# A Memory Reliability Enhancement Technique for Multi Bit Upsets

Alexandre Chabot[1,2], Ihsen Alouani[3], Réda Nouacer[2], Smail Niar[1]

**Abstract**

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 meeting 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 techniques referred to as DPSR: Double Parity Single Redundancy. DPSR is designed to enhance computing systems resilience to SBU and MBU. Based on a thorough fault injection experiments, DPSR shows promising results; It detects and corrects more than 99.6% of encountered MBU and has an average time overhead of less than 3%.

*Keywords:* Reliability, MBU, Fault Injection, Memory

## 1. Introduction

Thanks to manufacturing process and integration improvements, modern mobile and embedded systems are now able to execute complex applications with advanced functionalities, such driver assistant systems in autonomous automotive, drones etc.

Consequently, System-on-chip (SoC) architectures are becoming increasingly complex and the underlying hardware has a particular impact on the energy consumption, performance and reliability. In fact, soft errors phenomenon represents a serious challenge to new computing systems. Soft errors result from a voltage transient event induced by alpha particles from packaging material

☆

*Email addresses:* alexandre.chabot@cea.fr (Alexandre Chabot), ihsen.alouani@uphf.fr (Ihsen Alouani), reda.nouacer@cea-list.fr (Réda Nouacer), smail.niar@uphf.fr (Smail Niar)

[1]LAMIH, UMR CNRS, Université Polytechnique Hauts-de-France, France

[2]CEA-LIST, France

[3]IEMN, UMR CNRS, Université Polytechnique Hauts-de-France, France

or neutron particles from cosmic rays [1]. The event is created through the collection of charge at a p-n junction after a track of electron–hole pairs is generated. A sufficient amount of accumulated charge in the struck node may invert the state of a logic device, such as a latch, static random access memory (SRAM) cell, or logic gate, thereby introducing an error into the hit circuit. In past technologies, this issue was considered in a limited range of applications in which the circuits are operating under aggressive environmental conditions like aerospace applications. Nevertheless, shrinking the transistor size and reducing the supply voltage in new technologies result in a remarkable decrease of the capacitance per transistor leading to a higher vulnerability within circuits nodes[2]. Hence, soft errors become a serious challenge in complementary metal–oxide–semiconductor (CMOS) circuits, especially for memories. Moreover, the Semiconductor Industry Association (SIA) roadmaps indicate that embedded memories are exceeding 90% of the chip area [3]. Consequently, the overall systems reliability is considerably affected by the memory immunity to errors. Despite of the numerous published works, memories reliability enhancement is still an open problem especially for critical applications.

For this reason, next generation embedded systems have to be more resilient to transient faults. 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).

However, in most of the existing approaches real environmental factors are not taken into account. Moreover, 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 .

In this paper, we present a memory reliability technique and provide a comparison with related techniques based on different metrics. In Section 2, we expose a state of the art about fault models, fault injection techniques and memory reliability techniques. In Section 3, we present our first contribution, which corresponds to an improved version of the Double Parity Single Redundancy technique. In Section 3 we present, our second contribution, a new methodology to inject fault during simulation. Our simulation-based fault injection methodology is detailed in this section. Thanks to this methodology we compare different memory reliability enhancement techniques. Finally, we conclude our work in Section 4.

## 2. State of the Art

In this section, we first give basic definitions. we then survey existing methods to model and simulate single and multiple bit upset in SoC. We also present existing methods to improve system reliability.

### *2.1. Fault Types*

Embedded systems are subject to faults whose distinction is made upon their duration. The three types are the following [4] [5]:

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 changing the hardware. Due to functioning or fabrication issues, permanent fault occur mainly due to three different causes [6]:
   - Manufacturing and Design Time: those faults comes from error in the design or in the manufacturing process of the Hardware and manifest as stuck at one/zero and delay.
   - Wearout Mechanisms: those mechanisms are influenced by the aging of the system. Negative-Bias temperature instability, hot carrier injection, time-Dependent dielectric breakdown and electro-migration 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.
2. **Intermittent Fault**. Intermittent faults occur sporadically. They do not appear continuously but rather at irregular intervals. Intermittent faults are often considered as early indicators of potential permanent faults.
3. **Transient Fault**. *Transient faults* are logical faults in circuit's nodes that occur in a random manner mainly due to charged particle emissions [7]. The fault is manifested by one or more bit flips or computation error. This change is called a *single event* and can cause a single or a multiple upset Transient faults are non-permanent faults. The system is only perturbed during a small amount of time. The time of the perturbation is reduced to an instruction at the application level. The metric used to evaluate the sensitivity of the system to its environment is the soft error rate (SER) [8]. 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. Besides, the atmospheric neutrons are usually separated in two categories based on their energy: atmospheric neutrons with an energy inferior to 1 MeV and those with an energy higher than 1 MeV. In the space environment, it exists different radiation sources

such as: Van Allen radiations, solar activity and cosmic radiations [9]. Energies of those cosmic particles vary between some MeV and up to $10^{30}$.

The main focus of our study concerns transient faults. SER determines the number of soft errors per unit time. SER unit is the Failure In Time (FIT); which represents the number of failures expected for a device during one billion functioning hours.

### *2.1.1. Multiple Bit Upsets*

To maintain the Moore law prediction with the reality [10], transistor size has been reduced. This size shrinking has a direct impact on the sensitivity of Hardware to soft errors with the apparition and the raise of multiple faults observed for newest technologies. This new phenomena is firstly highlighted in [2] which shows that transistor miniaturization goes with the rise of single event multiple bit upsets.

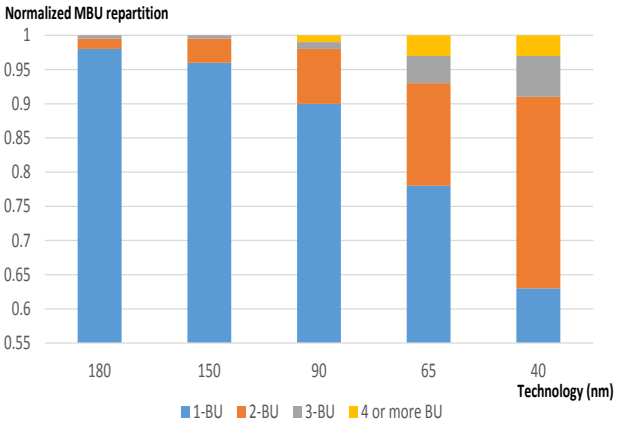


Figure 1: Single and Multi Bit Upset (BU) percentages by technology nodes in nm for SRAMs [2]

Figure 1 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 [2]. Usually, single event were linked to a single upset. With the rise of multiple upsets, Figure 1 shows that this hypothesis is valid only only for previous technologies.

In Table 1, we present results obtained by Radaelli et al. [11] regarding the distribution of soft errors in 150nm commercial available SRAMs. The second line of the table is linked to the Table 2. For example, a 1-2 configuration corresponds to all upsets where two horizontally adjacent bits are flipped. This table shows that for single event 2-bit upsets, it is more frequent to observe two horizontally adjacent flips than two vertically adjacent ones. Multi-cell upset

Table 1: Cumulative 2-Bit Event Count Normalized to 1000 for 150nm SRAMs for Different Particle Strikes Energy (MeV) [2]

| 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: Pattern Injection Square

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

events tend to be a concern especially for patterns that flip multiple bits in the same row [2].

*2.1.2. Probabilistic Model*

Even though the bit SER saturate or even decrease for latest technologies, the system SER 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. In our work we focused our study on soft errors impacting memory.

First, depending on the impacted memory region, the flip operation may alter either a data value or an instruction code, but this information is not taken into account when creating the fault appearance probabilistic model. The reliability law is given by Equations (1) and 2, where $\lambda$ is the constant failure rate, $R$ is the reliability distribution, $MTTF$ is the mean time to failure and $t$ is the time.

$$R(t) = \exp(-\lambda * t) \tag{1}$$

$$MTTF = 1/\lambda \tag{2}$$

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 [13] and [14] who considered environmental conditions. In [13], failure rates are defined based on temperature while in [14], authors take power consumption into account. FIDES global electronic reliability engineering methodology guide is a generic approach to compute architectures failure rates [15]. Based on FIDES guide [15], physical and process impacts have to be considered for a precise failure rate $\lambda$ 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 $\lambda$, environmental conditions are modeled by providing: ambient temperature,

temperature cycles, relative humidity, vibrations, saline pollution, environmental pollution, application pollution and chemical protection.

$$AF_{temperature} = exp(\frac{Ea}{Kb}(\frac{1}{T_0} - \frac{1}{T})) \tag{3}$$

$$AF_{humidity} = (\frac{H}{H_0})^p * exp(11604 * Ea * \frac{1}{T_0 + 273} - \frac{1}{T + 273}) \tag{4}$$

$$AF_{vibrations} = (\frac{G_{RMS}}{G_{RMS0}})^p \tag{5}$$

In Equations 3, 4 and 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 $= 0.5 G_{RMS}$

This model is usable in different processes to evaluate system reliability at different stages of the system development life-cycle. In particular, we inspired our fault model used during our fault injection to the presented state of the art. Our fault model will be exposed in the Section 4.

### *2.2. Fault Injection Techniques*

Fault Injection has been studied since decades now. Up to our knowledge, the first paper dates of 1967 [16]. 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 [5]. Finally, reducing the risk to discover unexpected scenario after the commercialization of the product.

### *2.2.1. Definitions*

All fault injection environments are usually composed by the following components [17], [6], [18]:

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.

The components just presented vary in implementation 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 [19], 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 specific component in the circuit is very complicated as the technology evolves. For instance, in [19] on 60nm technology, means used to target specific transistor were very complicated to set up and costly.
2. Virtual platform or Simulation-based Fault Injection: When used, this technique imposes the development team to dedicate time to develop a simulation tool representative of the hardware expected [18]. Once done, the Fault injection can be applied at different levels:from transistor up to algorithm level. 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 simulator 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 [17]. The lack of hardware behavior consideration is the major drawback of this technique as it is not representative of the final system. Furthermore, we modify the software, we thus need to be careful when removing the added code by ensuring the system remaining reliable. During certifications processes, the final system is evaluated and issues may happen when the code furnished is not the one tested.

*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 [20] 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 challenges about simulation is to select the correct level of abstraction [21]. Choosing the level of abstraction depends on the type of information the designer would like to obtain and the desired speed of simulation. Different abstraction levels of simulation exists in system simulators [22]: High level System, Transnational, Timed Transnational, Register-Transfer (RTL), Gate, Transistor, etc.

Simulating a SoC at a high level of abstraction allows to modelize easily a complex system and permits important speed in running the application on the simulated architecture. However, it does not make possible to measure execution time in the simulator nor to extract memory behaviour. At the opposite, low level simulation, such as RTL or transistor level, allows to have accurate measurements at the cost of low execution rate of instruction by the simulator. Ideally having a trade-off between measurement accuracy and simulation speed is interesting. For this reason, the timed transaction-level modeling (or timed

Table 3: Comparison of Fault Injection Tools

| Name | Injection Level | Determinism | MBU |
|---|---|---|---|
| LEON3 [20] | architecture | random | yes |
| FERRARI [25] | software | free choice | possible |
| J-SWIFT [26] | software | random | possible |
| BITFIT [27] | prototype | model-based | possible |
| SASSIFI [28] | control flow | random | no |
| GeFIN [29] | micro-architecture | random | yes |

TLM) have been chosen in this work. Timed TLM [23, 24] offers the possibility to explore a large set of architectures in a relatively reduced period of time with a good level of accuracy.

Table 3 compares some of the existing fault injection method. The fault injection tools presented are usually associated with different way to determine the injection type, location and its plausibility. J-Swift [26] and Ferrari [25] are example of tools that propose fault injection to evaluate system robustness However, up to our knowledge, only one work has included multiple bit upsets into injection fault library [20]. This work has been made onto LEON3 architecture. The authors use random fault injection in time and in memory location.

### *2.3. Reliability Techniques and Means*

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 [18]:

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.

In the CLEAR project [30], authors compare cross layer reliability techniques. The authors assume that the environment impact on the memory is limited to single upsets.

In the rest of this section, we present some of the existing memory reliability techniques used to prevent single upsets.

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. Such as showed in Figure 2, 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.

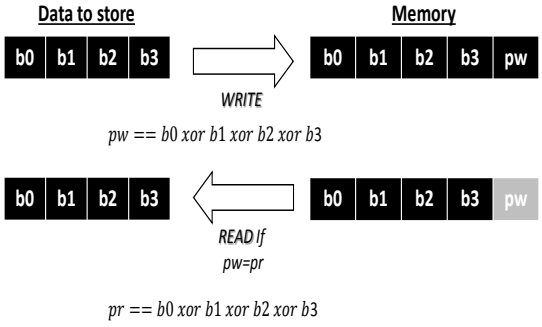


Figure 2: Parity Functioning

2. **Double and Triple Memory Redundancy (DMR/TMR)**. As shown Figure 3 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 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.

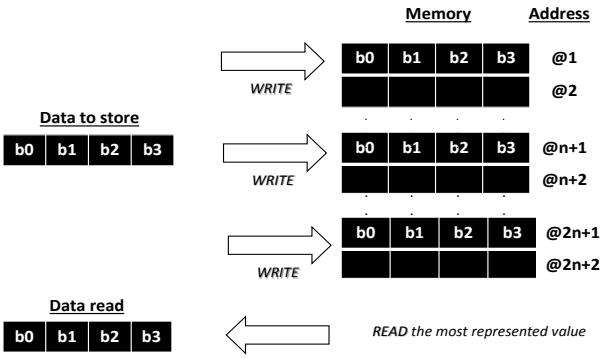


Figure 3: TMR Functioning

Other solutions have been developed to address specific need for robustness by replicating different part of the hardware. However, these solutions go with a rise in cost and complexity as sometimes a single erroneous bit makes a entire part of the memory unusable. With process variations increase, the solution seems to reach its limits [31], [32].

3. **Parity-Based Mono-Copy Cache (PmC2)**. In [33], authors propose to combine the double memory redundancy and the parity to create the PmC2 technique. Such as shown Figure 4 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.

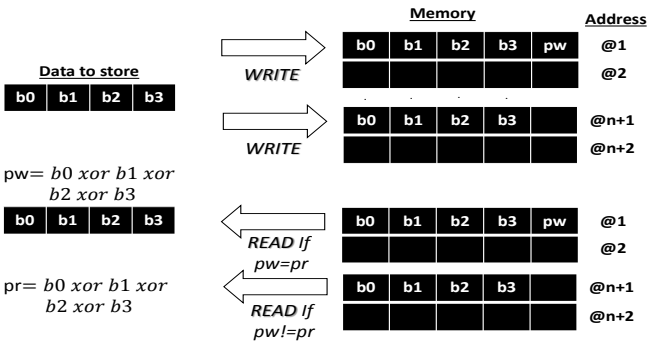


Figure 4: PmC2 Functioning

4. **Single Error Correction Double Error Detection (SECDED)**. Even if it exists optimized version of the Single Error Correction Double Error Detection mechanisms [34] the principle stays the same for all implementations. We base our study onto SECDED codes based on Hamming codes. Such as showed Figure 5 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 6, 7, 8, 9, and 10 give an example of SECDED implementation for 8 bits data words. During the write operations, $px$ bits are computed and stored together with the $dx$ bits. A last protection bit (called $p4$ Equation 10) is added that is a xor between all the other $px$ bits but is not represented in Figure 5. 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 when the data size to protect raises.

$$p0 = d0 \oplus d1 \oplus d3 \oplus d4 \oplus d6 \tag{6}$$

$$p1 = d0 \oplus d2 \oplus d3 \oplus d5 \oplus d6 \tag{7}$$

$$p2 = d1 \oplus d2 \oplus d3 \oplus d6 \tag{8}$$

$$p3 = d4 \oplus d5 \oplus d6 \oplus d7 \tag{9}$$

$$p4 = p0 \oplus p1 \oplus p2 \oplus p3 \tag{10}$$

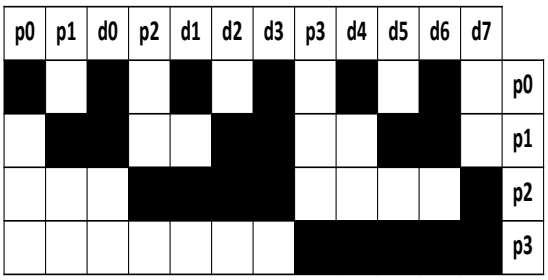


Figure 5: SECDED example for 8 bits data word

5. **Double Error Correction Triple Error Detection (DECTED)**

First time published in the beginning of 1980s [35], this technique is now used in safety critical systems. 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 detailed in [35] 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. In the literature we may find techniques that derive from SECDED or DECTED, such as [36]. They tend mainly to reduce time or hardware complexity of the encoding and the decoding.

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 [37]. 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 [38].

As we can identify here, solutions proposed to protect the memory are either not efficient against multiple bit upsets, or too complex and thus time and energy consuming or hugely impacting the memory size which is critical for embedded systems. In the next Section, 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.

## 3. Double Parity bit Single Redundancy

### *3.1. Presentation and Motivation*

As explained in previous Sections, the MBU phenomena is becoming more crotical with technology scaling down along with the high performance requirements for time-critical applications. To address this problem, we propose a new memory reliability enhancement technique considering MBU patterns.

TYhe proposed solution consists of a double parity bit associated with data redundancy. We refer to it as DPSR: Double Parity Single Redundancy. DPSR objective is to cope with most encountered MBU patterns in a comprehensive manner. Contrary to state-of-the-art techniques that assume that the MBU location is a totally random phenomenon, our technique takes into account spatial MBU pattern probabilities. However, as shown in Section 2.1.1, this assumption is not accurate. In fact, the particularity of multiple faults occuring within memory cells is the non uniformity of the spatial error distribution with respect to proximity of flipped bits [2]. Regarding this particularity, we suggest to use two parity bits for the detection. As showed later in Section 3.2.1, the

proposed technique detects more than 99.6% of encountered upsets. Adding a third bit would rise this percentage of 0.3% but requires more hardware resources to be implemented Using a fourth bit is enough to correct all considered fault patterns. However, it results in a high area overhead that is not only problematic from a resource utilization perspective, but also increases the circuit exposure to transient events. For this reason, we decided to stick to only two parity bits.

In terms of redundant data location, we decide to place the redundancy in a separate **non adjacent** memory location to the initial word to avoid its corruption within the same event. This redundant storage will be useful for data recovery in case of a detected corruption. Hence, our technique deploys 2 bits for fault detection and redundancy for error correction. It's worth noticing that depending on the required reliability, this technique can be adapted for detection only, or for detection and correction.

Figure 6 gives an illustrative overview on the proposed technique during a write operation in memory for an 8 bits-word. When data is stored, two parity bits are computed. Equations 11 and 12 give the formula for the even and the odd parity bits in the case of an 8 bits-word. Once both bits are computed, the write operation consists of storing the initial word, the redundant word as well as the parity bits. The redundant data, as mentioned beforehand, is not stored in adjacent addresses but rather in different memory location.

As shown in Figure 6, bits $p_o$ and $p_1$ correspond, respectively, to even and odd parity bits of the original data. During a read operation, as illustrated by Figure 7, the original data is read. Even and odd parity bits are computed for the read value. The computed even and odd parity bits are compared to the stored 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, either that data value or the parity bits has been corrupted. Therefore, the read operation returns the redundant data instead.

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 in [2] it represents less than 2% of 2-bit-upsets, which makes less than 0.6% of total observed patterns for 40nm SRAM technology. Moreover, in the case of a 3-bit-upset, the only pattern that may lead to corruption even with our solution is when three horizontally aligned bits are flipped. The probability 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 MBU.

$$p0 = b0 \oplus b2 \oplus b4 \oplus b6 \tag{11}$$

$$p1 = b1 \oplus b3 \oplus b5 \oplus b7 \tag{12}$$

### *3.2. Probabilistic comparison for DPSR*

In this section we evaluate existing memory reliability techniques and our technique against the data provided by [2] and exposed in Section 2. The

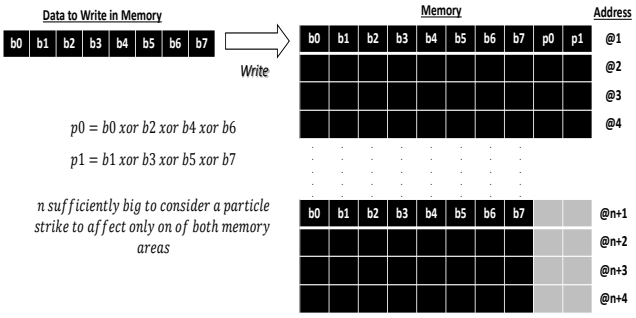


Figure 6: DPSR Write for 8 bits word

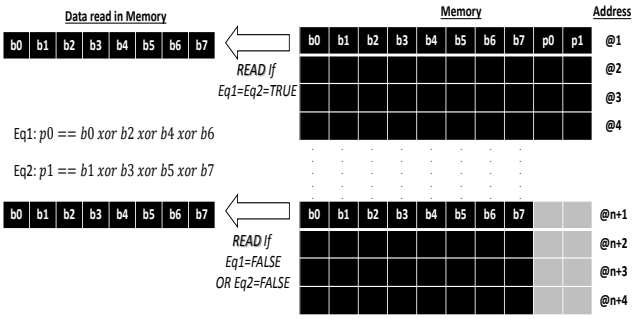


Figure 7: DPSR Read for 8 bits word

upset due to a single particle strike depends on the carried energy. To consider this parameter, the study of memory reliability techniques is made for different particle strikes energy. The choice of the implemented reliability technique depends on orthogonal parameters. For this reason, choosing a perfect reliability protection is impossible. Indeed, in the case of a multi-objective problem, it is impossible to maximize all parameters. The goal in such problems is to find solutions that offer interesting trade-off among all solutions. The choice of a memory reliability technique is crucial and has to be a trade-off between: error detection and correction, memory space, and delay induced by the reliability technique. In this probabilistic study, we evaluate the detection probability, the correction probability and the memory space used by the reliability technique. We assume that, due to the different locations of initial data and redundant data, faults occurring within the initial data do not result in bit flips in the redundant one. The probabilistic model is based on Equation 13 and data used are from [2]. In Equation 13, $p_{BU}$ and $p_{shape}$ correspond to the probability to

Table 4: 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 |

observe $1, 2, 3$ or 4 BU and to the shape of upsets to inject, respectively. They both depend on the memory technology and on the particle energy. Finally $p_{fault}$ is the probability to observe a given fault pattern.

$$p_{fault} = p_{BU}(technology, particle\ energy) * p_{shape}(technology, particle\ energy) \tag{13}$$

### *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 proposed technique with: parity technique (that has the same detection rate as the PmC2 technique), DMR, SECDED, DECTED techniques. The detection probability $p_{detection}$ is computed using Equation 14, where $p_{detection fault}$ equals to 1 if the technique detects the type of fault, and to 0 elsewhere. We assume a single fault affecting only one memory area. As shown in Table 4, all techniques have the same rate for detecting single faults. However, this rate goes down when multiple upsets appear. The DMR is the best technique to detect multiple faults, the DPSR that is our proposed technique is close to what SECDED and DMR achieve for detection rate.

$$p_{detection} = \sum_{fault} (p_{fault} * p_{detection fault}) \tag{14}$$

### *3.2.2. Correction*

The correction rate is the probability of a technique to detect and correct an error in a given environment. The correction rate $p_{correction}$ is computed following Equations 15 and 16 where $p_{correction fault}$ equals to 1 if the technique corrects the type of fault and to 0 elsewhere. In this Section, we compare DPSR correction rate with $PmC2$ [33], TMR and SECDED techniques. Table 5 shows that the proposed technique outperforms SECDED. In fact, SECDED detects up to 2 upsets but is only able to correct single fault in a line. TMR has the best correction rate but DPSR is close to its results.

$$p_{correction} = \sum_{fault} (p_{fault} * p_{detection fault} * p_{correction fault}) \tag{15}$$

$$p_{correction} = p_{detection} * p_{correction fault} \tag{16}$$

Table 5: 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 6: Memory space overheads (in bits) for different techniques function of the considered 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 Overhead*

In this Section we compare memory techniques presented in Section 2.3 for 4 memory word sizes. For this purpose, we compare the overhead of $PmC2$, TMR, SECDED, DECTED techniques with DPSR. As shown in Table 6 and Table 7, SECDED has the lowest memory overhead when protecting wider words. DPSR uses one more bit to protect data than $PmC2$ but as shown earlier, it has better detection and correction rates. The worst is obviously the TMR technique because of the resource overhead.

Overall, we showed that DPSR offers high reliability with low timing overhead. In fact, DECTED is a technique that guarantees high robustness to errors but has a significant timing overhead mainly caused by the propagation delay of the error correction codes circuitry. This time overhead is systematic, i.e., it is consumed every single memory access regardless of the error occurring. However, in our case, the parity circuitry is much lower in size as shown earlier. The time overhead resulting from reading the redundant data is susceptible to happen only in the case of a detected error in the initial word. Since these events are rare by nature, this time overhead is practically insignificant overall. In the context of time-critical applications, this advantage is very valuable. The

Table 7: Relative memory space overheads for different reliability techniques function of the considered data size. Values in this table are computed using Table 6

| 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 |

second advantage of our technique is its flexibility. In fact, in the case of low to moderate criticality applications, the correction part of proposed technique that is implemented through redundancy can be dropped. Changing the reliability mode is as easy as using a single multiplexer and addressing the redundancy-dedicated space in the memory space.

### *3.3. RETG: Reliability Enhancement Technique Grade*

As we can identify in previous sections, different criteria are used to evaluate reliability techniques. Some criteria are antagonistic such as the correction probability and the memory space used. Others are highly correlated to each other such as the complexity of the algorithm and the power consumption. In the following, we propose a new metric to easily compare reliability techniques. First 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 and thus to consider only the computation overhead to represent both. Finally the memory overhead is also a criteria linked to the power consumption but raises also other concerns regarding embedded systems. Consequently, we take the memory space as a third criteria. Equations 17 and 18 are provided to understand our way to compute each of RETG criteria, regarding detection and correction.

$$RETG_d = \frac{p_{detection}}{MemOv * PerfOv} \tag{17}$$

$$RETG_c = \frac{p_{correction}}{MemOv * PerfOv} \tag{18}$$

In Equations 17 and 18, $PerfOv$ and $MemOv$ as computed thanks to Equations 19 and 20, 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.

$$PerfOv = \frac{Time_{protected}}{Time_{unprotected}} \tag{19}$$

$$MemOv = \frac{datasize + techniquesize}{datasize} \tag{20}$$

We want now to compute all those parameters for all reliability techniques previously presented. Such as stated in 2, we use a virtual platform to evaluate those techniques. In the next Section, we will explain the structure and the functioning of our fault injection tool.

## 4. Structure of the Fault Injector

### *4.1. Overview*

Our fault injection strategy is exposed in Figure 8. The main objective of this strategy 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?

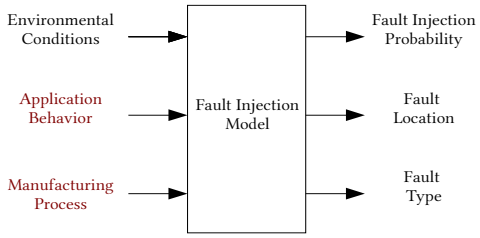


Figure 8: The Fault Injection Model

Our strategy is based on FIDES standard and on MBU patterns exposed previously in Section 2. To go further, we take also into account the locality of memory accesses that we are going to present and justify in the next Section.

### *4.2. Memory Accesses impact onto Fault Injection*

Indeed, during the simulation, we propose to take into account accesses. Depending on the technology and on operating conditions, the more a memory zone is accessed, the more likely an error occurs 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 zone dynamically. Therefore, the probability to inject is different for each memory zone as shown in Equation 21. In this equation, $f_i$ is the frequency of access to the i$^{\text{th}}$ memory zone. $\pi_i$ represents the fault injection probability in the considered zone depending on $f_i$. In the experiments $InjectionLocality$ is implemented using Equation 22 where $\alpha$ is a tunable coefficient to make the fault injection more or less focusing onto more accessed areas. In the experiments $\alpha$ has been set to 1.5 for experiments.

$$\pi_i = InjectionLocality(f_i) \tag{21}$$

$$InjectionLocality(f_i) = \alpha \frac{f_i}{\sum_{i=1}^{n} f_i} \tag{22}$$

Authors in [39] 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 [40]. 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, is considered for the first time in simulation-based reliability evaluation. We believe that it is important to inject both single upsets and multiple upsets to improve representativeness of the proposed fault injection model. To achieve accurate representation of MBU phenomenon, we identify the probability of MBUs patterns. As shown in [11], depending on the technology and the number of flipped bits during a particle strike, different spatial patterns have different likelihood to happen. Table 9 is an example of data measured for a 150nm SRAM regarding multiple bit upset [11]. 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 8. Finally, the total probability is computed and indicated Table 9 in Column *(1)*(2) Probability*.

Table 8: Pattern Injection Square

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Table 9: Pattern Flipping Probability for 150nm technology [11]

| 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 9 describes the proposed fault injection mechanism. The system failure rate is computed at the beginning of the simulation based on data provided by the user. Then, the fault injection location is computed thanks to the access profile. The shape of the fault is determined thanks to MBU patterns. Finally a diagnostic is established by comparing results in the corrupted run and golden run.

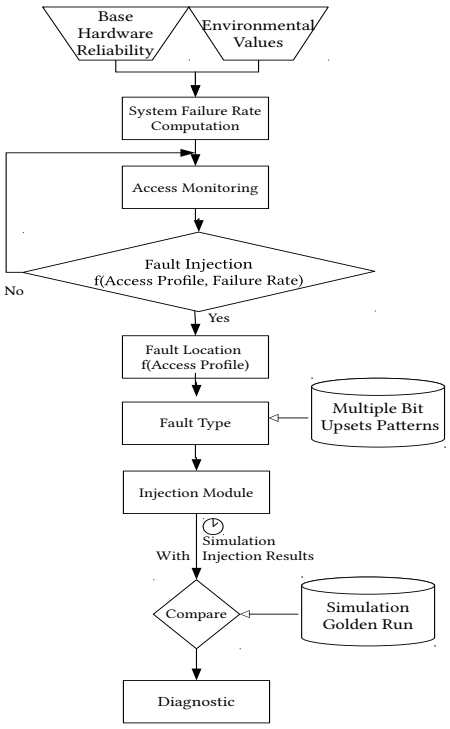


Figure 9: Fault Injection Strategy

## 5. Experimental Results

### *5.1. Experimental Setup*

For evaluating memory reliability techniques impacts on performance we used the Mi-bench applications [41]. We focus onto 5 applications in Mi-Bench with the large input data.

- Qsort: Efficient sorting algorithm, still used today in a large variety 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: image recognition and modification application depending on the mode selected. Two modes allow to detect edges and corners, the other spreads the input image.

All these applications have been cross-compiled to work properly onto our armv7 simulator. A large number of simulators [42] 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 [23], 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 [24] 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. Services range from debuggers, loaders, monitors, host hardware abstraction layer and of course our fault injection module.

We use UNISIM-VP as our virtual platform because of it's transactional model that enable 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 regard of different criteria and metrics:

1. The efficiency: our approach is able to raise reliability issues and cover the system testability (Section 5.2.1).
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 5.2.2).
3. The simulation speed (Section 5.2.3).

#### *5.2.1. Efficiency*

Figure 10 shows obtained results for 11000 runs with four different injection experiments on Susan benchmark. Similar results have been obtained for other benchmarks. For all cases, we inject faults during writes in memory and we divided the global memory in 262144 areas. In the **SBU** experiments only single bit flips are injected while in the **MBU** experiments, multiple bit flips using the probabilistic model given in Table 9. 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 monitoring in the fault injection (SBUA and MBUA), more result and behavioral corruptions occur. Indeed, for the same number of injections, Figure 10 shows that, SBUA (respectively MBUA) increase the behavioral corruptions by 8.26% (respectively 7.7%) compared to SBU (respectively MBU). By consequence, rising

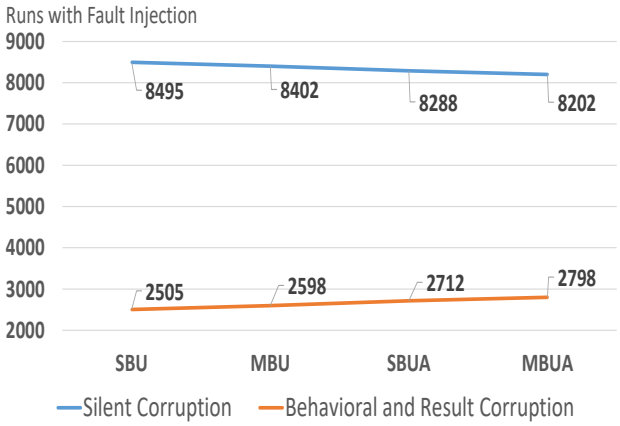


Figure 10: Type of observed corruption with different injection procedures on Susan smooth bench with 11000 runs for each procedure

the probability to inject into the most frequently accessed areas, widely used variables are effected by the fault injection.

The second conclusion 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, Figure 10 shows 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 these results to a pure random distribution of fault injection inside the memory would be inappropriate. Indeed, 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. We thus comparing our results to state of the art technique that is SBU. We remind the reader that during SBU injections, only accessed parts are modified by the injection.

Figures 11 and 12 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 are ranked and presented in these two figures (Figure 11 and 12) representing results obtained for Susan Corner (respectively Smooth). For these 2 benchmarks, 2000 runs (respectively 4000) have been done. For Susan Corner Figure 11 (respectively Susan Smooth 12), our UNISIM fault module injected 1500 faults (respectively 4000). As we can see in Figures 11 and 12, 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 that are swapped together in Figure 11, but it is not observable in Figure 12. 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. Even if not presented in given figures, we can also notice that almost all areas will be perturbed during a fault injection campaign. By consequence the injection model matches with the statistical testing mind spirit and also solves the concern to miss some memory areas.

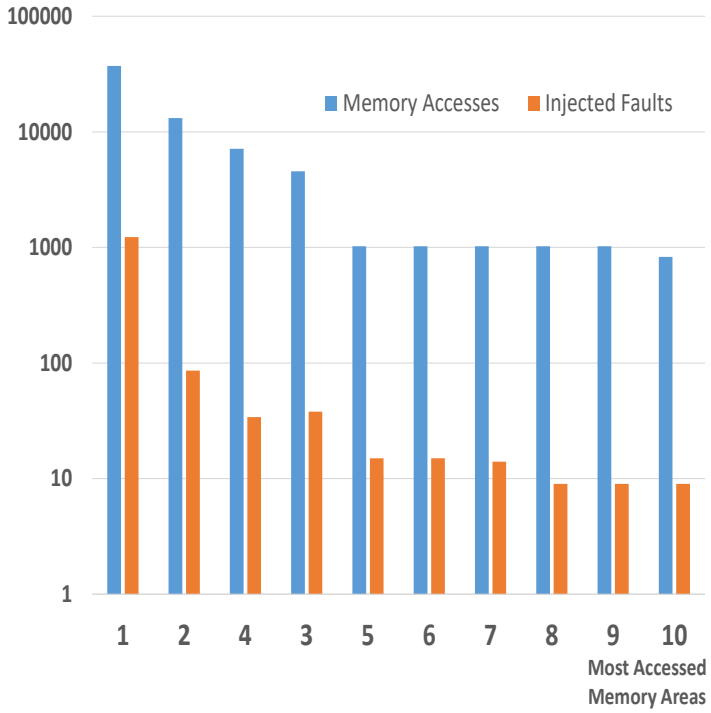


Figure 11: Number of memory access compared to number of injection in most accessed memory areas for Susan Corner Mode Application. Values are given for the 10 (x-axis) most accessed memory zones.

*5.2.2. Representativeness*

Figure 13 shows two distributions of fault patterns. The first is the patterns distribution given Table 9. The second distribution is the result of 443 injections made on different MiBench applications using the MBUA procedure. Figure 13 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%

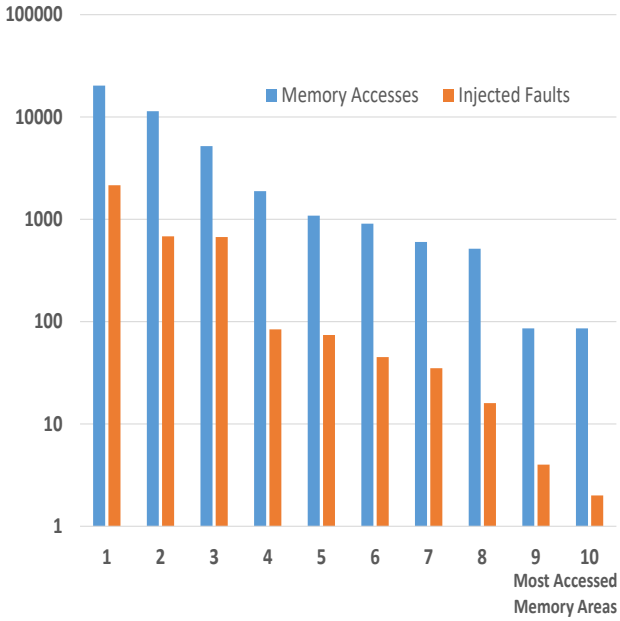


Figure 12: Number of memory access compared to number of injection in most accessed memory areas for Susan Smooth Mode Application. Values are given for the 10 (x-axis) most accessed memory zones.

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 8) is located on the extreme right bit of a line, then it's not possible to inject a 1-2 MBU. In a case like that, 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 decision has been made to avoid losing simulations runs and explains the difference between both distributions presented Figure 13. Furthermore, Figure 13 helps also to understand the difference observed between SBU(A) and MBU(A) in Section 5.2.1. As we base our MBU on a realistic model exposed Table 9, the difference is not as sensitive as if we would have considered only MBU in MBU and MBUA injections procedures.

Figure 14 shows the normalized distribution of corruptions regarding the single bit upset (top graphic) and the 2 bits upset (bottom graphic). Results have been obtained running the MBUA injection procedure based on MBU patterns exposed in Table 9. First, all type of injections have in majority resulted to an

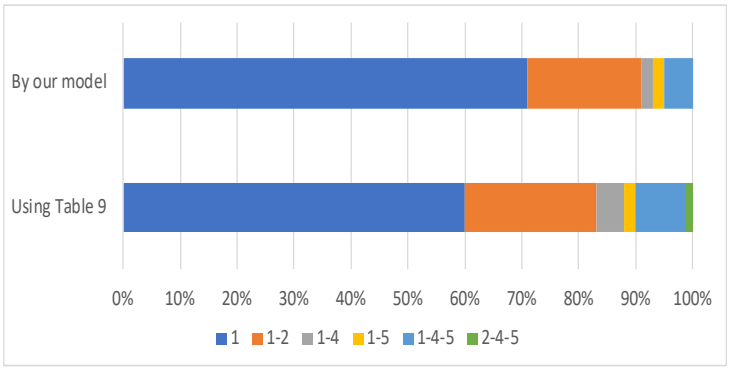


Figure 13: Distribution of injected upsets patterns given by [11] in Table 9 and those injected by our model in the simulator

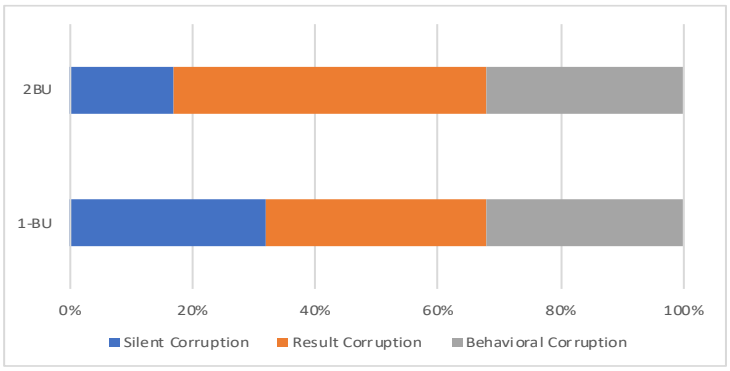


Figure 14: Single and 2-BU corruptions distribution

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. these 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 final 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 scenarios leading to reliability decrease are discovered. As explained 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.

### 5.2.3. Simulation Overhead

Figure 15 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. these applications represent a mix of instructions and computation intensive applications. First, Figure 15 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 slim drastically the sorting of access regarding the address accessed.

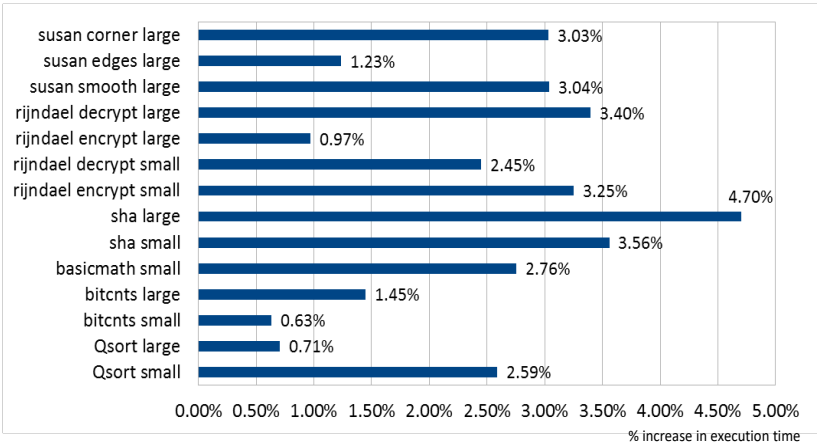


Figure 15: Simulation time increase after implementing fault injection module

This simulator is accompanied with a fault injection module presented in [43]. This fault injection module is configurable with environmental conditions as well

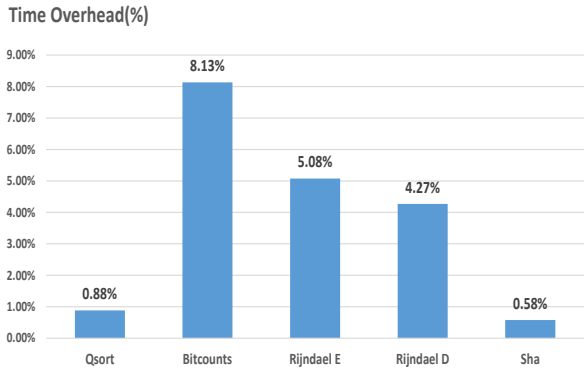


Figure 16: Simulation overhead due to read and write monitoring without memory protection

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 [43] authors suggest fault injections to be only made during write operations. This limitation has been over-passed in the used version of the simulator because we improved the injector to be enable to inject both during read and write operations without considerable impact on the simulation performance. As exposed in Figure 16, 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 [43], the simulation overhead on large benches was in the worst case of 5% when only write operation were subject to fault injection.

Moreover, as shown in Figure 17, three outcomes are possible for a system under a fault injection: the system crashes (STOP FUNCTIONING), or the system ends but with a result different from the golden result (RESULT CORRUPTION) or the system ends with the same result as with a fault free simulation (NO IMPACT). Figure 17 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 [43] where more than 20% of simulation fault injection runs on unprotected systems resulted in no corruption.

### *5.3. RETG computation*

#### *5.3.1. Memory Reliability Techniques impact on performance*

Figure 18 shows simulation time on different applications when different reliability techniques are applied. Data are collected among 5 benchmarks presented Section 5.1. For all reliability techniques implemented and for each application 25 runs were done to compute the mean simulation time. The mean

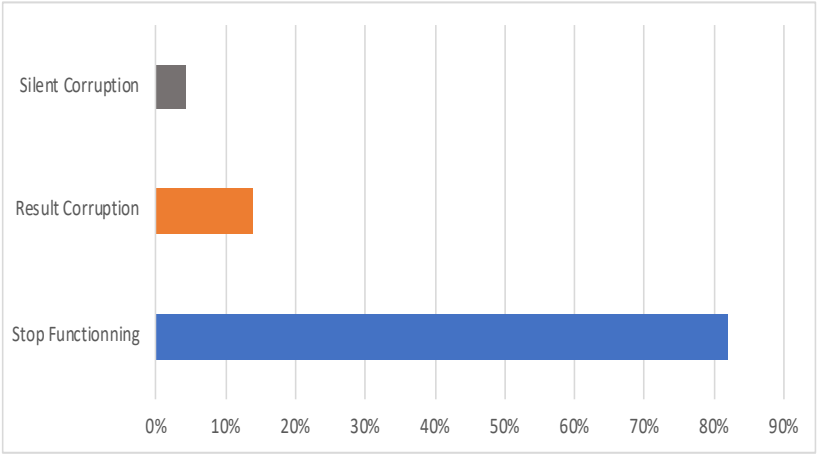


Figure 17: Distribution of simulation results after read fault injection on unprotected system for different application on 19000 runs

Table 10: Performance Overhead due to memory reliability techniques

| PmC2 | TMR | SECDED | DPSR | DECTED |
|---|---|---|---|---|
| 1.015 | 1.020 | 1.09 | 1.025 | 1.138 |

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 fault injection is realized and no monitoring is realised. 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. This figure exacerbates two important points. First, all benchmarks present less than 15% of 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 and redundancy combined techniques and DPSR. These techniques have relatively similar simulation overheads. The second group is composed with SECDED, DPSR and DECTED. In this second group, we see a slight overhead increase, especially for computing intensive applications where the number of memory accesses is important 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, DPSR technique is part of the less performance impacting reliability techniques.

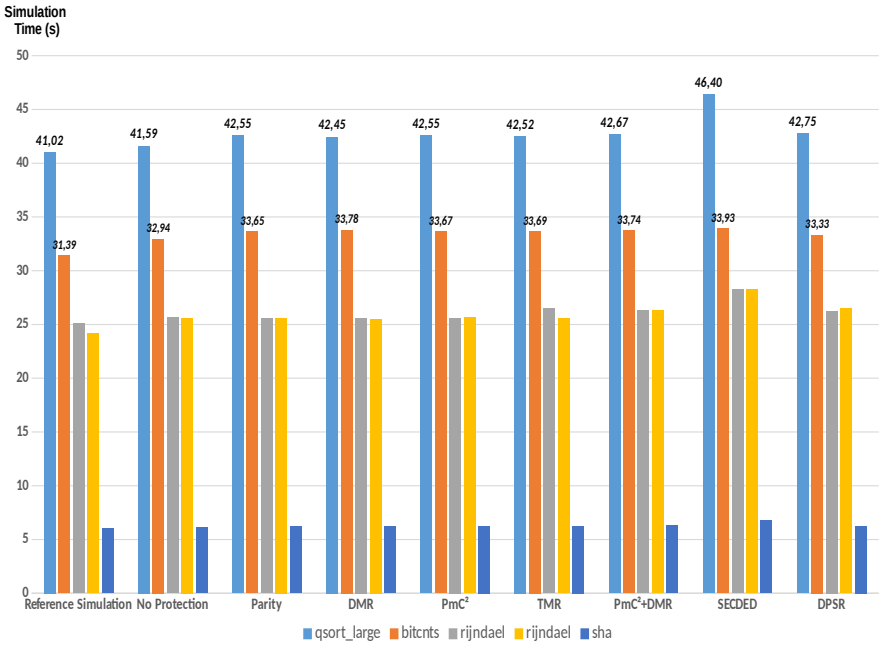


Figure 18: Simulation time with fault injection for different reliability techniques

### *5.3.2. Memory Reliability Techniques Comparison*

In this section we summarize all results obtained so far and give a global view to the reader about the rank of our reliability technique in the existing spectrum of memory reliability techniques. Table 11 is a sum up of all data found during our work for an 8-bits word size. We can clearly see that DPSR comes in a good place and is an intermediate between a soft protection represented by the DECTED technique where the goal is more onto the memory size and the 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 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 ranks itself to be a decent trade-off between protection, memory space and performance.

Table 11: Memory Reliability Techniques Comparison for an 8-bits word size

| | 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%** |

Table 12: RETG detection 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 |

#### *5.3.3. RETG estimation*

RETG allows to compare the different techniques. Table 12 gives all RETG detection values computed for different memory word size and different memory reliability techniques. We see that our technique shows promising results and is comparable to DECTED technique. In our opinion the choice between both techniques has to be made on the need of protection and memory overhead compared to the time overhead, along with the need for flexibility. In a averagely critical application our DPSR technique appears to be a good choice. However, to garantee the highest level of reliability, 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 12 shows also the scaling of techniques compared to memory size word to protect. While DPSR uses the redundancy, it offers the flexibility of using different reliability levels: detection with/without correction.

## 6. Conclusion

In this work, we present a new memory reliability enhancement techniques called DPSR. This technique has the advantage to be easily implementable. It provides an interesting trade-off between correction and detection probability, memory and time overhead and implementation complexity. Moreover, we propose a fault injection tool and methodology to evaluate reliability of the system. This methodology has the advantage to be tuned by the user while maintaining a small performance overhead. The fault injection tool has been used to precisely measure the usage of different memory fault protections. In future work, we plan to extend the work by taking into account power consumption as a different metric to evaluate memory reliability techniques. Moreover, we

would like to extend our memory fault injection tool to all of the processor components.

## References


[1] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, C. W. Wahaus, Ibm experiments in soft fails in computer electronics (1978–1994), IBM Journal of Research and Development 40 (1) (1996) 3–18.

[2] A. Dixit, A. Wood, Impact of new technology on soft error rates, Reliability Physics Symposim (IRPS) (2011) 486–492.

[3] Semiconductor industry association, international technology roadmap for semiconductors, http://www.itrs.net.

[4] S. Rehman, M. Shafique, J. Henkel, Reliable software for unreliable hardware: A cross layer perspective, 2016. doi:10.1007/978-3-319-25772-3.

[5] L. Pintard, From safety analysis to experimental validation by fault injection - case of automotive embedded systems, Ph.D. thesis, University of Toulouse, France (2015).

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

[7] M.-C. Hsueh, T. K. Tsai, R. K. Iyer, Fault injection techniques and tools, Computer 30 (4) (1997) 75–82.

[8] P. Hazucha, C. Svensson, Impact of cmos technology scaling on the atmospheric neutron soft error rate, Nuclear Science, IEEE Transactions on 47 (2001) 2586 – 2594. doi:10.1109/23.903813.

[9] R. Velazco, P. Fouillat, R. Reis, Radiation Effects on Embedded Systems, Springer-Verlag, Berlin, Heidelberg, 2007.

[10] G. E. Moore, Creaming more components onto integrated circuits, Electronics 38 (8).

[11] D. Radaelli, H. Puchner, S. Wong, S. Daniel, Investigation of multi-bit upsets in a 150 nm technology sram device, IEEE Transactions on Nuclear Science 52 (6) (2005) 2433–2437.

[12] S. Borkar, Designing reliable systems from unreliable components: the challenges of transistor variability and degradation, IEEE Micro 25 (6) (2005) 10–16. doi:10.1109/MM.2005.110.

[13] A. S. Hartman, D. E. Thomas, 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.

[14] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Transactions on Computers 58 (10) (2009) 1382–1397.

[15] FIDES-Group, Reliability Methodology for Electronic Systems, 2010.

[16] F. H. Hardie, R. J. Suhocki, Design and use of fault simulation for saturn computer design, IEEE Transactions on Electronic Computers EC-16 (4) (1967) 412–429. doi:10.1109/PGEC.1967.264644.

[17] M. Kooli, A. Bosio, P. Benoit, L. Torres, 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.

[18] M. Kooli, G. D. 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.

[19] S. Anceau, P. Bleuet, J. Clédière, L. Maingault, J. luc Rainard, R. Tucoulou, Nanofocused x-ray beam to reprogram secure circuits, in: Cryptographic Hardware and Embedded Systems – CHES 2017, Vol. 10529 of Lecture Notes in Computer Science, Springer, 2017, pp. 175–188. doi:10.1007/978-3-319-66787-4_9.

[20] H. Abbasitabar, H. R. Zarandi, 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.

[21] P. Benjamin, M. Erraguntla, D. Delen, R. Mayer, Simulation modeling at multiple levels of abstraction, in: 1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274), Vol. 1, 1998, pp. 391–398 vol.1. doi:10.1109/WSC.1998.745013.

[22] D. D. Gajski, R. H. Kuhn, New vlsi tools, Computer 16 (12) (1983) 11–14. doi:10.1109/MC.1983.1654264.
URL http://dx.doi.org/10.1109/MC.1983.1654264

[23] Accellera, Systemc standard download page (2011).
URL http://www.accellera.org/downloads/standards/systemc

[24] J. Aynsley.

[25] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, Ferrari: a flexible software-based fault and error injection system, IEEE Transactions on Computers 44 (2) (1995) 248–260.

[26] B. P. Sanches, T. Basso, R. Moraes, J-swfit: A java software fault injection tool, in: 2011 5th Latin-American Symposium on Dependable Computing, 2011, pp. 106–115.

[27] D. Li, J. S. Vetter, 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.

[28] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, J. Emer, 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.

[29] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, D. Gizopoulos, Differential fault injection on microarchitectural simulators, in: Workload Characterization (IISWC), 2015 IEEE International Symposium on, 2015.

[30] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, S. Mitra, CLEAR: cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores, CoRR abs/1604.03062. arXiv:1604.03062.
URL http://arxiv.org/abs/1604.03062

[31] S. Ozdemir, D. Sinha, G. Memik, J. Adams, H. Zhou, 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.

[32] Nhon Quach, High availability and reliability in the itanium processor, IEEE Micro 20 (5) (2000) 61–69. doi:10.1109/40.877951.

[33] I. Alouani, S. Niar, F. Kurdahi, M. Abid, 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.

[34] M. K. Qureshi, 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.

[35] C. L. Chen, M. Y. Hsiao, Error-correcting codes for semiconductor memory applications: A state-of-the-art review, IBM J. Res. Dev. 28 (2) (1984) 124–134. doi:10.1147/rd.282.0124.
URL http://dx.doi.org/10.1147/rd.282.0124

[36] L. Saiz-Adalid, P. Reviriego, P. Gil, S. Pontarelli, J. A. Maestro, Mcu tolerance in srams through low-redundancy triple adjacent error correction, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 23 (10) (2015) 2332–2336.

[37] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, J. Hoe, 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.

[38] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, J. Hoe, 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

[39] M. Bagatin, S. Gerardin, A. Paccagnella, C. Andreani, G. Gorini, C. Frost, Temperature dependence of neutron-induced soft errors in srams, Microelectronics Reliability 52 (1) (2012) 289 – 293.

[40] Y. Kagiyama, S. Okumura, K. Yanagida, S. Yoshimoto, Y. Nakata, S. Izumi, H. Kawaguchi, M. Yoshimoto, Bit error rate estimation in sram considering temperature fluctuation, in: Thirteenth International Symposium on Quality Electronic Design (ISQED), 2012, pp. 516–519.

[41] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite (2001) 3–14.

[42] T. E. Carlson, W. Heirman, L. 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), 2011, pp. 52:1–52:12.

[43] A. Chabot, I. Alouani, S. Niar, R. Nouacer, 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.

[44] A. Chabot, I. Alouani, S. Niar, R. Nouacer, A new memory reliability technique for multiple bit upsets mitigation, in: Proceedings of the 16th ACM International Conference on Computing Frontiers, CF '19, ACM, New York, NY, USA, 2019, pp. 145–152. doi:10.1145/3310273.3321564. URL http://doi.acm.org/10.1145/3310273.3321564