesis is valid only only for previous technologies. Results exposed in Figure 2.2 are obtained thanks to a neutron beam representing cosmic neutron flux at 10^8 the intensity of what is observed at the earth surface. Such as explained in the previous part, accelerated test are necessary to obtain results quickly than a simple exposure to a classical environment.

In Table 2.1, we present results obtained by Radaelli et al. [49] regarding the distribution of soft errors in 150nm commercial available SRAMs. The second line of the table is linked to the Table 4.1. For example, a 1-2 configuration corresponds to all upsets where two horizontally adjacent bits are flipped. Even if the study has been realised onto 150nm SRAMs, results are valid for future generations. Recent data are complex to obtain due to the cost of the facilities needed to obtain those results. This table shows that for

TABLE 2.1: Cumulative 2-Bit Event Count Normalized to 1000 for 150nm SRAMs for Different Particle Strikes Energy (MeV) [49]

Energy	Double-bit pattern			
	1-2	1-4	1-5	Others
22 MeV	773	136	80	11
47 MeV	681	180	117	22
95 MeV	653	192	132	23
144 MeV	686	156	133	25

TABLE 2.2: Pattern Injection Square

1	2	3
4	5	6
7	8	9

single event 2-bit upsets, it is more frequent to observe two horizontally adjacent flips than two vertically adjacent ones. Multi-cell upset events tend to be a concern especially for patterns that flip multiple bits in the same row [19].

Even though a large sets of these results have been observed while the memory has been studied in static mode, the work presented in [54] shows the trend of MBU is also cocerning for the reliability of memory working into a dynamic mode.

As explained in [19], SRAM multi-cell upsets are much more common in newer technology nodes, and microprocessor designs need to be protected against them. To do so, probabilistic model are proposed and will be explained in the next Section.

2.1.2 Probabilistic Model

Even though the bit SER saturates or even decreases for latest technologies, the SER of the system is exponentially growing due to the high level of integration [12]. It is thus mandatory to consider soft errors during the development of a critical system. To be able to study soft errors, a probabilistic model can be used. A probabilistic model is a mathematical model allowing exploration tools to take into account faults. It can take into account different inputs such as environmental values, applications technology, manufacturing processes. In our work we focused our study on soft errors impacting memory. Such as previously explained in Introduction, we have decided to focus on the memory as it represents more than 80% of electronic components area. Nevertheless, all the study can be muted to study other hardware parts of the system.

First, depending on the impacted memory region, the flip operation may alter either a data value or an instruction code, but this difference between data or instruction alteration is not taken into account when creating the fault appearance probabilistic model. The reliability law is given by Equations (2.1) and 2.2, where λ is the constant failure rate, R is the reliability distribution, $MTTF$ is the mean time to failure and t is the time.

$$R(t) = \exp(-\lambda * t) \tag{2.1}$$

$$MTTF = 1/\lambda \tag{2.2}$$

This model is based on a prior evaluation of the system failure rate and does not depend on system environmental conditions. Evolution of the fault model have been proposed in [28] and [57] who considered environmental conditions. In [28], failure rates are defined based on temperature while in [57], authors take power consumption into account. FIDES global electronic reliability engineering methodology guide is a generic approach to compute architectures failure rates [21]. Based on FIDES guide [21], physical and process impacts have to be considered for a precise failure rate λ computation. FIDES work is a sum up of what can be found in the literature regarding all criteria impacting the environment impact onto the failure rate. To compute the physical impact on λ, environmental conditions are modeled by providing: ambient temperature, temperature cycles, relative humidity, vibrations, saline pollution, environmental pollution, application pollution and chemical protection. Equations 2.3, 2.4 and 2.5 respectively respresent the Acceleration Factor of Temperature, Humidity and Vibrations. Thoses factors are used in the reliability evaluation of a system and can be positive or negative. These acceleration factor are most of the time forgetten when talking about electronics reliability and thus impact the probabilistic model drastically.

$$AF_{temperature} = exp(\frac{Ea}{Kb}(\frac{1}{T_0} - \frac{1}{T})) \tag{2.3}$$

$$AF_{humidity} = (\frac{H}{H_0})^p * exp(11604 * Ea * \frac{1}{T_0 + 273} - \frac{1}{T + 273}) \tag{2.4}$$

$$AF_{vibrations} = (\frac{G_{RMS}}{G_{RMS0}})^p \tag{2.5}$$

In Equations 2.3, 2.4 and 2.5:

- Ea is the Activation Energy
- T_0 is the reference temperature in which the base failure rate has been computed, usually 20° C.
- T is the temperature of the environment
- Kb is the Boltzmann Constant $= 8.617.10^{-5} eV/K$
- H is the relative humidity of the environment
- H_0 is the reference relative humidity in which the base failure rate has been computed, usually 70%
- p is the power of acceleration for each factor.
- G_{RMS} is the efficient vibration.
- G_{RMS0} is the reference vibration, usually $= 0.5G_{RMS}$

The presented probabilistic model can be used in different processes to evaluate system reliability at different stages of the system development lifecycle. In particular, the presented fault probabilistic model is a base of our fault model presented later in the thesis. Indeed, it helps us to compute a fault probability for the entire system and we have adapted this fault probability to our needs. Our fault model will be exposed in the Section 4.

2.1.3 Fault Consequences

To illustrate the subject, I will use an example code avaible in Table 2.3 that describes a factorial computation of a number n defined before the compilation phase of the application. This application is smally protected by the if statement at the line 9 which checks if the variable result is still superior to 0. Indeed, the result of a factorial is strictly superior to 0, this test is thus never wrong unless there is an exterior intervention. In a case of an exterior intervention, the result returned is -1 which corresponds basically to an error signal send to the caller of the function. Using this application, I will give example of different outputs that are possible after a fault occurence. Once the fault happens, different outputs are possible for the application:

1. Silent Corruption: In this case, the fault has impacted the hardware but has no impact on the executed task. For the given example code it can be a fault that impact the value i while the return call is done. This will have no impact on the application and the fault will be masked as the variable i is a variable of the called function.

2. Detected Corruption: In this case, the fault is detected by the system. The fault is thus seen at the system level. Actions that can be taken by the system are: pointing out the fault, correcting the fault, restarting the current routine, etc. In our example in Table 2.3, the fault can be detected line 9. Let's suppose the fault changes the value of the variable "result" and set it to 0 during the for loop. Then the instruction at the line 9 will detect this modification. The factorial function will thus return -1. This value returned by the factorial function can be interpreted by the caller of the function as an error result.

3. Result Corruption: In this case, the fault is not detected by the system but changes the output of the system. In our example Table 2.3, imagine the fault happening onto the result variable during the for loop and not making the value of result to be inferior to 0. Then the fault will not be detected and the result will be false. The false result is finally send to the caller and nothing is raised by called to alert the caller of a possible error.

4. Behavioral Corruption: In this case, the fault is not detected by the system but changes the behavior of the system. In our example Table 2.3, if the fault happens onto the code that increments the i value (ligne 7) then the for loop is infinite and the function factorial would never end.

TABLE 2.3: Code Example 1

```
# define n 15
int factorial (void)
{
    int i, result;
    i = 2;
    result = 1;
    for (i=2; i<=n; i++)
    {
        if(result > 0)
        {
            result = result*i;
        }
        else
        {
            return -1;
        }
    }
    return result;
}
```

It causes the system to be blocked into an infinite loop. In this particular case, the beahvior of the entire system is impacted and stuck inside a loop.

2.2 Fault Injection Techniques

Fault Injection has been studied since decades now. Up to our knowledge, the first paper dates of 1967 [26]. Nowadays, fault injection is used at different levels and for different applications such as Operating System, Smart Card, Web services, etc. There are three different objectives when realising a fault injection campaign. First, ensuring the correct functioning of error detection and correction mechanisms. Second, evaluating the overall robustness of the system [47]. Finally, reducing the risk to discover unexpected scenario after the commercialization of the product.

2.2.1 Fault Injection Overview

All fault injection environments are usually composed by the following components [38]:

1. Fault injector that modifies the system current state.
2. Fault library that stores different fault types, fault locations, fault times, and appropriate hardware semantics or software structures.
3. Workload generator which generates and stores different workload with different data input.
4. Controller and monitor, that control and track the injection target.
5. Data collector and analyzer which perform data collection, analysis and processing.

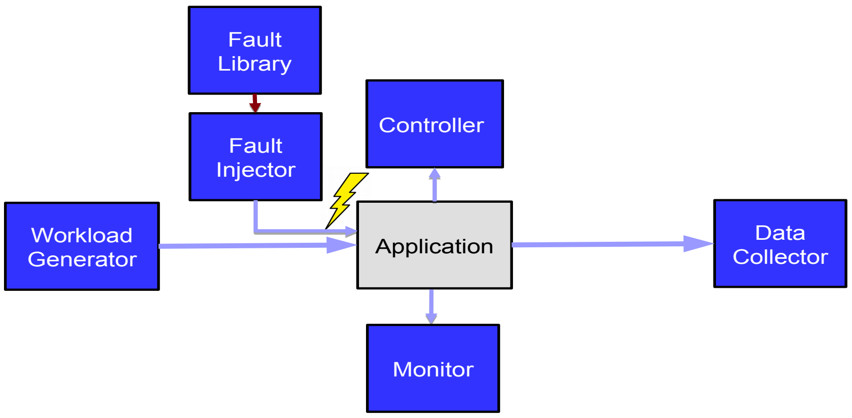


FIGURE 2.3: Representation of Fault Injection Environment

The components just presented and exposed in Figure 2.3 vary in complexity regarding the type of fault injection technique used. Indeed, existing fault injection techniques can be classified in four major types:

1. **Hardware Fault Injection**: In this technique external equipment is used to introduce faults into the hardware. We can cite laser for the Smart Card testing. We can also cite the recent work made onto the use of X-Ray in [5], which improves the injection by laser by making possible to target a transistor precisely. This technique is only usable in middle and late design phases as the software must run onto the chosen

hardware to be able to run experiments. This technique has the advantage to be extremely representative of what can happen in a real system. However, targeting a special scenario becomes harder and harder as the technology evolves. In [5] on 60nm technology, the tool used to target specific transistor is pretty hard to set-up and need a specific facilities; it causes also a huge rise in price.

2. **Virtual platform or Simulation-based Fault Injection**: When used, this technique imposes to dedicate time to develop a simulation tool representative of the desired hardware [37]. Once the simulator available, the Fault injection can be applied at different levels:from transistor up to algorithm level depending on the abstraction level of the simulator. The main advantages of this solution is the early access of the testing procedure and the possibility to test during different development phases or scenario. It allows to target easily time and location of the injection. Main drawbacks are the simulation time that can be long if the system is fully simulated and the time needed to develop the simulator.

3. **Emulation-based Fault Injection**: The purpose is to rise the match between the simulated hardware and the real one while maintaining a decent testing time. This approach requires however more design time. Field Programmable Gate Arrays (FPGA) are most of the time used in this approach to represent the future hardware and the injection is realised thanks to software modules. The main advantage is the correlation between the simulation engine and the future hardware and the speed-up compared to Virtual platform low level. The drawback is of course the time used to develop the emulator and the time consumed by the update needed during the development of the product.

4. **Software Fault Injection**: this technique injects fault in the running software either during the debugging phase or by adding source code [38]. The lack of hardware behavior consideration is the major drawback of this technique as it is not representative of the final system. Furthermore, the final software is modified to allow the injection, when tested, the real softwatre is not really tested because modified. This

TABLE 2.4: Fault Injection Methodologies comparison

Fault Injection	Development Time	Fault Representativeness	Repeatable	Speed
Hardware	High	High	Hard	Slow
VP, Simulation	High	Medium	Easy	Quick
Emulation Based	Medium	Medium	Easy	Very Quick
Software	Short	Small	Easy	Very Quick

modification imposes a careful verification when removing the added code to ensure the system remaining reliable. During certifications processes, the final system is evaluated and issues may happen when the code furnished is not the tested one.

Table 2.4 is a resume of pros and cons of different fault injection presented just above. In our work we have only considered Fault Injection by Virtual Platform. This technique has been imposed to us due to research constraints but has major advantages such as the repetability of operations. We have tried to reduce the medium representativennes by improving the probabilistic model. The hardware fault injection is almost never done in real environemental conditions and is most of the time accelerated to gain money and time. A comparison of our obtained results may be a good follow-up to our presented work.

2.2.2 Simulation-Based Fault Injection

In this work, we focus on virtual platform-based fault injection. This group of fault injection technique can be split in two different approaches:

1. **Deterministic fault injection:** The fault injection is directly processed by the designer. Hence, all characteristics of injection are provided by the designer to the fault injector. Indeed, the fault library in this case is replaced by critical scenarios. This method is used to focus the analysis onto a critic code part or instruction of the application. It is also used to replay a scenario that have been proved to exacerbate issues when the non deterministic fault injection find the scenario.

2. **Non Deterministic Fault Injection:** This injection mode can be either applied at run-time [2] or at compile-time. If applied at compile-time,

faults are injected in the target hardware or in the executed code. This procedure is more used to test a given scenario that have raised concerns regarding the system reliability. The non deterministic characteristic of this injection comes from the impossibility to know before the run of the system the time and the location of the fault. Indeed, time, location and type of fault are determined by a probabilistic model. At run-time, the fault injection type, instant and location are determined by the *Fault Library*. This technique is more used to test the system as an entire entity and to evaluate the system reliability in its environmental conditions. It serves also to discover problematic scenario unexpected.

One of the main challenge about simulation is to select the correct level of abstraction [10]. Indeed, determining the correct level of abstraction refers to selecting the quantum of information included in the model to answer questions asked. There exist different abstraction levels for simulation which are [22]: High System level, Transaction level, Timed Transactional level, Register-Transfer level, Gate level and Transistor level. Historically the simulation was at the same abstraction level for all components but this limitation has been overpassed to allow users to determine abstraction level of different part of the simulated system. Indeed, each part of the system is considered has a black box and is connected to other system components by communicaitons protocols. In conclusion, the abstraction level can be fixed for the entire simulated system or can be mixed regarding the level of precision needed. If we want to ensure the correct functioning of an application on a platform without performance limitation a high level of abstraction is valid, however if a precise performance measurement is needed, then the abstraction level must be as close as possible of the transistor abstraction level. The process of reducing the abstraction used decreases the simulation performance and thus takes more time to complete. It is thus important to balance the need of precision with the time allowed to the simulation step. Table 2.5 presents a comparison of existing and commonly used fault injection tools. The fault injection tools presented are usually associated with different way to determine the injection type, location and its probability. J-Swift [51] and Ferrari [35] are examples of tool that proposes fault injection to evaluate system robustness. However, up to our knowledge, only one work has included

TABLE 2.5: Comparison of Fault Injection Tools

Simulator Name	Injection Level	Determinism	MBU
LEON3 Simulator [2]	architecture	random	yes
FERRARI [35]	software	free choice	possible
J-SWIFT[51]	software	random	possible
BITFIT [40]	prototype	model-based	possible
SASSIFI [27]	control flow	random	no

multiple bit upsets in their injection fault library [2]. This work has been made for LEON3 architecture. The authors use random fault injection in time and in memory location.

2.3 Reliability Techniques and Means

A classical system representation is given with four layers. Figure 2.4 shows the four classical layers of a system. The hardware is explicit and contains all chips, connexions, I/O from the physical point of view. The Driver layer is set up upon and is the link between the hardware and the Operating System. The Operating System is the layer allowing the Application layer to work correctly onto its hardware, it also handles the memory management and the scheduling of application. Finally the Application layer is the layer that makes the job wanted by the user. It contains mainly the visble part of the iceberg.

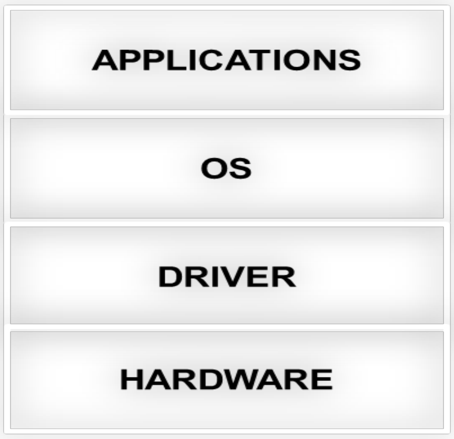


FIGURE 2.4: Representation of a System

To protect systems against MBU, reliability techniques and means are used at different levels of the system. Reliability techniques have different goals when implemented which are [37]:

1. Prevention: avoiding the fault to occur on the system.
2. Tolerance: prevent failures when faults are present in the system
3. Correction: prevent failures by correct faults before propagation
4. Forecasting: evaluate the system behavior and comparing it to faulted behavior. The goal is to determine if the technique reaches expected results of Prevention, Detection and/or Correction.

2.3.1 Global View of Reliability Enchancement Techniques

As the system is composed of different layers, it exists reliability techniques for every layers. It exists excellent survey talking about this subject and grouping a lot of reliability techniques such as [42].

One of the main concept associated to reliability is redundancy. Redundancy is the fact to multiply an instruction, a function, a storage or even a component to be able to compare information obtained by different object and compare them together to be able to detect a fault.

In the work presented in [52], authors duplicates L1 caches to be able to detect if an error occur. The methodology is especially efficient to raise reliability as L1 caches are highly accessed, they are more subject to propagate faults. To tackle the problem, they duplicate caches. With the same spirit, the work [56] duplicates information in the L1 cache in the same cache but at two different places. In this work, they use dead cache block (block not used for a long time) to hold replications of newly cache stored information. This technique is more efficient than the previous one as less memory impacting but is usable only when cache is not fully used. In those presented works, only detection is faisible as variables are only stored twice in memory, it becomes thus impossible to determine which caches has been impacted. In the work presented in **??**, authors propose to chose between a double or a triple cache redundancy. If the double redundancy is used, only detection is possible, in the case of a triple redundancy, the correction is possible by voting for the more present value. The redundancy is also used during scheduling. Works like [18] and [24] uses the parallelism of multi cores application to replicate certains functions or certain functions parts in other cores and compare results at the end to determine if a fault has occured. Those techniques have been optimized in [32] where replications of tasks is not static but dynamic and are induced by different mechanisms such as a result checker and time/memory usage valdiation. Indeed, using two cores to compute the same set of instructions is not efficient. Moreover, most of critical systems developped onto multi cores platforms are linked with static scheduling. This staticness is due to constraints of worst case execution time ordered by critical systems. It is far more complicated to ensure worst case execution time matching with a dynamic scheduling than with a dynamic one.

Another main concept associated to reliability is separation. Separation consists in spliting information to reduce the risk that a fault affects the meaningfull information. In the work [23], they cut the meaningfull information in two parts. The first part is stored followed by 0 and the second part is stored preceded with 1. In a case of a fault, the probability is divided by two to modify the meaningfull value. However, two lines in memory are subjects to fault, this modification raise the probability of a transient faults to happen but is negligible compared to the gain in reliability for the system.

Another concept associated to reliability is called write back. Write back consists in writing data in another memory section after a certain amount of time. Works like [6], [41] stored data from L1 cache to other place (Mainly in other Cache Levels, sometimes in global memory) after a certain amount of time. The time where information is written back is determined by a number of clock cycles.

All those reliability enhancement techniques are set at different layers of the system. An emerging idea is to combine those different techniques by taking best advantages of all of them to raise the reliability in different situations while maintening a good level of speed, memory overhead and power consumption.

An impressive work has been realised during the CLEAR project [17], in which authors compare cross layer reliability techniques together. Techniques have been applied at diffeWith the previsouly part regarding Multiple Bit Upsets, we shown their assumption to be less and less true with the improvement in manufacturing process and an evolution of their work would be to consider MBU.

Finally, in the next section, we focus onto memory reliability techniques as memory represent more than 80% of electronic area and is thus more sensible to particle strike. We will give an overview of global concepts and ECC (error correcting code) to protect memory.

2.3.2 Global Memory Reliability Techniques

We present now some memory reliability techniques used to fight against single and multiple upsets that we want to study all along our work. Some techniques used to fight against single upsets are still used nowadays. In all figures for this part, b0 represents the bit 0 of the work (thus the most significant bit or the less significant bit).

1. **Parity**. This technique consists in adding a bit to the memory line or column to compute the number of 1 or 0 stored in the line or the column. As shown in Figure 2.5, during the write operation, a XOR operation is realized between all bits to store and the result is added to the stored bits. This technique detects all single bit upsets but cannot determine the position of the corrupted bit in memory. The correction of the error is thus not possible.

2. **Double and Triple Memory Redundancy (DMR/TMR)**. As shown in Figure 2.6 DMR/TMR techniques consist in doubling or tripling the data that is stored. In the case of the double respectively triple redundancy, the data is stored twice respectively three times in memory. Memory areas where data are stored have to be separated enough to consider a particle strike modifying only one stored version. DMR does not allow to correct the value perturbed as it is impossible to know the value modified. The triple redundancy however allows to determine

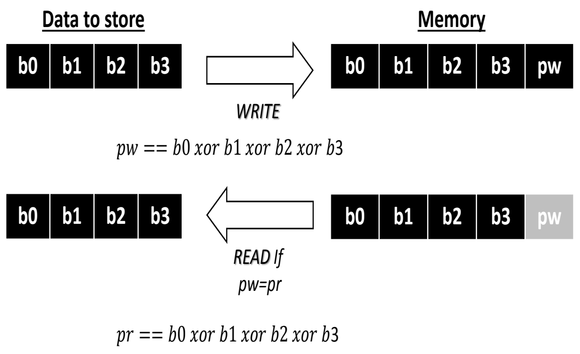


FIGURE 2.5: Parity Functioning

the line that has been perturbed. Indeed, the value is stored three times in memory during the write operation. During the read operation, a voter is associated to decide the correct value between the three proposed and the majority determines the real value. With the hypothesis of the gap sufficiently big between redundant memory areas, the DMR allows to detect all kinds of errors and the TMR allows to detect and correct all kinds of errors. The main disadvantage of this memory technique is its memory space usage.

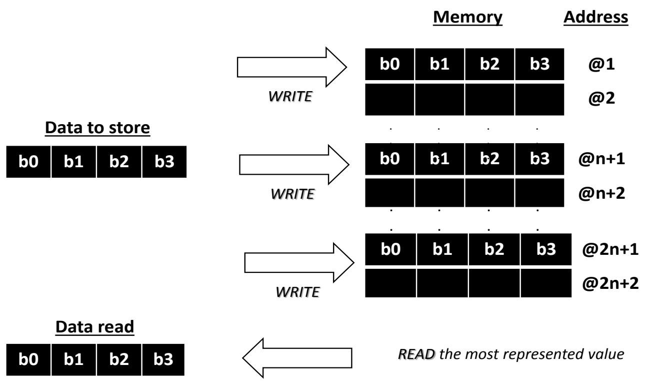


FIGURE 2.6: TMR Functioning

With the same spirit, solutions have been developed to address specific need of robustness by replicating different parts of the hardware such as column, row or even part of the memory. However those solutions are more used to protect against manufacturing errors and not against soft errors. In addition, those solutions goes with a rise in cost and complexity as sometimes a single erroneous bit make an entire part of the memory unusable. With process variations increase, the solution seems to reach its limits [46], [44].

3. **Parity-Based Mono-Copy Cache (PmC2).** In [4], authors propose to combine the double memory redundancy and the parity to create the PmC2 technique. Such as shown Figure 2.7 In this technique, during write operations, the parity bit is used and associated with a redundancy procedure to store the data in another memory location. During the read operation, the parity bit of the value read is compared to the parity bit stored, if there is a difference, the value taken is the one stored redundantly. This technique is a trade-off between single parity bit and the TMR, it uses the power of detection of the parity bit and use the redundancy to correct the fault once detected.

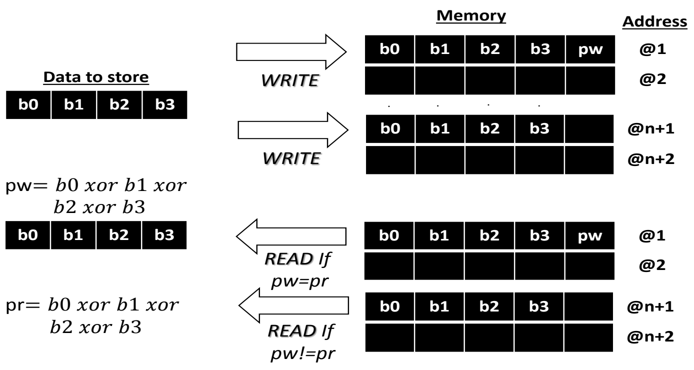


FIGURE 2.7: PmC2 Functioning

4. **Single Error Correction Double Error Detection (SECDED).** Even if it exists optimized versions of the Single Error Correction Double Error Detection mechanisms [48] the principle stays the same for all implementations. We base our study on SECDED codes based on Hamming codes. As shown Figure 2.8 the SECDED protection can be seen as an extension of the parity bit allowing to detect double error and correct single error. Data word represented by bx bits are protected by adding extra information represented by the px bits. Equations 2.6, 2.7, 2.8, 2.9, and 2.10 give an example of SECDED implementation for 8 bits data words. During the write operations, px bits are computed and stored together and mixed with all the dx bits. A last protection bit (called $p4$ Equation 2.10) is added that is a xor between

all the other *px* bits but is not represented in Figure 2.8. During the read operations, the same operations are done to ensure that the value protected have not been modified between the read and the write. This solution is expensive in terms of computation time, the main advantage of this technique is that it scales very well in terms of memory footprint when the data size to protect raises. Indeed, less and less bits are needed when the data size raise, for example, we need 2 bits to protect 2 bits but we only need, 5 bits to protect 16 bits. SECDED functionning is represented in 2.8, all black boxes are XOR operations realized between bits that give the value for the parity bit. When the data is read, all parity bits are once again computed and compared to previously stored ones. If it exists a difference, a fault has occured.

$$p0 = d0 \oplus d1 \oplus d3 \oplus d4 \oplus d6 \tag{2.6}$$

$$p1 = d0 \oplus d2 \oplus d3 \oplus d5 \oplus d6 \tag{2.7}$$

$$p2 = d1 \oplus d2 \oplus d3 \oplus d6 \tag{2.8}$$

$$p3 = d4 \oplus d5 \oplus d6 \oplus d7 \tag{2.9}$$

$$p4 = p0 \oplus p1 \oplus p2 \oplus p3 \tag{2.10}$$

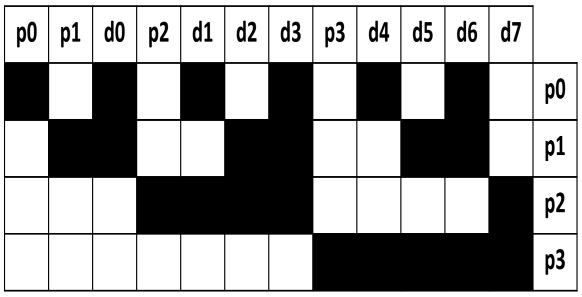


FIGURE 2.8: SECDED example for 8 bits data word

5. **Double Error Correction Triple Error Detection (DEC-TED)**

First time published in the beginning of 1980s [16], this technique is now used in the critical system development industry. Indeed, such as explained in the Section 2.1.1, the number of MBU presence is constantly rising with the reduction of transistor size. Thus, for systems needing a strong reliability aspect, they evolve from a SECDED error correcting code to a DEC-TED code. Far more complex to implement and thus more performance downgrading, this technique sets itself as an intermediate between the existing SECDED and the TMR. In our experiments, we implement the one proposed in [33] because of its widely usage. The extra data stored are separated in three categories and we are going to give an example for 32 bits data word to protect that induces 16 bits of protection:

(a) The first group is composed by 7 bits evenly distributed. This group has the same power of correction and detection of SECDED (with more bits used).

(b) The second group is composed by 8 bits similar to the first group, but in this case, 8 bits are used and those bits are computed differently from the first group. Due to this feature, the system is capable to detect triple error.

(c) The final group is composed by a single bit that is the parity of the bit in the second group. It allows to detect single error that may happen onto check bits and thus reduce the number of false positive.

Even if the optimized number to double correct and triple detect faults is 11, this scenario in real implementations is far more realistic as it exists a granularity for memory and memory are most of the time composed by power 2 data storage capacity.

6. **Physical Bit Interleaving**. As multiple faults number increase, and the complexity of techniques used to fight against multiple faults will not stop to rise, the physical bit interleaving is a solution less complex. The principle of this solution is to interleave words together on the same line and thanks to this procedure, multiple faults on the same line are reduced to smaller multiple faults and thus less complex error correcting code are enough to correct errors. However, during a read, the

entire line is read and a operation has to be made to obtain the desired word. A table of corresponding position is stored in memory and two interleaved words has to be accessed at two different time. it is also more power consuming [36].

As we can identify here, solutions proposed to protect the memory are either not efficient against multiple bit upsets, either complex and thus time and energy consuming either hugely impacting the memory size that is unacceptable for embedded systems. In the next part, we propose a new memory reliability technique developed with awareness about multiple bit upsets and embedded systems constraints. We will follow this proposition with a new metric to compare reliability enhancement techniques.

Chapter 3

Double Parity bit Single Redundancy

In this Chapter, we first present and motivate our proposition of a new memory reliability techniques called DPSR in Section 3.1. We then, mathematically analyze our techniques compared to other techniques presented in the previous Chapter. Section 3.2 presents the detection and the correction probability of our techniques compared to other ones, we also analyze the memory overhead of different memory reliability enhancements techniques. In Section 3.3 we present a global metric to compare with one metric memory reliability techniques.

3.1 Presentation and Motivation

As mentionned earlier, the phenomena of MBU is becoming more urgent with technology scaling down and the critical systems high performance requirements. Actually multiple bit upsets tend to be the major type of upsets observed [19]. Critical systems have to conserve their reliability level. To address the problem we propose a new memory reliability enhancement technique considering MBU patterns.

We provide the solution of a double parity bit associated with a simple redundancy. We call it DPSR (Double Parity Single Redundancy). DPSR has the objective to cope with most encountered MBU patterns. Contrary to different techniques considering multiple faults to be totally random, our technique takes into account MBU pattern probabilities. The particularity of multiple faults happening on memory is the proximity of flipped bits [19].

Regarding this particularity, we suggest to use two parity bits for the detection. Such as showed later in Section 3.2.1, the proposed technique detects more than 99.6% of encountered upsets. Adding a third bit would raise this percentage of 0.3% but appears hard to implement as most of memory are based on power 2 size. Using a fourth bit is enough to correct all patterns studied in our works and is conceivable but it also raises the memory area. This area rise is problematic as it makes particle strike on the system more frequently modifying the memory state. That explains why we decided to stick with only two parity bits. Finally as explained in [4], the parity bit is easy to set up. It can be done on different system layers and can be accelerated in different ways.

With the same spirit as what has been achieved in [4], we decide to add a redundancy for the word in a separate **non adjacent** memory location. This redundant storage will be useful for data recovery in case of a detected corruption. Hence, our technique deploys 2 bits to detect the fault and redundancy for error correction. It's worth noticing that depending on the required reliability, this technique can be used for detection only, or for detection and correction.

Figure 3.1 explains the functioning of the technique during a write operation in memory for an 8 bits word. When data is stored, two parity bits are computed. Equations 3.1 and 3.2 give the formula for the even and the odd parity bit in the case of an 8 bits word. Once both bits are computed, the word is stored twice in memory. They are not stored in adjacent addresses but rather in different memory locations. As shown in Figure 3.1, bits po and $p1$ correspond, respectively, to even and odd parity bits are stored within the original data.

During a read operation, as illustrated by Figure 3.2, the original is read. Even and odd parity bits are computed for the read value. The freshly computed even and odd parity bits are compared to stored even and odd parity bits. If they match, we consider the data to be fault free and the read operation carries on. In the case of a mismatch, the value or the parity bits has been corrupted. The read operations returns the redundant data.

The choice of interlaced bits to compute the two parity bits comes from the observation that it is very unlikely to find a 2 bit upsets that have a gap between the two flipped bits. Regarding the work [19] it is less than 2% for 2 bits upsets that make less than 0.6% of total observed patterns for 40nm SRAM technology. Moreover, in the case of a 3 bits upsets, the only pattern that may lead to corruption even with our solution is when three horizontally aligned bits are flipped. The chance to observe this pattern for a 3 bits upset is less than 0.28% that represents less than 0.028% of total observed upsets for 40nm SRAM technology. In the next parts, we compare with more details the proposed technique to other ones in presence of multiple bits upsets.

$$p0 = b0 \oplus b2 \oplus b4 \oplus b6 \tag{3.1}$$

$$p1 = b1 \oplus b3 \oplus b5 \oplus b7 \tag{3.2}$$

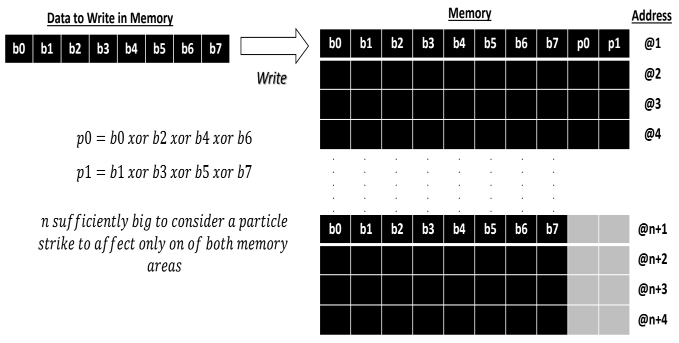

Data to Write in Memory
b0 b1 b2 b3 b4 b5 b6 b7
Write
Memory
b0 b1 b2 b3 b4 b5 b6 b7 p0 p1
Address
@1
@2
@3
@4
p0 = b0 xor b2 xor b4 xor b6
p1 = b1 xor b3 xor b5 xor b7
n sufficiently big to consider a particle strike to affect only on of both memory areas
b0 b1 b2 b3 b4 b5 b6 b7
@n+1
@n+2
@n+3
@n+4

FIGURE 3.1: DPSR Write for 8 bits word

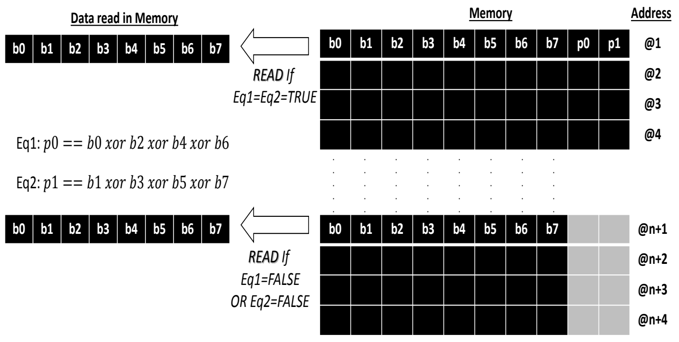


FIGURE 3.2: DPSR Read for 8 bits word

3.2 Probabilistic comparison between existing memory reliability techniques and DPSR

In this section we evaluate used memory reliability techniques and our technique against the data provided by [19] and exposed in the Section 2. The study of memory reliability techniques is made for different particle strikes energy. The choice of the implemented reliability technique depends on different parameters. Thus choosing a perfect reliability protection is pratically impossible.

Indeed, in the case of a multidimensional problem, it is almost impossible to maximize all parameters. The goal in multidimensional problems is to find solutions that are trade-off between all solutions. The choice of a memory reliability technique is crucial and has to be a trade-off between: detection, correction, memory space and speed of the reliability technique. In this probabilistic study, we evaluate the detection probability, the correction probability and the memory space used by the reliability technique chosen. The only hypothesis made is that faults can occur on every bits of the word, however even if the added data is faulted, the detection has to happen. The probabilistic model is based on Equation 3.3 and data used are from [19]. In Equation 3.3, p_{BU} and p_{shape} corresponds respectively to the probability to observe a 1, 2, 3 or 4 BU and to the shape of upsets to inject. They both depend on the memory technology and on the particle energy. Finally p_{fault} is the probability to observe a given fault pattern.

$$p_{fault} = p_{BU}(technology, particle\ energy) * p_{shape}(technology, particle\ energy) \tag{3.3}$$

3.2.1 Detection

The detection rate is the probability of a memory technique to detect a fault in a given environment. In this Section we compare the parity (that has the same detection rate as the PmC2 technique), with the classical DMR, SECDED techniques and with the DPSR. The detection probability $p_{detection}$ is computed following Equation 3.4 where $p_{detectionfault}$ equals 1 if the technique detects the type of fault else it is 0. We assume a single fault affecting only one memory area.

As shown in Table 3.1, all techniques have the same rate for detecting single faults. However, this rate goes down when multiple upsets appear.

TABLE 3.1: Detection probability of memory reliability techniques for 22 MeV particle strikes

	Parity	DMR	SECDED	DPSR	DECTED
1BU	1	1	1	1	1
2BU	0.22	1	1	0.999	0.999
3BU	0.059	1	0.999	0.994	0.999
4BU	0.010	1	0.990	0.976	0.999
Mean	0.704	1	0.999	0.996	0.999

The DMR is the best technique to detect multiple faults due to its capacity to detect all kinds of faults, the DPSR that is our proposed technique is close to what SECDED and DMR propose for detection rate.

$$p_{detection} = \sum_{fault} (p_{fault} * p_{detectionfault}) \tag{3.4}$$

3.2.2 Correction

The correction rate is the probability of a memory technique to detect and correct a fault in a given environment. The correction rate $p_{correction}$ is computed following Equations 3.5 and 3.6 where $p_{correctionfault}$ equals 1 if the technique corrects the type of fault else it is 0. We assume a fault affecting only one memory area. For this Section, we compare correction rate of the PmC2 from [4] paper, with the classical TMR and SECDED techniques and finally with the DPSR technique. Table 3.2 shows that the proposed technique overpasses expectation by being better than SECDED, indeed, SECDED detects up to 2 upsets but is only capable to correct single fault in a line. TMR has the best correction rate but DPSR is close to its results.

$$p_{correction} = \sum_{fault} (p_{fault} * p_{detectionfault} * p_{correctionfault}) \tag{3.5}$$

$$p_{correction} = p_{detection} * p_{correctionfault} \tag{3.6}$$

TABLE 3.2: Correction probability of memory reliability techniques for 22 MeV particle strikes

	PmC2	TMR	SECDED	DPSR	DECTED
1BU	1	1	1	1	0.9993
2BU	0.22	1	0.226	0.999	0.9993
3BU	0.059	1	0.0646	0.994	0.9992
4BU	0.010	1	0.010	0.976	0.9988
Mean	0.704	1	0.708	0.996	0.9991

TABLE 3.3: Memory Space (bits) of memory reliability techniques function of data size

Data Size	PmC2	TMR	SECDED	DPSR	DECTED
8 bits	9	16	5	10	9
16 bits	17	32	6	18	11
32 bits	33	64	7	34	13
64 bits	65	128	8	66	15

3.2.3 Memory Space

In this Section we compare memory techniques presented in Section 2.3.2 in front of 4 memory word size. For this purpose, we compare the PmC2 of the [4] paper, with the classical TMR, SECDED, DECTED techniques and with the DPSR technique. As shown in Table 3.3, SECDED is the best for scaling when protecting wider words. The DPSR uses one more bit to protect data than the PmC2 techniques but as shown earlier, it has better detection and correction rates. The worst in terms of energy and ressource utilization is obviously the TMR technique but its detection and correction rates are very high.

TABLE 3.4: Memory Space Overhead of memory reliability techniques function of data size

Data Size	PmC2	TMR	SECDED	DPSR	DECTED
8 bits	2.125	3	1.625	2.25	2.125
16 bits	2.062	3	1.375	2.125	1.687
32 bits	2.031	3	1.2185	2.0625	1.406
64 bits	2.016	3	1.125	2.031	1.234

3.2.4 Discussion

In previous Sections, we have aknowledge that DPSR shows promising results regarding Detection, Correction and Memory Space. Those criteria have been largely studied during my work and are what my reflexion is based on. DPSR shows of course due to its simplicity a good candidate to be implemented. However, as no implementation have been realized during the work, expressing a clear affirmation on the difficulty to implement is hard to do. Moreover, as defined in the Introduction, a critical system is a system that must be highly reliable even after evolution. This definition is subject to subjectivness and area of usage. For example, banking system are considered to be critical system but the consequences are not the same if a payment terminal is deficient or if a stock markets handler system is subject to faults. In one case, financial impact in the case of a fault is smaller than in the other. Thus, choices are made to match requirements regarding the benefit compared to the risk. As our technique is not perfect in terms of detection and correction, DPSR may not be considered in stock markets systems but could be a good choice for payment terminals. Our technique does not solve MBU rising issues alone but provides a new solution for reliability engineers to answer faults in given circumstances.

3.3 RETG: Reliability Enhancement Technique Grade

As we can identify in previous Section, different criteria are used to compare reliability techniques together. Some criteria are antagonistic such as the correction power and the memory space used. Some criteria are really close to each other such as the complexity of the algorithm and the power consumption. We propose a new metric to compare easily reliability techniques. We strongly think that we need to separate correction and detection as it is impossible to correct without detecting but it is possible to detect without correcting. Moreover, with cross-layer techniques, the detection is sometimes enough for a bunch of applications. We consider power consumption and computation overhead as correlated metrics thus we stick to the computation overhead. Finally the memory overhead is also a criteria in relation to the power consumption but raises also other concerns, we thus take the memory space as a third criteria. Equations 3.7 and 3.8 are provided to understand our way to compute each of RETG criteria, regarding detection and correction.

$$RETG_d = \frac{p_{detection}}{MemOv * PerfOv} \tag{3.7}$$

$$RETG_c = \frac{p_{correction}}{MemOv * PerfOv} \tag{3.8}$$

In Equations 3.7 and 3.8, $PerfOv$ and $MemOv$ as computed thanks to Equations 3.9 and 3.10, where $Time_{unprotected}$ stands for the mean execution time without protection and $Time_{protected}$ stands for the mean execution time with the reliability enhancement technique use.

$$PerfOv = \frac{Time_{protected}}{Time_{unprotected}} \tag{3.9}$$

$$MemOv = \frac{datasize + techniquesize}{datasize} \tag{3.10}$$

From now, we want to compute all those parameters for all reliability techniques previously presented. Such as stated in Section 2, we use a virtual platform to evaluate those techniques. In the next Section, we explain how works our fault injection tool and how it is representative of the environment.

As presented beforhead, DPSR appears to be a good trade-off between highly memory impacting techniques and small correction and detection rate. We also showed that some techniques does not scale very well when the number of errors raise. For example, the parity bit detects only 0.704% of MBU when double and triple upsets are taken into account which can be unacceptable for critical systems. Then, we presented an homemade grade to evaluate reliability enhancement techniques called RETG. The goal of this grade is to be able to judge a technique with only one grade by grouping different parameters. To compute this grade for our technique we need to determine the performance overhead induced by reliability enhancement techniques. In the next chapter we will present our proposed tool and detailled its functionning.

Chapter 4

Structure of the Fault Injector

In this Chapter, we present first an overview of our methodology in Section 4.1. Then in Section 4.2 and in Section 4.3, we present respectively how our model takes into account memory accesses and how we take into account the MBU phenomena. Finally, we detail in Section 4.4 how our algorithm works from the top up to the injection. To explain how our testing campaign works, we provide two screen shots of all parameters available of our virtual platform and of a simple script used to run campaign, respectively in Annexe A and Annexe B. Finally, we detail other injection modes implemented that are not detailled in our work as already largely studied in the State of the Art but we compare our results to random injection.

4.1 Overview

Our fault injection path of thinking is exposed in Figure 4.1. The main objective of this path of thinking is to answer the three main questions during a fault injection testing campaign:

1. What is the corresponding fault probability?
2. Where and when fault injection takes place in the memory unit ?
3. What kind of fault do we want to inject, SBU or MBU?

A major modification in our path of thinking compared to what is usually done is that we had the question of the fault type injected. This question is in our opinion mandatory due to the multipek bit upsets apparition increase.

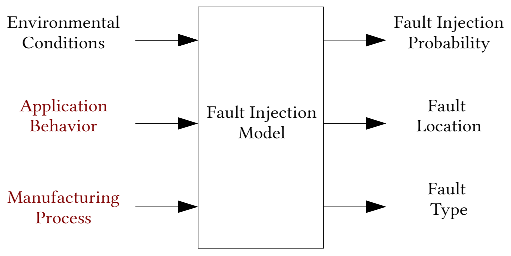


FIGURE 4.1: The Fault Injection Model

Our strategy started on FIDES standard to compute a base reliability for the system. We ameliorate the model by taking into accounts MBU patterns exposed previously in Chapter 2 missing in fault models and we take into account the locality of memory accesses. Both improvements will be detailled and justified in the next Section.

4.2 Memory Accesses Impact onto Fault Injection

During the simulation, we propose to take into account memory accesses. Depending on the technology and on operating conditions, the more a memory zone is accessed, the more likely an error or not within this zone. Hence, the memory access frequency impacts fault injection mechanism by weighting the value of the failure rate for each memory area.

To do so, we divide the memory into different zones and then track each access to the zones dynamically. Therefore, the probability to inject is different for each memory zone as shown in Equation 4.1. In this equation, f_i is the frequency of access to the i^{th} memory zone. π_i represents the fault injection probability in the considered zone depending on f_i. In the experiments $InjectionLocality$ is implemented using Equation 4.2 where α is a tunable coefficient to make the fault injection more or less focusing onto more accessed areas. In the experiments α has been set to 1.5 for experiments.

$$\pi_i = InjectionLocality(f_i) \tag{4.1}$$

$$InjectionLocality(f_i) = \alpha \frac{f_i}{\sum\limits_{i=1}^{n} f_i} \tag{4.2}$$

Authors in [9] mentioned a correlation between temperature and soft error rate. Temperature can increase soft error rate by up to 20%. Thus, soft error rate variation driven by temperature is valid to consider [34]. The memory thermal profile is directly related to the power density, and thereby to the memory access frequency. In our model, we consider memory access frequency as a parameter that directly impacts temperature and by consequence reliability.

4.3 Multiple Bit Upsets in the Model

As explained in Section 2.1.1, Multiple Bit Upsets is a phenomenon that, to the best of our knowledge, has been rarely integrated during simulation reliability evaluation using fault injection. We believe that it is important to inject both single upsets and multiple upsets to improve representativeness of results obtained thanks to fault injection . To achieve accurate representation of MBU phenomenon, we identify the probability of MBUs patterns. As shown in [49], depending on the technology and the number of flipped bits during a particle strike, different spatial patterns have different likelihood to happen. Table 4.2 is an example of data measured for a 150nm SRAM regarding multiple bit upset [49]. An x-y-z upset means that bits x,y and z are flipped simultaneously during the fault injection. As all memory cells accessible at a given address have the same probability to flip, a random draw determines the location of cell 1 for pattern in Table 4.1. Finally, the total probability is computed and indicated in Table 4.2 in Column *(1)*(2) Probability*. A fact to point in this Section is that technology has an impact of presented probability. Indeed, with the scaling down of transistors, transistors are getting closer and closer to each other and thus particle strike energy affect more and more closer transistor and the probability to get multiple bit upsets raise. However, to get data from very recent technologies is pretty hard to get. First, facilities to make studies of MBU on recent technologies are pretty expensive. Finally, getting results on ultra recent technologies for critical systems as it takes years and sometimes decades for newest hardware technologies to reach critical systems.

TABLE 4.1: Pattern Injection Square

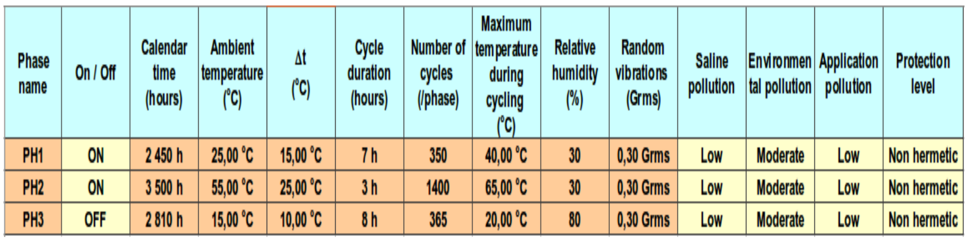
1	2	3
4	5	6
7	8	9

TABLE 4.2: Pattern Flipping Probability for 40nm technology [49]

Fault Type	(1) Type Probability	
1-BU	0.6	
2-BU	0.3	
3-BU	0.1	
Upset Patterns	(2) Pattern Probability	(1)*(2) Probability
1	1	0.6
1-2	0.773	0.2319
1-4	0.147	0.0441
1-5	0.08	0.024
1-4-5	0.92	0.092
2-4-7	0.062	0.0062
1-7-8	0.015	0.0015
1-4-7	0.003	0.0003

4.4 Global Algorithm

The flowchart in Figure 4.3 represents the proposed fault injection mechanism. From the top to the Injection Module, it is our fault injection strategy that is presented. In Figure 4.3, trapeziums represent data provided by the user, rectangles stands for injection algorithm steps, diamonds represents tests and cylinder represents database provided by the user and fixed for all simulations. First, the system failure rate is computed based on data provided by the user. Indeed, data are provided by the user regarding the FIDES standard such as shown in Figure 4.2. The values used nowadays in our fault



Phase name	On / Off	Calendar time (hours)	Ambient temperature (°C)	Δt (°C)	Cycle duration (hours)	Number of cycles (/phase)	Maximum temperature during cycling (°C)	Relative humidity (%)	Random vibrations (Grms)	Saline pollution	Environmental pollution	Application pollution	Protection level
PH1	ON	2 450 h	25,00 °C	15,00 °C	7 h	350	40,00 °C	30	0,30 Grms	Low	Moderate	Low	Non hermetic
PH2	ON	3 500 h	55,00 °C	25,00 °C	3 h	1400	65,00 °C	30	0,30 Grms	Low	Moderate	Low	Non hermetic
PH3	OFF	2 810 h	15,00 °C	10,00 °C	8 h	365	20,00 °C	80	0,30 Grms	Low	Moderate	Low	Non hermetic

FIGURE 4.2: Example of a set of parameters (extracted from FIDES standard)

injector are the following:

- Phase name: A system while functioning is submitted to different working conditions. Phases help to cut its life cycle into similar phases. An example of life phases is given in the next paragraph.

- ON/OFF: The system during its life cycle can be also in a OFF state. This parameter is here to determine the state.
- Calendar Time(h)
- Ambiant Temperature: It expresses the environnement ambiant mean temperature.
- Delta Temperature: It expresses the maximum temperature gap for the given cycle.
- Cycle Duration: It expresses the cycle duration.
- Maximum Temperature: It expresses the maximum temperature encountered by the system in its environnement.
- Relative Humidity: It expresses the mean relative humidity per phase in the environnement.
- Random Vibrations: It expresses the mean random vibrations per phase in the environnement.

To sum up those informations, we can see that the life cycle of our system in a given environnement are separated into phases that are separated into cycles. The example given Figure 4.2 is pretty simple, but the methodology is able to match other life cycle. If we take the example of a satellite life, a first phase matches the launch of the satellite, a second phase matches phases where the satellite is not hidden by the earth from the sun and a third one when the satellite is hidden from the sun by the earth. All parameters that we can see in Figure 4.2 are not an obligation to be set for every simulation but of course are possible to parametrize if wanted to. To accelerate the simulation, the computation of the failure rate is only computed once at the beginning of the simulation when the first memory access is realized. Indeed, this computation needs a file openning and computationnal treatments, so we suppose phases to be fixed for the entire simulation.

Second, when launched, each memory accesses are tracked. Of course, memory accesses regarding controller or monitor are not stored in memory, only memory accesses induced by the application are kept in memory. To improve simulation speed, memory is divided into memory areas that represents a power of 2 of the memory size. We have decided to realize this optimisation to reduce the memory footprint of the simulation and also to reduce the impact of the memory monitoring on the simulation time. Indeed, letting the free choice to the user of the number of memory areas makes the simulation far more slow than only masking the address to determine the area accessed by a read or write operations.

Third, when the random draw determines the injection moment using the failure rate, the accesses monitoring stops and we use the formula exposed in Section 4.2 to determine the injection location of the fault.

Fourth, the only thing missing to be able to inject the fault is the type of fault injected. The shape of the fault is determined thanks to data provided by the user. In Table 4.3 an example of data that can be provided by a user is given. The first Column represents the number of bits flipped. The second column represents the probability to flip this number of bits and the last column provides the combinaison of the probability to flip the number of bits and the probability to flip a precise combinaison if the number of bits flipped is the one chosen. For example, injecting the 2-4-7 patterns has $0.1 * 0.062 = 0.0062$ probility to happen. For the moment, a maximum of three bits upset is possible to inject, this can be easily improved if needed by a user. We kept this limitation as far less than 0.2% of observed patterns for 40nm technology are composed by 4 or more bits upsets. We have kept this limitation as we based our injection onto 3x3 square for patterns, the modification can be easily overpassed if needed for future technologies.
Five, the injection is realized during either a read or a write operation. This feature is also parametrizable by the user of the simulator. We advice the user to inject during both operations if the goal is the representativeness, or to inject during read operations if the goal is the number of perturbated runs. This will be justified in the next Section.

Finally the simulation is run up to its end (that can be the normal or the

TABLE 4.3: MBU example

Number of Cells	Probability	Combinaison Probability
1	0.6	1(1)
2	0.3	1-2(0.773) 1-4(0.147) 1-5(0.080)
3	0.1	1-4-5(0.920) 2-4-7(0.062) 1-7-8(0.015) 1-4-7(0.003)

stoped end) and then both the golden run and the corrupted run results are compared. The classification is today made by hand or with the help of a script but it's not part of the simulation engine. Critical applications have different output and different way to express them, we decided thus to keep this output comparison which is more a problem of post treatment than a reliability related problem.

4.5 Other Injection Modes

4.5.1 Random Injection

In this mode, the injector is not anymore based on a set of data provided by the user. In fact, the user is asked to provide only the failure rate. This base failure rate allows the injector to determine the moment of the injection. This mode is the historically mode used to test the system reliability. Once again the injection is made during a read or a write operation. However, this mode is not representative of the run-time environment as no information about the system and its environment are provided. This injection is used to test the system in a first step and is quicker than the global algorithm. The main drawback of its mode is the randomness of the injection. All memory areas are equally tested and is not representative of the application behavior. It could however be used if the functioning environment is not well know by the user.

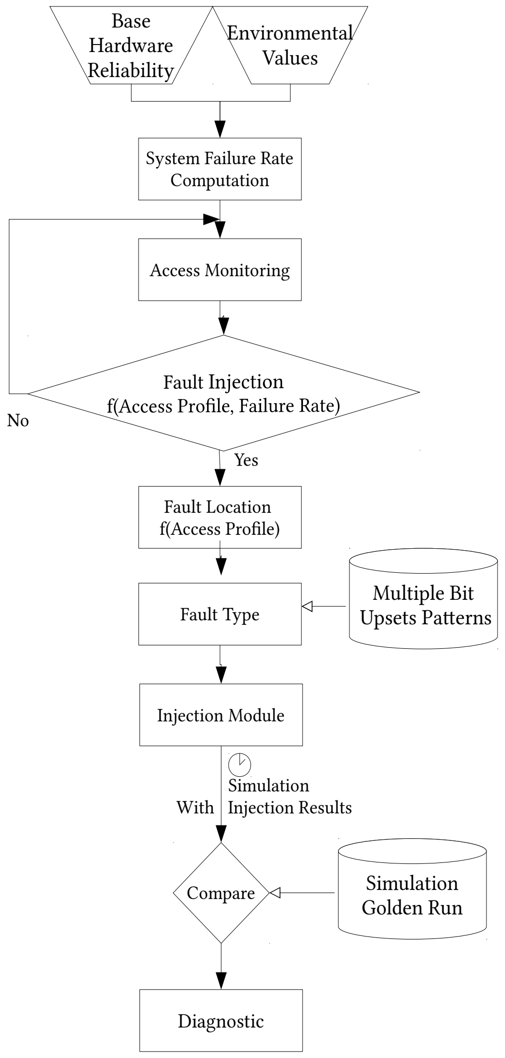

Base Hardware Reliability
Environmental Values
System Failure Rate Computation
Access Monitoring
Fault Injection f(Access Profile, Failure Rate)
No
Yes
Fault Location f(Access Profile)
Fault Type
Multiple Bit Upsets Patterns
Injection Module
Simulation With Injection Results
Compare
Simulation Golden Run
Diagnostic

FIGURE 4.3: Fault Injection Strategy

4.5.2 Determined Injection

In this mode, the injection is entirely or almostly determinisitc. Indeed, the simulator allows the user to enter a list of program instructions to execute before stopping the application execution. In this mode, the user can either inject faults at the desired instruction or can inject faults starting at desired instruction. This mode is used in the industry to test a given part of the application. We know that the critic code represents in mean 15% of an entire system [11]. This mode helps to focus more onto critic code parts. Moreover, this mode can also be used to test again a path that has lead to an undesired result following a fault injection. We don't focus on this mode but it is implemented and usable in our fault injector if desired by a future user.

This chapter has presented all injections modes implemented in our tool. In the next chapter we will use our proposed non-deterministic injection taking into account memory access and multiple bit upsets to evaluate different memory reliability techniques. Moreover, we will evaluate the efficiency and the representativeness of our injection tool while looking at the performance overhead of the simulation induced by the previously exposed methodology.

Chapter 5

Experimental Results

In this chapter, we first expose our experimental setup (Section 5.1). Then, we evaluate our tool regarding its efficiency, its representativeness and the performed overhead induced by the injection module in Section 5.2. After the evaluation of our injection method, we look at the missing variable to compute RETG: the performance overhead caused by memory reliability techniques. Section 5.3 exposes first a comparison between result expected by mathematical approach and results obtained during simulation, it exposes then, the performance overhead caused by memory reliability techniques. We then conclude this chapter by computing the RETG for memory reliability techniques studied all along this thesis.

5.1 Experimental Setup

For evaluating memory reliability techniques impact on performance we conducted experiments on well-known Mi-bench benchmarks [25], all representative of branching and computing intensive algorithms. We focus on 4 applications and all of them were run with large inputs.

- Qsort: efficient sorting algorithm, still used today in a large varieties of situations
- Bitcounts: this algorithm counts the number of bits in an array of integer in different ways. Used mainly to test the capacity of the processor to manipulate bits.
- Rijndael (encryption and decryption): an implementation of the well-known Advanced Encryption Standard.
- Sha: Encryption algorithm used to cipher a given input. It is used mainly to exchange keys and to cipher some data.
- Susan: stands for Smallest Univalue Segment Assimilating Nucleus and refers to algorithms used to filter image noise and/or find edges and/or corners. In this thesis, edges and corners finding have been widely used while the image noise filltering have been less used.

All those applications have been cross-compiled to work properly onto our armv7 simulator [45]. We remind to the reader that it exists a difference between ARMv7 and ARM7, the difference is that ARM7 corresponds to a family of processor cores with a three-stage pipeline and a von Neumann memory interface. In opposite, ARMv7 corresponds to the Instruction Set Architecture (ISA) version 7 and this type of ISA can be found in different Cortex family of cores. Figure 5.1 and 5.2 shows two representations of respectively UNISIM simulator and a simplified example of ARM7 architecture. In Figure 5.1, we can see that the ISA is perfectly simulated and corresponds exactly at the real one, however, the memory is set to be a ram only and thus all accesses to the memory (caches or global memory) are done inside the same adressed space. Moreover, a ELF loader is associated to the simulator to be able to use any application on the simulator. Finally, system call are translated for the hosting operating system that avoids to use a bare metal OS and faciliate the usability of all C applications even with system calls. Figure 5.2 shows a simplified representation of an ARMv7 architecture that is based on a ARMv7 ISA. This representation is far more complex than the UNISIM one as a lot of external components are available and adressable by the processor. However, main components are simulated or replaced when running an application onto UNISIM-VP. A large brands of simulators [13] exists on the market and have different characteristics. UNISIM-VP provides full system structural computer architecture simulators of electronic boards and System-on-Chip (SoC) using a processor instruction set interpreter. The whole software stack, consisting of the user programs, the operating system and its hardware drivers, is executed directly on the simulator.

UNISIM-VP is a component-based software and is thus modular. Hardware components, written in the SystemC language [3], model the real target hardware components, such as CPU, memories, Input/Output, buses and specialized hardware blocks. Hardware components communicate with each other through SystemC TLM-2 [8] sockets that act like the pins of the real hardware. The service components are not directly related to pure computer architecture simulation. They allow initializing and driving of simulation.

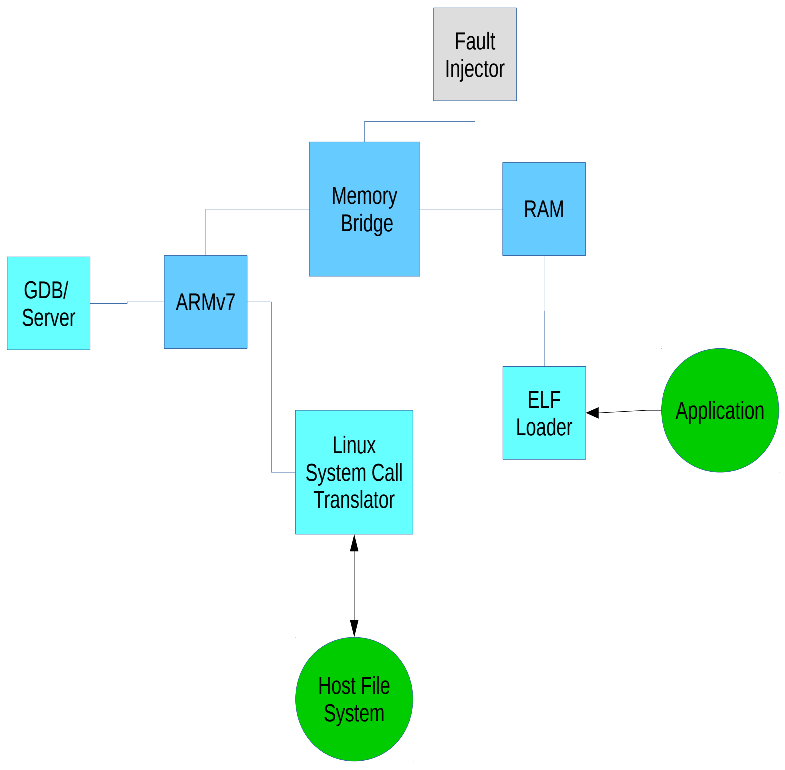


FIGURE 5.1: Representation of UNISIM-VP environment for ARMv7 simulations.

Services range from debuggers, loaders, monitors, host hardware abstraction layer and of course our fault injection module. To make the integration and use of Mi-Bench applications easier, a reduced version of linux operating system have been used especially for input and output files handling. Thus applications are not runned bare-metal but thanks to an existing operating system. Our armv7 simulator is based on the following architecture : the correct instruction set associated with different software components that represent different components of the system. A Linux operating system has been adapted to handle file opening, reading, writing and closing. It makes easier to get a functionning OS than to develop a bare-metal one. A major abstraction of the system is that we consider the memory to be a global memory and it exists no difference between caches, rom and ram. The choice of this simulator was associated to the thesis and we thus decided to make a work easily adaptable for other C++ based simulators.

UNISIM-VP has still major advantages like its transactional model that

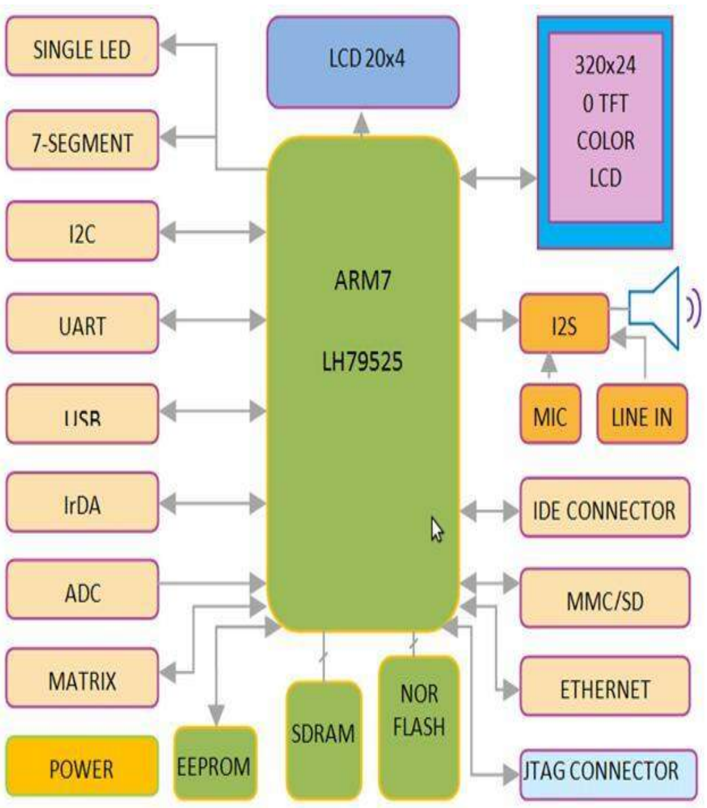


FIGURE 5.2: Representation of ARM7 architecture [39] which use a ARMv7 simulator.

enables to build representative and efficient simulators. Moreover, its modular architecture enable re-usability and portability of our work to other simulation platforms.

5.2 Evaluation of our Injection Tool

Experimental results are presented in terms of different criteria and metrics:

1. The efficiency: our approach is able to show reliability issues and ensuring the system to be entirely tested (Section 5.2).

2. The representativeness: Ensuring the fault injection representativeness of the environment and proving its added value to take into account not only single upsets but also multiple upsets (Section 5.2).

3. The simulation speed (Section 5.2.1).

Efficiency

Figure 5.3 shows obtained results of 11000 runs with four different injections procedures on Susan that is representative of observed behavior with other applications. For all cases, we inject faults during write operations in memory and we divided the global memory of 16MO in 262144 areas. It represents 16 accessed addresses per area. In the **SBU** experiments only single bit flips are injected and this is the type of experiments usually run during fault injection simulation test campaign. In the **MBU** experiments, multiple bit flips are injected using the probabilistic model given in Table 4.2. In the **SBUA** and **MBUA** experiments, fault are injected taking into account memory area access frequency.

The first conclusion that can be made is that by introducing memory access

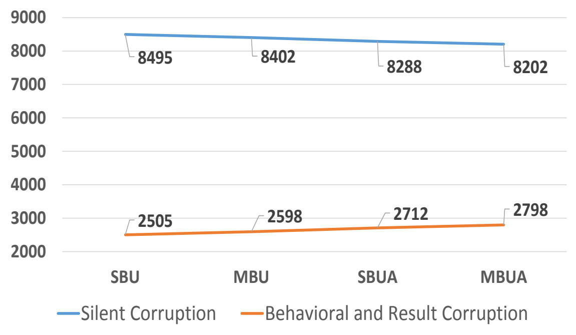


FIGURE 5.3: Type of observed corruption with different injection procedures on Susan smooth bench with 11000 runs for each procedure

monitoring in the fault injection (SBUA and MBUA), more result and behavioral corruptions occur. Indeed, for the same number of injections Figure 5.3 shows that, SBUA (respectively MBUA) increase the number of result and behavioral corruptions by 8.26% (respectively 7.7%) compared to SBU (respectively MBU). By consequence, rising the probability to inject faults into the most frequently accessed areas, widely used variables are effected by the fault injection.

The second conclusion that can be made is that by considering multiple bit upsets in the fault injection (MBU and MBUA), more result and behavioral corruptions occur compared to procedure where only single bit upset are considered (SBU and SBUA). For the same number of injections, Figures

5.3 exacerbates that MBU and MBUA increase the number of non-silent corruptions by 3.2% in average.

Our procedure considering MBU and access frequency has been proved to increase the number of non-silent corruptions by 11.7% for the same number of injections compared to SBU. Comparing those results to a pure random distribution of fault injection inside the memory would be inappropriate. In fact, among the 262144 different areas, only few of them (less than 100) are accessed by the application and thus the difference would have been enormous. Therefore, we compare our results to state of the art technique that is SBU. We notice that during SBU injections, only accessed parts are modified by the injection. Those promising results represents a proof of concept as Susan smooth mode is a representative image processing application that can be found in critical automotive application that our fault injection procedure shows more reliability concerns. However it has to be tested on a concrete and bigger application to encounter limits of access monitoring.

Figures 5.4 and 5.5 give the number of accesses and of injected faults for each memory area for Susan application with two different modes: the corner and the smooth mode. In the experiments, among the 262144 zones, less than 100 are significant. The 10 most accessed zones for Susan Corner (respectively Smooth) are ranked and presented in Figure (Figure 5.4 and 5.5).For these 2 benchmarks, 2000 runs (respectively 4000) have been done. For the first bench Figure 5.4 (respectively 5.5), our UNISIM fault module injected 1500 faults (respectively 4000).
As we can see in Figures 5.4 and 5.5, the most accessed area is highly prioritized during the choice of the injection location. As expected there is also a correlation between the memory accesses and the number of injected faults per area. Only a small deviation is observed for the fourth and the third areas (probably to the sampling choice) that are replaced in Figure 5.4, but it is not observable for Figure 5.5. This is due to the statistical model that is associated to our algorithm and as more runs have been made on the smooth mode exotic result does not appear. The relative small number of fault injections made in those areas due to the small number of runs makes the inversion

possible. From those figures, we can also notice that almost all areas have been perturbed during experiments. By consequence the injection model matches with the statistical testing mind spirit and also solves the concern to miss some memory areas, that is unacceptable for a good system reliability evaluation. Obviously in the context of critical system, injecting onto no accessed by the application memory part can have an impact. However, in our work we do not explore this aspect.

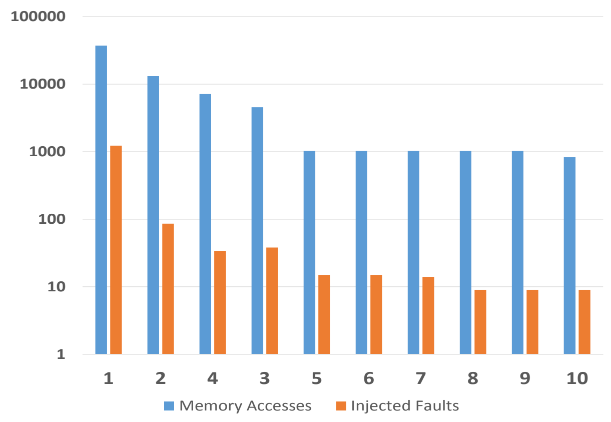


FIGURE 5.4: Memory access locality compared to Injection locality for Susan Corner Mode Application. Values are given for the 10 (x-axis) most accessed memory zones.

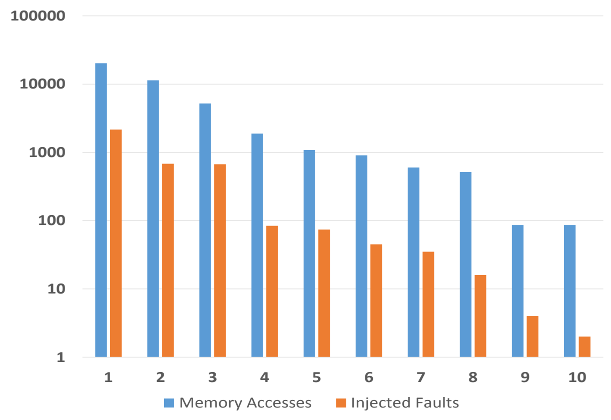


FIGURE 5.5: Memory access locality compared to Injection locality for Susan Smooth Mode Application. Values are given for the 10 (x-axis) most accessed memory zones.

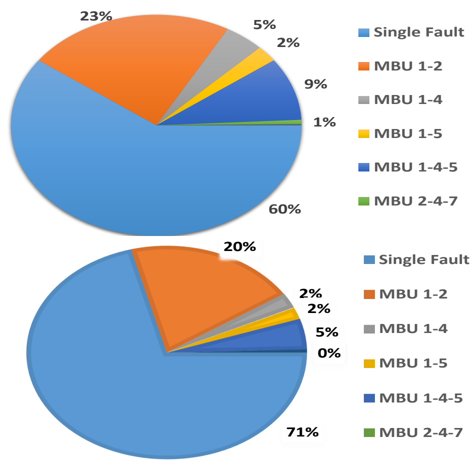


FIGURE 5.6: Repartition of injected upsets patterns given by Table 4.2 (top) and those injected by our model (bottom)

Representativeness

Figure 5.6 shows two distributions of fault patterns. The first is the patterns distribution given Table 4.2, the second distribution is the result of 443 injections made on different MiBench applications using the MBUA procedure (representing the injection of more than 90% of patterns). Our benchmark suite is composed of Quicksort, BasicMath, Sha, Susan (corner, edges, and smooth mode), Rijndael and Bitcount applications with different input sizes . Figure 5.6 indicates a correlation between both distributions, however the correlation is not perfect. By digging into details we can see that there is an augmentation of 11% of single bit injections and a decrease of around 11.5% of multiple bit injections. This imperfection is due to boundary conditions for the virtual platform global memory representation. As the memory is represented by an array of 64 bits line, multiple bit injections located on boundary conditions are impossible to realize. For example if the cell where the pattern injection square (Table 4.1) is located on the extreme right bit of a line, then it's not possible to inject a 1-2 MBU. In such a case, we decide to still inject a fault but to reduce the number of bits flipped until the pattern is able to fit into the memory at the desired memory cell. In the case of our example we thus reduce the 1-2MBU injection to a SBU injection. This choice was made to avoid losing simulations runs and explains the difference between both distributions presented Figure 5.6.

Furthermore, Figure 5.6 helps also to understand the 3.2% difference observed between SBU(A) and MBU(A) for Figure 5.3 described in Section 5.2. As we based our MBU onto a realistic model exposed Table 4.2, the difference is not as sensible as if we would have considered only MBU in MBU and MBUA injections procedures.

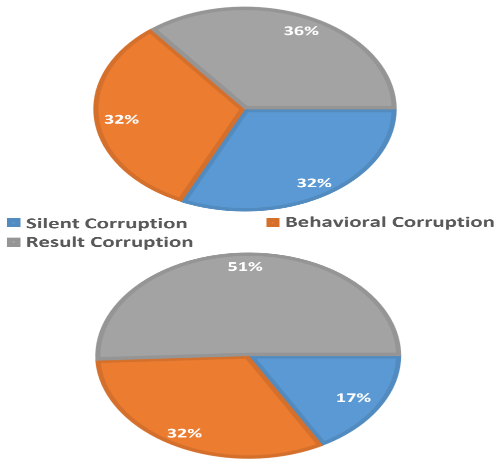


FIGURE 5.7: Single (top) and 2-BU (bottom) corruptions distribution

Figure 5.7 shows the normalized distribution of corruptions regarding the single bit upset (top graphic) and the 2 bits upset (bottom graphic). Results were obtained through the MBUA injection procedure based on MBU patterns exposed in Table 4.2. The benchmark is composed of Quicksort, BasicMath, Sha, Susan (Corner, Edges, and Smooth mode), Rijndael and Bitcount applications with different inputs size. Results have been obtained after a run of 1500 injections for each applications tha represent around 6500 single injections and 2500 double injections.

First, all type of injections have in majority resulted to an unwanted behavior. In 68.10% of cases for single bit upset and in 83.10% of cases for 2-bit upset, the result of the application is not conform to the golden run. Those results are explained by the absence of robustness mechanisms in our tested applications. Second, it is showed that a 2-Bit upsets injection has a higher chance to make the result different from the golden one. Indeed, 2-Bit injections have leaded to 15.0% more of non-silent data corruption compared to single bit injection. This recrudescence is due to the fact that the memory is more modified with a 2-bit injection than with a single bit injection and thus new weaknesses are discovered. Such as exposed in Section 2, multiple bit upset represents 40% of observed phenomena in 40nm technology and it's going to increase with the transistor miniaturization. It is thus mandatory to include MBU injection in new injection procedures.

5.2.1 Simulation Overhead

Figure 5.8 shows the time increase after implementing our fault injection module as a service for the UNISIM armv7 virtual platform. We compared simulation time of two runs. The first run made without the injection service and the second made with a single fault injection following the MBUA procedure (access monitoring is stopped after injection). Runs ended with a behavioral corruption have been removed from results as they are not representative of our injection module performance. Indeed, a behavioral corruption may lead to an infinite loop or to an execution issue and thus to a crash of the simulation that is not meaningful for our time performance purpose. We have compared simulation times for Susan corner, edges and smooth modes with a large input, Rijndael encrypt and decrypt mode for large and small inputs, Sha with large and small inputs, Basicmath for small inputs, Bitcounts for large and small inputs, and QuickSort for large and small inputs. Those applications represent a mix of instructions and computation intensive applications.

First, Figure 5.8 shows that the addition of our injection module has impacted the simulation time under 5.0% in the worst case and by less than 3.0% in mean.

Second, the input size does not impact the same way the simulation time. Indeed, for the QuickSort application, the simulation time augmentation is smaller for a larger than for a smaller input. However the Sha application exacerbates the opposite behavior. We attribute this simulation augmentation to the dynamic monitoring of memory accesses prior to injection.

Third, the Simulation time is not modified by the number of memory areas wanted by the user as the memory division is base on a base-2 division, this allows to reduce drastically the sorting of access regarding the address accessed. This simulator is accompanied with a fault injection module pre-

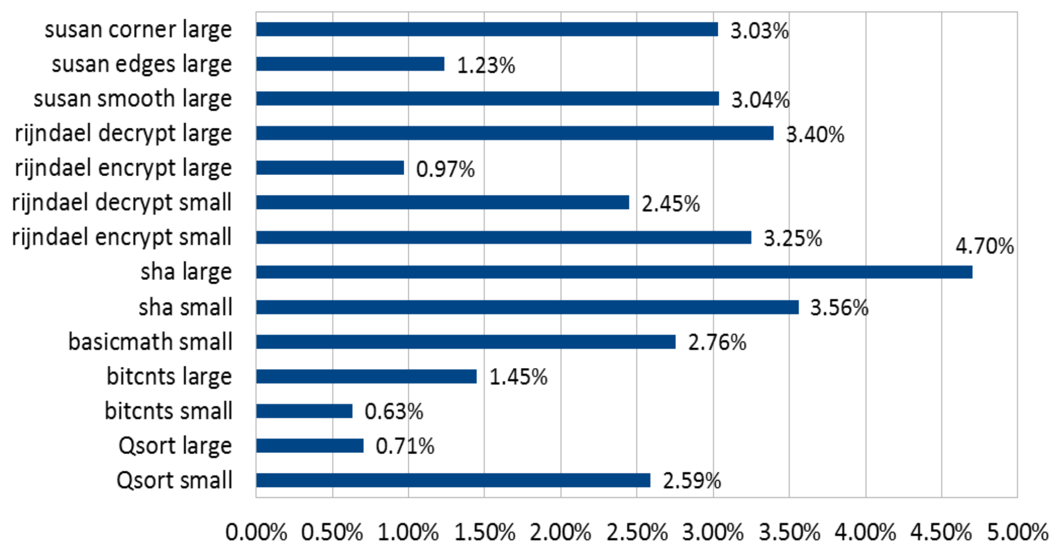


FIGURE 5.8: Simulation time increase after implementing fault injection module

sented in [14]. This fault injection module is configurable with environmental conditions as well as the probability to observe different MBU patterns. Moreover, the injection module takes into account the behavior of the application by monitoring memory accesses and influencing the injection to be in highly accessed memory areas to be as efficient as possible. The choice of memory areas influence the time and the location of the fault injection.

In [14] fault injections are only performed during write operations. This limitation has been over-passed in the used version of the simulator because we improved the injector to be able to inject both during read and write operations without considerable impact on the simulation performance. As exposed in Figure 5.9, the worst case is an increase of 8.13% simulation time for one run with monitoring and injection compared to a free run without accesses monitoring and without fault injection. This result is acceptable for a highly linked application to the memory as in [14], the simulation overhead on large benches was in the worst case of 5% when only write operation were subject to fault injection. Results comparison between Figure 5.8 and 5.9 is impossible to make as improvements of the simulation engine has been made and as experimental conditions have changed (PC older of more than one year, modification of the compiler, better understanding of performance tests processing). We can only make conclusions about each Figure but it is impossible to compare them.

Moreover, as shown in Figure 5.10, three outcomes are possible for a system under fault injection: the system crashes (STOP FUNCTIONING), or it ends but with a result different from the golden result (RESULT CORRUPTION) or it ends with the same result as with a fault free simulation (NO IMPACT). Figure 5.10 shows that 95% of simulation runs made on a not protected system with different applications are useful when injecting onto read operations. This is more precise than with only injection onto write operations where more than 20% of simulation fault injection runs on unprotected systems resulted in no corruption.

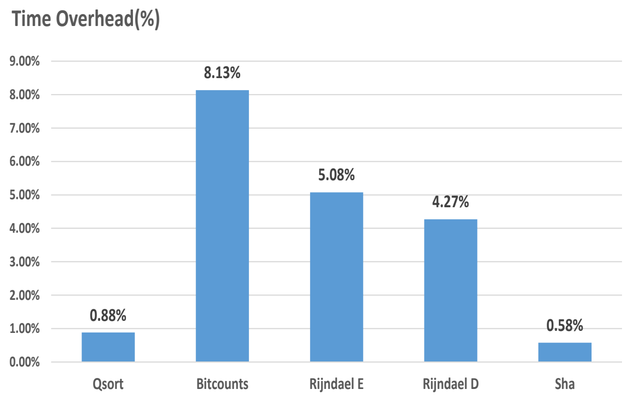
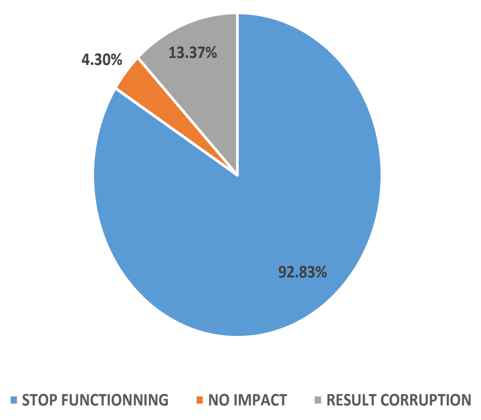


FIGURE 5.9: Simulation Overhead due to read and write monitoring without memory protection

The user has the choice to set up the number of memory areas, however, the number of areas is set to be a power of 2. Indeed, if the choice of areas is set to be free for the user, the simulation speed overhead is to high. Indeed, letting the free choice to the user of the number of memory areas make the access address handling far more less effcient. On the other hand, scaling with a power of 2 makes the address access handling based on a simple masking of the address to determine the access area. As we can see in Figure 5.11,



e

FIGURE 5.10: Distribution of Simulation Results after Read Fault Injection on unprotected system for different application on 19000 runs

the computation time overhead is almost stable when scaled onto a power 2 of memory area even with a large numbers of memory areas. This was not the case when scaled onto a free memory area number, previously obtained results are accessible through **??**.

5.3 RETG computation

5.3.1 Memory Reliability Techniques impact onto performance

Figure 5.12 shows simulation time on different applications when different reliability techniques are applied. Data are collected among 5 benchmarks presented Section 5.1. For all implemented reliability techniques and for each application 25 runs were done to compute the mean simulation time. The mean simulation time is of course made on simulations that have terminated correctly. The "Reference Simulation" time corresponds to the simulation time for different benchmarks when no faults are injected and no monitoring is made. The "No Technique" times correspond to Simulation time when injections are realized but no reliability techniques are used to protect against fault injection. The difference between Reference Simulation times and No Technique corresponds to the overhead due to fault injection algorithm and is the same for all runs comparing reliability techniques.

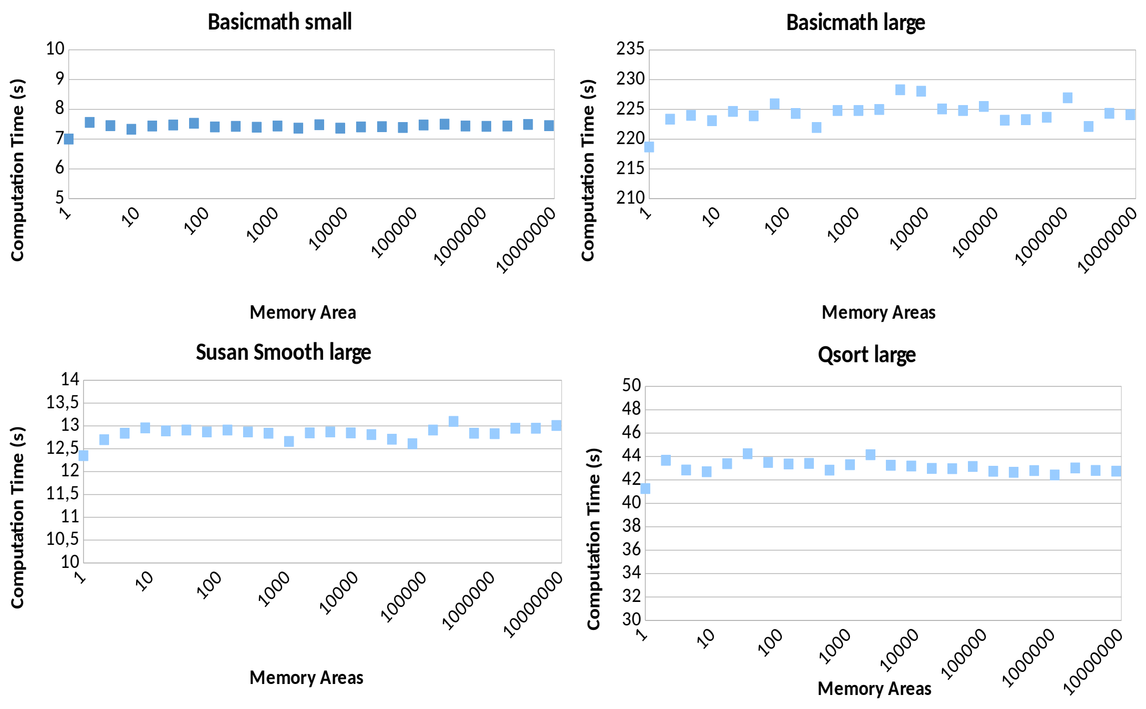


FIGURE 5.11: Computation Time modification due to memory areas number

This figure exacerbates two important points. First, all benchmarks present less than 15% simulation time overhead for all techniques. More accurately we can point out two groups. The first one is composed by Parity, DMR,, PmC2, TMR, Parity+Redundancy and our technique DPSR-opti, all simulation times are really closed to each other. The second group is composed with SECDED, DPSR and DECTED where we can see a slight overhead increase, especially for computing intensive applications that communicate a lot with the memory such as QSort. DPSR shows an overhead similar to SECDED and DECTED due to the decomposition of the value stored in bits to be able to compute different parity bits. However, unless what has been presented in [15], the new version of the DPSR technique is part of the less performance impacting reliability techniques. Indeed, optimizations have been put in place such as what was expected in the conclusion of our previous article [15]. Indeed, we have change our way to compute parity bit and more than using a switch case on the storage size, we have used a loop well known by binary expert that consists in cutting the stored word in two parts and xor each part together up to getting two bits. Both bits represents even and odd parity bit of the word and it makes us win a lot of computation time.

TABLE 5.1: Performance Overhead due to memory Reliability techniques

PmC2	TMR	SECDED	DPSR	DECTED
1.015	1.020	1.09	1.025	1.138

All other techniques have been implemented following at the same level and inside the memory, only SECDED and DECTED are based on the Hamming approaches.

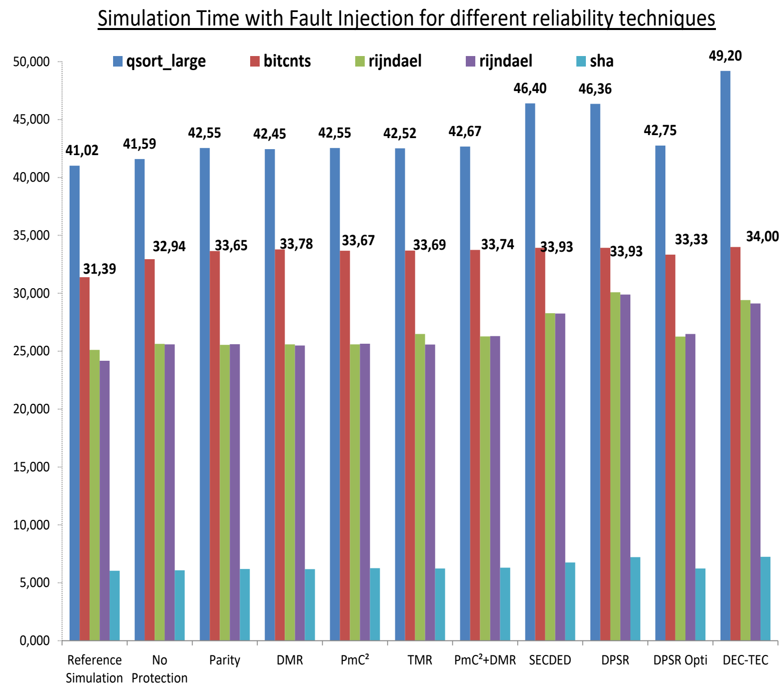


FIGURE 5.12: Simulation Time (seconds) with fault injection for different reliability techniques

5.3.2 Mathematical Approach Verification

In this section, the goal is to ensure the mathematical approach to be valid and to compare memory reliability techniques presented in previous Section with the DPSR. Our experimental setup keeps the fault injector presented in the previous Chapter. We run at least 300 times our algorithm for each reliability enhancement technique and for 7 seven different mi-bench applications by injecting one fault for each run and we separate the results of the simulation in three categories. First, the reliability technique permits to detect the fault but can not correct it. Second, the reliability technique permits to detect and correct the fault. Third, the fault is not detected by the reliability technique. The injection has been done with MBU patterns up to 3 bit upsets, we do not have injected more than 3 bit upsets.

Detection Probability

As shown in Figure 5.13, the detection probability modeled Section 3.2.1 and the one observed after simulation are close to each other. As expected, Parity does not detect a lot of bit upsets, however, SECDED and DMR detects everything on our runs. DPSR has not detected 2 faults injected on more than 300 for 7 different applications injections. Small deviations from the mathematical approach are due to the probabilistic model used to inject faults. By injecting more faults we would have closed the negligible gap existing between probabilistic and simulation approaches. By only adding one more bit in memory space for the DPSR detection we have improved by more than 30% the detection rate of the parity bit.

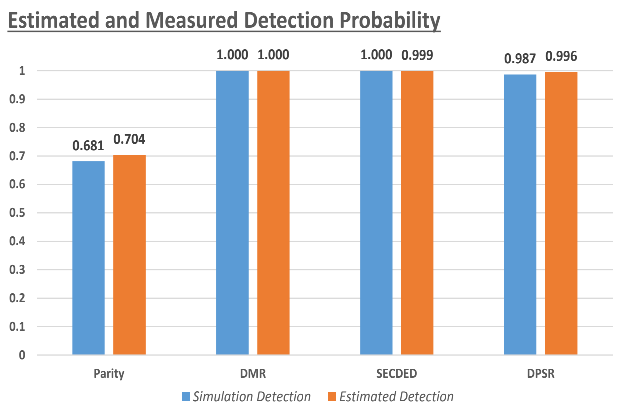


FIGURE 5.13: Estimated and Observed Detection Probability of different reliability techniques during simulation with a realistic fault injection

Correction Probability

As shown in Figure 5.14, the correction probability mathematically obtained Section 3.2.2 and the one observed after Simulation are close to each other (even if there is a slight difference for SECDED due to the probabilistic model). We can observe a efficiency reduction of SECDED for the correction compared to the detection. Indeed, in our model inspired by the work in [49], it's pretty rare to observe a double bit upset, that is not in the same line, this type of multiple upsets is not corrected with the use of SECDED even if it detects it. Such as expected, the DPSR has the same probability to detect and to correct a fault. It remains close to the TMR correction rate that is the most expensive in terms of memory space.

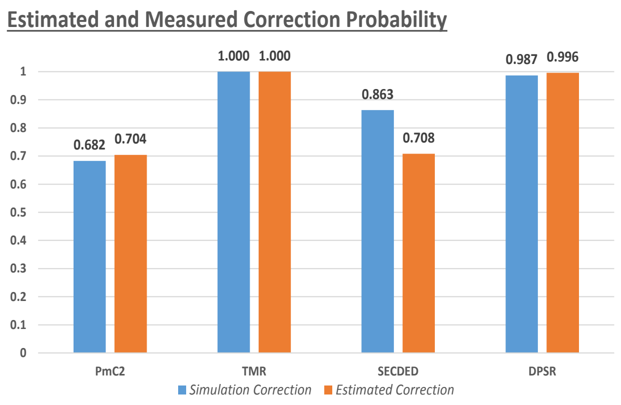


FIGURE 5.14: Estimated and Observed Correction Probability of different reliability techniques during simulation with a realistic fault injection

5.3.3 Memory Reliability Techniques Comparison

In this section we summarize all results obtained so far and give a global view about the efficiency of our reliability technique compared to the existing spectrum of memory reliability techniques. Table 5.2 is a sum up of all data found during our work on 8 bits word size. We can clearly see that DPSR shows promising results and is an intermediate between a protection represented by the DECTED technique where the objective is to reduce the impact onto memory size and allow a higher scalability and the TMR where the goal is set onto the reliability at the cost of the memory space. The closest concurrent of DPSR is DECTED, Regardless the main advantage of DECTED to be extremely scalable, DPSR overpasses DECTED from the performance and easiness to implement points of view. Another concurrent of DPSR is SECDED but has been over-passed in all criteria expect from the memory size usage point of view. In a context of critical systems and in technology

TABLE 5.2: Memory Reliability Techniques Comparison

	PmC2	TMR	SECDED	DECTED	DPSR
Detection	0.6824	1	1	1	**0.9867**
Correction	0.6824	1	0.8633	0.999	**0.9867**
Memory Space	9	16	5	9	**10**
Simulation Overhead	2.826%	2.623%	6.473%	13.8%	**2.50%**

improvement our solution will be better and better as more and more multiple bits upsets would be induced by a single particle strike.

DPSR is an intermediate choice between an extreme protection using a lot of memory space and a poor correction rate. DPSR presents an interesting trade-off between protection, memory space and performance. Performance optimization of the DPSR technique made after the publication of our previous work [15] has for sure improve its quality.

5.3.4 RETG estimation

Finally, we have all data necessary to compute the RETG estimation for all presented techniques and thus be able to classify them. We use data obtained in Table 5.2 to compute RETG. Table 5.3 gives all RETG detection values found for different memory word size and different memory reliability techniques. In this section, we can see that our technique comes into a good place and is a concurrent to the DECTED technique. The choice between both techniques has to be based on the requirements in terms of protection and memory overhead compared to the time overhead. In a averagely critical application our DPSR technique appears to be a good choice. However, for a huge need of reliability a TMR has to be selected. Finally, for a embedded critical system without execution time barriers, the DECTED seems to remain the good trade-off.

Table 5.3 shows also the scaling of techniques compared to memory size word to protect. We can see that our technique scales pretty well but less

TABLE 5.3: RETGc of memory reliability techniques function of data size

Data Size	PmC2	TMR	SECDED	DPSR	DECTED
8 bits	0.229	0.249	0.259	0.304	0.306
16 bits	0.234	0.249	0.285	0.316	0.354
32 bits	0.237	0.249	0.305	0.323	0.393
64 bits	0.238	0.249	0.318	0.326	0.421

than DECTED and SECDED that are recognised for their scalability. Indeed, DPSR uses the redundancy thus the scalability cannot be as good as other techniques scalability.

This chapter is concluded by an RETG estimation of different memory reliability techniques in which we included our proposed technique : DPSR. In the last chapter of our work we will conclude about the contributions brought to the state of the art during my thesis and how my work can be improved in the future.

Chapter 6

Conclusion

Critical systems development has always been a challenging and complex task. Regarding the importance to reach a high reliability level while maintaining a good level of performance, several techniques have been developped and studied during the last decades. From tools and development methodologies to concrete propositions of reliability enhancement at different layers of systems, it is not an easy task to chose between all those propositions to match the system requirements. For highly critical systems, it exists different qualification processes that must be passed to allow a product to be commercialized.

To cope with those requirements, we have presented our two contributions. Firstly, we work on memory reliability enhancement techniques by designing a new memory reliability technique called DPSR. This technique has the advantage to be easily implementable and to respect the hardware constraints by always demanding multiple of 2 more data and two more bits to protect the critical data. It presents to be a good trade-off between all important parameters involved in critical systems design, namely:correction and detection probability, memory and time overhead and implementation complexity. With the exposed evaluation metric called RETG we showed that our technique sets itself in a good trade-off between memory consumption, time overhead, reliability increase. Of course, the impact on the momory is pretty high and can be limited if only the detection is mandatory, we can then only use two parity bits to detect errors. If the desired behavior is the corruption then the redundancy has to be used.

Secondly, we proposed a fault injection tool and methodology to evaluate systems reliability by injecting faults into the memory at run-time. This technique has the advantage to be tunable by the user while maintaining a small performance overhead. This technique has been implemented as an extension of UNISIM-VP an instruction set simulator. The extension comes with minor changes to the simulator but is easily transposable to other simulation platforms. Indeed, the extension is just bounded to the simulator, the big modification inside the simulator is limited to memory calls. We have added a flag to these calls to be able to determine if a fault has to be injected. The fault injection is today linked to the memory, but it can be linked to other components of the simulated system. Finally, we used our new version of the simulator to evaluate the effectiveness of memory fault reliability enhancement techniques included our DPSR technique. Moreover, we have ensured that our mathematical prediction of detection and correction probability was correct by measuring them also under fault injections.

As a perspectives, we propose to consider the power consumption as a different metric to evaluate memory reliability techniques. Given the limited power budget of embedded systems, power consumption is also an important metric to consider. This thesis focused on memory fault injection, even if it is justified by the area occuped by the memory for newest systems, it could be interesting to transpose our methodology to other system's components. This would have the main advantage to allow the user to compare reliability techniques on different components and why not combined techniques. Working on multi-cores platforms could also bring different evolutions to our work and should be a follow-up to our work as it seems to be the future of critical systems with the development of standard in the automotive and aeronautic areas. We also think that prototyping our two contributions on an FPGA could give more accurate results.

Finally, with the development of higher computation possibilities, artificial intelligence and especially machine learning trends to come into the world of critical systems especially in space exploration. The issue for the moment is the data provided to the alogirthm to learn. A virtual platform may be the way to furnish a data input to the machine learning solver.

Appendix **A**

Appendix A

A.1 List of Usefull Parameters to Configure Fault Injection

```
          SystemC 2.3.1-Accellera --- Feb  8 2017 15:04:00
                Copyright (c) 1996-2014 by all Contributors,
                ALL RIGHTS RESERVED

Starting simulation.
The Application has been perturbated during its execution by read
injecting ;a single fault; at ;1072300616; area ;261793

Program exited with status 0
Simulation finished
Simulation run-time parameters:
FaultInjection.acceleration-factor                      1
# allow to tune the number of fault injected
FaultInjection.activation-energy                        0.001
# Energy Activation given in eV
FaultInjection.allow-multiple-injection                 false
# Do we make possible multiple injection
```

```
FaultInjection.base-fail-lambda                  1e-05
# It is the base failure value
FaultInjection.enable-debug-info                 false
# Is the simulator prints debug info
FaultInjection.full-access-written               false
# Do we print all accessed areas
FaultInjection.humidity-acceleration-power       1
# Acceleration power for humidity constraint
FaultInjection.immediate-injection               false
# Do we inject a fault for the next memory access
FaultInjection.injection-method                  4
# Injection Method Chosen (1 purely random, 2 FIDES,
# 3 given probability, 4 FIDES+access storage)
FaultInjection.mechanical-power                  1.5
# Acceleration power for the mecanichal constraint
FaultInjection.multiple-bit-upset                false
# Enable The Injection Of Multiple Bit Upset
FaultInjection.path-to-MBU-file
/path/mbu_config.csv
# Allow to gives the path to MBU file
FaultInjection.path-to-access-storage-file
/path/access.csv
# Path to the access storage file
FaultInjection.path-to-injected-results-file
/path/path/TechniqueImpactOnPerformance.csv
# Path to the Injected Results file
FaultInjection.reference-temperature             20
# Reference Temeperature given in K
FaultInjection.reference-vibrations              0.5
# Reference Vibration level in the considered environment
FaultInjection.storage-sizing                    1048576
# Determine the number of parts for the memory access storage size
#(must be a power of 2)
```

```
.
.
.
.
memory.bytesize                                          4294967295
# memory size in bytes
memory.cycle-time                                        31250 ps
# memory cycle time
memory.initial-access-number-value                       0
# Global number of access at the start of the simulation
memory.initial-byte-value                                0x00
# initial value for all bytes of memory
memory.injection_fault_allowed                           true
# Is the fault injection allowed
memory.mbu-allowed                                       true
# Is Multiple Bit Upset allowed
memory.memory-access-storage                             true
# Memory Access stored
memory.memory_temperature                                25
# Temperature of the memory
memory.org                                               0x00000000
# memory origin/base address
memory.read-latency                                      31250 ps
# memory read latency
memory.read-only                                         false
# enable/disable read-only protection
memory.robustness-technique-used                         9
# Choice of the robustness technique used
memory.verbose                                           false
# enable/disable verbosity
memory.write-latency                                     0 s
# memory write latency
path_to_results_file
/home/ac250711/Documents/results.csv
```

```
# Path to csv Results file

Simulation formulas:

Simulation statistics:
FaultInjection.platform-reliability 6.81102e-06
# Computed Value of the Platform Reliability (0 if not computed)
cpu.cpu-time                        54451725218750 ps
# The processor time
cpu.instruction-counter             517005940
# Number of instructions executed.
memory.access                       351091838
# Number of access to memory
memory.debug-access                 0
# Number of memory debug access
memory.dissasembly-access           0
# Number of memory dissasembly access
memory.fault-disable-access         513996
# Number of memory fault disable access
memory.fault-enable-access          350577842
# Number of memory fault enable access
memory.loader-access                0
# Number of memory loader access
memory.memory-access                0
# Number of memory memory access
memory.os-access                    0
# Number of memory os access
memory.read-counter                 263704676
# read access counter (not accurate when using SystemC TLM 2.0 DMI)
memory.undefined-access             0
# Number of memory undefined access
memory.write-counter                86873158
# write access counter (not accurate when using SystemC TLM 2.0 DMI)
```

```
simulation time: 52.77 seconds
simulated time : 54.4517 seconds (exactly 54451725218750 ps)
host simulation speed: 9.79735 MIPS
time dilatation: 0.969115 times slower than target machine
```

A.2 Explications

This list is a sort of usefull parameters printed at the end of the simulation in which some of them are let to the user to be able to customize as wanted the simulation. All the lines starting with a # are comment lines that help the suer understand each parameters. So, I will not list all of them. The pattern for arguments is pretty simple to handle. the first word is the a keyword that stands for the component of the simulator that is targeted. For example «memory.size in byte» stands for the memory size allowed to the memory of the simulated platform. If not indicated, it means the variable is global for the simulator (such as the path to results file). This set of parameters is also a way for the user to gain information about the simulation such as the memory.read counter that gives the total number of meaningfull read operations made during the simulation. At the end of the previous Section we can also see: the simulation time, the simulated time, and the dilatation of the simulation time compared to the simulated time. The dilatation is an indicator of the speed of the simulation as it is the fraction between the simulated time and the simulation time. It gives information about the computation power needed to complete the simulation. Finally, when the simulation is perturbated by a fault injection, a sentence like the one «The Application has been perturbated during its execution by read injecting ;a single fault; at ;1072300616; area ;261793». In this case, the fault is a single upset and has been injected into the area 261793 at the address 1072300616.

This resume of the simulation is the best way for the user to both ensure everything has worked fine and also to check if all parameters have been correctly setup.

Appendix B

Appendix B

B.1 Example of a Script Used During Test Campaign

```
# Parameters Declaration
ArmemuPath=/home/ac250711/Documents/test1/armemu
BenchAutomotivePath=path
BenchSecurityPath=path
CrossCompilerPath=path/arm-cortex_a9-linux-gnueabihf
FaultImpactPath=path/TechniqueImpactOnPerformance.txt
NumberOfIterations=10
NumberOfIterations2=1
MaximalDurationOfAnIteration=60
DurationOfSleep=1
FaultedResultFile=output_small.smoothing_faulted.pgm
STORAGE_SIZING_MAX=1048576
STORAGE_SIZING=1048576

# turn on debug mode: set -x
##################################################
                    #### QSORT LARGE ####
```

```
#################################################
cd path/qsort
export CROSS_COMPILE=path/arm-cortex_a9-linux-gnueabihf-
make clean
make qsort_large
i=0
while [ $i -ne $NumberOfIterations2 ];
do
$ArmemuPath/bin/unisim-armemu-0.8.0 -s linux-os.utsname-release=3.0.
-s cpu.enable-dmi=false
-s memory.injection_fault_allowed=false
-s memory.memory-access-storage=false
-s FaultInjection.enable-debug-info=false
-s FaultInjection.storage-sizing=$STORAGE_SIZING
qsort_large input_large.dat > output_large.txt
i=$(($i+1))
done
#Here Without protection
i=0
while [ $i -ne $NumberOfIterations2 ];
do
timeout -s SIGINT $MaximalDurationOfAnIteration
$ArmemuPath/bin/unisim-armemu-0.8.0
-s linux-os.utsname-release=3.0.4
-s cpu.enable-dmi=false
-s memory.injection_fault_allowed=true
-s FaultInjection.injection-method=4
-s memory.memory-access-storage=true
-s FaultInjection.enable-debug-info=false
-s FaultInjection.base-fail-lambda=0.00001
-s FaultInjection.storage-sizing=$STORAGE_SIZING
-s FaultInjection.path-to-injected-results-file=path
-s memory.robustness-technique-used=0
qsort_large input_large.dat > output_large_faulted.txt
```

```
# We examine the output of the run with the injection and
# compare it to the run without fault injection.

if [ -f output_small.smoothing_faulted.pgm ]
then
DIFF=$(diff output_large.txt output_large_faulted.txt)
if [ "$DIFF" != "" ]
then
FaultThatHasAnImpact=$(($FaultThatHasAnImpact +1));
DIFF2=$(diff output_large.txt output_large_faulted.txt| wc -l)
#echo $DIFF2;
if [ "$DIFF2" -gt "8" ]
then
echo "The␣fault␣injected␣has␣modified␣the␣behavior"
 >> $FaultImpactPath;
FaultThatHasModifiedTheBehavior=
$(($FaultThatHasModifiedTheBehavior+1));
else
echo "The␣fault␣injected␣has␣modified␣the␣result"
 >> $FaultImpactPath;
fi
else
echo "The␣fault␣injected␣has␣not␣impacted"
 >> $FaultImpactPath;
fi
rm "$FaultedResultFile"
else
echo "The␣fault␣injected␣has␣stopped␣the␣correct␣functionning"
 >> $FaultImpactPath;
fi
#To avoid multiple times the same seed
sleep $DurationOfSleep
i=$(($i+1))
done
```

```
##Here with DECTED protection
i=0
while [ $i -ne $NumberOfIterations ];
do
timeout -s SIGINT $MaximalDurationOfAnIteration
$ArmemuPath/bin/unisim-armemu-0.8.0
-s linux-os.utsname-release=3.0.4
-s cpu.enable-dmi=false
-s memory.injection_fault_allowed=true
-s FaultInjection.injection-method=4
-s memory.memory-access-storage=true
-s FaultInjection.enable-debug-info=false
-s FaultInjection.base-fail-lambda=0.00001
-s FaultInjection.storage-sizing=$STORAGE_SIZING
-s FaultInjection.path-to-injected-results-file=path
-s memory.robustness-technique-used=9
qsort_large input_large.dat > output_large_faulted.txt

# We examine the output of the run with the injection
#and compare it to the run without fault injection.

if [ -f output_small.smoothing_faulted.pgm ]
then
DIFF=$(diff output_large.txt output_large_faulted.txt)
if [ "$DIFF" != "" ]
then
FaultThatHasAnImpact=$(($FaultThatHasAnImpact +1));
DIFF2=$(diff output_large.txt output_large_faulted.txt | wc -l)
#echo $DIFF2;
if [ "$DIFF2" -gt "8" ]
then
echo "The␣fault␣injected␣has␣modified␣the␣behavior"
>> $FaultImpactPath;
```

```
FaultThatHasModifiedTheBehavior=
$(($FaultThatHasModifiedTheBehavior+1));
else
echo "The␣fault␣injected␣has␣modified␣the␣result"
>> $FaultImpactPath;
fi
else
echo "The␣fault␣injected␣has␣not␣impacted␣the␣application"
 >> $FaultImpactPath;
fi
rm "$FaultedResultFile"
else
echo "The␣fault␣injected␣has␣stopped␣the␣correct␣functionning"
>> $FaultImpactPath;
fi
#To avoid multiple times the same seed
sleep $DurationOfSleep
i=$(($i+1))
done
```

B.2 Explications

The previous section exposes an example of scripts used to obtain results thanks to our fault injection methodology. This example is extracted from a script evaluating the DECTED techniques exposed in Chapter 2. The parameters section helps the user to determine the number of runs for each loop and set up paths to different files. Then comes the cross compilation (available in the UNISIM-VP package). In this example, we first run the simulation without protection and without fault injection to obtain a base for the comparison of performance overhead. Then we change to *true* the parameter allowing the fault injection (memory. fault injection allowed). We use our methodology to inject faults by changing the injection-method to 4 and run this simulation. Of course, in this case no protection are used, we thus have the overhead bringed to the simulation time by the injection module. We finally protect the memory by modifying the robustness technique used to 9 which corresponds to DECTED.

This type of script is mandatory to evaluate reliability of a system with non-deterministic fault injection. Indeed, a large numbers of runs is needed and cannot be handled by human. I consider to arround one million my number of runs realized during my thesis on large benchmarks.

Appendix C Publications

1. Chabot A., Alouani I., Niar S., Nouacer R. (2021). A Memory Reliability Enhancement Technique for Multi Bit Upsets. Journal of Signal Processing Systems, 93(4), 439-459 March

2. Chabot A., Alouani I., Niar S., Nouacer R. (2019). A New Memory Reliability Technique For Multiple Bit Upsets Mitigation. ACM International Conference on Computing Frontiers, Sardinia, Italy, may

3. Chabot A., Alouani I., Nouacer R., Niar S. (2018). A Comprehensive Fault Injection Strategy for Embedded Systems Reliability Assessment. IEEE International Symposium on Rapid System Prototyping (RSP), october

4. Chabot A., Alouani I., Niar S., Nouacer R. (2018). A Fault Injection Platform for Early-Stage Reliability Assessment. Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools RAPIDO'18, Manchester UK, january .

Bibliography

[1] *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*. IEEE Computer Society, 1998. ISBN: 0-8186-9221-9. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5939.

[2] H. Abbasitabar, H. R. Zarandi, and R. Salamat. “Susceptibility Analysis of LEON3 Embedded Processor against Multiple Event Transients and Upsets”. In: *2012 IEEE 15th International Conference on Computational Science and Engineering*. 2012, pp. 548–553. DOI: 10.1109/ICCSE.2012.81.

[3] Accellera. *SystemC Standard Download page*. 2011. URL: http://www.accellera.org/downloads/standards/systemc.

[4] I. Alouani et al. “Parity-based mono-Copy Cache for low power consumption and high reliability”. In: *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*. 2012, pp. 44–48. DOI: 10.1109/RSP.2012.6380689.

[5] Stéphanie Anceau et al. “Nanofocused X-Ray Beam to Reprogram Secure Circuits”. In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 175–188. DOI: 10.1007/978-3-319-66787-4_9.

[6] G. . Asadi et al. “Balancing Performance and Reliability in the Memory Hierarchy”. In: *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* 2005, pp. 269–279. DOI: 10.1109/ISPASS.2005.1430581.

[7] A. Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. ISSN: 1545-5971.

[8] John Aynsley. *OSCI TLM-2.0 language reference manual*. JA32. Open SystemC Initiative. 2009.

[9] M. Bagatin et al. “Temperature dependence of neutron-induced soft errors in {SRAMs}”. In: *Microelectronics Reliability* 52.1 (2012), pp. 289 –293. ISSN: 0026-2714.

[10] P. Benjamin et al. “Simulation modeling at multiple levels of abstraction”. In: *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*. Vol. 1. 1998, 391–398 vol.1. DOI: 10.1109/WSC.1998.745013.

[11] Alessandro Birolini. *Reliability Engineering: Theory and Practice*. Mar. 2010. ISBN: 978-3-642-14951-1. DOI: 10.1007/978-3-642-14952-8.

[12] S. Borkar. “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation”. In: *IEEE Micro* 25.6 (2005), pp. 10–16. ISSN: 0272-1732. DOI: 10.1109/MM.2005.110.

[13] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. “Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2011, 52:1–52:12.

[14] A. Chabot et al. “A Comprehensive Fault Injection Strategy for Embedded Systems Reliability Assessment”. In: *2018 International Symposium on Rapid System Prototyping (RSP)*. 2018, pp. 22–28. DOI: 10.1109/RSP.2018.8631986.

[15] Alexandre Chabot et al. “A New Memory Reliability Technique for Multiple Bit Upsets Mitigation”. In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF ’19. Alghero, Italy: ACM, 2019, pp. 145–152. ISBN: 978-1-4503-6685-4. DOI: 10.1145/3310273.3321564. URL: http://doi.acm.org/10.1145/3310273.3321564.

[16] C. L. Chen and M. Y. Hsiao. "Error-correcting Codes for Semiconductor Memory Applications: A State-of-the-art Review". In: *IBM J. Res. Dev.* 28.2 (Mar. 1984), pp. 124–134. ISSN: 0018-8646. DOI: 10.1147/rd.282.0124. URL: http://dx.doi.org/10.1147/rd.282.0124.

[17] Eric Cheng et al. "CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores". In: *CoRR* abs/1604.03062 (2016). arXiv: 1604.03062. URL: http://arxiv.org/abs/1604.03062.

[18] M. Cirinei et al. "A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370342.

[19] Anand Dixit and Alan Wood. "Impact of New Technology on Soft Error Rates". In: *Reliability Physics Symposim (IRPS)* (2011), pp. 486–492.

[20] Japan Electronics and Information Technology Industries Association. *JEITA SER Testing Guideline*. 2005, p. 36.

[21] FIDES-Group. *Reliability Methodology for Electronic Systems*. 2010.

[22] D. D. Gajski and R. H. Kuhn. "New VLSI Tools". In: *Computer* 16.12 (Dec. 1983), pp. 11–14. ISSN: 0018-9162. DOI: 10.1109/MC.1983.1654264. URL: http://dx.doi.org/10.1109/MC.1983.1654264.

[23] K. R. Gandhi and N. R. Mahapatr. "Energy-Efficient Soft-Error Protection Using Operand Encoding and Operation Bypass". In: *21st International Conference on VLSI Design (VLSID 2008)*. 2008, pp. 45–51. DOI: 10.1109/VLSI.2008.116.

[24] A. Girault and H. Kalla. "A Novel Bicriteria Scheduling Heuristics Providing a Guaranteed Global System Failure Rate". In: *IEEE Transactions on Dependable and Secure Computing* 6.4 (2009), pp. 241–254. DOI: 10.1109/TDSC.2008.50.

[25] M. R. Guthaus et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: WWC '01 (2001), pp. 3–14.

[26] F. H. Hardie and R. J. Suhocki. "Design and Use of Fault Simulation for Saturn Computer Design". In: *IEEE Transactions on Electronic Computers* EC-16.4 (1967), pp. 412–429. ISSN: 0367-7508. DOI: 10.1109/PGEC.1967.264644.

[27] S. K. S. Hari et al. "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation". In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 249–258. DOI: 10.1109/ISPASS.2017.7975296.

[28] A. S. Hartman, D. E. Thomas, and B. H. Meyer. "A case for lifetime-aware task mapping in embedded chip multiprocessors". In: *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2010, pp. 145–154.

[29] A. Hava et al. "Integrated circuit reliability prediction based on physics-of-failure models in conjunction with field study". In: *2013 Proceedings Annual Reliability and Maintainability Symposium (RAMS)*. 2013, pp. 1–6. DOI: 10.1109/RAMS.2013.6517737.

[30] Peter Hazucha and Christer Svensson. "Impact of CMOS technology scaling on the atmospheric neutron soft error rate". In: *Nuclear Science, IEEE Transactions on* 47 (Jan. 2001), pp. 2586 –2594. DOI: 10.1109/23.903813.

[31] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82. ISSN: 0018-9162.

[32] Jia Huang et al. "A Framework for Reliability-Aware Design Exploration on MPSoC Based Systems". In: *Design Automation for Embedded Systems* 16 (Nov. 2012). DOI: 10.1007/s10617-013-9105-6.

[33] IBM. *IBM DS8880 Architecture and Implementation*. 2019. URL: https://books.google.fr/books?id=nSyKDwAAQBAJ&pg=PA376&lpg=PA376&dq=DECTED+ibm&source=bl&ots=76oc2xFLRQ&sig=ACfU3U3pLE0Vood2WAdVNMCxpYmpTTpuhw
hl=fr&sa=X&ved=2ahUKEwjhteDp39PpAhVlyoUKHcp0BuAQ6AEwBXoECAsQAQ#v=onepage&q=DECTED%20ibm&f=false.

[34] Y. Kagiyama et al. “Bit error rate estimation in SRAM considering temperature fluctuation”. In: *Thirteenth International Symposium on Quality Electronic Design (ISQED)*. 2012, pp. 516–519.

[35] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. “FERRARI: a flexible software-based fault and error injection system”. In: *IEEE Transactions on Computers* 44.2 (1995), pp. 248–260. ISSN: 0018-9340.

[36] J. Kim et al. “Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 197–209. DOI: 10.1109/MICRO.2007.19.

[37] M. Kooli and G. Di Natale. “A survey on simulation-based fault injection tools for complex systems”. In: *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2014, pp. 1–6.

[38] M. Kooli et al. “Software testing and software fault injection”. In: *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2015, pp. 1–6.

[39] Ravi Kumar. “ARM Architecture - Working Features”. In: (2013). URL: https://www.eeweb.com/profile/ravi-kumar-6/articles/arm-architecture-working-features.

[40] D. Li, J. S. Vetter, and W. Yu. “Classifying soft error vulnerabilities in extreme-Scale scientific applications using a binary instrumentation tool”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.29.

[41] Lin Li et al. “Soft error and energy consumption interactions: a data cache perspective”. In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758)*. 2004, pp. 132–137. DOI: 10.1109/LPE.2004.240852.

[42] S. Mittal and J. S. Vetter. “A Survey of Techniques for Modeling and Improving Reliability of Computing Systems”. In: *IEEE Transactions*

on Parallel and Distributed Systems 27.4 (2016), pp. 1226–1238. DOI: 10.1109/TPDS.2015.2426179.

[43] Gordon E. Moore. "Creaming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[44] Nhon Quach. "High availability and reliability in the itanium processor". In: *IEEE Micro* 20.5 (2000), pp. 61–69. ISSN: 0272-1732. DOI: 10.1109/40.877951.

[45] Reda Nouacer, Gilles Mouchard, and Daniel Gracia-Perez. "UNISIM Virtual Platforms". In: (Jan. 2012). RAPIDO'12 - 4th Workshop on: Rapid Simulation and Performance Evaluation: Methods and Tools.

[46] S. Ozdemir et al. "Yield-Aware Cache Architectures". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 15–25. DOI: 10.1109/MICRO.2006.52.

[47] Ludovic Pintard. "From safety analysis to experimental validation by fault injection - Case of automotive embedded systems. (Des analyses de sécurité à la validation expérimentale par injection de fautes ? Le cas des systèmes embarqués automobiles)". PhD thesis. University of Toulouse, France, 2015.

[48] M. K. Qureshi and Z. Chishti. "Operating SECDED-based caches at ultra-low voltage with FLAIR". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–11. DOI: 10.1109/DSN.2013.6575314.

[49] D. Radaelli et al. "Investigation of multi-bit upsets in a 150 nm technology SRAM device". In: *IEEE Transactions on Nuclear Science* 52.6 (2005), pp. 2433–2437. ISSN: 0018-9499.

[50] S Rehman, Muhammad Shafique, and J Henkel. *Reliable software for unreliable hardware: A cross layer perspective*. Jan. 2016, pp. 1–192. DOI: 10.1007/978-3-319-25772-3.

[51] B. P. Sanches, T. Basso, and R. Moraes. "J-SWFIT: A Java Software Fault Injection Tool". In: *2011 5th Latin-American Symposium on Dependable Computing*. 2011, pp. 106–115.

[52] Seongwoo Kim and A. K. Somani. “Area efficient architectures for information integrity in cache memories”. In: *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. 1999, pp. 246–255. DOI: 10.1109/ISCA.1999.765955.

[53] Y. Song et al. “Experimental and analytical investigation of single event, multiple bit upsets in poly-silicon load, 64 K*1 NMOS SRAMs”. In: *IEEE Transactions on Nuclear Science* 35.6 (1988), pp. 1673–1677. DOI: 10.1109/23.25520.

[54] G. Tsiligiannis et al. “Multiple-Cell-Upsets on a commercial 90nm SRAM in dynamic mode”. In: *2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2013, pp. 1–4. DOI: 10.1109/RADECS.2013.6937429.

[55] Raoul Velazco, Pascal Fouillat, and Ricardo Reis. *Radiation Effects on Embedded Systems*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 1402056451.

[56] Wei Zhang et al. “ICR: in-cache replication for enhancing data cache reliability”. In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings*. 2003, pp. 291–300. DOI: 10.1109/DSN.2003.1209939.

[57] D. Zhu and H. Aydin. “Reliability-Aware Energy Management for Periodic Real-Time Tasks”. In: *IEEE Transactions on Computers* 58.10 (2009), pp. 1382–1397. ISSN: 0018-9340.