UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

LUIZ GUSTAVO CASAGRANDE

Soft Error Analysis With and Without Operating System

Work presented in partial fulfillment of the requirements for the degree of Master in Microelectronics

Advisor: Prof. Dr. Fernanda G. de Lima Kastensmidt

Porto Alegre
June 2016

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Casagrande, Luiz Gustavo

Soft Error Analysis With and Without Operating System / Luiz Gustavo Casagrande. – 2016.

103 f.:il.

Orientador: Fernanda Gusmão de Lima Kastensmidt.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2016.

1. Efeitos da Radiação em Processadores. 2. Metodologias e ferramentas de injeção de falhas. 3. Emulação de processadores. I. Kastensmidt, Fernanda Gusmão de Lima. II. Soft Error Analysis With and Without Operating System.

ACKNOWLEDGMENTS

The loneliness of the decision to the responsibility of the gesture. Not even the intention escapes the judgment. What would be nobler then? Deliver your best days and steady arms to the social claim or bear in mind the very free and tragic image? To decide something, whether or not conducted by Opinion, is performing a morbid act. It is an impulse, a spark who disguises herself as motivation and ideal. This is all anger and hopelessness, the insoluble existential doubt urging cease.

In this process, the head is an island, but I am thankful to have open the ports to many people that, even sometimes unknowingly, contributed with words, experiences, actions or provocations. Among them, I mention Caroline Araujo for her constancy, availability and beautiful smile. My mother, Maria, for unquestioned support and my Father, Ivo, for the silent and unconditional love, know that I am very proud of what you represent and that without you nothing would be possible. Thanks to my supervisor, Fernanda Kastensmidt, for the patience. To my current and past colleagues for the technical discussions and philosophical clashes. To my friends (including the previously mentioned), for the comic relief and delight of oblivion. My professors Delfim Luiz Torok, for the confidence and youthful spirit, Ewerton Cappelatti, for the fellowship, Paulo Piber, for the placidity and Ronaldo do Espírito Santo, for the character.

ABSTRACT

The complexity of integrated system on-chips as well as commercial processor's architecture has increased dramatically in recent years. Thus, the effort for assessing the susceptibility to faults due to the incidence of spatial charged particles in these devices has growth at the same rate. This work presents a comparative analysis of soft errors susceptibility in the commercial large-scale embedded microprocessor ARM Cortex-A9 single core, widely used in critical applications, performing a set of 11 applications developed for a bare metal environment and the Linux operating system. The soft errors analysis is performed by fault injection in OVPSim simulation platform along with the OVPSim-FIM fault injector, able to randomly select the time and place to inject the fault. The fault injection campaign reproduces thousands of bit-flips in the microprocessor register file during the execution of the benchmarks set, with a diverse code behavior ranging from control flow dependency to data intensive applications. The analysis method is based on comparing applications executions where faults were injected with a fault-free implementation. The results show the error rate classified by their effect as: masked (UNACE), crash or loss of control flow (HANG) and silent data corruption (SDC); and by register locations. By separating latent errors by its location in the results and exceptions detected by the operating system, one can provide new better observability for a large-scale processor. The proposed method and the results can guide software developers in choosing different code architectures in order to improve the fault tolerance of the embedded system as a whole.

Keywords: Open Virtual Platform (OVP). Soft error. ARM Cortex-A9. Bare metal. Linux operating system. Embedded processor.

RESUMO

A complexidade dos sistemas integrados em chips bem como a arquitetura de processadores comerciais vem crescendo dramaticamente nos últimos anos. Com isto, a dificuldade de avaliarmos a suscetibilidade às falhas em decorrência da incidência de partículas espaciais carregadas nestes dispositivos cresce com a mesma taxa. Este trabalho apresenta uma análise comparativa da susceptibilidade à erros de software em um microprocessador embarcado ARM Cortex-A9 single core de larga escala comercial, amplamente utilizado em aplicações críticas, executando um conjunto de 11 aplicações desenvolvidas para um ambiente *bare metal* e para o sistema operacional Linux. A análise de *soft errors* é executada por injeção de falhas na plataforma de simulação OVPSim juntamente com o injetor OVPSim-FIM, capaz de sortear o momento e local de injeção de uma falha. A campanha de injeção de falhas reproduz milhares de *bit-flips* no banco de registradores do microprocessador durante a execução do conjunto de benchmarks que possuem um comportamento de código diverso, desde dependência de fluxo de controle até aplicações intensivas em dados. O método de análise consiste em comparar execuções da aplicação onde falhas foram injetadas com uma execução livre de falhas. Os resultados apresentam a taxa de falhas que são classificadas em: mascaradas (UNACE), travamento ou perda de controle de fluxo (HANG) e erro nos resultados (SDC). Adicionalmente, os erros são classificados por registradores, separando erros latentes por sua localização nos resultados e por exceções detectadas pelo sistema operacional, provendo novas possibilidades de análise para um processador desta escala. O método proposto e os resultados obtidos podem ajudar a orientar desenvolvedores de software na escolha de diferentes arquiteturas de código, a fim de aprimorar a tolerância à falhas do sistema embarcado como um todo.

Palavras-chave: Plataforma virtual (OVP). *Soft error*; ARM Cortex-A9. *Bare metal*. Sistema operacional Linux. Processador embarcado.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACP	Accelerator Coherence Port
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AVF	Architectural Vulnerability Factor
CAN	Controller Area Network
CISC	Complex Instruction Set Computer
COTS	Commercial of the Shelf
CPU	Central Processor Unit
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Design Under Test
ESA	European Space Agency
FI	Functional Interrupts
FIM	Fault Injector Module
FIT	Failure In Time
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GP	General Purpose
GPP	General Purpose Processor
IC	Integrated Circuit
ICM	Innovative CPU Manager
IDE	Integrated Development Environment
IO	Input/output
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
JEDEC	Joint Electron Device Engineering Council
JIT	Just in Time Compiler
JTAG	Joint Test Action Group
MBU	Multi Bit Upset
MIPS	Million Instructions per Second

MOS	Metal-Oxide Semiconductor
MPE	Media Processing Engine
OCD	On-Chip Debug
OS	Operating System
OVP	Open Virtual Platform
PC	Personal Computer
PTM	Production Test Mode
PVF	Program Vulnerability Factor
RAM	Random Access Memory
RHBP	Radiation Hardening By Design
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SBU	Single Bit Upset
SDC	Silent Data Corruption
SEE	Single Event Effect
SEGR	Single Event Gate Rupture
SEL	Single Event Latch Up
SER	Soft Error Rate
SET	Single Transient Effect
SEU	Single Event Upset
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multiprocessor
SP	Stack Pointer
SPI	Serial Peripheral Interface
SWI	Software Implemented
TAP	Test Access Port
TID	Total Ionizing Dose
TLB	Translation Lookaside Buffer
TLM	Transaction Level Modeling
UART	Universal Asynchronous Receiver/Transmitter
VFP	Vector Floating Point Unit
VHDL	VHSIC Hardware Description Language
VMI	Virtual Machine Interface
WCET	Worst-Case Execution Time

SUMMARY

1 INTRODUCTION

Failures in integrated circuits (ICs) such as complex embedded systems can be responsible for endangering human life. Because of this, the demand for secure systems has grown considerably in recent years, a phenomenon that can be observed by the continuous research activity in the area. A secure system is characterized by several attributes, including reliability, availability and debugging capabilities [1], [2]. Among these systems, microprocessors are present in most digital systems due to its reprogramming flexibility, performance and especially cost, which is still not comparable with other devices such as FPGAs. These reasons led microprocessors to various application areas such as medical, automotive and aerospace, where the safety factor should be considered the most important.

Safe-critical applications seek high reliability. They are usually composed of single or multi-core processors that may run embedded software developed with or without operating systems. Bare metal software is a term used in embedded systems when the program code is directly hosted on the target microprocessor and is self-managed. They allow high controllability of hardware and software flow. Additionally, maintenance, debugging and troubleshooting of code is easier and less time consuming. Operating systems can provide the availability of hardware resources for applications with a high level of complexity. The increasing need of this complex operations and resource sharing demands better using of the available capabilities as well as better programmability.

However, erroneous behaviors in this type of systems have been observed since the early 70s. Phenomena such as soft errors can result in serious failures, and their incidence is worsened due to the significant decrease in the ICs’ transistor size [3]. Soft errors occurs due to transient faults and are mainly observed as bit-flips in memory elements operating under a radiation environment [4], also called single event upsets (SEU). The radiation environment can be space or on Earth, where neutrons interact with the material provoking secondary particles such as alpha to provoke transient faults [5]. Soft errors becomes as a representative issue while projecting a microprocessor, since the effects of SEUs can be noticed by failures in data results or deviations in the application flow. Not only the physical architecture, but the code architecture itself influences directly its performance in front of soft errors, which are the main aspects to be considered in a reliable efficient design.

In this way, it is mandatory a preliminary analysis of the embedded systems microprocessor behavior in critical applications without and with operating system [6][7], and fault injection experiments have shown to be one of the most effective techniques for reliability evaluation of an IC [7][8]. However, a fault injection environment is costly and complex, as it requires compliance of numerous parameters, such as injection model, list of failures, workload, data acquisition mechanism and data analysis methodology, which directly influence coherence and results interpretation [9].

Therefore, in this work we adopt the use of a virtual platform capable of simulating embedded systems running real application code with performance comparable to a traditional system, in the order of hundreds of MIPS. Originally developed to accelerate the development of software, such simulators typically offer a range of CPU models and memory systems for exploitation of resources during development stage [10], allowing the analysis and implementation of different applications or operating systems for several architectures, including multiprocessor systems. Currently UFRGS conducts research using the Open Virtual Platform (OVP) OVPsim, which has shown great applicability for environments development for fault simulation. Compared to the others, which reach rates of around 200 KIPS [11], the OVPsim can achieve simulation speeds up to 100 MIPS. This factor was decisive in choosing this platform to model the embedded system used in this work.

In view of these considerations and along with OVPSim virtual platform, this work aims to compare soft error susceptibility in embedded software that was developed for a bare metal environment and under an operating system. It also analyses the architectural vulnerability factor (AVF) of the register file in the well-known embedded microprocessor ARM Cortex-A9, widely used commercially in low power or thermally constrained applications. The soft error analysis is performed by fault injection with the OVPSim-FIM module that was proposed by [12], capable of simulating SEU type faults on the processor registers bank, producing bit-flips with controlled locality and temporality, allowing investigation of the failures detected and treated by both the platforms. Another contribution of this work, besides the comparative analysis of application environments, is the improvement of the mentioned fault injector, allowing Operating System analysis from processor architectural perspective to software masking level.

A set of 11 applications was used as benchmarks that diversify control and data flow behavior. Those applications were compiled for the target architecture and run directly on the processor for the bare metal tests. The chosen operating system for the comparative tests was Linux, mainly because it has open code, which significantly facilitates the controllability and observability required by the experiments. Beyond that, it is worth mentioning that Linux is already considered as an option of a commercial purpose operating system, which is being included in a wide range of applications, including those with reliability requirements [13].

During the fault injection campaign, it was injected 100,000 faults in the register file to evaluate each benchmarks. Results were classified into masked faults (UNACE), silent data corruption (SDC) and hang. Results show that applications running in Linux environment are usually less prone to failures due to bit-flips in the register file, while bare metal applications have shown to be more susceptible to this faults, specially silent data corruption (SDC) failures.

This work is divided as follows: Chapter 2 evaluates faults in embedded systems, going through fault injection techniques commonly used in these environments, failure classification and presenting the OVPSim platform, discussing its operation and organization. Chapter 3 describes the proposed embedded system architecture, including the ARM Cortex-A9 CPU and its modeled functionalities, the Linux operating system and the fault injector structure, with failure classification and special features. Chapter 4 introduces the test platform and used benchmarks, explaining the fault distribution methodology and system observability. Chapter 5 exposes the results in many forms, such as environment comparisons, detailed classification, Linux error mechanism observation and overall error rate analysis. Finally, Chapter 6 presents the conclusions.

2 FAULT EVALUATION IN EMBEDDED SYSTEMS

The radiation effects in embedded systems have become a concern for safe critical software developing, which includes land level applications like medical, military and aerospace applications. Soft errors arise because of these radiation interactions, causing undesirable changes in memory elements that may lead to catastrophic failures. In most cases radiation tolerant COTS processors are not financially viable, so even devices used in such applications are the same we can find in regular cellphones or computers. It is therefore essential that embedded systems can be evaluated with respect to their susceptibility to faults in order to determine its limitations and, based on this, develop outline methods such as tolerant software techniques.

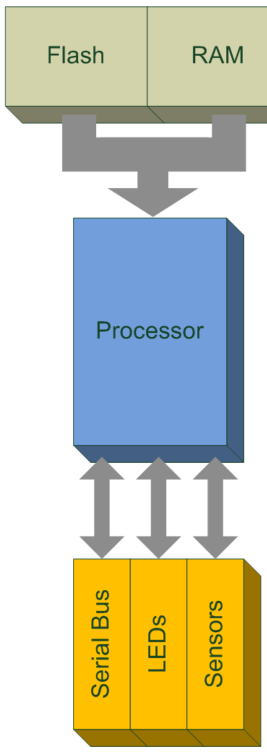
2.1 Defining an Embedded System

An embedded system comprises a broad field of study and therefore there are many definitions of what it is. Depending on what's the study aims to emphasize the definition can vary, but in Noergaard [14] we found a short and objective designation that is aligned with the work proposes, defining an embedded system as an applied computer system.

An embedded system can be seen as a resource manager, which provides an orderly allocation of processors, memory, input devices and output data. It is found in a wide range of devices and combines software and hardware. However, different than an operating system, it is a physically limited computer system, usually with size, power and memory constraints, which has limited and specific, functions [16]. A notable example of embedded systems was the Apollo Guidance Computer, a redundant hardware used in the spacecraft control, with error recovery routines, meaning that that it one of the first embedded systems was fault tolerant [17].

An embedded system is depicted in Figure 2.1 with its main divisions: CPU, memory and peripherals. Additionally, a communication layer and the embedded code can be listed, since the system must interact with external elements and its tasks shall be defined [18].

Figure 2.1 – Embedded system with CPU, memory and peripherals



Source: Author

An embedded system has one or more CPUs, which are responsible for fetching and executing instructions from memory. It can be considered the main system component, and has a well defined execution cycle, that includes searching instructions from memory, decode them to determine the operation and operands, execution, external memory accesses and writing back to updated values. Each CPU has a specific set of instructions and all application code shall be translated to this specific set before loaded into the embedded system. Furthermore, as the access time to memory to fetch instructions or operands is bigger than the time to perform them, the CPUs have internal registers to store key variables and temporary results [16]. Embedded systems CPUs, as commented, are intended to specific tasks, thus it is common to found several devices models, which should be considered depending on the application, such an Application Specific Instruction-set Processor (ASIP), a RISC or CISC architecture, a Digital Signal Processing (DSP) [19] or intended for low power applications [17].

Another major component in an embedded system is the memory. Ideally, the memory should be non-volatile, inexpensive and faster than the processor, so that the CPU was not delayed, but today technology does not meets these requirements. Therefore, the system memory is organized in layers, where the faster, smaller, and proportionally more expensive memory, are closer of the CPU in terms of access, while memories with greater storage capacity are more distant. The memories order, from the slower to the faster, and technologies typically are internal registers, cache memory, external RAM and Flash memories or hard drives [14] [16]. From the application execution point of view, memory can be organized as instruction and data memories, which represents only a logical separation to determine how the CPU will organize the instructions under execution (internal registers, cache and RAM), the application code (Flash or hard drive) and the operations results (all layers).

Peripherals or I/O devices may consist of two parts: a controller and the device itself. The controller tend to have standard interfaces with the CPU, which helps to share communication resources between them, but as each peripheral has its own purpose, typically a driver should be used. The devices are commonly used as auxiliary communication, I/O driving, sensing, status or additional storage, adding more functionality to the embedded system [17].

Finally, the non-physical part of an embedded system is the embedded application, which uses all the described parts of the system. The code developed by the programmer must be compiled to the intended architecture, being converted into machine code, which will reside in the physical platform. The CPU has a initial state and starts to fetch instructions from a known place that depends on the processor model. Thereafter, the sequence of instructions performed by the CPU coordinate the operation of the embedded system, including communication with peripherals[17].

2.2 Fault Injection in Embedded Processors

To understand the implications and assumptions around fault injection in embedded processors, definitions regarding terminology must be defined, as well as the nature of a fault, like what provoke them and how faults react in a system and become an error. As well, concepts concerning how it interacts with operating systems and measurement metrics are discussed in this section.

2.2.1 Faults, Errors and Failures

The literature define the terms that surround this work in different ways, trying to explain a common phenomena. To avoid misunderstandings, we use as reference the article developed by [71].

As the author suggests, a failure is defined as an unexpected behavior or unwanted operation of the target system that is perceived by the user. The error concept is understand as an state of the system that may lead to a failure, i.e., the error create a condition in which the subsequent computing operations may fail. Finally, the error state is achieved by the effect of a fault, that is defined as the error physical reason.

2.2.2 Faults in Embedded Systems

A system is said to be fault-tolerant when it has the ability to perform properly their duties regardless of the occurrence of faults [21]. Even a system that has apparently been developed properly is subject to failures due to wear of the devices or design errors that result in a logic failure. Two types of failures may occur in a system: the first is the natural failures caused by wear, internal or external sources or a component; second, the failures resulting from an accidental or intentional human action [22]. A situation in which these two types of failure can occur is, for example, an operator setting the parameters of an installation to perform an overloaded task. This systematic operation will cause the installation, inevitably, to malfunction due to wear of its parts. In this particular example, an error caused by human action generated a natural lack. The occurrence of these faults does not imply that other type of failure may not happen.

However, fault-tolerant systems should be able to drive the system to a safe state when the error is detected, or allowing the system to operate in degraded form, but meeting the minimum requirements of the same. Given this context, there are some techniques to give proper treatment to this events: (a) redundant structures to mask components that suffered an error; (b) error control codes and double or triple voting mechanism to discover or correct erroneous information; (c) diagnostic techniques to locate defective components; (d) and techniques for automatic conversions to replace a subsystem that has crashed [22]. In 1984, Johnson defined the three main techniques to try to improve or maintain the normal operation of a system: fault avoidance, fault masking and fault tolerance [21].

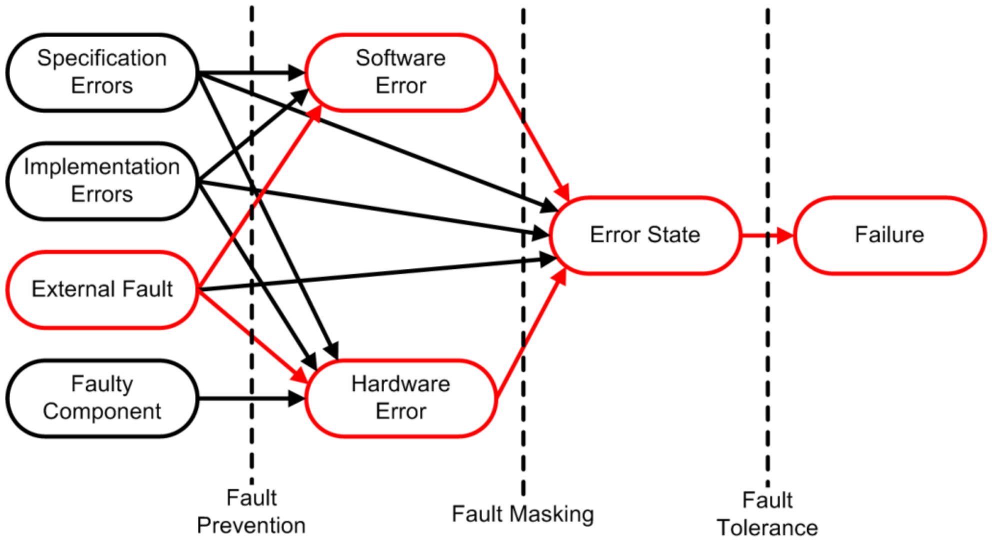
Fault avoidance aims to block the fault occurrence or even its effect that is the error. It can be achieved through good project management practices and system testing for design mistakes not observed in the detailing phase. The idea is to prevent the system to be released to the market with a “bug”, which in safe critical applications can be catastrophic. In this regard, extensive testing of the application is used. However, the time spent in the testing phase will increase the "time to market" (amount of time from conception to launch the product on the market).

The fault masking is applied in real-time system and aims to prevent the error from spreading to other system components or leading to an error state causing a failure. One way to implement this technique is through the implementation of voters.

The fault tolerance, unlike the others, is not intended to eliminate the presence of the fault or its propagation, but drive the system to a state that is acceptable in terms of execution and security. There are four principles on which is based the use of the fault tolerance: detection, location, containment and recovery.

Figure 2.2 shows a flowchart where the techniques described by Johnson can be used to increase the guarantee proper functioning of the system [21].

Figure 2.2 – Johnson’s Fault Tolerant System Flow



Source: Adapted from [21]

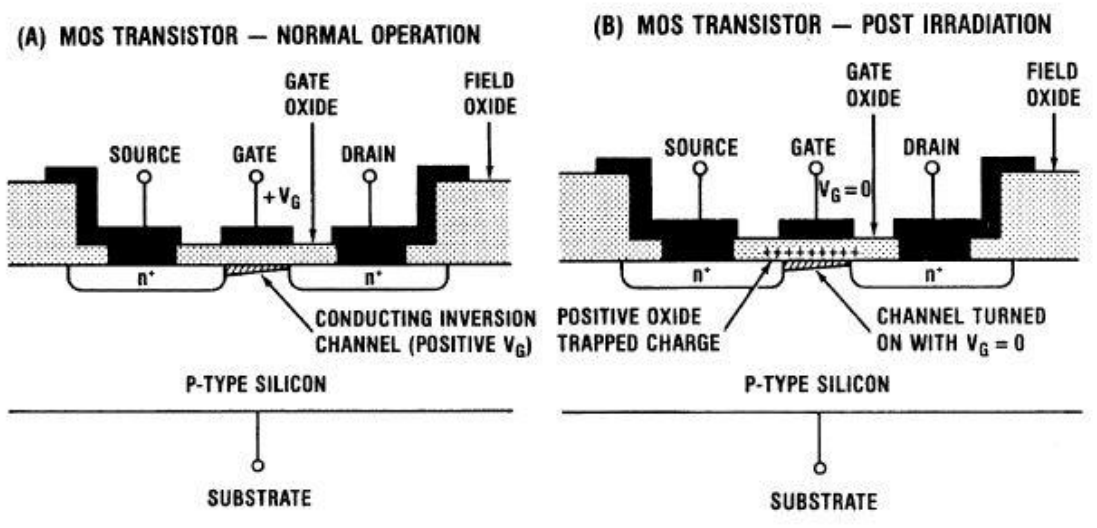
In this work, our attention is around the path marked in red, except that we are not proposing any mitigation technique, but in fact, exposing an embedded system weakness to soft errors by injecting external disturbances, analyzing if it manifests in hardware or software and classifying the arising failures.

2.2.3 Radiation Effects in Embedded Processors

One of the main concerns around failures in embedded processors relies on external sources, especially the radiation effects that may cause soft errors. They can be divided in effects due to Total Ionizing Dose (TID) and SEEs (Single Event Effects). As discussed by [36], this phenomena occurs due to the impact of atmospheric particles, like neutrons, that collides with silicon atoms of the device, causing nuclear reactions that generates an internal ionization. These phenomena and its impact in embedded systems is the central theme of this work, but it is necessary to understand the type of fault in which the device is exposed.

The TID effect is cumulative, resulting in degradation of characteristics of the electrical parameters of the electronic device over time due to the deposit of charges. They are long-term effects that depends on the intensity of radiation and the time that the circuit is exposed to this radiation. In a transistor gate, for example, the TID may cause an increase of the skew and leakage current due to changes in the voltage thresholds. A big exposition to this particles can result in error of the electronic device [20], as shown in Figure 2.3 below.

Figure 2.3 – Induced Charge by Radiation in the Gate Oxide of a N channel MOSFET: (a) normal operation, (b) after irradiation


(A) MOS TRANSISTOR — NORMAL OPERATION
GATE OXIDE
FIELD OXIDE
SOURCE
GATE
DRAIN
$+V_G$
n^+
n^+
CONDUCTING INVERSION CHANNEL (POSITIVE V_G)
P-TYPE SILICON
SUBSTRATE
(B) MOS TRANSISTOR — POST IRRADIATION
GATE OXIDE
FIELD OXIDE
SOURCE
GATE
DRAIN
$V_G = 0$
n^+
n^+
POSITIVE OXIDE TRAPPED CHARGE
CHANNEL TURNED ON WITH $V_G = 0$
P-TYPE SILICON
SUBSTRATE

Source: [24]

The total ionizing dose is measured in rad (radiation absorbed dose). The rad represents 100 ergs of energy deposited per gram of material, which can be correlated with

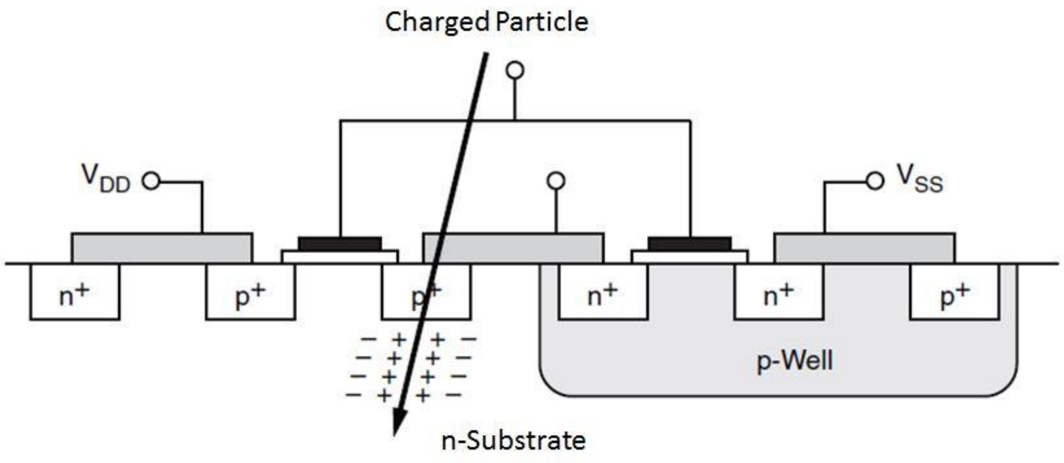
Joule, such that 1 = erg 0,1μJ [25]. Since the energy depends on the material, the units need to be expressed in this terms, such as the silicon: 100 rad (Si) [20].

The TID is specified in units of rads in silicon (rad(Si)), where one rad is defined as the absorbed radiation dose. A rad is a measure of the amount of energy deposited into the material and is equal to 100 ergs (6.24E4 10 eV or NJ) of energy deposited per gram of material [25]. The energy deposited in a device must be specified for the material of interest. Thus, for a transistor of a metal oxide semiconductor (MOS), the total dose is measured in units of rad (Si) or rad (SiO2). The ability to resist to TID in reinforced components against radiation is also defined in rads or krads [28].

The effect can be reduced by shielding or by changing the default device fabrication process. When using shielding the idea is to include a layer of a material, like aluminum, to absorb a range of charged particles, specially low energy ones. Depending on the environment profile, this technique may not be effective, since high energy particles can still cross the shield layer. Another solution, as mentioned, is to modify the fabrication process by including, for example, an epitaxial layer in the high-doping regions of the transistors or an oxide layer in the substrate [20].

The category that is analysed in this work and that may result in undesirable effects in the system functionality are the Single Event Effects (SEE), described by [20] also as electrical disturbances due to colliding particles, but now the high energetic particle does not need to accumulate to generate an observable effect, it has such energy that it can change the state of a given device. Figure 2.4 depicts the effect of a single charged particle passing through the structure of a PMOS transistor.

Figure 2.4 – Particle Crossing the Substrate of a Transistor



Source: [20]

As mentioned, the particle must have enough energy to transfer the charge necessary in the transistor gate to make state changes in its outputs. This charge is represented in the following equation by Qcrit.

$$Qcrit = Cnode \times Vnode \qquad (2.1)$$

According to [20], in equation (2.1), Cnode denotes the capacitance between the nodes of transistors and Vnode is the transistor operating voltage. Today, reducing the size of transistors and the significant increase in transistor density on a single chip increases sensitive to SEE, in part since the chances of collision are bigger, in the other side the capacitance and stored charges of the devices decreases, which results in a lower Qcrit [27] and for consequence the range of potential charged particles necessary for generating an upset increases.

The SEE can occur in different ways such as a transient effect (SET), upset (SEU), latch up (SEL), door break (SEGR), among others. A typical case is in a circuit with memory elements, for example, where a single event may provoke that an incorrect value is stored and may produce an error that will persist until the value is overwritten. Also, the SEE may flip a signal in a combinational logic, generating incorrect values in arithmetic operations or even producing a wrong output, such an interruption [20]. In this work, we will analyze the Single Event Upset effects, since it is one of the most common persistent and latent errors.

2.2.3.1 Single Event Upset

The term Single Event Upset and the correlation with charged particles appeared in [26] for the first time as it is known today. The acronym has been used to describe disorders in digital circuits of memory devices, caused by radiation or cosmic rays and they are the focus of this work

They differ from the SETs, since they are not transient, associated with the inversion bit memory elements. They are also referred to in the literature as soft errors and can have an indefinite duration or may be fixed after one or more clock cycles. The SEU is called single bit upset (SBU) if only one memory element inverts its state and multi-bit upset (MBU) if more than one element changes [20].

A characteristic of SEUs is that they are random events and, therefore, they may occur at unpredictable times. For example, they can damage the contents of a processor register during the execution of an instruction, which in case of a pipelined processor may result in an instruction being correctly executed in the first pipeline stages and then corrupted in the middle of operation, avoiding any pre decoding correction mechanism and causing unpredictable effects in the data flow.

2.2.4 Failure Classification, Metrics and Operating System Impact

As mentioned, radiation effects can lead to soft errors in electronic devices according to ionization characteristics. In the firsts occurrences it is not permanent since only repeated events in the same point can damage the silicon structure, and so the error is called “soft”.

Not all soft error occurrences generate a manifested failure. They are the originating factor that results in the appearance of an error. When a soft error occurs, the system can or cannot be driven to an inconsistent/incoherent state, which characterizes the existence of an failure. In a given system, [30] defines the probability that the soft error has to generate a failure, as shown in equation (2.2).

$$\sigma = \sum_i \sigma_{raw} \times V_i \tag{2.2}$$

The author introduces new terminologies to define the probability. The cross-section is represented by σ [cm^2]. It defines the radiation susceptible area in which an error happens if a particle collides. Depending on the technology, the radiation sensitivity varies and it is expressed by the constant σ_{raw}. Finally, the probability to generate an error is represented by V, which means de vulnerability factor.

The vulnerability factor is often estimated using radiation test campaigns with accelerated particles as a reason of the number of errors observed and the particles flux, which is the amount of particles per unit area. In some cases, it is possible to estimate the vulnerability using simulation if a detailed description of the design is available to the tester.

During the test campaigns, either radiation tests or simulation, the most usual metric is the soft error rate (SER), which indicates the number of errors manifested in the functionality of the system during a defined time, commonly represented by the terminology Failure in Time (FIT) and expressed by [31] in the following equation.

$$Soft\ Error\ Rate = \frac{\eta}{N} \times \varphi \times 10^9 \qquad (2.3)$$

The equation shows that under a fluency of particles, expressed by N, the number of errors observed, indicated by η, and the total particle flux, the φ in the equation, is possible to determine the soft error rate in terms of FIT. Since the Failure in Time express one failure per 10^9 hours, the scale is also included in the equation.

The SER is the main parameter to evaluate in a device as it indicates how tolerant it is to radiation. A well-controlled environment is required to determine this metric, since the flux shall be constant and the error properly detected. That is why simulation techniques has had so much visibility, allowing practical and precise measures as long the device model is correctly represented.

To guaranty a good evaluation, in an embedded system context the failures can be divided in two types: functional interrupts (FI) and silent data corruption (SDC). The functional interrupts means that the system or application running in the processor hanged or crashed and may lead the system to an unresponsive state, eventually requiring the reboot or a power cycle to recover. Silent data corruption means that application has finished but its result differs from the expected one. Note that not only the produced results of an algorithm, for example, is important but the entire memory space, since latent errors may occur due to data corrupted that not necessarily was used after soft error event, remaining undetected for a long period.

According to [32] the vulnerability factor can take into account the SDC and FI classification, as they represent different criticality depending on the system. In this manner, it is possible to classify the vulnerability into FI factor and SDC factor that indicates the probability of a soft error in a defined element to generate a functional interrupt or a silent data corruption, respectively.

This classification became even more important when an Operating System (OS) is involved. The work presented in [33] has shown that errors in the presence of an OS tend to cause FIs, and SDC rate is not significantly influenced as its occurrences has the application as the major contributor.

2.2.5 Fault Tolerance Evaluation Techniques

Fault injection is the most common and widely accepted approach for analysis of safe critical systems in the presence of faults [34]. The fault injection techniques try to provide information about the reliability of the design under test (DUT). This information can be about: a) compliance validation with reliability requirements; b) detection of weaknesses in the adopted fault tolerance techniques and; c) DUT behavior forecast in the presence of faults.

Fault injection is related to all techniques that simulate or emulate the device behavior in the presence of a fault. Among these techniques are those applied mainly to processor systems in which software changes are made to simulate the occurrence of a fault in the hardware device. Thus, it is important to have a distinction between fault injection in hardware and software, the latter is outside the scope of this work.

The fault injection techniques are not limited on the injection approaches. The techniques involve the complete process that comprises the entire environment required for startup of the DUT, selecting the appropriate workload, the capture of relevant data, comparison with the fault-free DUT data, classification of the effects and monitoring of the whole process. As will be discussed later, the approach of this works involves these steps, where the analysis is made according to a comparison between a fault-free run of the system and a faulty run, which is a run under the effects of the fault injections.

The fault injection method depends on the type of DUT being tested. In the case of memories, for example, the effect being examined by injecting faults will be predominantly SEUs. Another factor to consider is the possibility to cause real faults on the device, i.e., physical faults in the real device or if only fault models (logical) will be applied. The abstraction level of the fault type is directly related to the DUT that can be a COTS, a prototype or a design model. Observability is another important parameter, since it interferes directly on the quality of the extracted results. Finally, the results will be direct influenced by the chosen injection technique, since more realistic techniques like tests under radiation has less observability than simulation methods that, on the other hand, are less precise in terms of device behavior.

The fault injection campaign according to [35] can be divided in four basic elements that are applicable to all techniques:

- Fault Model: Injected faults are used to model the SEEs at different abstraction levels. At least the level of the system behavioral model shall be used. When a COTS is being evaluated, different fault models can be applied, depending on the test needs and device observability.
- Workload: Used to emulate the typical operation of the system under test.

- Observation Elements: Those are applied in the system to classify the effects of the injected faults. Once a failure is identified, it may be classified as any of the many types of failures definable. It is also common to classify the fault itself.
- Measurements: States the degree of dependability of the evaluated system and can be connected with the metric employed (commonly the SER).

This section provides an overview of all existing fault injection techniques. Techniques are presented in two groups, namely those working with physical fault injection and injection techniques that address at the logical level.

2.2.5.1 Physical Fault Injection

Physical fault injection methods use external sources to allow tests with radiation, electromagnetic noise and aging of ICs. The purpose of these tests was to analyze the strength of a COTS or qualify a prototype as hardened against the effects caused by radiation.

The cosmic radiation is the main source of SEE in ICs. Therefore, performing tests with the device at high altitude or even in space is the most realistic way to assess the sensitivity of the ICs to SEE. However, considering the low probability of error, it would require hundreds of thousands of samples of the DUT to obtain a valid measurement, making the time and cost of this approach unfeasible. The particle accelerators are used to classify these products, performing tests that last for hours or days [34]. In these tests, various types of particles are used with different energy values in order to cause different effects. The types of particles and typical energy values used to generate a SEE are detailed in [36]. These parameters are standardized according to the final application of the device. Spatial intended devices are exposed to a different radiation environment than medical equipment, for example, and are defined by space agencies or committees as the JEDEC who created the standard JESD89 [37], which defines the requirements and procedures for testing of SERs for ICs.

Fault injection using laser rays is similar to the method that uses heavy ions in the sense of the beam is applied directly on the silicon surface. However, the laser beam is much more accurate, and thus it is possible to inject faults in specific places in a more controlled way. The incidence of the laser beam in silicon can cause effects similar to those caused by particles of cosmic rays and this method is usually associated with the state change of circuit elements and is used to test the effect of SEU events. With the help of a special microscope, the use of laser to fault injection provides high accessibility as you can select the region to execute the campaign in the circuit. Thus, the fault injection is cheaper than systems who employ radiation tests. Furthermore, the laser needs a simpler fault injection environment since it is not necessary to isolate the DUT or the components in the periphery of the DUT that are not being tested and they didn't suffer disturbances caused by radiation, which would compromise the results of the analysis.

Unlike other physical techniques of fault injection presented, the method by induction values at the DUT pins requires physical contact between the test platform and the DUT. The method seeks to replicate the effect of a natural fault by forcing changes of the logical value of an IC pins. Considering the complexity of modern systems, fault injection in the pins is very limited due to accessibility limitation of the method. This technique is often employed for testing tools in combination with other techniques in order to extend the results. Solutions as Messaline, MARS, FIST, RIFLE described in [38] employ features of fault injection method in circuit pins.

2.2.5.2 Logical Fault Injection

The results of physical injection methods provide realistic values of SER. It is widely used for hardening qualification of critical applications in harsh environments, like space. However, installations with neutron accelerators, for example, charge in the range of hundred to a thousand dollars per hour of exposure [39], i.e., physical fault injection is extremely expensive and thus it is necessary analysis solutions that can be applied earlier in the circuit design. The fault injection methods in logical level exploit resources available in order to insert the effect that a fault in hardware may provoke.

Between logic fault injection, software implemented (SWI) method is directly linked to the execution of specific pieces of software that modify internal elements (accessible by the user) causing the effect of a fault occurred in the hardware. The SWI method is related mainly to embedded processors systems where the normal application software is stopped for the execution of a fault injection code that changes an element such a register, a memory data, or even an application instruction. SWI also appears in the DUT behavior analysis to evaluate communication problems or interaction with other systems, for example, repeated/missing messages or inaccurate information, failure to read from memory, among others. This type of fault injection can be applied in different moments, which can be at compile time or at runtime. At compile time, changes in the software image are made and, when executed, it enables the simulation of an error in hardware. This type of fault does not need additional code to be executed and is generally used to emulate permanent errors. In the case of runtime fault injection, trigger mechanisms are used to notify the time of the fault injection. Timers or specific instructions in the code are used for stopping the implementation and starting the fault injection task. In [40] two processors PowerPC 750 running LynxOS were used along with Xception tool to emulate the effect of SEUs in the system. The Xception tool exploits the advanced features for monitoring performance and errors present in the processors and uses the exception mechanisms of the processor itself to identify the failures. Another tools in [38] like FERRARI, use timers to trigger a fault injection routine. The FIAT tool may delay tasks, corrupt messages or complete them abruptly. Finally, FTAPE tool can add fault injection drivers in the operating system.

The injection of simulation-based faults uses a model of the system being analyzed. Simulation models can use hardware description languages like VHDL and Verilog or a higher abstraction level like SystemC. In the simulation method, faults can be injected by the simulation tool or by changing the description of the hardware model. In the latter case, dedicated modules can be added in the model with the unique purpose of injecting faults or the use of components with the presence of a known error. In the higher abstraction level, is possible to adapt the simulation tools in order to use commands to control internal signals of the system model. This method is dependent on the simulator functionalities and available commands. However, there is no need to change the hardware model. In [41], a model of LEON3 Gaisler processor used by the European Space Agency (ESA) described in SystemC is used for fault simulation. This model described in SystemC is implemented at transaction level (TLM) and employ dedicated modules that are used to simulate the effect of faults in the memory sections (buffers, instructions and data memory, etc.) of the processor. The MEFISTO tool presented in [42] uses saboteurs and simulation commands to execute a more detailed analysis of the error behavior in the VHDL model. Tools like VERIFY propose an extension of VHDL description to add features for fault injection in [38].

Current processors have specific resources to support testing and debugging. These features, known as on-chip debug (OCD), also enable the fault injection and observation of the effects externally to processor. The FIMBUL tool developed in [43] uses the test access port (TAP) of the Thor processor for fault injection. The TAP allows access to internal scan chains and the periphery of Thor processor, thus transient faults are injected throughout where the chain have access. As a comparison, the MEFISTO tool that uses the simulation technique has a slightly higher error coverage than FIMBUL tool that uses the TAP technique, but in the other hand, it proved to be a hundred times faster. The work presented in [44] proposes changes in the debug infrastructure (OCD) in order to enable fault injection to support the verification of fault tolerant mechanisms. The OCD oriented to fault injection is based on the NEXUS standard and it is an additional hardware that automatically triggers in the occurrence of predefined conditions. The instruction address that will trigger the fault injection is generated at random from one of the values present in the address space of instruction memory. The same applies to the address of the data memory that will have its value changed when the mechanism has been triggered. After the faults have been injected, the results are recovered after the completion of all experiments.

The last logical fault injection method is the emulation FPGAs based hardware. It became popular in ASIC verification, and more recently has been exploited for rapid injection campaigns. The term emulation in this context is related to prototyping the circuit to be analyzed in FPGAs. Many techniques that employ these devices use it only as a mean to perform fault injection and quickly identify if the method can be implemented in other technologies. The controllability in most cases is achieved accessing the configuration memory, causing virtual bit-flips. Another approach consists in adding hardware blocks to support the fault injection. An early work using FPGA for fault emulation is shown in [45], which implements permanent errors (stuck-at) connecting signals at constant values. For each injected fault, the circuit was synthesized again and thus the bit-stream had to be reloaded in the FPGA to a new emulation. Among the newer methods that use FPGAs for fault emulation two types of approaches can be perceived. The first method is called instrumentation; it inserts logic in some way in the circuit in order to increase the control and observability of the DUT during its execution [46]. The second uses existing partial reconfiguration capabilities available in newer devices to identify and recover part of the internal circuit [47].

This work uses the simulation-based method and employs OVPSim tool, as shown in the following section.

2.3 Open Virtual Platform (OVP)

A virtual platform is a complete simulation of a computer system. All hardware is abstracted into a model, in which a regular software is able to run, i.e., the platform should be modeled in a way that compiled application is no different from the actual physical hardware.

The virtual platform can be seen as an Instruction Set Simulator (ISS), which simulates not only the CPU, but also the complete system, including memory, cache and buses. The modeling of virtual platforms is defined as modeling in electric system level [48]. This environment can be placed between a RTL level and the functional level description, and is mostly used when the cycle accurate RTL is considered to be too detailed, but still requires a truthful simulation of at least instruction level.

Thus, a virtual platform is designed to: a) enable the development of software more effectively and with better prediction; b) minimize dependence on hardware and; c) increase the test efficiency.

Open Virtual Platform (OVP) is the tool that is used for the implementation of the virtual platform in this work. The Imperas™ Software Limited announced the formation of the alliance OVP to service the needs of the market. This alliance has created models of CPUs, scan tools, debugging and analysis, in addition to simulation platforms. According to the company, the platform allows the development of virtual prototypes with instruction accuracy of up to 100 MIPS in conventional computers [49].

The OVP has three main components:

- APIs that allow C models Additional modules are developed;
- A collection of models of processors and peripherals on open source;
- The OVPsim simulator, responsible for the implementation of these models.

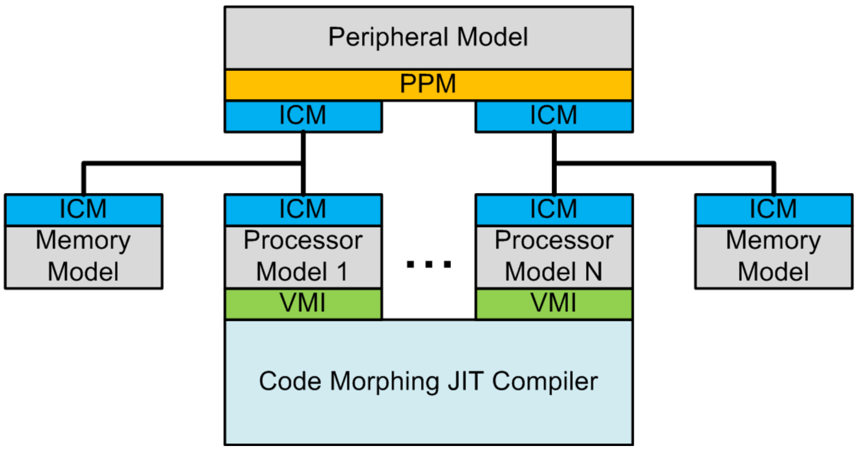
2.3.1APIs

Modeling an environment requires development of the main parts of a real embedded system, such processors, memory and peripherals, but also the platforms and the integration environment. The platform is responsible to promote the integration of the behavioral components. Processors execute the compiled code and the peripherals enable a way to represent the environment in which the application will interact. Thus, the OVP is constructed of four interfaces:

- CPU Manager or Innovative CPU Manager (ICM);
- Virtual Machine Interface;
- Interface with behavioral model of the hardware;
- Interface for programming the peripheral model.

The interaction between these interfaces is shown in Figure 2.5.

Figure 2.5 – OVP Interfaces



Source: Adapted from [50]

The ICM is an API codded in C used to create the platform interconnection environment and netlist with the modeled system to use along with the OVPsim simulator. Through, it is possible to instantiate multiple processors, buses, memories and peripherals. The ICM interface is also responsible for linking these modules and it allows applications to be loaded and executed on instantiated memories [50]. Figure 2.5 locates the ICM in the platform environment as described above in which it interfaces between various modules in the platform.

The Virtual Machine Interface (VMI) is interfaced with the C based processor model, allowing communication with the simulation kernel and other system components. The VMI is the centerpiece for the high performance provided by the OVP, since the processors use a morphing code approach that is coupled to a just-in-time (JIT) compiler that maps the model instructions to native local machine instructions. Between the processor model and compiler there is a set of optimized commands where the processor operations are mapped, being responsible for the OVPsim to provide fast and efficient interpretation of the native machine capabilities. Some of the VMI abilities are [51]:

- Provide virtualization features, such as file I / O, allowing direct execution on the host using the standard libraries that are provided with the environment;
- Instruction set templates encapsulation with OVPsim, if they export some basic characteristics such as availability of Instruction Set model as a shared object;
- The VMI can be used for both RISC processors and CISC processors, supporting any instruction format;
- Finally, it allows the modeling of L2 caches and other extensions around the processor.

2.3.2 Hardware Behavioral Models

The Open Virtual Platform has many processor models like the Altera NIOS, ARC, ARM Cortex A, ARM Cortex M and ARM Cortex R, Xilinx MicroBlaze, several MIPS models, Power PC, Renesas and NEC, most of them provided by the processor manufacturer. In addition, a large number of available devices allow the assembly of a complete platform, including various types of memories, bridges, DMAs and UART modules.

As mentioned, the processor models are instruction level accurate. Behavioral models based on instruction set simulators aim to implement the instructions considering the real number of cycles to execute and perform the IO operations along with each step of each instruction. However, OVP processor models execute each instruction in a functional approach, disregarding timing. The precision level of OVP allows internal registers to store properly the values at the end of each instruction, generating all expected effect. The platform executes an instruction at a time, but abstracts everything regarding the pipeline, for example. The main disadvantages of the platform rely on the high abstraction level of the architecture, that doesn't permit an clock cycle analysis of the tests, as well deeply architecture examination.

The OVPsim also supports description of multicore platforms. It can simulate multiple processors including configurations with shared memory, and heterogeneous multiprocessor systems. The model performance depends on variants of the chosen processor and the nature of the application, but it is possible to achieve speeds of hundreds of millions of instructions per second. As OVP models can be compiled as common objects, they can also be encapsulated in any simulation environment capable of using shared objects. This includes C, C ++ and SystemC environments.

3 CASE-STUDY, PROCESSOR-BASED SYSTEM AND FAULT INJECTOR ARCHITECTURE

An embedded processor can be defined as a processor that is responsible for computation and control of devices that are not regular computers. Embedded processors are used in several product fields, including automotive, networking, mobile, medical and aerospace, and according to [52], they has the following design criterias: (a) performance, represented by the maximum of data that they can process per unit of time and can be achieved by parallel architectures, multi-level caches, function specific internal modules, clock frequency, etc.; (b) power, that depends on the technology process, die area, number of transistors, clock frequency, gate sizing and core voltage; and (c) cost, in which the manufacturability is the key point, being related with the circuit complexity and area. In this way, modern processors needs to balance the use of special architectures, such as multipliers, additional caches, memory controllers and branch predictors, depending mainly on the market share they want to reach.

In this chapter, the ARM Cortex-A9 series processors is evaluated, a mobile processor constrained for the embedded market. This processor has its own OVPSim model provided by ARM itself, making this analysis more reliable in terms of functional equivalence to the physical device. This processor was chosen because it is present in many commercial platforms such as Zynq™-7000 All Programmable SoC from Xilinx [67] and Cyclone V from Altera [66]. These FPGA based platforms allows failure analysis methodologies on the real target, and can be used further on to correlate and validate our results. The Linux operating system is discussed in terms of how its architecture can contribute to processor dependability. Finally, the fault injector organization, embedded system integration characteristics and the main injection phases are explained.

3.1 ARM Cortex-A9 Architecture

The ARM Cortex-A9 has a specific set of features and functionality beyond the traditional ARMv7 architecture in which it was based, in order to provide a high performance and low power consumption. The main features of the architecture are described in this section.

3.1.1Main Functionalities and Relevant Features

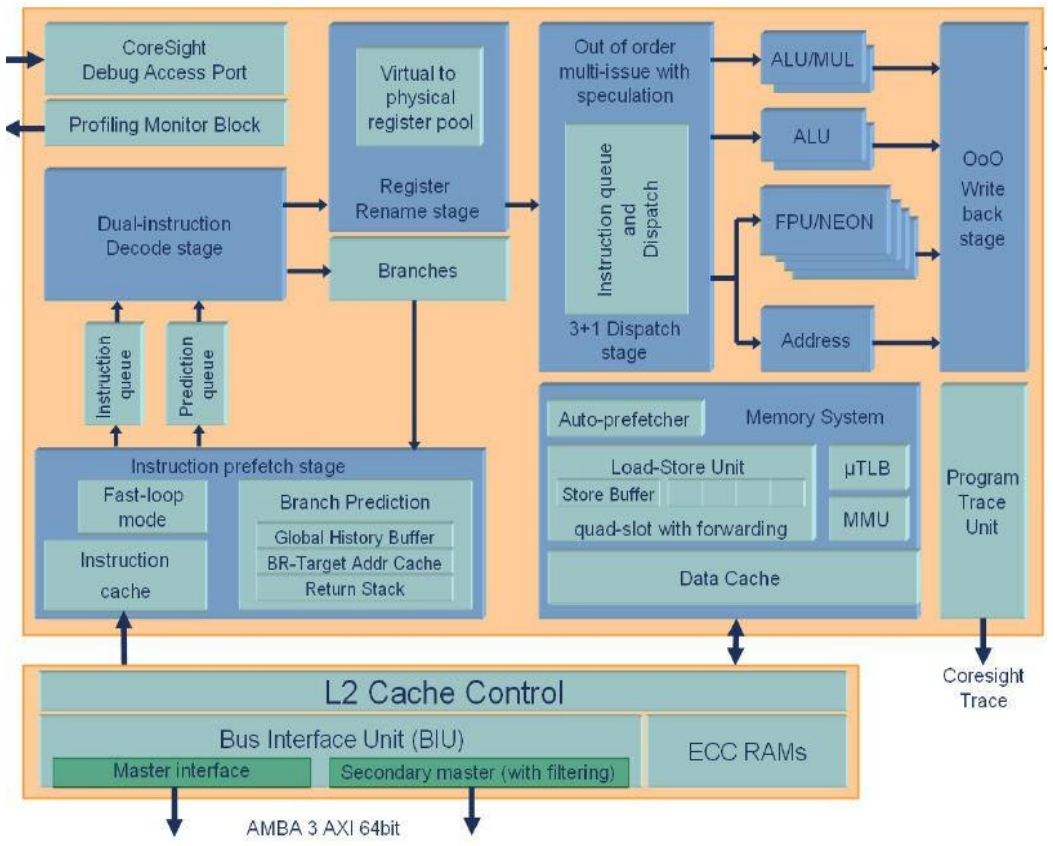
The processor ARM is based on a RISC architecture, being the most popular embedded processor for mobile applications. It happens due to its architecture that can provide high performance and operates within the mobile power acceptable range, which is under 5W of constant power consumption. With its high performance ARM macro equipped with an L1 cache subsystem, it enables the use of a full virtual memory system, reaching around of 98% of mobile devices and 75% of all 32-bit embedded processors [53] [52].

All processors of the Cortex-A family are designed for mobile purposes, but is also intended for televisions and TV receivers. The Cortex-A9 uses an advanced version of the instruction set of the Cortex-A8, based on the ARMv7 ISA, with extended FPU and ability to perform data level parallelism, since it has multiple internal processing elements. The parallelism allows the core to perform several optimizations. The ones that are considered relevant for this work is:

- Out of order and speculative execution, that may result in a more allocated register file;
- Dynamical renaming of physical registers, promoting autonomy from the code and leading to a non obvious register allocation.
- Hardware based unrolling of loops, that spreads control information among available registers .

Figure 3.1 present the top level diagram of the Cortex-A9. The described functions can be located after the decoding stage.

Figure 3.1 – ARM Cortex-A9 Block Diagram



Source: [52]

The Cortex-A9 MPCore processor is the one modeled for the OVP environment and includes an enhanced version of the ARM MPCore technology, which provides the flexibility for vendors to implement between one and four CPUs in a cache coherent system architecture design, as in the Zynq™-7000 SoC that implements 2 cores.

The ARM Cortex-A9 embedded in Zynq and Cyclone FPGAs are a dual-core ARM that can be configured to work as single or dual core and to access an embedded SRAM memory, caches level 1 and level 2 and a flash-based memory located outside the chip. In this work, we use the model of the single core Cortex A9.

3.2 OVP Model of the ARM Cortex-A9 Architecture

As described by Imperas© [55], the processor models used along with the OVP simulator are shared object files, represented by the .dll files in Windows and by the .so files in Linux. The models may include support for SystemC integration.

The exclusive purpose of an OVP processor models is code conversion. It has the ability to translate the original application code previously compiled for the target architecture into the simulator own abstraction layer and for the just-in-time code mechanism, available in its VMI API. This API is used for disassembling and generation of the simulator interface, breaking the correlation with the original application, through a specific decoder that knows the real processor instruction set. The simulator is responsible for all the other tasks, leaving the execution complexity for the part of the environment that do not depends on the models being used. It implements the mentioned just-in-time mechanism that converts dynamically the original instructions into the simulator ones, memory allocation and other simulation dependencies [55].

3.2.1 Modeled Functionalities

The used OVP model is a single core processor modeled by ARM Ltd to be used along with OVP simulator, called “Cortex-A9MPx1”. Its instruction model is accurate, and the simulation results of a single thread program will correctly match the actual hardware that it models. However, not all actual architectures characteristics are modeled, these being dependent on the processor used, since the existence of a given function is a developer's decision and not necessarily a limitation of the platform. Therefore, it is possible to find processors with and without cache implemented, for example, but as mentioned the Cortex-A9 model is provided by ARM with the purpose to provide means of simulated debugging software so that it can later be used in a real processor, a factor that indicates the equivalence between the model and the device is reliable.

3.2.1.1 Pipeline e Cache L1

The Cortex-A9 instruction pipeline is not modeled, so that the bus may not be accessed in the proper sequence, but it is assumed that all instructions conclude immediately when read from memory and in the same reading order, disregarding all of the steps of load, fetch and store, for example. This means that the instructions of the type Instruction Barrier and Data Barrier, such as ISB and DSB, are treated as NOPs, except for instructions with defined behavior at runtime only. The model also does not implement the functionality of a speculative fetch.

For this reason, typically caches are not modeled, although the platform has the resources to create both L1 as the shared L2, L3, etc. The ARM used in this work does not have caches or modeled buffers, nor the branch cache shown in real processor architecture. Cache handling instructions are implemented as NOPs, but again except for instructions with defined behavior at runtime, which are modeled. However, the cache control registers are present, allowing applications that normally make use of this resource, to run normally [56].

3.2.1.2 Performance and Timing

The OVP model approximate the processor performance, but does not model accurately the execution time of the real processor, since it would need to be cycle accurate. It is given an estimative of MIPS that represents the same parameters of real processors, but in the processor model perspective, each instruction is executed in one unit of time, meaning that a fair comparison can be done between application running in the same environment, but a real hardware can produce very different performance results. The idea of time used in the OVP simulator is based on simulation time, that is incremented at the end of each iteration, which is global across the simulator and is called simulation quantum [55].

3.2.1.3 Special Functions

According to the description of the real architecture [56], the special functions that were modeled in the ARM Cortex-A9 are:

- Specific instructions for memory reduction;
- Java byte code execution for JIT operations;
- Single Instruction, Multiple Data (SIMD) are supported;
- Functions for NEON media processing;
- Vector Floating Point Coprocessor (VFP) is modeled;
- Extensions for data security modules (TrustZone);
- Address translation for secured and unsecured virtual memory.

3.2.1.4 Modeled Registers

To support the implementation of a real processor application, whole register file and support registers are implemented, enabling a better reproduction of the real behavior.

Table 3.1 lists the modeled registers of the ARM Cortex-A9 architecture.

Table 3.1 – ARM Cortex-A9 Register File

Name	# of Bits	Initial Value	Operation Type	Function
R0	32	0	R/W	General Purpose
R1	32	0	R/W	General Purpose
R2	32	0	R/W	General Purpose
R3	32	0	R/W	General Purpose
R4	32	0	R/W	General Purpose
R5	32	0	R/W	General Purpose
R6	32	0	R/W	General Purpose
R7	32	0	R/W	General Purpose
R8	32	0	R/W	General Purpose
R9	32	0	R/W	General Purpose
R10	32	0	R/W	General Purpose
R11	32	0	R/W	Frame Pointer
R12	32	0	R/W	General Purpose
SP	32	0	R/W	Stack Pointer
PC	32	0	R/W	Program Counter

Source: [56]

3.3 Linux Operating System

The Linux kernel was selected as a target for our experiments as it provides a representative Operating System. The mainly reason for choosing this OS is that it is an open source environment, which significantly facilitates the controllability and observability required by the internal experiments. Beyond that, it is worth mentioning that Linux is already considered as an option of a commercial purpose operating system, which is being included in a wide range of applications, including those with reliability requirements [13].

The controllability of the OS enables our failure classification, which will be discussed later, through the observations made after each experiment. It is correct to say that based on the error detection mechanisms of Linux we draw part of our conclusions.

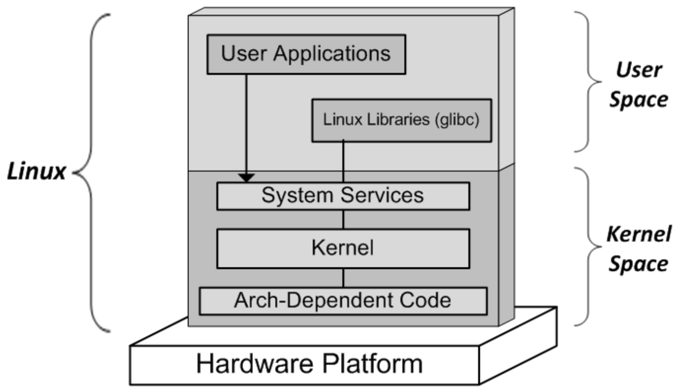
In this section, the focus is the Linux Kernel analysis and discussion on the OS exception handling mechanism.

3.3.1Conceptual Architecture

The Linux is a UNIX-like operating system based on the open source concept and, according to [57], it is composed of four major subsystems. The first is the user space, which includes the user applications in a specific Linux installation, a region where they are executed and the Linux libraries, which promote mechanisms for allocation and communication of the applications and the kernel . The next subsystem is the O/S services, gathering all OS interfaces like command shell, kernel interface for programming. The hardware controllers provide interface for physical devices, like communications systems, memory architecture, disks and the CPU itself, making this subsystem dependent on the hardware architecture. Finally, the Linux kernel, that is the main interest of this work, since it provide an abstraction layer for hardware managing, specially the processor, making this subsystem independent from the architecture.

The described topology can be seen in Figure 3.2.

Figure 3.2 – Linux System Major Subsystems



Source: Adapted from [57]

Each subsystem can only communicate with the subsystem that is immediately adjacent to it. In addition, the dependencies between subsystems are from the top to bottom, i.e., a subsystem near the top depends on lower subsystems, but subsystems nearer the bottom do not depend on those at the top [57].

Being our major interest, the Linux kernel can interface with user code through an abstraction layer, allowing applications to run without hardware specific concerns, that’s why the application running in Linux doesn't need pre cross compilation.

The Operating System used in this work is provided by Imperas Software Company, along with the model of the ARM Cortex-A9 processor and is a reduced Linux v2.6 kernel.

The codes are compiled using the standard g++ Gnu Compiler Collection and automatically run from the file system that, in the simulation platform, is a directory.

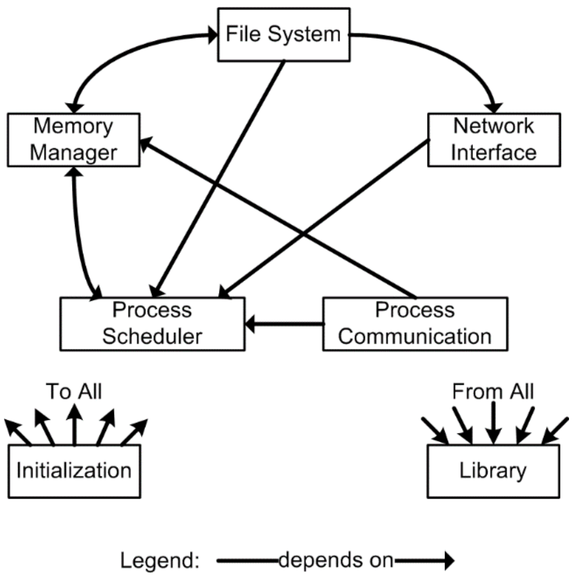
Linux adopts a monolithic kernel strategy, meaning that it has a single block compiled with all services provided by the system. The kernel also is composed by subsystems [57], where each one is responsible for implementing operating system functions, as follows:

- Process Scheduler: manages CPU resource sharing by the kernel and running applications. In real time OS versions, it is responsible for guaranty that each task is being executed on time;
- Memory Management: similar to the Process Manager, this subsystem manages accesses to the system main memory, supporting virtualization;
- Virtual File System: provides a common interface for all the data stored in hardware devices, supporting different formats of file systems;
- Network Layer: allows connection to other network devices;
- Process Communication: implement communication management between active processes in the OS;

Additionally, to support the described subsystems, the kernel also include specific libraries to sustain processes dependencies and an initialization subsystem, for setting up user dependent configurations.

The described architecture and how the subsystems interact is shown in Figure 3.3.

Figure 3.3 – Linux Kernel Architecture



Source: Adapted from [57]

3.3.2 Exception Management

One of the advantages of the Linux is the ability to detect natively exceptions by implementing dedicated kernel functions that monitors each application operation before execution. Kernel calls functions are generated via interrupts and manifest themselves in 4 ways: (1) an interruption issued to the processor by a hardware device indicating that it requires attention, (2) an exception indicated by the processor due an error, (3) a kernel call or system call issued by an application, or (4) a kernel thread [58]. The activation of internal kernel functions is not defined only by the mentioned events, but also the current kernel state. In this work, we focus on the second and third items (kernel calls issued by an error in the processor or by an application), since the injected faults can result in both behaviors and are reflected similarly in the operating system mechanisms [59].

In this scope, a process may result in a hardware exception generation when, for example, it attempts to divide by zero or fails translating a virtual address. In a UNIX based operating system, this event automatically changes the processor context to start the execution of an exception handler in the kernel. In this cases kernel are able to manage the exception, like in a 'page fault' that happens when there is a mapped address page which is not loaded in fact, but the kernel can pre-identify the problem before an illegal access actually occur. As a result, the application exits, but if possible the page will be allocated, remapped and the application flow can follow.

When an exception is detected and cannot be handled by the kernel's internal mechanisms, then it is identified and a default trigger mechanism is called in order to handle these exceptions. This mechanism is called signals, where the system sends a notification to the application with the type of exception that occurred [58][60]. Examples of such exceptions that triggers default signals are:

- Division by zero: a division by zero error exception (SIGFPE) is generated by making the kernel sending this signal to the application.
- Segmentation Fault: access to a memory address out of virtual address space, making the kernel notify the application through a SIGSEGV signal, since he cannot know the right address.

By sending a signal back to the application indicating the exception, it may contain a handler to treat it. The existence of these handlers depends on the software developer, and if it does not exist, the operating system can terminate the application. In this work, all benchmarks were edited by including a default signal handler able to capture any signal propagated by the Linux kernel with no corrective action, but only generating an external message to the OVPSim platform in order to provide failure classification for further result gathering.

3.4 Fault Injection Environment

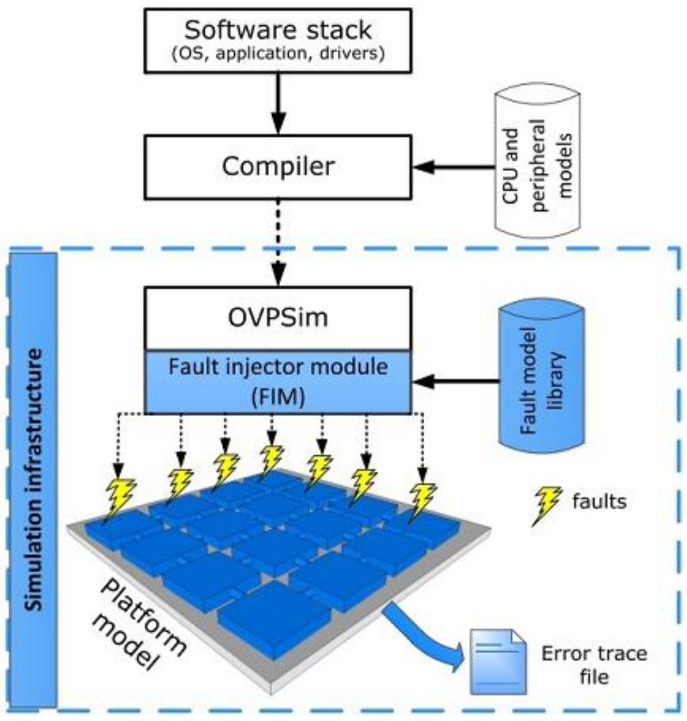
The embedded systems emulation method for fault injection envisions the reduction of limitations when compared to other techniques. Since the architecture model is available in software, it is possible to obtain a high degree of control and observability in the experiments provided by the source code emulator. Through this, it is possible to emulate soft errors in the memory elements of the available processor model. In contrast to several fault injection techniques, this approach does not require application modifications that will run in the target processor for fault injection, except for diagnosis improvements regarding software masking effects. Another relevant aspect is the emphatic run time reduction of the experiments compared to hardware description simulation through tools such as ModelSim. The software validation for state of the art processors also makes this approach attractive, since the manufacturers do not normally provide hardware description for COTS devices.

As previously defined at Section 3.2.1.4, all the faults in this work are injected into the processor’s register bank. The ARM processor embedded at the studied SoC is a single-core ARM Cortex-A9. The system is simulated with the OVPSim simulator, defined at Section 2.3. The fault injector is the OVPSim-FIM and its technique will be introduced at Section 3.4.2. Each benchmark execution has one random fault injection, as defined at Section 3.4.2.1. The model used to simulate the ARM Cortex-A9 was the model developed by ARM Ltd itself to be specially used at OVPSim, called "Cortex-A9MPx1". This model is validated by use, since it is largely used by embedded software developers along with the OVP simulator, which means that realistic simulations can be possible with this environment.

As the fault injections are limited to the ARM's register bank, the only part of the SoC simulated is the ARM processor itself and the minimum infrastructure, i.e., memory and bus models. Other elements such as I/O, DMA and special interfaces are not of importance for this work.

Figure 3.4 depicts the composition of all this elements that composes the environment.

Figure 3.4 – Proposed fault injection framework organization


Software stack
(OS, application, drivers)
Compiler
CPU and peripheral models
Simulation infrastructure
OVPSim
Fault injector module (FIM)
Fault model library
faults
Platform model
Error trace file

Source: [12]

This section will define the fault model used in both approaches (bare metal and Linux OS) and how it is implemented by the Fault Injector module by discussing its architecture and detailing the Figure 3.4.

3.4.1 Fault Model Definition

The fault model used is called bit-flip or upset, which is nothing but a modification of the contents of a storage cell during program execution.

In a processor, the regions likely to be affected by this kind of faults are the internal memory cells, logic registers, control registers and registers that cannot be accessed by the instruction set. In addition, register files and integrated caches can be included in this list. Despite the wide range of memory devices, the models only allow access to the register bank, as shown in Table 3.1, therefore the faults are emulated only in these registers.

Despite its relative simplicity, the bit-flip model is widely used for real faults, since it accurately corresponds to the default behavior in a real case [62].

3.4.2 Fault Injector Module

The simulation platform already described in this work provides an API called ICM, briefly described in Section 2.3.1. Through this API is granted read and write access at any time in any part of the architecture implemented on the platform, such as the processor, bus, memory, or cache. Access is possible during the simulation, exception handling or by calling specific platform functions. Thus, it is possible to trigger an access when a certain event occurs, which generates a function call by the simulator capable of manage any architecture element in accordance depending on the function features.

The Fault Injection Module (OVPSim-FIM) used in this work was introduced by [12], and developed with the support of the microelectronics post-graduation program of UFRGS. It uses the ICM resource and is responsible for inserting SEUs, capturing the processor model exceptions for results elaboration and reports generation. This module is individually connected to an OVP processor model, that in this work is the ARM Cortex-A9 with one core, as mentioned, providing independence between simulations, i.e., numerous platform instances comprising processor, memories and fault injector can be run concurrently, as each processor model includes one fault injection event.

The fault injector module uses the instruction bus monitoring at runtime and can be used in any processor model. The monitoring function has been developed using the Callback API of the OVP platform, which is called when any previously configured event occurs. Then, after the detection of this event, the simulator calls the data handling function, pointing to the fault injector module.

The callback function triggers the execution sequence of the fault injector by accessing the fault information, position, location and mask pattern that will always be a single-bit flip. To ensure a better distribution of faults, the module checks if the position to inject the fault is the same as the previous one. If it is valid, another ICM function is used to read the selected register value and store its value in a temporary variable. By using a masking operation, it is possible to invert the target bit, emulating an SEU. After that, the resulting value is written back in the selected register and simulation returns where it left off.

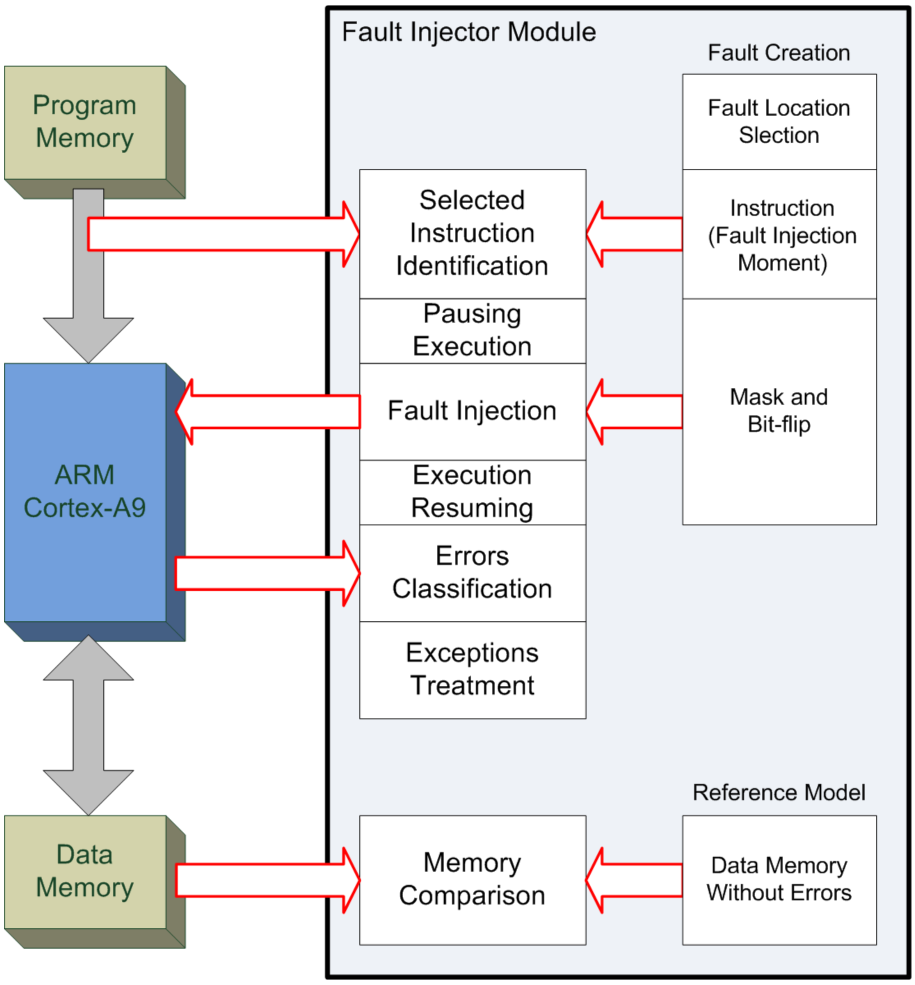
Depending on the target register, the erroneous value can lead to an exception in the processor, such as a shift in the program counter register that may cause a simulation crash, and is an intrinsic behavior of the OVP processor model to be interrupted until this exception be handled. Several situations may occur due to the random fault injection, like the processor fault manager activation or read/write events beyond the extent of memory accepted by the platform, which is limited to 500 MB for any application. Each one of these errors generates a specific exception in the model. The fault injector module allows you to separate the different exceptions individually and to continue running from the stop point, as long the OVP scheduler are returned to the processor´s model when an exception is captured.

It is worth emphasis that the objective is to identify possible failures arising from the impact of a SEU and not to treat or mitigate them. To do this, as many environments run simultaneously, we must remove the processor that has failed without affecting others. The icmFreeze function is responsible for remove a specific processor, allowing the simulation to be resumed for all pending models. This process is repeated until all of the processors complete their operation or generate an exception. Eventually the generated error might lead the application to fall into an infinite loop, generating an excessive memory consumption and simulation time; to avoid such a situation the application, every time the current simulation run 10% more instructions than the expected number, it is considered incorrect, the processor is removed and the simulator generates an exception.

If no exception is identified during the execution, the memory still needs to be compared. Thus, the model recovers the memory map attached to the processor model and compares with a known map without errors and this process is repeated for all running simulation instances. The collection of exceptions and differences between memories in the simulations are compiled in a final report.

Figure 3.5 shows the described structure of the fault injector module.

Figure 3.5 – Fault injector module architecture



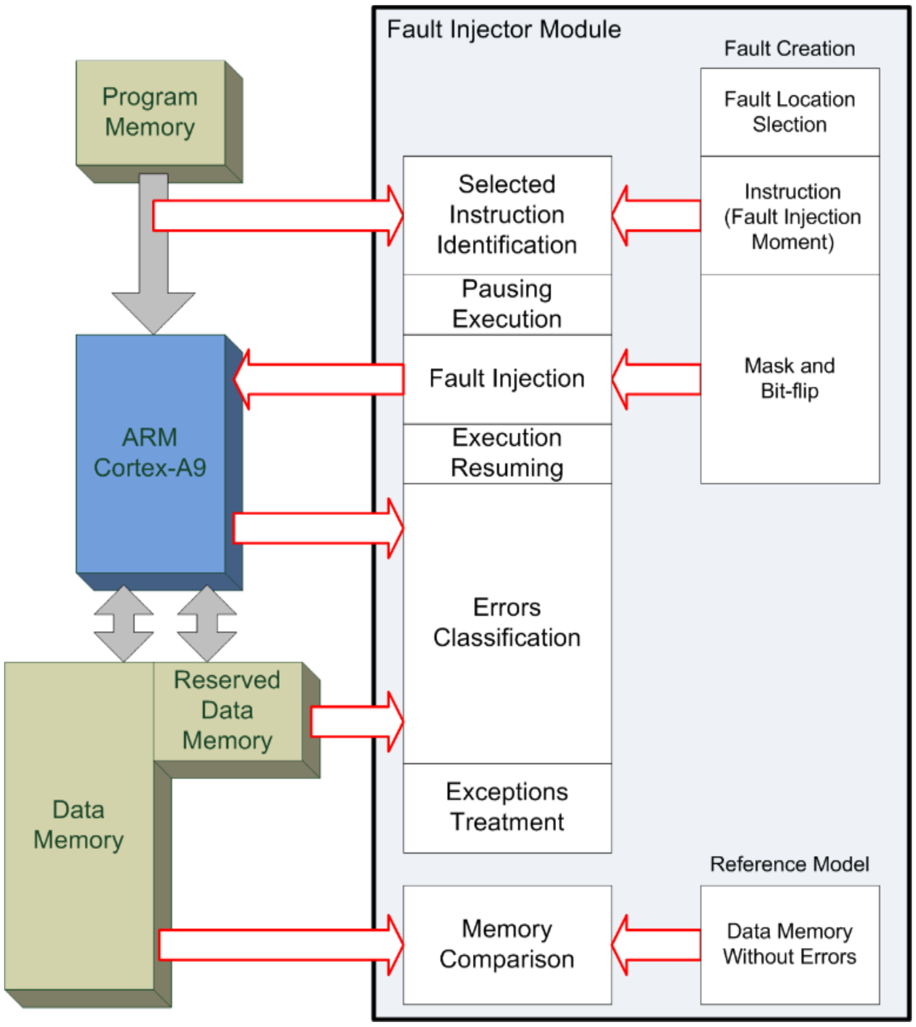
Source: Author

However, the presence of an operating system may cause that some errors are masked. As described in the exception handling Linux mechanisms, an exception indicated by the processor because of an error can generates a kernel call and the OS are able to manage the issue and resume the application without crashing or hanging the system. In the eyes of the processor, no error was reported and consequently the platform or the presented fault injector can identify no failures.

To work around this phenomenon, a different fault injector was proposed when the Linux OS is on top of the embedded system. Each application needs to be edited by including a default signal handler able to capture any signal propagated by the Linux kernel. The signal handler should have no corrective action; it only generates an external message to the OVPSim platform in order to provide an failure classification for further result gathering. The message is able to write and classify each Linux signal in a reserved memory area with no code or application data. During results collection phase, the fault injector can access this memory and classify the items together with default platform reports.

Figure 3.6 shows the proposed fault injector structure to support Linux exception signals.

Figure 3.6 – Fault injector module architecture for embedded system with Linux



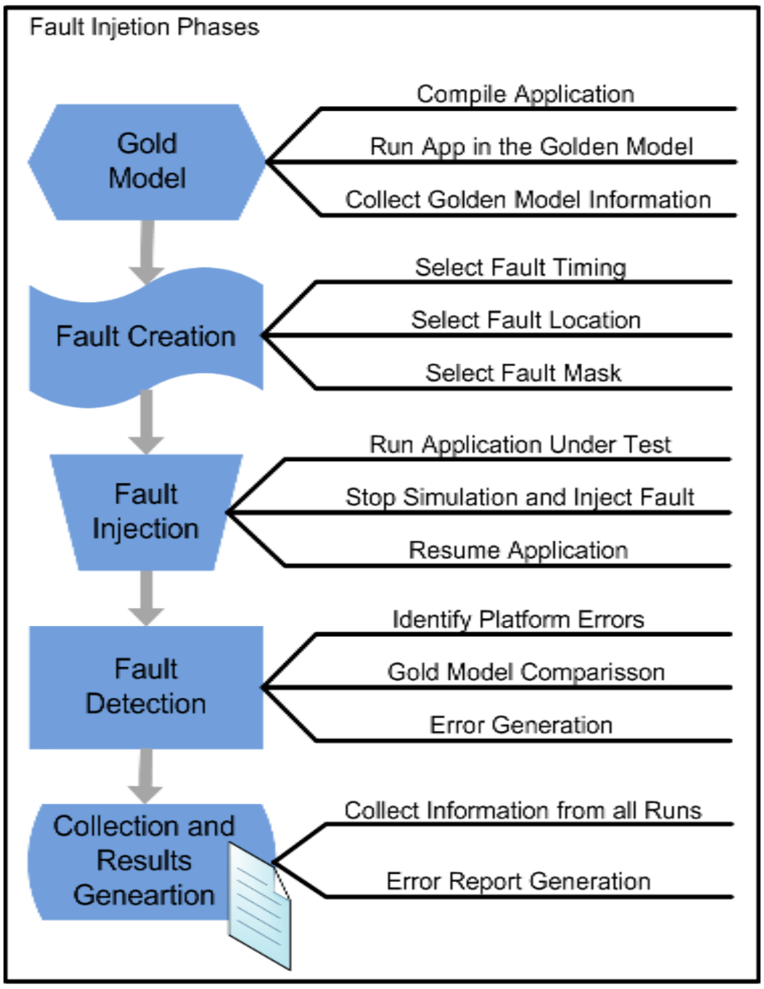
Source: Author

3.4.2.1Fault Injection Phases

For evaluation, fault injection comprises five steps: executing a simulation without injecting faults, fault creation, fault injection, failure checking and result collection.

Figure 3.7 present all the steps and its subdivisions.

Figure 3.7 – Fault Injector Operation Flow



Source: Author

The execution of the environment without fault injection, also called gold model, is performed with the target platform and code without any intervention, so that from this simulation the essential information such as instruction count and memory map are collected. The same application that is compiled using the proper cross-compiler, is executed in the gold model and in the fault campaign platforms. The instruction count is used in the next step as a reference for pseudorandom register selection where the fault is injected. The memory map file contains a copy of the memory content, containing all addresses that have changed in comparison to the initial memory, avoiding that the full memory space is copied, that is, even if the memory model used on the platform have 10 MB, if an application uses a smaller quantity of data, the copied memory map will contain exactly what the application uses, not all 10 MB.

The second phase comprises the fault creation where the SEU that will be injected is implemented in pseudo-random manner. This stage selects the timing, location and fault mask. As mentioned above, the OVP platform has granularity of an instruction (instruction accurate) and consequently the smallest element where a fault can be injected is during execution of an instruction, ignoring at which clock cycle the fault happen. The fault injector in this work selects a pseudorandom position that is, in practice, the execution time of the application. This position is nothing more than a number between 0 (zero) and the last instruction that were extracted during the previous step for the bare metal environment. For the Linux environment, the position starts where the application starts in memory. Once the insertion time has been selected, it is necessary to select a location, i.e., a flip-flop in the register file to introduce a single fault. The process for choosing the register is also pseudorandom and follows a similar concept as the timing selection, but now from a list containing all registers.

As previously discussed, the processor model provides only registers that can be observed from the application standpoint, not including therefore registers that exist in the physical architecture, as those present in the pipeline. In this work, faults are injected into the entire register file as listed in Table 3.1 and are implemented as a single bit flip. To do this, a bit mask to "1" is used to inject this event, and only one bit is selected for masking to "0" (zero), exchanging its value by doing an XNOR operation between the mask value and the actual register value. For example, to change the third bit in a 32-bit register, the mask value would be 0xFFFFFFFB. As in the other steps, the generation of these masks is also pseudo-random. After mask value definition, the selected position receives its opposite value, ensuring that the fault will produce an error, since the value in a real situation caused by SEU can be the same as the bit already has, and does not represent an actual error. In this third stage, beyond the fault injection, any exceptions that the simulator accuse due to error caused by unexpected behavior of the processor model is captured. As multiple instances of the simulation environment can be triggered independently, each one can simulate a fault injection and executes the described process.

In the fourth stage, the simulation results are compared to the results obtained from the gold model. In this work, a fault is counted as a failure when there is a change in the application behavior or when data stored in memory is different from the expected.

The last step comprises the information collection where results are gathered from all simulations platforms and compiled in a final report.

3.4.3Failure Classification

In the last OVPSim-FIM step, the final report is generated and the simulation campaign shall then be interpreted. The most important information for this work’s analysis is the number of failures and their classifications. In order to establish a relationship between both environments, bare metal and Linux failure classification are equivalent. The failure types are defined, alongside with their categorization, which helps to understand better the effect of the errors on the application run, as follows [12]:

- Masked_Fault: The fault produce an error that was masked, meaning that simulation finishes normally, without processor signaling or memory incoherencies with gold model;
- Control_Flow_Data_OK: Divergence between executed instructions during test and gold phase, but memory is correct;
- Control_Flow_Data_ERROR: Divergence between executed instructions during test and gold phase, and also between memories;
- REG_STATE_Data_OK: Internal state is incorrect, but memory is correct;
- REG_STATE__ERROR: Both internal state and memory incorrect;
- SDC: Silent data corruption failure, meaning that simulation ended with no noticeable events, but memory ended different than gold model;
- RD_PRIV: Read privilege exception;
- WR_PRIV:Write privilege exception;
- RD_ALIGN: Read align exception;
- WR_ALIGN: Write align exception;
- FE_PRIV: Fetch privilege exception;
- FE_ABORT: Fetch an inconsistent instruction;
- ARITH: Arithmetic exception in an instruction;
- Hang: Application presumably in an infinite loop;

The above classification does not take into account the OS failures. As mentioned in Section 3.4.2.1, the Fault Injector was modified to handle Linux signals. The fault injection campaigns have shown that between operating system available signals, Linux was able to detect only segmentation fault, illegal access and floating point exception violations, and so, only these signals were included in results.

- SEG_FAULT: Segmentation fault error (the application tries to address a non-allocated memory area);
- ILLEGAL_INST: Illegal instruction error (the processor attempts to execute an illegal, malformed, unknown, or privileged instruction);
- FLOAT_PT_EXC: Floating Point Exception (code generates an overflow, underflow, or divide-by-zero error).

Nevertheless, for practical reasons those types of failures are re-classified on three different groups, based on their behavior and consequences to the application execution: HANG, SDC and UNACE. Those groups are defined below.

- SDC: Those failures are detected due to a difference in the final memory from the golden phase execution and the fault-injected test executions.
- HANG: Failures that cause the application to be stuck in a certain point. Not necessarily in a given instruction, but also in an infinite loop.
- UNACE: Any error that had no influence at all at the final memory state, i.e., memory is just as if it was predicted to be by the golden phase execution.

Table 3.2 classifies the previously defined Failures into those new classifications.

Table 3.2 – Failure Group Classification

Failure Group	**Failure Types**
HANG	Hang, RD_PRIV, WR_PRIV, RD_ALIGN, WR_ALIGN, FE_PRIV, FE_ABORT, ARITH, SEG_FAULT, ILLEGAL_INST and FLOAT_PT_EXC
SDC	SDC, Control_Flow_Data_ERROR and REG_STATE_Data_ERROR
UNACE	Masked_Fault, Control_Flow_Data_OK and REG_STATE_Data_OK

Source: Author

This classification helps to better identify the type of failure and what may have caused it. Note that not all of the nineteen types of failures are defined at Table 3.2. That is because this failure grouping definition does not take into account the errors that were caught by Linux.

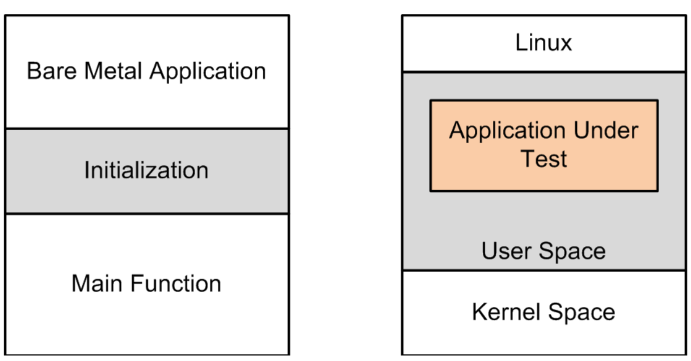
4 APPLICATIONS ENVIRONMENT

In this section, it is defined the environments in which the tests were proceeded. For both platforms, bare metal and Linux, a brief description of compilation methodology and fault timing selectivity is presented, since the presence of an Operating System requires particular techniques to provide fair comparisons. These different environments are intended to define the level of implementation complexity, which for bare metal application is considered lower, providing means to evaluate the impact these approaches. In total, eleven benchmarks are presented and discussed in terms of functionality but particularly its data behavior diversity. The simulation report structure is also presented, providing an overview on the platform observability.

4.1 Testing Platform

The structure of each system that is target of faults can be seen in Figure 4.1. In each environment, the complete set of benchmarks is run to exercise the registers in the emulated processor's register file. The bare metal system, Figure 4.1 (a), is set in two layers, a processor initialization and the main function, which is responsible for implementing the applications. The Linux system, Figure 4.1 (b), considered the most complex system between the environments, has a Kernel v3.7.1 and it is structured in two main levels: user space and kernel space. In the user space, a process for benchmark execution is defined, and again the purpose is to generate debug messages to indicate the condition of the system that is running in the emulator. The user space naturally shares resources with the kernel space, whose functions was described above.

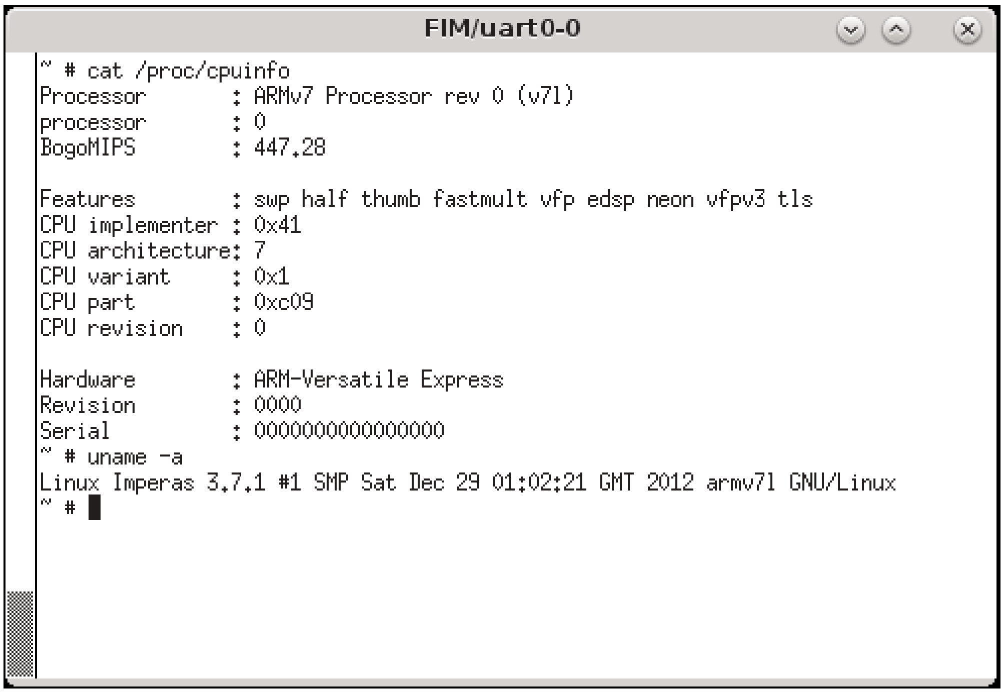
Figure 4.1 – Test Environments Structure



Source: Author

The system setup on where OVPSim runs is not relevant on the simulation results, but it shall influence only the simulation execution time. It also has no influence at all in the simulation model execution. The benchmarks execution relies only on the defined target architecture being simulated, in this case the ARM Cortex-A9 in an ARMv7 architecture version. Figure 4.2 shows the system information given by the simulator itself.

Figure 4.2 – Emulated CPU Information



```
~ # cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 0 (v7l)
processor       : 0
BogoMIPS        : 447.28

Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x1
CPU part        : 0xc09
CPU revision    : 0

Hardware        : ARM-Versatile Express
Revision        : 0000
Serial          : 0000000000000000
~ # uname -a
Linux Imperas 3.7.1 #1 SMP Sat Dec 29 01:02:21 GMT 2012 armv7l GNU/Linux
~ # █
```

Source: Author

The tests were executed on a personal computer with an Intel® Core™ i5-5200U CPU @ 2.20GHz processor, 4GB of RAM and running CentOS 7 Linux as OS. The test execution time for each benchmark is very variable, as some are more complex and computing demanding than others. All tests were automated by script, allowing parallel simulation with multiple environment, each one composed of only one embedded processor and memory.

4.2 Benchmarks

In this work, we choose to work along with Worst-Case Execution Time (WCET) analysis project [63] that performs research in static WCET. In this research project, the application does not run in a conventional way, instead it derives the WCET information by analyzing the characteristics of the program code and the target hardware. The focus is on methods for deriving safe information on the possible executions of a program, like iteration limits of loops and dependencies between conditional statements. The group maintains a large number of WCET benchmark programs, used to evaluate and compare different types of analysis tools and methods. The benchmarks are collected from several different research groups and tool vendors. Each benchmark is provided as a C source file.

All benchmark programs are single path programs. While different input data cause the code to execute on different execution paths with different execution times, the single-path approach avoids this uncertainty by ensuring that the code has only a single execution path. To do this, the approach uses code with loops with invariable iteration counts and branches dependent only on local data, avoiding I/O operation. By using this benchmark, we guaranty predictability on execution time making the bit-flip localization equally distributed in all simulations during fault injection campaign.

Ten benchmarks from WCET group were used in the tests. Additionally, an Imperas benchmark was included since its intention is to achieve maximum CPU performance. They were chosen intentionally in order to exploit the diversity of behavior and to attempt to reach failure detection classification variety, as described below:

- binary_search: Binary search relies on a divide and conquer strategy to find a value within an already-sorted collection. It divides a range of values into halves, and continues to narrow down the field of search until the unknown value is found. It can be used to access any sorted collection data quickly;
    - Loop based;
    - Uses arrays and/or matrixes;
- bitManipulation: To act on data at bit level or set of bits using boolean operations is called bit manipulation. In computing, this technique is used for low-level operations such as device control, or in some algorithms such as detection and error correction and encryption, as well as for optimization. The operations used in bit manipulation are boolean operations like AND, OR, XOR, NOT, the arithmetic and logic shifts and rotations;
    - Uses arrays and/or matrixes;
    - Bit operations
- bubble: The bubble sort makes multiple passes through a list, comparing adjacent items and exchanging those that are out of order. Each pass through the list places the next largest value in its proper place, doing it repeatedly until

no swaps are needed, which indicates that the list is sorted. It is a good example of an O(n^2) complexity algorithm;

- Nested loops;
- Uses arrays and/or matrixes;

- compress: The algorithm is based on the common Unix compression utility. This version has been modified to its work in memory, rather than reading and writing files, to avoid disk IO. Only compression is done from a buffer containing random data to another one;
  - Loop based;
  - Uses arrays and/or matrixes;
- crc: Implements a cyclic redundancy check computation on 40 bytes of data by using a 16-bit CRC CCITT standard with polynomial $x^{16} + x^{12} + x^5 + 1$, which is now widely used for a CRC checksum;
  - Single path stereotype;
  - Bit operation;
- factorial: Simple algorithm that repeats 50 times the factorial operation of 25. This algorithm was chosen in order to exploit recursion;
  - Nested loops;
  - Recursion;
  - Data interdependence;
- fdct: The forward discrete cosine transform is a technique for converting a signal into elementary frequency components, similar to the discrete Fourier transform. It is widely used in image compression;
  - Uses arrays and/or matrixes;
  - Floating point operations;
- harm: implementation of a harmonic series defined by equation (4.1)

$$\sum_{n=1}^{\infty} \frac{1}{n} \tag{4.1}$$

  - Loop based;
  - Floating point operations;
  - Data interdependence;

- matrixMult: This program multiplies two 20x20 square matrices resulting in a 3rd matrix. It uses compiler's resources to handle multidimensional arrays and simple arithmetic and exploits looping code behavior.
    - Single path stereotype
    - Nested loops;
    - Uses arrays and/or matrixes;
- mdc: The Euclidean algorithm is a way to find the greatest common divisor (the largest positive integer that divides the numbers without a remainder) of positive integers;
    - Loop based;
    - Data intensive;
- peakSpeed: This is the Imperas benchmark to achieve maximum CPU performance. It consists of 500 interactions of sequential variables attributions, concluding with the sum of all compromised values;
    - Loop based;
    - Data interdependence;

Table 4.1 shows the number of instructions executed in bare metal and in Linux for the set of applications. The Linux boot executes 1,176,574,110 instructions. Note that the number of instructions of the application executed in bare metal and in Linux is very similar. The difference comes from the fact that for bare metal we use a compiler for the ARM Cortex-A9 architecture, while for Linux we use the default GNU compiler, since the application is on top of the OS, what results in a different instruction count.

Table 4.1 – Number of Instructions Executed in Bare metal and in Linux for the set of Applications

Applications	Bare Metal # of Exec. Instructions	Linux # of Exec. Instructions
binary_search	138,293	130,010
bitManipulation	305,211	293,520
bubble	236,548	236,164
compress	196,736	189,141
crc	201,812	223,448
factorial	310,343	310,061
fdct	507,696	537,709
harm	449,123	448,839
matrixMult	343,373	343,172

mdc	658,172	510,295
peakSpeed	19,859	19,574

Source: Author

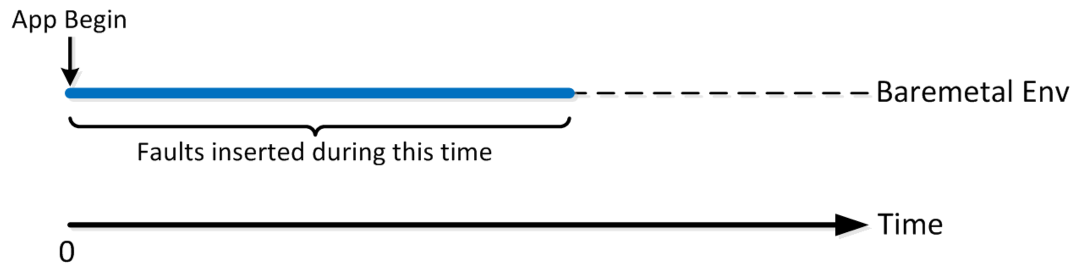
4.3 Fault Distribution

The fault injection process in 15 registers of the Cortex-A9 architecture was made in a random distributed manner, i.e., register bank were injected with 100000 faults that guaranty an approximate distribution for each register. Thus, 1.1 million faults were injected during the campaign of each environment, since we are working with 11. To ensure a known condition for the comparison step of the results in the fault injector, at the end of each benchmark running the whole system is returned to the initial value, therefore, no more than one fault is injected per application execution. After that procedure, a new run is initiated, in order to emulate a new fault.

As mentioned, due to Linux ability to handle errors, OVPSim-FIM classification was extended to include signals triggered by kernel categories. The signal handler added to each application code has the ability to write and classify each signal in a reserved memory area with no code or application data and after that, resumes the simulation. During results collection phase, the fault injector can access this memory and classify the items together with default platform reports.

When executing an application in bare metal mode, all instructions in memory belong to the application and the code is compiled directly for the architecture. As a result, it is guaranteed that any injected fault will aggress a register with an application content. Figure 4.3 presents the bare metal fault distribution approach.

Figure 4.3 – Bare Metal Fault Distribution over Time



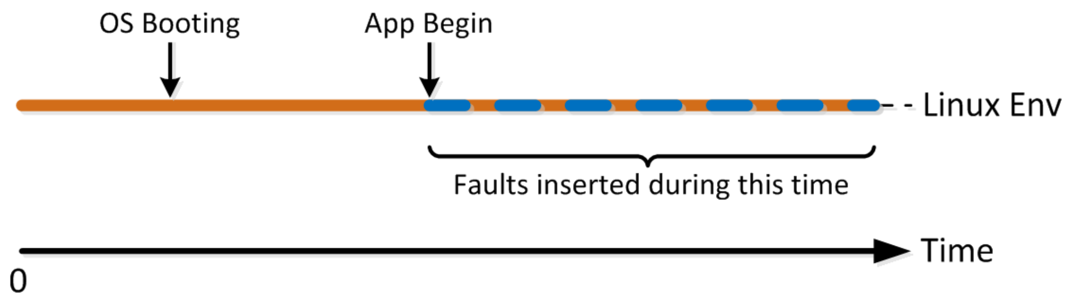
Source: Author

Applications that run in Linux shall be compiled for the specific operating system and the resulting memory occupation is very different from bare metal, as it includes the OS data. This works aims to analyze the application behavior under faults in both platforms, but in this case much OS Linux architecture runs alone and a fault injected in this moment would check only the Operating System. In order to run the benchmark, the Linux shall be booted first, resulting in a much higher number of instructions than in bare metal version. This behavior generates an unfair comparison between platforms, since random fault injections are most likely to happen during Linux boot than in the application itself. Safe critical applications that use operating systems are commonly initialized before being exposed to the susceptible environment, thus booting should not be taken into account. In addition, when performing the OS boot, all the latent errors are cleaned.

OVPSim-FIM was modified to identify application beginning and guaranty that fault injection happens only after it starts, but after that, Linux kernel is also exposed to faults along with the application, since it shares the processor resources even running with only one application.

Besides that, the application signal handler is an additional code used as a solution to overcome the inability of the platform to capture exceptions since Linux was able to prevent. This strategy may lead to an unbalanced analysis if these additional functions are take into account during fault injection, as bare metal applications do not include this code. So, in addition to finishing the simulation when a Linux exception signal is captured, it also informs to fault injector the moment where injecting faults is allowed, that is after signal handling functions initialization. Figure 4.4 presents the Linux fault distribution approach.

Figure 4.4 – Linux Fault Distribution over Time



Source: Author

4.3.1System Observability

Each simulation, independently the environment it is running, results in three tables. The first table summarizes all executions of a given benchmark in the following characteristics:

- Fault number;
- Selected instruction number for fault injection;
- Selected register;
- The mask to determine the position in the register that will suffer the bit-flip;
- The value that the program counter stopped when simulation ends due to normal or exceptional reasons;
- The result of comparison between the golden model memory and the one under analysis;
- The failure detection, and;
- The simulator flag returned at the end of the simulation.

The second table summarizes all executions in terms of architecture, i.e., the place where the fault was inserted. This analysis is of great importance as it highlights the critical points in microprocessor architecture. The table lists in order, the register where the fault is injected, the injected total number of faults in that register, the percentage of injected faults, the number of failures that occurred due to an inserted fault and the error rate.

The third table is complementary to the second one, as it depicts the failure classification per register, i.e., the failure gamma that arises in each register during the campaign. This information is valuable, since along with critical registers highlighted by the second table, the know of failure types in a given register can provide means to understand how each application allocate processor resources and consequently knows which one to protect.

Second and third result tables will be presented and discussed in Chapter 5. Table 4.2 presents an excerpt of the first table from an example report.

Table 4.2 – Result Example with Faults Details

Fault #	**Instruction #**	**Register**	**Bit-Flip Mask**	**Final PC**	**Memory Comparison**	**Failure Detection**	**Simulator Flag**
0	130,811	R1	0xFFFFFFFF	276,735	0	0	3
1	160,830	R11	0xDFFFFFFF	160,830	1	1	12

2	210,709	R7	0xFFFFFFDF	276,735	1	1	3
...							
18	28,631	R3	0xFDFFFFFF	276,595	0	1	3
...							

Source: Author

In Table 4.2, always the first line corresponds to the simulation in which no fault has been inserted and which is used as reference for the other simulations, i.e., the golden model. It is for this reason that the mask in line 0 (zero) is 0xFFFFFFFF, corresponding to no bit-flip. It is also through this implementation reference that can be seen the highest value that the program counter takes, indicating that this is the last instruction number of the test application. The simulator flag always finishes with number 3 and 12 in all tests, which means, respectively, that the simulation has ended normally and the simulation was stopped by icmFreeze function previously described.

Based on this information we can still notice some behaviors resulting from the fault injection. In line 1, a failure in the R11 register (frame pointer) is expected, since this record helps a function return along with stack pointer. This type of failure is noticed by the platform, causing a simulation crash and forcing the environment to kill it. This event generates two actions in the report: failure detection and final memory discrepancy, as the execution does not come to an end and therefore not finished writing in memory.

Some errors, as the depicted in line 2, will not cause the simulation interruption and will execute completely. However, the comparison between the final memories warns discrepancy in the results, indicating that the bit-flip probably happen in a register with data stored in it that was written back to the memory or that was used in a result calculation.

The line with index 18 shows the opposite; a simulation that was not killed by the simulator (simulator flag 3) due to the error, ended prematurely, but at the same time executed enough code to complete the simulation with correct memory data. This case is known for demonstrating the fact that an error may manifest later in the execution (fault injected in the first instructions, but which generates locking near the end of the application) and that was not detected in terms of data generated erroneously.

5 RESULTS AND ANALYSIS

In this chapter, the results of the test campaigns discussed in Section 4 are presented. At first, our test environment and methodology is compared to a real hardware setup in order to establish a relationship between simulated and physical testing.

After that, all fault injection procedures were repeated for both environments, bare metal and Linux OS, and the results aimed to establish comparison between them. Initially, we evaluated the types of failures that take place in each application, classifying them according to Section 3.4.3, comparing them with each other and between environments. The classification proposal aims to synthesize the behavior under a functional perspective, but the complete classification can be seen in Appendix A. At first, both platforms will be analyzed equivalently, without concerning the operating system features. Also in this context, will be possible to analyze from the perspective of the ARM Cortex-A9 processor architecture the location of these errors and recognize the most sensitive points of the register bank to some benchmarks. In a second step, we refine the analysis comparing the final memory, separating the effects of latent errors from the ones that generate wrong results.

Finally, an analysis on architectural and application vulnerability factor will be presented in order to sustain the work developed in this study, along with the overall error rate between environments for all benchmarks.

5.1 Interruption-based Physical Environment Comparison

Before comparing the results between Linux and bare metal environment achieved by simulation in our OVP environment, we must establish a correlation between the real processor in an equivalent methodology. This comparison is based on the study developed by Lins [68], where a Xilinx Zynq-7000 Processing System, that includes an embedded single core ARM Cortex-A9, was tested against an interruption-based environment, commonly called hardware emulation. This approach exploits an identical strategy used in this work, where a bit-flip is forced in an internal memory element triggered by a pseudo random interruption. In [68], Lins evaluate the behavior of three applications by injecting faults in the processor’s register file, from R0 to R12.

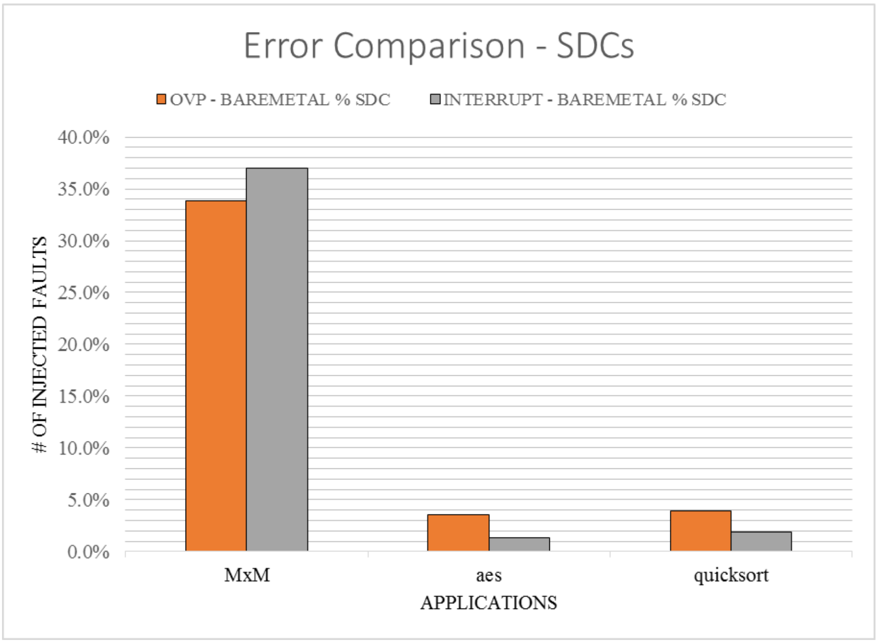
This method is based on the idea published by Raoul Velazco [69] that is based on adding functions to your code that are able to inject upsets in the register file, PC and SP registers. These functions, as implemented by Lins, are triggered by hardware interruptions, which randomly select fault location and time. By doing this, the method include instruction sequences to inject faults on critical control registers, often not directly accessible by the instruction set. As depicted by Velazco, the approach has the same limitations as our simulation based method, since not all possible sensitive elements can be reached and the effects of SEUs cannot be seen during instruction execution.

In the referred environment, when the application is completed the output is compared with a previously executed application with no faults inserted. The interruption-based platform is able to categorize two possible failures: (a) a mismatch between executions are identified as a Silent Data Corruption (SDC) and (b), whenever the ARM becomes unresponsive or sends garbage through the serial communication used for result collection, a Single Event Functional Interruption (SEFI) is detected by a watchdog in the host PC, causing a system reboot. Those classifications are equivalent to the SDC and HANG categories previously explained in section 3.4.3. The platform also allows logging the register where the fault was injected in the cases where it caused corruption of the expected output value.

The applications under test were a Matrix Multiplication (MxM), Advanced Encryption Standard (AES) and Quicksort algorithms. All of them were executed approximately 100000 times each in the interrupt-based platform, and exactly 100000 times each in OVP platform. For comparison purposes, the same application sources were used, compilation optimization was set to level 2 and exposed registers (R0 to R12) are the same in both environments. However, the software compilers and assemblers are different, since Xilinx provide its own development and integration platform, despite having the same processor architecture.

Figure 5.1 depicts the SDC error comparison between platforms for all mentioned applications.

Figure 5.1 – SDCs Comparison with Interruption-based Platform

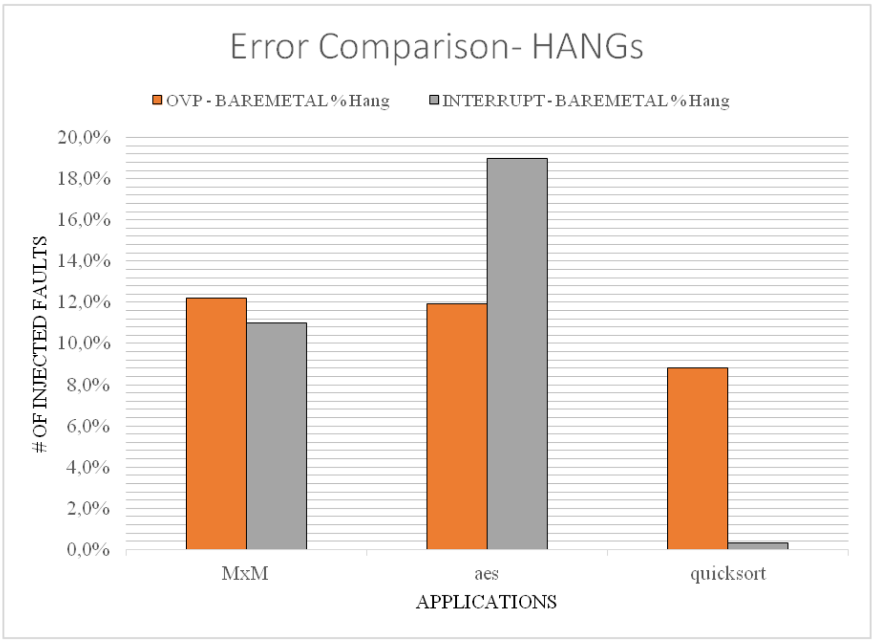


Source: Author

Matrix multiplication, AES and Quicksort applications have failures rates around 35%, 3% and 2.5%, respectively. As can be noted, there is a correlation between the OVP platform and the hardware emulation by interrupt-based fault injection in terms of proportion, i.e., in a single application, both environments have no more than 3% of difference from the total amount of injected faults. Considering that and the amount of executions performed in both environments, it is possible to say that the application has a similar flow and data management, which means coherence between real processor's architecture and modeled one. Observing the results from this perspective we cannot conclude that the observed difference is a result of the environment discrepancies, since both methods use pseudo random fault distribution, which can lead to differences even bigger between applications in the same environment. However, if we compare the amount of failures, it is possible to have a different interpretation, where OVP environment has 2 times more failures for Quicksort and 2.3 times more failures for AES algorithm. Despite the applications behave similar in the processor, we cannot say that data distribution in the register file are analogous, because the results shows that interrupt-based platform is twice times less susceptible to SEU faults. One possible reason to this behavior is the compilation process, since we were not able to compile the code for both platform in a single compiler. The result can be different optimization strategies, register allocation and/or distribution, which reflects on the error rate.

Figure 5.2 shows the comparison between errors that results in a crash or undesirable behavior of the same applications.

Figure 5.2 – HANGs Comparison with Interruption-based Platform

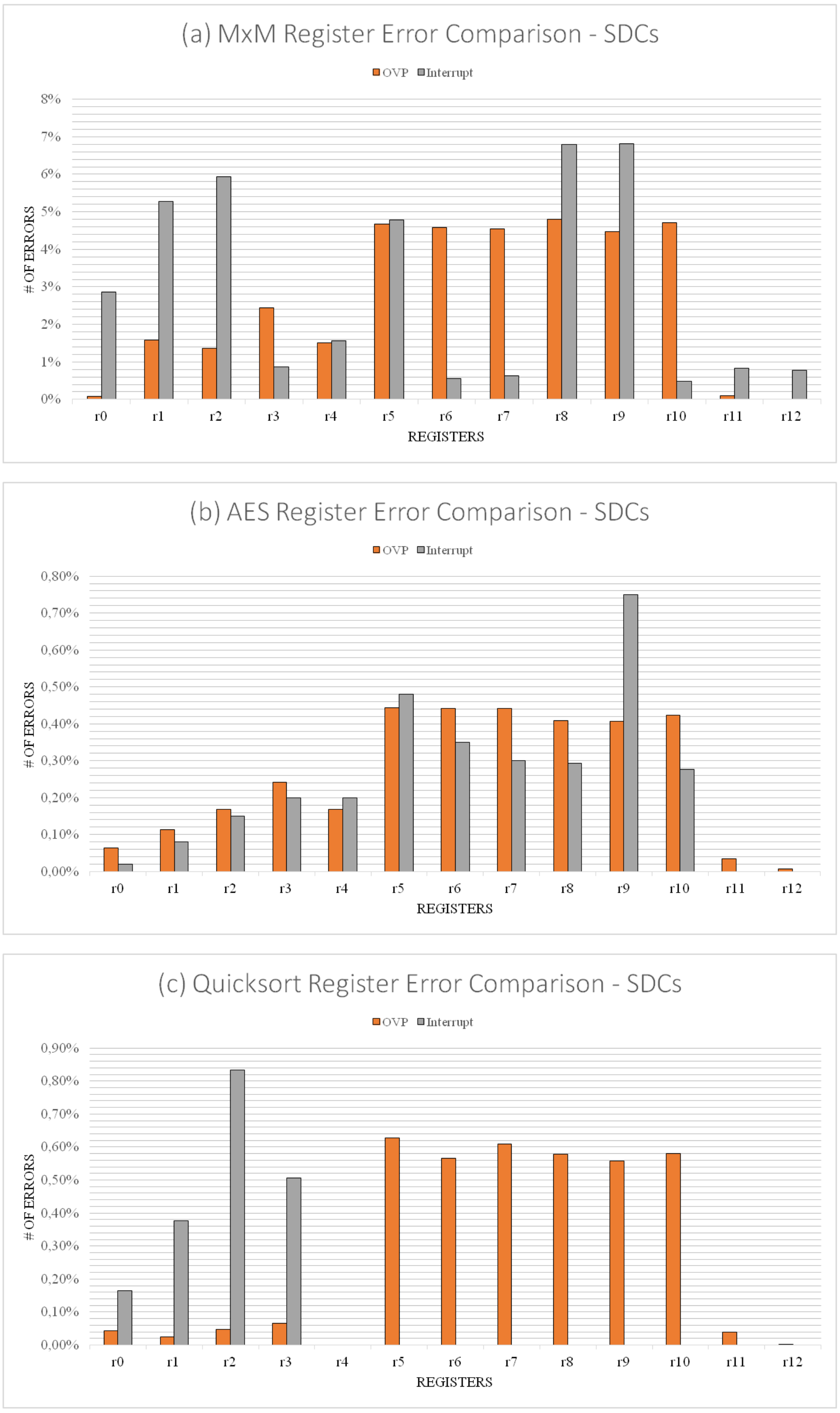


Source: Author

Now, a different behavior can be observed, since there is more deviation in the results for mostly applications even from the total amount of injected faults perspective. MxM still have a coherent result, with 1.2% of difference, but AES and Quicksort algorithms presented a significant deviation, ranging from 7.1% and 8.5%, respectively. Again, if we consider the amount of failures, we'll see interrupt-based platform with 50% more failures in AES application and OVP with approximately 9 times more failures for Quicksort. These discrepancies again reflects the fact that the we have different compilers in both scenarios, resulting in different behaviors in front of the same fault injection pattern.

To enforce this assumption, an analysis at register level gives us a better understanding of the register susceptibility in each platform. Figure 5.3 bellow presents the error rate for all applications in each tested register that produced SDC failures.

Figure 5.3 – SDCs Comparison at Register Level: (a) MxM, (b) AES and (c) Quicksort

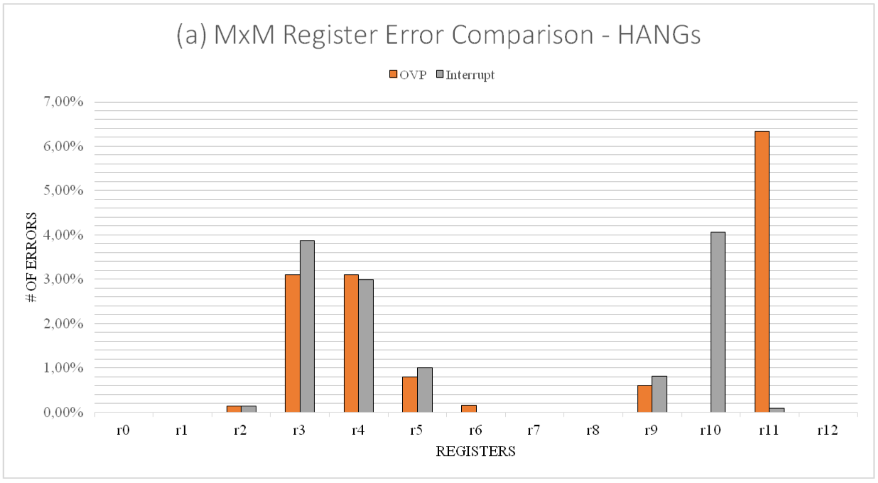


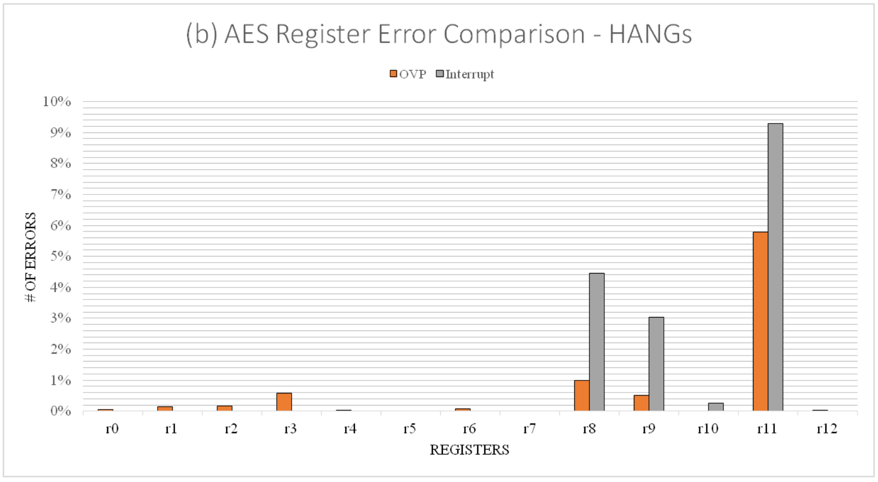
Source: Author

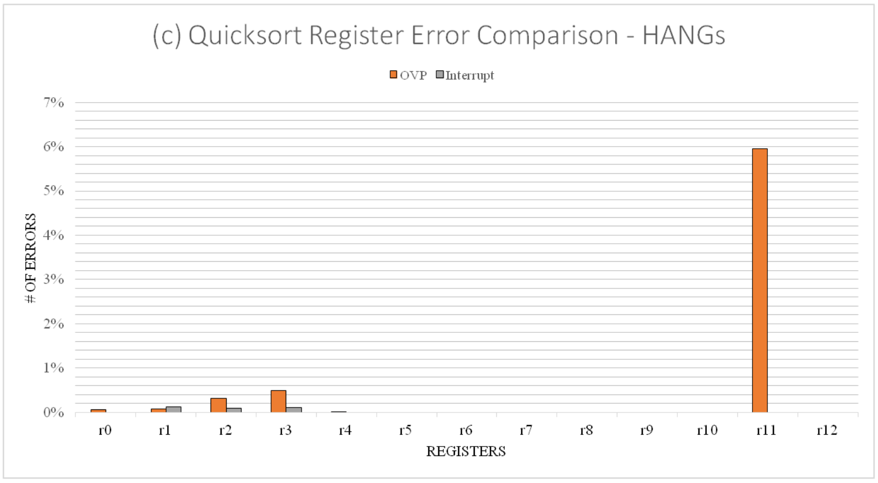
The SDCs results showed to be coherent in the overall error analysis, and this correlation can be confirmed again in Figure 5.4. Although at the first sight many differences can be noticed from register to register in the same application, the average error rate of all applications gives the results discussed earlier. A closer look shows that for MxM, the interrupt-based environment tend to distribute data in R0 to R2 and R8 to R10, while OVP compiler prefers to distribute data in R5 to R10. The same phenomena happen in the AES and Quicksort application, where OVP platform locate data between R5 to R10. In the other hand, interrupt-based platform distributed data along the available registers for AES, while concentrate in R0 to R3 for Quicksort. A main conclusion can be drawn from this analysis, that is, despite the results noticed for overall rate are similar, register allocation can be very different, justifying the relative error previously explained.

In the same way, the tests produced the fault sensibility result for all applications in each tested register that generates HANG failures. Figure 5.4 bellow presents the results.

Figure 5.4 – HANGs Comparison at Register Level: (a) MxM, (b) AES and (c) Quicksort







Source: Author

In the graphs presented in Figure 5.4, a memory element with noticeable error rate means that flow control information is allocated in that register. This allocation, as mentioned, is intrinsically related with the compiler, who maps this information to the available register resources. Depending on the optimization and the compiler algorithm itself, the register mapping will differ, and the same application will allocate different resources, despite the processor being the same. MxM application has a similar error rate along the register bank, and it is noticeable that OVP platform tend to use R11 for control flow operations, while interrupt-based platform prefers R10. Therefore, this regular distribution of both environments produced a similar error rate, as seen in Figure 5.2. The OVP tendency to allocate R11 (frame pointer) is recurrent in AES and Quicksort, while interrupt-based platform compiler find different solutions to resolve the application control flow. For example in AES, where R0 to R7 doesn't generate any failure like in OVP environment, which are locate mostly in R8 to R11. Quicksort application shows a singular situation, where both environments used R0 to R4 to locate the control flow data. However, exceptionally in OVP platform, the frame pointer was abused by the OVP compiler, which results in a high error rate due to the register AVF, which is expected to be high.

Based on the presented results we cannot conclude that the environments have any kind of equivalency. The approach of using different compilers brought a variable that adds uncertainty to the results and unfortunately, it was not possible to perform a new fault campaign, meaning that OVP platform results can only be analyzed in the scope of the simulation environment.

5.2 Environment Comparison

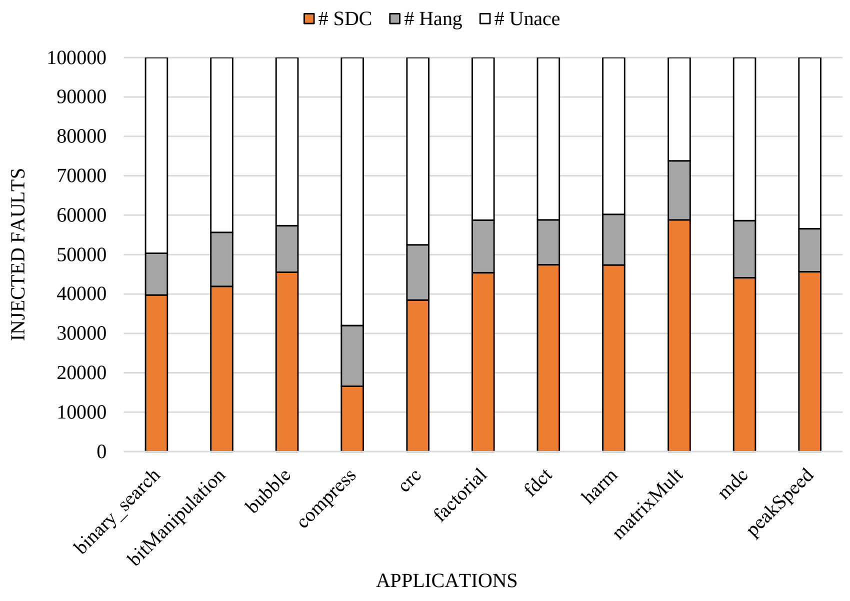
This section will present the comparative results of SEUs injection from the viewpoint of the failures classification in a functional perspective. As mentioned, the failure analysis compares each application running under fault injection with a gold reference to detect errors. A failure would be reported when a mismatch is found in the application control flow or data results.

The platform OVPSim is able to report an UNACE event, when no error is detected and the result memory is identical to golden execution, a HANG, meaning the application or system hanged or crashed and a SDC, informing that application has finished but the data memory mismatch with the golden memory. For SDC failures, note that not only the application results stored in memory are compared but the entire memory space. This is important to analyze any latent error that may occur in the memory.

Figure 5.5 shows the number of UNACE events (white columns), HANG (grey columns) and SDC (orange columns) failures for each bare metal application. The campaign of 100,000 faults in each application reached an average error rate slightly above 50 % of the total number of injected faults. The number of HANGs ranges from 10% to 15%, showing that regardless of the application behavior, it tends to crash at a relatively constant rate. This shows a good performance considering that the injections are done in registers in their active storage cycle. The low rate of HANGs is obtained due to the WCET benchmark characteristics in the bare metal environment, since they were designed to worst case execution time analysis, they are all single data path algorithms, with a few number of control code and mostly data intensive, meaning that the SEUs are less prone to cause a loss of flow.

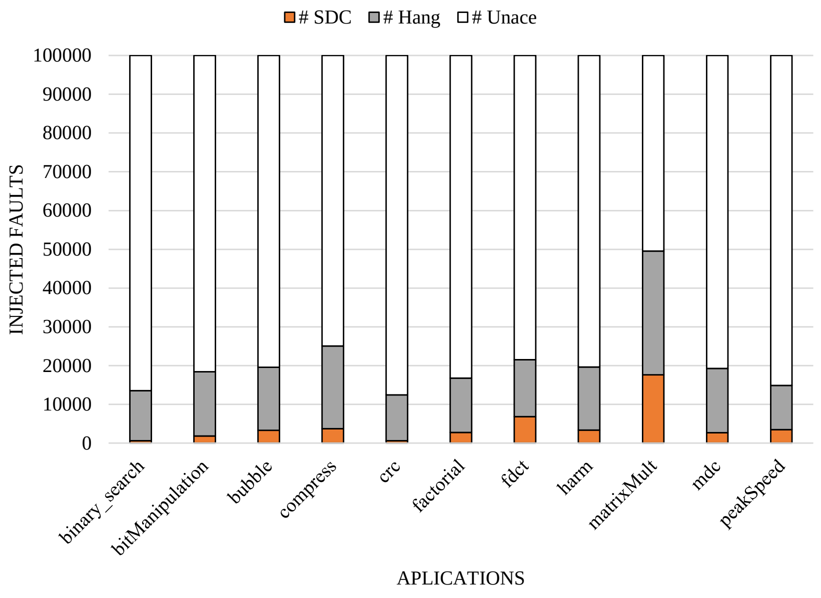
In the same Figure, SDC failures in most benchmark cases range from 40 % to 47 % of total injected faults. A substantial number of failures were detected, and again it is a result of the benchmark composition, strongly based on data manipulation, which constantly allocates mostly registers for data storing. The variation on fault susceptibility is mainly because the application particularities. It is important to remark, for example, that the compress application shows to be less susceptible to SDC failures with 16.6 %, due to the extensively use of external memory and less necessity of spreading data over registers. The matrixMult in the other hand, shows a SDC error rate of 58.8 % because it abuses of nested loops, which is highly benefited from the register renaming mechanism that does loop unrolling, exposing more the data in the register file.

Figure 5.5 – Bare Metal Error Classification



Source: Author

Figure 5.6 – Linux OS Error Classification



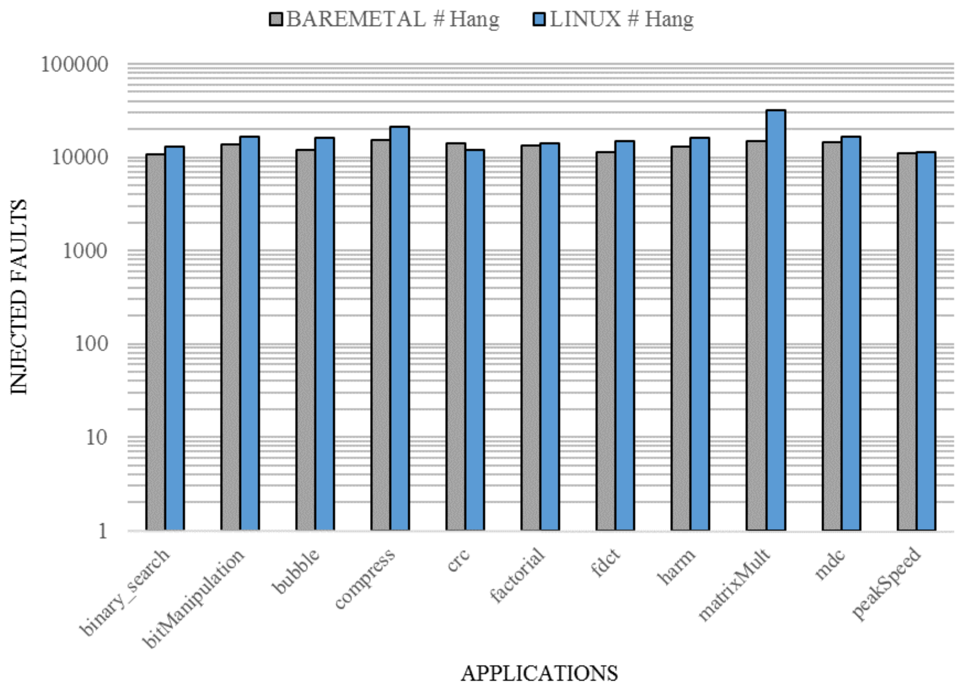
Source: Author

As mentioned, the Linux kernel has the capacity to manage errors before they manifest due to memory virtualization and is able to pre evaluate the execution of an operation. The impact of this treatment is presented in Figure 5.6. In Linux environment, the majority of the applications have an error rate ranging from 12 % to 25 %. HANGs in this case can also happens in OS kernel as it is exposed to faults during application execution, and so they prove to have a regular distribution, from 12 % to 18 % of total injected faults. It is important to mention that in this category, we also included hangs that the operating system was able to signalize. However, two applications in this group, compress and matrixMult, draws attention by having 21 % and 32 % hangs, respectively, this may be due to the fact of the register allocation for Linux and the application that exposes more frequently control flow registers due to resource sharing.

In the SDC category, most applications in Figure 5.6 stayed under 6 % of total injected faults. It happens because in Linux the application is statistically less likely to have a fault, since the application shares the register file resources with operating system, i.e., part of the register file is allocated to Linux and part for the application, so the chances to hit a register file that causes a SDC in the application is reduced. As expected, matrixMult still has more SDC failures than the average due to hardware loop unrolling.

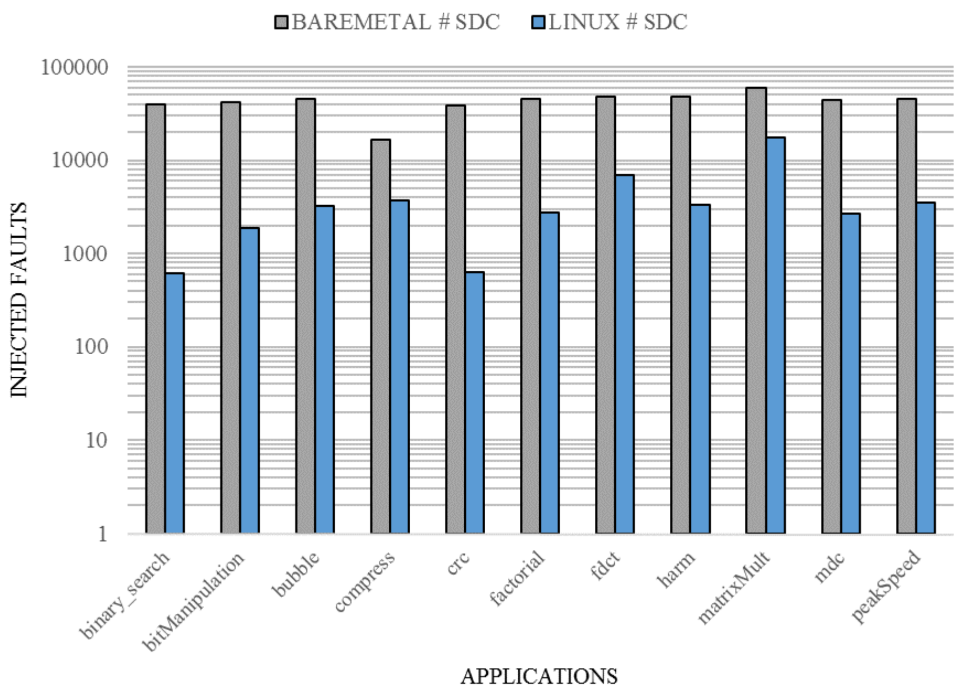
To better understand the presented results, Figure 5.7 and Figure 5.8 depicts comparison graphics with HANG and SDC results, respectively.

Figure 5.7 – HANG Comparisons Between Environments



Source: Author

Figure 5.8 – SDC Comparisons Between Environments



Source: Author

Figure 5.7 shows that bare metal and Linux environments present a similar behavior considering HANG type of failure. One of the reasons is the fact that this type of failure is also manifested due to violations in the execution flow that causes an invalid state of the processor, not depending on the characteristics of the application itself. Also, the fact that ARM Cortex-A9 has a build in register renaming mechanism makes control variables to be equally distributed between registers, causing control variables stored in registers to be spread and consequently equally prone to errors. However, Linux OS have shown on average a slightly higher rate comparing with bare metal, which make sense, since resource sharing with the kernel increase the control flow exposition to SEUs.

However, in neither case we observed such a significant impact on HANGs rate produced by SEUs in registers. This result can be explained by the applications morphology and how the registers are allocated during execution. One of the major factors for HANGs appearance are the branch instruction, there are always conditional or compulsory jumps in the code. Loops in code generate these instructions, but they use few records to allocate its controlling while it is operating. As most benchmarks have shown few of these operations throughout the execution, it is plausible that HANG error rate is low. However, nested loops start loops inside another loop and take up more registers, making the control instructions more susceptible to SEUs, as in matrixMult application. Another factor that causes an increase in the number HANGs is the code needed to access external memory, which increases the number of control instructions and thus HANGs susceptibility, as we noted in the case of the compress algorithm, which also showed a higher HANG rate.

In Figure 5.8, it is noticed a considerable difference in SDCs detected in bare metal environment than in Linux. Note that the graph is in logarithmic scale; otherwise, it would be difficult to see the difference. The high rate of errors in the bare metal environment takes place due to WCET benchmark characteristics again, which in addition to limiting the algorithms to single path, they are data manipulation intensive, such as bit manipulation, matrixes and floating points. These characteristics causes that mostly instructions in the application manipulate data that will be part of the result. Of course, a large concentration of such instructions greatly increases the chances of SEUs, causing SDC failures.

On the other hand, we can obtain a significant improvement on using Linux OS, since SDC rates are much lower, as noted in Figure 5.8. In the operating system case there is still a lot of data manipulation, therefore this is not the reason that explains the difference seen in the environments. In fact, two types of masking effects can be observed in presence of OS. One is the exchange of context by sharing hardware resources with application and Linux kernel, making the application less likely to be hit by a fault, since the total run time increases. The other is the increasing number of accesses to the external memory, which constantly moves data to and from the registers whenever a thread takes precedence, causing the data to resides less time inside the processor.

5.2.1 Register Fault Susceptibility Analysis

The Fault Injector module was improved to allow detailed reports on the error coverage by register. By doing this, it was possible to identify critical points in the applications and to determine how compiler allocates the register bank. The information can also be used for programmers to improve software reliability.

The results presented in Table 5.1 are expressed in terms of failure percentages classified between HANGs and SDCs failures per register. Among the benchmark, a set of 4 applications was subjected to this analysis.

Table 5.1 – Manifested Failures by Register

		compress		fdct		harm		mxm	
pc	SDC	Bare metal	1.27%	Bare metal	26.87%	Bare metal	3.23%	Bare metal	15.00%
		Linux	0.00%	Linux	0.00%	Linux	2.86%	Linux	3.51%
	Hang	Bare metal	68.35%	Bare metal	67.16%	Bare metal	79.03%	Bare metal	73.33%
		Linux	80.00%	Linux	83.87%	Linux	82.86%	Linux	80.70%
sp	SDC	Bare metal	8.96%	Bare metal	0.00%	Bare metal	74.29%	Bare metal	1.49%
		Linux	7.14%	Linux	2.94%	Linux	30.56%	Linux	4.62%
	Hang	Bare metal	2.99%	Bare metal	4.05%	Bare metal	11.43%	Bare metal	1.49%
		Linux	3.57%	Linux	9.32%	Linux	50.00%	Linux	1.54%
r0	SDC	Bare metal	0.00%	Bare metal	0.00%	Bare metal	7.58%	Bare metal	56.52%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	47.83%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	10.61%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	3.03%	Linux	0.00%
r1	SDC	Bare metal	1.54%	Bare metal	12.79%	Bare metal	0.00%	Bare metal	60.32%
		Linux	0.00%	Linux	2.94%	Linux	0.00%	Linux	33.33%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	1.59%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	33.33%
r2	SDC	Bare metal	1.72%	Bare metal	29.03%	Bare metal	0.00%	Bare metal	37.50%
		Linux	0.00%	Linux	35.71%	Linux	0.00%	Linux	13.04%
	Hang	Bare metal	1.72%	Bare metal	1.61%	Bare metal	1.19%	Bare metal	0.00%
		Linux	0.00%	Linux	10.71%	Linux	5.41%	Linux	27.54%
r3	SDC	Bare metal	18.57%	Bare metal	68.12%	Bare metal	1.75%	Bare metal	60.87%
		Linux	2.86%	Linux	44.12%	Linux	0.00%	Linux	29.82%
	Hang	Bare metal	1.43%	Bare metal	2.90%	Bare metal	3.51%	Bare metal	0.00%
		Linux	34.29%	Linux	38.24%	Linux	7.41%	Linux	29.82%
r4	SDC	Bare metal	76.27%	Bare metal	0.00%	Bare metal	8.20%	Bare metal	76.19%
		Linux	17.95%	Linux	100.00%	Linux	0.00%	Linux	25.81%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	15.87%
		Linux	43.59%	Linux	0.00%	Linux	0.00%	Linux	69.35%
r5	SDC	Bare metal	2.99%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	93.75%
		Linux	0.00%	Linux	100.00%	Linux	0.00%	Linux	30.00%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	6.25%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	68.57%
r6	SDC	Bare metal	15.15%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	84.48%
		Linux	14.71%	Linux	100.00%	Linux	0.00%	Linux	43.55%
	Hang	Bare metal	63.64%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	10.34%
		Linux	64.71%	Linux	0.00%	Linux	0.00%	Linux	51.61%
r7	SDC	Bare metal	3.28%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	100.00%
		Linux	3.70%	Linux	100.00%	Linux	0.00%	Linux	1.45%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	0.00%
r8	SDC	Bare metal	2.74%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	100.00%

		Linux	4.65%	Linux	100.00%	Linux	0.00%	Linux	5.80%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	0.00%
r9	SDC	Bare metal	100.00%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	100.00%
		Linux	0.00%	Linux	100.00%	Linux	0.00%	Linux	4.17%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	0.00%
r10	SDC	Bare metal	1.49%	Bare metal	100.00%	Bare metal	100.00%	Bare metal	100.00%
		Linux	3.57%	Linux	100.00%	Linux	0.00%	Linux	2.94%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	0.00%
r11	SDC	Bare metal	6.06%	Bare metal	0.00%	Bare metal	4.41%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	1.37%
	Hang	Bare metal	93.94%	Bare metal	100.00%	Bare metal	92.65%	Bare metal	100.00%
		Linux	96.88%	Linux	100.00%	Linux	95.35%	Linux	98.63%
r12	SDC	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	34.29%
		Linux	0.00%	Linux	100.00%	Linux	0.00%	Linux	25.40%
	Hang	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%	Bare metal	0.00%
		Linux	0.00%	Linux	0.00%	Linux	0.00%	Linux	0.00%

Source: Author

As expected, the registers program counter, stack pointer and r11 (frame pointer) have the highest rate of hangs. The first occurs because it keeps the sequence of instructions and code jumps and tends to an error rate close to 100%. The second and third pointers are the callback functions that compute the locations in memory for both arguments as well as local variables, which also tend to have a high error rate. A noticeable amount of SDC failures can be seen in these registers, which means that these registers are used to compute data offsets that result in result miscalculation.

It is clearly shown that starting from r0 to r10 and r12, the HANG are near to 0 % in all applications. It happens as this registers are almost used exclusively for data storing, which result in SDC failures, as depicted in Table 5.1 . The exceptions for this are applications that require a more complex code flow, like compress and matrix multiplication, that uses external memory accesses and nested loops, respectively. This code behavior causes the compiler to allocate more registers, in this case r0 to r6, for control, provoking HANGs. The same registers, specially r5 to r10 have a high SDC rate (around 100 %) in both bare metal and Linux code, meaning that they are preferable by the compiler to store temporary and result data. This finding can drive developers to distribute registers better along the code or optimizes code, in order to reduce the vulnerability of this registers.

It is also noticeable that Linux has a better performance over bare metal for SDCs failures, but in contrast it has a worst HANG error rate in same registers. It is expected that Linux change software context from kernel to the application, leading to a higher data masking, since it is moved to and from external memory. However, doing this the kernel code is also exposed, and since it has high vulnerability for being the operating system core, it consequently causes the software to crash.

From a broader perspective, it is possible to say that in addition to the characteristics of Cortex-A9 processor and the system in which it is enclosed, the reliability of an embedded processor depends strongly on the application running on it. Otherwise, it would be possible to observe regularity in the amount of failures for all applications, which is not true, as they tend to vary in a coherent manner to the behavior described in Section 4.2. As already shown by [65], it´s perceived that the architecture vulnerability factor (AVF) estimates only the processor reliability, but for a system perspective the program vulnerability factors (PVF) is equally important, since soft-error susceptibility can be aggravated by this aspect. This perspective will be discussed later in this chapter.

This dependence is expected and has been observed previously in [64], in which a uniform set of faults were used, i.e., only Single Event Upset faults were injected into each of the studied processor registers in the same quantity, showing that despite the regularity of architecture and standard injections, the behavior of the system varies dramatically. Assuming this behavior, it is also expected different results being observed in the same application when it runs on bare metal or on an operating system. Although the algorithm is the same, the compiler is not, since Linux is cross-compiled for the ARM architecture, but the application that runs it uses the default parameters of the GCC compiler. In fact, and due to this, the instructions running in both environments are not completely identical, allied to this, resources sharing that exists in the operating system adds a level of complexity and a degree of fault masking.

5.3 Silent Data Corruption Classification

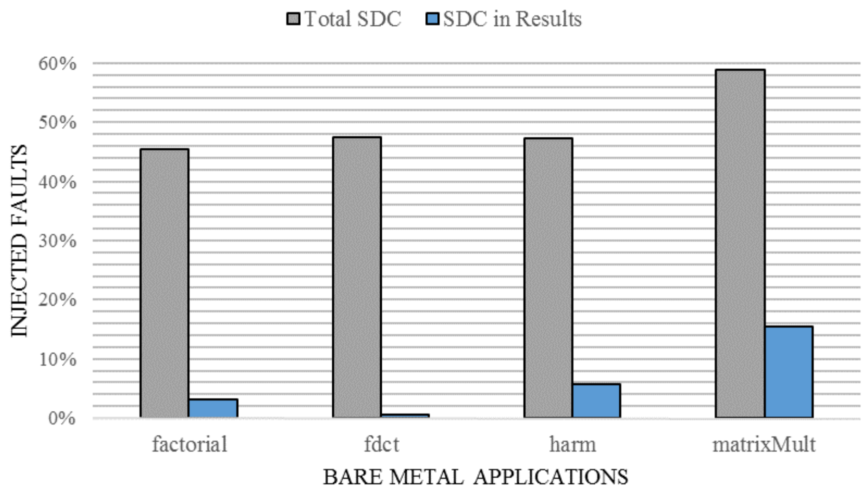
Silent Data Corruption is a critical failure type that can lead to disastrous events in an embedded system. SDC refers to operations returning a wrong result without any indication (for all intents and purposes the result is valid, yet is actually not correct). SDC is especially difficult to mitigate in a safety critical system, as to do so requires that every result be checked by both repeating the operation and checking that the result matches, or by including some kind of algorithmic check analysis comparing the final memory, separating the effects of latent errors from the ones that generates wrong results. This Section proposes to implement this separation, identifying the relationship between total number of failures and latent errors.

As mentioned, the OVPSim Fault Injector Module detects SDC failures by comparing application final memory with a reference model from an execution without faults. The process compares the entire memory that includes instructions (that are not affected by faults in this work), software data, temporary data and the result. By doing this, it is not possible to identify in which location the SDC happen. In the real world, it means that an error in the results will return a wrong data to the system in the current execution, but an error in another part of the memory, called latent error, can manifest a failure in a future code execution, continuing propagating its error every time the algorithm is executed. However, a latent error can be easily mitigated by reloading application content to the main memory, for example from a flash device.

The OVPSim-FIM was modified to identify the memory region where the result was placed after execution, being possible to identify the discussed types of SDC failures.

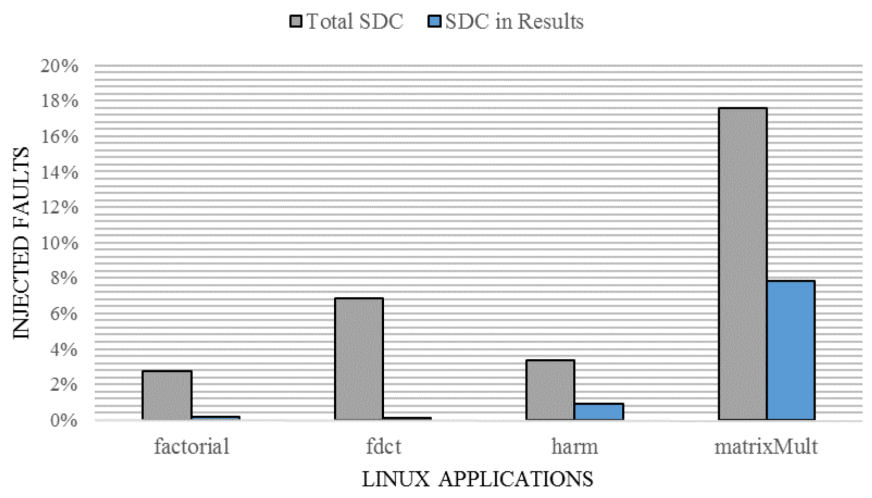
Figure 5.9 and Figure 5.10 depicts comparison graphics between total number of SDC and in results for bare metal and Linux environments, respectively.

Figure 5.9 – Bare Metal Total SDC vs SDC Only in Results



Source: Author

Figure 5.10 – Linux Total SDC vs. SDC Only in Results



Source: Author

Only four applications were chosen in this analysis, since each fault injection campaign takes about 12 hours in Linux setup. For both environments and all applications, it was possible to identify that the number of SDCs present only in results is much smaller than total number, meaning that the differences in the final memory are likely to happen outside the result region. This phenomenon is a probabilistic matter, since during application execution a large number of data is moved in and out the external memory and not necessarily is part of results. Another reason is fault injection timing, as during algorithm execution information that was already used by the application can still in the register until it is needed again, thus a fault that happens in this moment will not lead to a failure, but if the value is written back to the memory, the SDC failure will be noticed.

5.4 Linux OS and Error Mitigation

Error reports triggered by Linux kernel signals comprise a specific set. As detected and reported by the operating system they can be handled by functions added in the application code, allowing correction, discarding of unreliable results or even restart of the application. As mentioned in Section 4.2, each application was modified by including signal handling functions with no corrective action, but with communication with the OVPSim-FIM to provide feedback from the inside the application, feature that is not supported by the simulator itself.

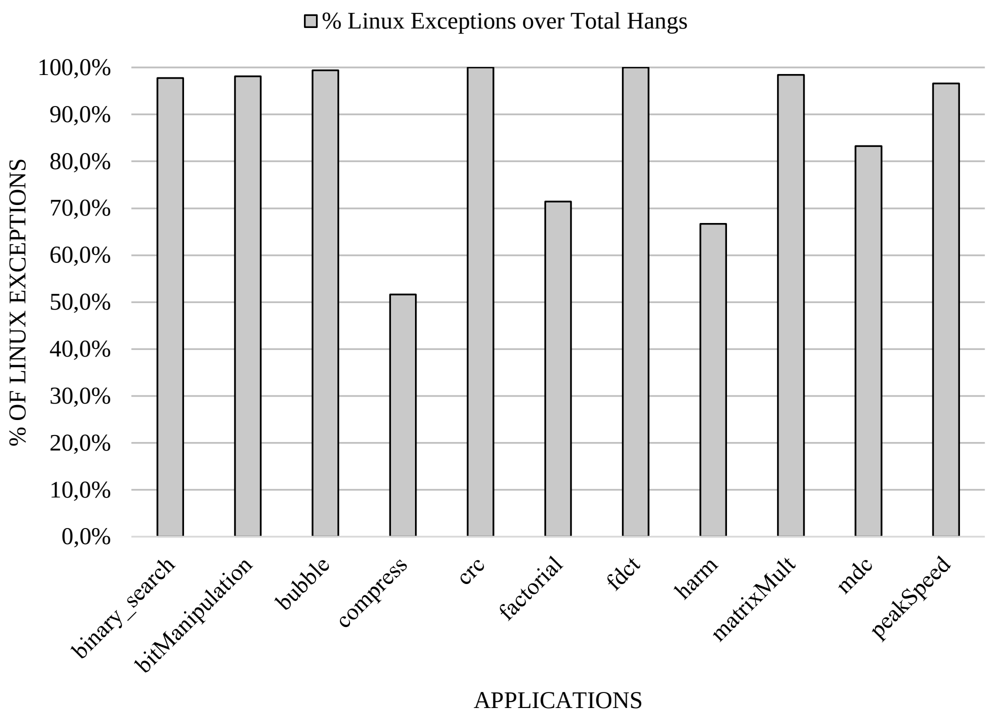
The fault injection campaigns have shown that between operating system available signals, Linux was able to detect only segmentation fault, illegal access and floating point exception violations.

Exceptions detected by Linux can be considered either HANGs or SDCs. It is a SDC because the Operation System does not stop working as it can recover for these occurrences, but the application result is wrong or even does not exists in memory. It is also a HANG because the application that is running under Linux crashed. This reason justifies the need for identifying OS exceptions between results presented.

The Linux signals was treated as HANGs before, since strictly they are from the application perspective. However, the OS provide the ability to treat them by the application, even if it is not being done here, and therefore should be counted apart.

Figure 5.11 depicts Linux Hang detection rate.

Figure 5.11 – Linux Exception Signal Rate Over Total Detected HANGs



Source: Author

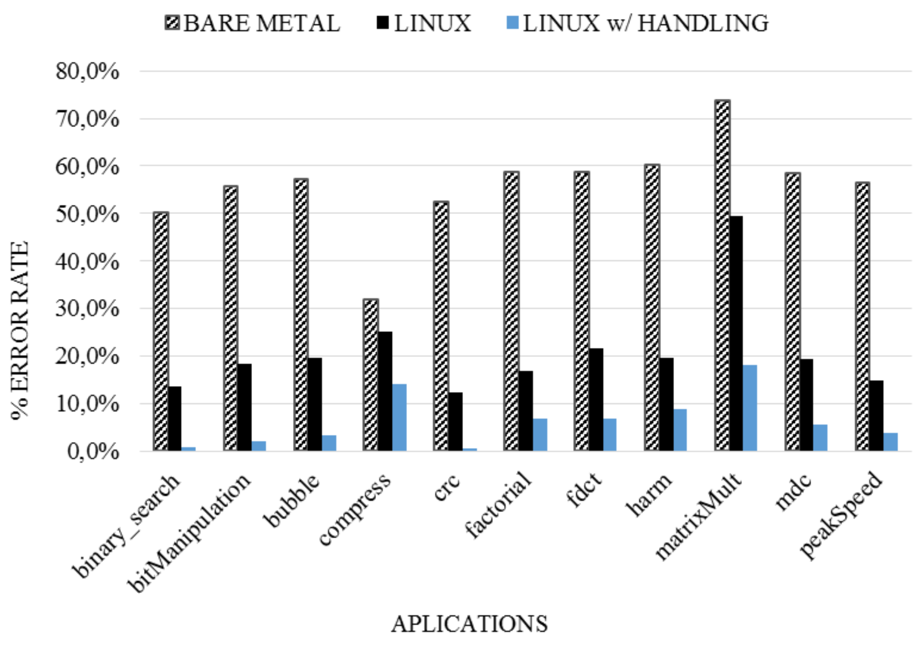
Figure 5.11 correlate the rate of identified soft errors by the operating system among total number of manifested HANG errors. Surprisingly, Linux shows to have a high rate of HANG perception, reaching 100% of detection in the case of the fdct application. As expected, analyzing the reports in details we found that around 90% of detected failures are Segmentation Faults, when the application tries to address a non-allocated memory area, making this more perceptive in applications that recursively access external data, like bit manipulation and matrix multiplication. In a regular distribution between applications, Illegal Access signal happened when the processor attempts to execute an illegal, malformed, unknown or privileged instruction, what's is an unsurprising behavior since we are modifying randomly internal registers values. Harm, fdct and mdc are the only applications with floating point operations in the benchmark, and so Linux triggered Floating Point Exception signal.

5.5 Overall Error Rate

Finally, considering the results presented, it is possible to determine the overall error rate comparison between bare metal and Linux applications, which is the sum of SDC and HANGs.

Figure 5.12 depicts the results.

Figure 5.12 – Linux Exception Signal Rate Over Total Detected HANGs



Source: Author

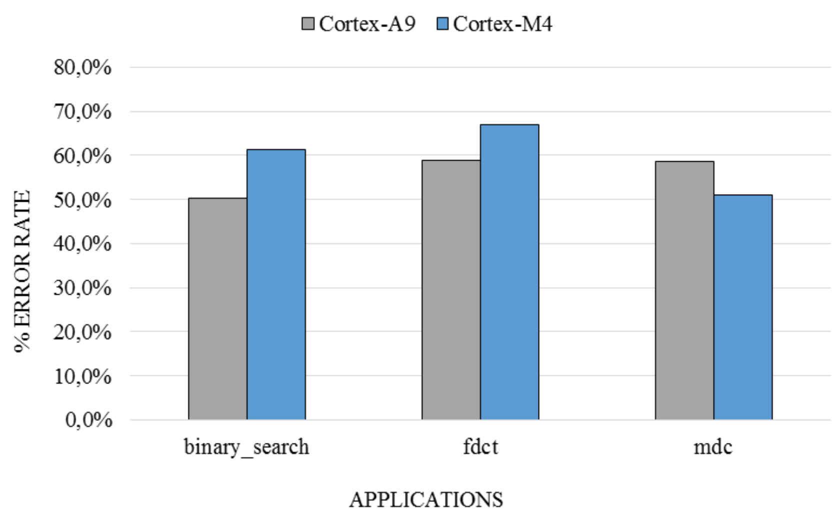
Results show clearly that all applications running in bare metal environment are more prone to failures in general than under Linux OS when faults are injected in the register file. It is known that ARM Cortex-A9 is a complex out-of-order architecture with many more internal registers. For a complete evaluation, these registers must be taken into account during fault injection, which unfortunately is not supported by the used simulator. However, the processor observability allows the architectural vulnerability factor analysis, which makes this evaluation meaningful considering the ability to examine in an isolated manner the register file of a large-scale commercial processor.

Taken the bare metal environment result in Figure 5.13 as an example, we can say that architectural vulnerability factor ranges from 32% to 74 %. This results show that each tested register, individually, should be taken into account to determine AVF, and what is seen is the average result. Yet, not only the register itself, but to generate a failure, the register content must be relevant in the moment of the fault. This means that depending on the register allocation, the error rate can vary and, consequently, architectural vulnerability factor varies according to the application.

This statement articulates the error rate result in a more complex perspective, since it is not only the hardware architecture the single factor taking into account, but also software architecture. As the software has its own structure and workflow, it is reasonable to talk in terms of application vulnerability factor. In fact, the presented error rates are a function of both factors, showing that a program has an inherent fault masking.

As a case study, a simple experiment is conduced to enforce this analysis, as depicted in Figure 5.13.

Figure 5.13 – Error Rate of Three Applications Running on ARM Cortex-A9 and Cortex-M4



Source: Author

Figure 5.13 shows that architectural/application vulnerability factor of a register file when running three different applications on two processors, the ARM Cortex-A9 and ARM Cortex-M4, both available for Imperas simulator and with identical register file structure, but different overall architecture. To sustain what was said, Figure 5.13 reveal that the vulnerability factor depends on both workload and hardware configuration, since results for the same application are different between processors, but also results for the same processor are different between applications.

Figure 5.13 shows clearly that applications running in bare metal environment are more prone to failures in general than when running in Linux OS version when faults are injected in the register file. Additionally, it depicts the total Linux environment error rate and the error rate excluding exceptions detected by the kernel, in black and blue respectively. Therefore, according to what was discussed in Section 5.4, in case of the Operating System, exceptions are handled, and they not produce HANGs or SDCs, the error rate of a Linux environment can be even better.

6 CONCLUSIONS

The work analyzed in a comparative approach the susceptibility of several applications under SEUs in two environments: bare metal and on the top of the Linux operating system. It demonstrates the use of an embedded system simulation platform that uses microprocessors models with architectural elements and functionality equivalent to the physical device, the OVPSim platform, in order to perform the sensitivity analysis. Starting from the feasibility study of the simulated environment, target architecture choice and fault model, the OVPSim-FIM fault injector was used and adapted to comply with the observability needs in this study. The fault injector was able to inject bit-flips in the register file of the ARM Cortex-A9 processor followed by the failure classification collected by the platform with report features that includes Linux exception signals. The methodology promotes statistical results separating the main groups of failures between control flow (HANGs), silent data corruption and Linux captured errors.

The studied OVP platform has shown great versatility for creation and system integration as it provides numerous validated processors models supported by the manufacturers. The main factor that led to the choice of this platform was the availability of an API for control of the entire architecture, allowing the fault injector strategy. The microprocessor ARM Cortex-A9 was chosen because of its degree of fidelity to the original architecture, including specific functions of the physical architecture and access to all the registers visible by the application. Furthermore, since it is a large-scale industrial and commercial application processor, the same test can be replicated in the FPGA embedded version from Altera and Xilinx manufacturers in order to establish a correlation between simulated tests and practice.

The goal was achieved by performing an analysis in terms of failure and not fault domain, since location is not the focus, but the variety of failures that can arise in similar faults. By doing this, we have shown that application running bare metal increase significantly the overall error rate for all 11 benchmarks that comprises a diverse scenario of data utilization. Despite this, considering different groups of failures, we found that the operating system has considerably less silent data corruption than bare metal environment, having the majority of failures located in HANG failure group. By saying this, it is possible to conclude that in our experiment the Linux OS environment tend to mask more frequently soft errors, while bare metal is more prone to corrupt the data of the application.

Additionally, the fault injector was modified in order to separate SDC failures in the results from the ones in other regions of the external memory. We observed that mostly of these errors are outside the results in both Linux and bare metal environments, which shows an opportunity to diminish these errors by restarting the application at each run.

The comparison with an interrupt-based environment developed by Lins [68], was an attempt to correlate the processor model with the real architecture in a fault injection method equivalent to the one used with OVP. Unfortunately, we were able to give credit to the results in the way they the tests were performed, showing us an important perspective of this work, that is the dependency on the compiler. The strategy developed in this work that is also presented in [70], was compared with the results in [32], where a correlation with an environment under radiation was analyzed. A different behavior was observed, since in [32] SDC failures in Linux was almost the same as in bare metal. What happens, and the paper itself points out, is that it happens mainly because the neutron flux also injected faults in the cache, where the application data is always exposed, leading in data corruption even before the content reaches the register file. Another reason is that in [32] the application is restarted continuously without restarting the operating system, a process where bit-flips that are not expressed in a particular execution may be manifested in the future.

Given the availability of the processor model, the only conclusion from the architectural perspective was the Architectural Vulnerability Factor of the register file. From the results, it is possible to identify that the AVF varies by register and not with the register file as a single entity. However, tests showed a variability in the AVF for each used benchmark, showing that the more correct is to determine the factor taking into consideration the application, establishing an Application Vulnerability Factor. Both measures provide a better understanding of the soft errors effects in embedded systems, especially a better observability of the application behavior, revealing their critical points and giving room for improvement.

An additional category of failure was analyzed by separating signals triggered by Linux kernel, and this approach found that the contribution of this group could be up to 100% of the total number of HANGs. Since the kernel reports to the application the signal, it is possible to implement signals handlers in the code in order to deal with the exception. The main contribution of this analysis is that, besides the natural SDC hardening that the Operation System promotes by sharing hardware resources with the application, this work shows that mitigation of soft error effects can be even better if corrective actions are adopted.

As a future work, some analysis limitations can be overcome, such as the impact on the results of using different compilers between environments and the running time factor that also influence the comparative results. A deeper analysis on the Linux architecture can also contribute with a better interpretation of the results concerning OS resource sharing mechanisms along with the investigation of the failure classification considering multiple applications running together. The failure distribution analysis will then help the elaboration of tolerant techniques focused on these aspects. This approach can be greatly supported by the failures classified by register, which allows depth analysis on how the application and compiler uses the register file, providing means to develop dedicated fault tolerance systems, according to each application's necessities and sensitivity to faults. Another proposal includes sharing processor and operating system resources by running multiple applications at the same time and evaluating soft errors masking effects with and without multicore platforms. Finally, the modeled embedded system can be better detailed to provide more fidelity compared with other discussed studies, which also uses the Xilinx Zynq SoC FPGAS, but also includes external and cache memories.

REFERENCES

[1] A. Benso and P. Prinetto, Eds., Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, vol. 23. Boston: Kluwer Academic Publishers, 2004.

[2] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” Computer, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[3] P. Vanhauwaert, R. Leveugle, and P. Roche, “Reduced Instrumentation and Optimized Fault Injection Control for Dependability Analysis,” in 2006 IFIP International Conference on Very Large Scale Integration, 2006, pp. 391–396.

[4] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, “A model of soft error effects in generic IP processors,” in 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005, 2005, pp. 334–342.

[5] T. R. Oldham and F. B. McLean, “Total ionizing dose effects in MOS oxides and devices,” IEEE Transactions on Nuclear Science, vol. 50, no. 3, pp. 483–499, Jun. 2003.

[6] P. Koopman, Eushiuan Tran, and G. Hendrey, "Toward Middleware Fault Injection for Automotive Networks," Fault Tolerant Computing Symposium, June, 1998, pp. 78 - 79.

[7] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, “Fault injection and dependability evaluation of fault-tolerant systems,” IEEE Transactions on Computers, vol. 42, no. 8, pp. 913–923, Aug. 1993.

[8] A. Benso, M. Rebaudengo, M. S. Reorda, and P. L. Civera, “An integrated HW and SW fault injection environment for real-time systems,” in 1998 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings, 1998, pp. 117–122.

[9] P. P. A. Benso, “EXFI: a low-cost fault injection system for embedded microprocessor-based boards,” ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 3, no. 4, pp. 626–634, 1998.

[10] Imperas. Open Virtual Platforms (OVP), 2014. http://www.ovpworld.org/, [acessado em 20 de Novembro de 2014].

[11] D. Sanchez and C. Kozyrakis, “ZSim: fast and accurate microarchitectural simulation of thousand-core systems,” ACM SIGARCH Computer Architecture News, vol. 41, no. 3, p. 475, Jul. 2013.

[12] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, “A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability,” in 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015, pp. 211–214.

[13] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau, “Analysis of the effects of real and injected software faults: Linux as a case study,” in 2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings, 2002, pp. 51–58.

[14] T. Noergaard, Embedded systems architecture: a comprehensive guide for engineers and programmers. Amsterdam ; Boston: Elsevier/Newnes, 2005.

[15] Mit and navigating the path to the moon. http://web.mit.edu/aeroastro/news/magazine/aeroastro6/mit-apollo.html. [Accessed: 11-Dec-2015].

[16] A. S. Tanenbaum, Sistemas Operacionais Modernos, Edição: 3[a]. Rio de Janeiro (RJ): Pearson, 2009.

[17] A. Sandstedt, Implementation and analysis of a virtual platform based on an embedded system. Linköping University, Department of Electrical Engineering, Computer Engineering 2014.

[18] Unknown author. NS7520 Hardware Reference. Net sillicon, 2003. Cited on pages 6, 7, and 58.

[19] D. Liu, Embedded DSP processor design: application specific instruction set processors. Amsterdam ; Boston: Morgan Kaufmann/Elsevier, 2008.

[20] F. H. Schmidt, "Fault tolerant design implementation on radiation hardened by design SRAM-Based FPGAs," Thesis, Massachusetts Institute of Technology, 2013.

[21] B. W. Johnson, "Fault-Tolerant Microprocessor-Based Systems," IEEE Micro, vol. 4, no. 6, pp. 6–21, Dec. 1984.

[22] A. Avizienis, "Toward systematic design of fault-tolerant systems," Computer, vol. 30, no. 4, pp. 51–58, Apr. 1997.

[23] Ray Ladbury. Osiris-rex radiation hardness assurance plan. Technical Report OSIRIS-REx-PLAN-0014, NASA Goddard Space Flight Center, Janeiro, 2012, pg. 32, 33, 35, 36.

[24] R. Oldham and F.B. McLean. Total ionizing dose e ects in MOS oxides and devices. IEEE Transactions on Nuclear Science, 50(3):483{499, June 2003.

[25] C. Claeys and E. Simoen, Radiation Effects in Advanced Semiconductor Materials and Devices. Springer Science & Business Media, 2002.

[26] Guenzer. C. S.; Wolicki, E. A.; Allas, R. G. "Single event upset of dynamic RAMs by neutrons and protons". IEEE Transactions on Nuclear Science, Dec. 1979.

[27] M. M. McCormack, "Trade study and application of symbiotic software and hardware fault-tolerance on a microcontroller-based avionics system," Thesis, Massachusetts Institute of Technology, 2011.

[28] J. R. Schwank, M. R. Shaneyfelt, and P. E. Dodd, "Radiation Hardness Assurance Testing of Microelectronic Devices and Integrated Circuits: Radiation Environments, Physical Mechanisms, and Foundations for Hardness Assurance," IEEE Transactions on Nuclear Science, vol. 60, no. 3, pp. 2074–2100, Jun. 2013.

[29] M. Caffrey, K. Morgan, D. Roussel-Dupre, S. Robinson, A. Nelson, A. Salazar, M. Wirthlin, W. Howes, and D. Richins, "On-Orbit Flight Results from the Reconfigurable Cibola Flight Experiment Satellite (CFESat)," in 17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009. FCCM '09, 2009, pp. 3–10.

[30] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, 2003, p. 29–.

[31] D. Alexandrescu, L. Sterpone, and C. Lopez-Ongil, “Fault injection and fault tolerance methodologies for assessing device robustness and mitigating against ionizing radiation,” in Test Symposium (ETS), 2014 19th IEEE European, 2014, pp. 1–6.

[32] T. Santini, P. Rech, L. Carro, and F. Rech Wagner, “Exploiting cache conflicts to reduce radiation sensitivity of operating systems on embedded systems,” in 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015, pp. 49–58.

[33] W. Gu, Z. Kalbarczyk, K. Iyer, and Z. Yang, “Characterization of linux kernel behavior under errors,” in 2003 International Conference on Dependable Systems and Networks, 2003. Proceedings, 2003, pp. 459–468.

[34] M. Nicolaidis, Ed., Soft Errors in Modern Electronic Systems, vol. 41. Boston, MA: Springer US, 2011, p. 335.

[35] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: a methodology and some applications,” IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[36] R. Velazco, P. Fouillat, and R. Reis, Eds., Radiation Effects on Embedded Systems. Dordrecht: Springer Netherlands, 2007.

[37] C. Slayman, “JEDEC Standards on Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray Induced Soft Errors,” in Soft Errors in Modern Electronic Systems, vol. 41, M. Nicolaidis, Ed. Boston, MA: Springer US, 2011, pp. 55–76.

[38] H. Ziade, R. Ayoubi, and R. Velazco, “A Survey on Fault Injection Techniques”. Int. Arab J. Inf. Technol., v. 1, n. 2, p. 171–186, 2004.

[39] S. Mukherjee, Architecture Design for Soft Errors. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[40] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, “Experimental evaluation of a COTS system for space applications,” in International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings, 2002, pp. 325–330.

[41] A. da Silva and S. Sanchez, “LEON3 ViP: A Virtual Platform with Fault Injection Capabilities,” in 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), 2010, pp. 813–816.

[42] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, “Fault injection into VHDL models: the MEFISTO tool,” in Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers, 1994, pp. 66–75.

[43] P. Folkesson, S. Svensson, and J. Karlsson, “A comparison of simulation based and scan chain implemented fault injection,” in Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998. Digest of Papers, 1998, pp. 284–293.

[44] A. V. Fidalgo, G. R. Alves, and J. M. Ferreira, “Real time fault injection using a modified debugging infrastructure,” in On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International, 2006, p. 6 pp.–.

[45] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, “Fault emulation: a new approach to fault grading,” in, 1995 IEEE/ACM International Conference on Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers, 1995, pp. 681–686.

[46] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, “Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation,” IEEE Transactions on Nuclear Science, vol. 54, no. 1, pp. 252–261, Feb. 2007.

[47] M. Alderighi, F. Casini, S. D’Angelo, M. Mancini, D. M. Codinachs, S. Pastore, C. Poivey, G. R. Sechi, G. Sorrenti, and R. Weigand, “Experimental Validation of Fault Injection Analyses by the FLIPPER Tool,” IEEE Transactions on Nuclear Science, vol. 57, no. 4, pp. 2129–2134, Aug. 2010.

[48] Mathieu Brethes, “Atome - Binary Translation for Accurate Simulation”, September 16, 2004.

[49] Frank Schirrmeister, “Delivering Software Productivity for Multi-core Systems on Chip”, www.imperas.com/news March 3, 2008.

[50] OVPSim and Imperas CpuManager User Guide – Version 2.0.1.

[51] OVP Processor Modeling Guide – Version 2.0.1.

[52] J. R. Eastlack, “Extending volunteer computing to mobile devices,” 2011

[53] ARM® Cortex-A9 Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388i/I1002917.html, [Accessed: 10-Dec-2014].

[54] ARM Holdings. CoreTile Express A9x4 - For the Versatile Express Family. ARM corporate website. [Online] www.arm.com.

[55] Imperas Software Limited. "OVP Guide to Using Processor Models". Imperas Buildings, North Weston, Thame, Oxfordshire, OX92HA, UK, January 2015. Version 0.5, docs@imperas.com.

[56] Imperas Software Limited. " Model Specific Information for variant ARM Cortex-A9MPx1". Imperas Buildings, North Weston, Thame, Oxfordshire, OX92HA, UK, July 2016. Version 0.5, docs@imperas.com.

[57] Bowman, Ivan. "Conceptual Architecture of the Linux Kernel". 1998. Available at http://www.stillhq.com/pdfdb/000524/data.pdf.

[58] “Linux.org.” [Online]. Available: http://www.linux.org/. [Accessed: 05-Oct-2015].

[59] P. C. P. Bhatt, An Introduction to Operating Systems: Concepts and Practice, 4 edition. Prentice-Hall of India. PHI Learning, 2014.

[60] “Linux manual page.” [Online]. Available: http://man7.org/linux/manpages/man7/signal.7.html. [Accessed: 13-Mar-2016].

[61] I. T. Bowman, R. C. Holt, and N. V. Brewster, “Linux as a case study: its extracted software architecture,” in Proceedings of the 21st International Conference on Software Engineering, 1999, pp. 555–563.

[62] M. Nicolaidis, “Time redundancy based soft-error tolerance to rescue nanometer technologies,” in 17th IEEE VLSI Test Symposium, 1999. Proceedings, 1999, pp. 86–94.

[63] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 01/2016

[64] F. de A. Geissler, “Metodologia de injeção de falhas baseada em emulação de processadores”, Universidade Federal do Rio Grande do Sul, 2014.

[65] R. Kost, D. Connors, S. Pasricha, "Characterizing the Use of Program Vulnerability Factors for Studying Transient Fault Tolerance in Multi-core Architectures", Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS 2009) Estoril, Portugal, Jun 2009.

[66] “Cyclone V Soc - Overview”. [Online]. Available at: https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html. [Accessed: 08-mar-2016].

[67] “Zynq-7000 All Programmable SoC”. [Online]. Available at: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#documentation. [Accessed: 08-mar-2016].

[68] F. M. Lins, L. Tambara, F. L. Kastensmidt, and P. Rech, “Register File Criticality on Embedded Microprocessor Reliability”, 2015, unpublished.

[69] S. Rezgui, R. Velazco, R. Ecoffet, S. Rodriguez, and J. R. Mingo, “Estimating Error Rates in Processor-Based Architectures”, IEEE Transactions on Nuclear Science, vol. 48, no. 5, pp. 1680-1687, 2001.

[70] L. G. Casagrande and F. L. Kastensmidt, “Soft error analysis in embedded software developed with without operating system,” in 2016 17th Latin-American Test Symposium (LATS), 2016, pp. 147–152.

[71] T. S. Weber, "Um roteiro para a exploração dos conceitos básicos de tolerância a falhas". [Online]. Available at: http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf . [Accessed: 23-aug-2016].

APPENDIX A – COMPLETE BENCHMARK RESULTS

As the benchmark results are extensive, it was decided to include them in this appendix. The results are presented with the error rates in both environments, considering the detailed classification presented in Section 3.4.3.

Table Appendix A.1 – Bare Metal Benchmark Results with Full Classification (a)

BARE METAL

	UNACE				SDC		
APLICATIONS	**MASKED FAULT**	**Control Flow Data OK**	**REG STATE Data OK**	**Total Number of faults**	**Control Flow Data ERROR**	**REG STATE Data ERROR**	**SDC**
binary_search	46161	3514	0	50325	212	39215	324
bitManipulation	39965	4412	0	55623	2345	38141	1472
bubble	42138	504	0	57358	5614	39001	906
compress	66096	1908	0	31996	1702	7741	7144
crc	45573	1932	0	52495	401	37819	255
factorial	39693	1574	0	58733	1562	40818	3041
fdct	40706	502	0	58792	1547	45261	608
fibonacci	43980	2614	0	53406	1206	40172	0
harm	36648	3116	0	60236	523	41232	5602
matrixMult	25600	607	0	73793	16020	27314	15472
mdc	32359	9042	0	58599	2184	40321	1616
peakSpeed	43118	320	0	56562	2499	42819	310

Table Appendix A.2 – Bare Metal Benchmark Results with Full Classification (b)

BAREMETAL

	HANG									
APLICATIONS	**Hang**	**RD PRIV**	**WR PRIV**	**RD ALIGN**	**WR ALIGN**	**FE PRIV**	**FE ABORT**	**ARITH**	**Hard Fault**	**Lockup**
binary_search	1065	4521	0	421	98	4145	324	0	0	0
bitManipulation	1421	5547	699	211	108	5214	465	0	0	0
bubble	925	5468	255	266	0	4625	298	0	0	0
compress	4820	4214	632	913	107	4617	106	0	0	0
crc	1431	5934	625	741	112	4923	254	0	0	0
factorial	8128	241	164	108	119	4416	136	0	0	0
fdct	204	5237	714	0	142	4238	841	0	0	0
fibonacci	6621	261	0	312	0	4625	209	0	0	0
harm	7332	104	116	416	329	4202	380	0	0	0
matrixMult	3002	4621	1153	564	0	4412	1235	0	0	0
mdc	7116	915	261	627	206	4139	1214	0	0	0
peakSpeed	5821	107	104	649	0	3952	301	0	0	0

Table Appendix A.3 – Linux OS Benchmark Results with Full Classification (a)

LINUX

	UNACE				**SDC**		
APLICATIONS	**MASKED FAULT**	**Control Flow Data OK**	**REG STATE Data OK**	**Total Number of faults**	**Control Flow Data ERROR**	**REG STATE Data ERROR**	**SDC**
binary_search	30479	2205	53796	13520	408	213	0
bitManipulation	38701	2820	40102	18377	1236	99	515
bubble	20928	384	59101	19587	2743	114	408
compress	26063	1617	47255	25065	1723	1516	492
crc	33863	2056	51648	12433	417	0	219
factorial	16691	1930	64622	16757	2314	205	197
fdct	27343	102	51026	21529	1117	5612	109
fibonacci	20612	0	15421	63967	4837	38920	1913
harm	22313	3142	54916	19629	2014	399	920
matrixMult	22339	108	28028	49525	6913	2859	7827
mdc	20458	8826	51482	19234	2039	537	129
peakSpeed	9872	325	74907	14896	1321	2154	0

Table Appendix A.4 – Linux OS Benchmark Results with Full Classification (b)

LINUX

	HANG										**Hang Detected by Linux**		
APLICATIONS	**Hang**	**RD PRIV**	**WR PRIV**	**RD ALIGN**	**WR ALIGN**	**FE PRIV**	**FE ABORT**	**ARITH**	**Hard Fault**	**Lockup**	**SEG FAULT**	**ILLEGAL INST**	**FLTPNT EXC**
binary_search	286	0	0	0	0	0	0	0	0	0	12507	106	0
bitManipulation	309	0	0	0	0	0	0	0	0	0	15714	504	0
bubble	98	0	0	0	0	0	0	0	0	0	15623	601	0
compress	10324	0	0	0	0	0	0	0	0	0	10923	87	0
crc	0	0	0	0	0	0	0	0	0	0	11768	29	0
factorial	4008	0	0	0	0	0	0	0	0	0	9327	706	0
fdct	0	0	0	0	0	0	0	0	0	0	14478	0	213
fibonacci	6435	0	0	0	0	0	0	0	0	0	11756	106	0
harm	5431	0	0	0	0	0	0	0	0	0	9111	645	1109
matrixMult	502	0	0	0	0	0	0	0	0	0	30916	508	0
mdc	2768	0	0	0	0	0	0	0	0	0	11615	412	1734
peakSpeed	387	0	0	0	0	0	0	0	0	0	10826	208	0

APPENDIX B – LATS 2016 PAPER

A paper related to the topic of this thesis is presented in this appendix. The paper also called “Soft Error Analysis in Embedded Software Developed with & without Operating System” has been accepted for publication in Latin American Test Symposium (LATS) in 2016 for oral presentation.

Soft Error Analysis in Embedded Software Developed with & without Operating System

Luiz Gustavo Casagrande and Fernanda Lima Kastensmidt
PGMICRO – Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{lgcasagrande, fglima}@inf.ufrgs.br

***Abstract*— This work presents a comparative analysis of soft error susceptibility in a well-known embedded microprocessor ARM Cortex-A9 single core, widely used along with safety critical applications, running embedded software developed for a bare metal environment and with operating system. The soft error analysis is performed by fault injection on OVPSim-FIM simulator platform. The faults injection campaign injects thousands of bit-flips in the microprocessor register file while executing a set of benchmarks with a diverse code behavior from control flow dependence to data intensive. Results present the percentage of masking faults, the classification of the errors and the number of exceptions detected by the operating system. The proposed method and the obtained results can help guiding software developers in choosing different architectures of the code in order to improve fault masking.**

Keywords—bare metal; operating system; Linux; soft errors; comparison; ARM Cortex-A9; embedded processor

I. INTRODUCTION

Soft errors are transient faults that are mainly observed as bit-flips in memory elements operating under radiation environment [1]. The radiation environment can be space or on Earth, where neutrons interact with the material provoking secondary particles such as alpha that can provoke transient faults [2]. Soft errors have emerged as a key challenge in microprocessor design. In microprocessors, the effect of Single Event Upsets (SEU) can be noticed by errors in the data or errors that change the application flow. The architecture of the program code is one of the most important and critical parameter in the embedded processor and it plays a major role to develop highly efficient and robust embedded systems.

Safe-critical applications seek high reliability. They are usually composed of single or multi-core processors that may run embedded software developed with or without operating systems. Bare metal software is a term used in embedded systems when the program code is directly hosted on the target microprocessor and is self-managed. It is known to be reliable, allowing high controllability of hardware and software flow. Additionally, maintenance, debugging and troubleshooting of code is easier and less time consuming.

Operating systems can provide the availability of hardware resources for applications with a high level of complexity. The increasing need of this complex operations and resource sharing demands better using of the available capabilities as well as better programmability. In addition, under soft errors bare metal software is not necessarily more robust than operating systems. For the best of our knowledge there are not many works in the literature that compare bare metal and operating systems in modern processor under soft errors.

Consequently, in order to understand the behavior of a microprocessor under soft errors, the use of fault injection mechanism is needed to emulate soft errors for different applications running in the microprocessor without and with operating system. Between the many different ways to inject faults, soft errors can be injected by using radiation accelerators, where the embedded system is exposed to the beam under a certain ion flux for a period of time. Another fault injection mechanism is by emulating soft errors by simulation, which was the choice of this paper. The embedded system is modeled in C language and faults are injected during the simulation by using the Imperas OVPSim simulator.

This paper aims to compare soft error susceptibility in embedded software that was developed for a bare metal environment and under an operating system. It also analyses the architectural vulnerability factor (AVP) of the register file in the well-known embedded microprocessor ARM Cortex-A9, widely used commercially in low-power or thermally constrained applications. The soft error analysis is performed by fault injection on OVPSim-FIM simulator, capable of simulating failures induced by cosmic particles on the processor registers bank, producing bit-flips with controlled locality and temporality, allowing a survey of errors detected and treated by both the platforms. The OVPSim fault injection environment was proposed by [3].

A set of 11 applications was used as benchmarks that diversify control and data flow behavior. Those applications were compiled for the target architecture and run directly on the processor for the bare metal tests. The chosen operating system for the comparative tests was Linux, mainly because it has open code, which significantly facilitates the controllability and observability required by the internal experiments. Beyond that, it is worth mentioning that Linux is already considered as an option of a commercial purpose operating system, which is being included in a wide range of applications, including those with reliability requirements [4].

During the fault injection campaign, it was injected 100,000 faults in the register file to evaluate the benchmark. Results were classified into unace or masked faults, silent data corruption (SDC) and hang. Results show that applications running in Linux environment are less prone to errors due to bit-flips in the register file, while bare metal applications have shown more susceptible to silent data corruption (SDC) errors.

The rest of the paper is organized as follows. Section 2 reports background information about error analysis in bare metal approaches and with the presence of an operating system. Section 3 describes processor architecture setup, outlining ARM Cortex-A9 relevant features and simulation model, as well as the Linux environment. In the section 4, the fault injection platform is presented, including the error model overview. Experimental campaign results and discussion including used benchmark is presented along with the error classification in the section 5. Finally, section 6 reports some conclusions.

II. Related Work

Many studies on error analysis were performed targeting embedded systems under soft error injection and most of them agree on using bit flip strategy as a representative error model of physical faults [5]. However, these studies focus on a single perspective and don't explore the system dependence in terms of software implementation on a comparative approach.

The bare metal software implementation in embedded processors has been largely investigated under soft. Works from Rebaudengo [6] show significantly susceptibility of the register file in the SPARC v8 architecture. Results from this work show that upsets in register file can provoke errors in the application generating up to 7 % of total SDC and hangs errors. The work from Reorda [7] also classify the soft errors effect in *silent*, *failure* and *time out* categories for a PowerPC 405 processor by injecting faults in any available bit of the general and special registers, reaching up to 16 % of error detection.

Some works have investigated embedded processors running operational systems. Sterpone et al. [8] evaluate dependability of real time applications under OS environment and define an error classification of arising faults. They use the microkernel scheduler of the µC-Linux in the Xilinx Microblaze® processor, injecting faults in the registers values in the configuration memory and they classify the results between *stall*, *crash*, *latent* and *silent* errors. In this work, they show more than 35 % of injected faults that can be detected by OS exception. Some works also investigate Linux operating systems under soft errors, like Fabre [9] and [10] which evaluate the Chorus ClassiX r3 microkernel reliability under different types of failure, including bit-flips, resulting in up to 70 % of error rate and a total of 38 % to 56 % of detected exception errors depending on the kernel fault location.

For a direct comparison between bare metal and operating systems, we found the recent work from Rech et al [11] that investigates the cache conflict effect in ARM-A9 to tolerate some soft errors. A set of applications running Linux kernel and with bare metal were tested under radiation. Results in terms of number of observed errors classified as SDC and hang are shown. However, because the experiment is under radiation, it is not possible to have a large observability of the errors and a more detailed classification as it can be done in a simulation environment as shown in this paper.

III. Soft Error Analysis Methodology and Setup

The proposed methodology includes a simulation platform, a microprocessor model, a set of meaningful benchmarks to simulate under faults and a dedicated module that allows the injection of faults and the collection of the results.

A. Microprocessor Model

We choose ARM Cortex-A9 processor in this study mainly because it represents an architecture widely used in industry primarily combined with operating systems. It is a single core processor modeled by ARM Ltd to be used along with OVP simulator, called "Cortex-A9MPx1". Based on the ARMv7-A architecture, it supports the 32-bit ARM instruction set with a multi issued architecture, dividing its operations into primary and secondary data processing, load and store functions and floating point operations. Among others, allows registers virtual renaming for better resource sharing.

The ARM Cortex-A9 processor model implemented in OVPSim platform is functionally equivalent to the real processor, but since its execution is accurate in an instruction level, structural characteristics are not modeled in the same way as the actual architecture [14]. The ARM A9 model uses an API to convert the target processor code into native code: it builds an instruction decoder for the target processor instruction set, provides the disassembly output and generates native code for the simulation without explicit reference to native instruction set. The platform strategy is code translation, and is responsibility of the simulator to implements the just-in-time translation algorithm, memory allocation, back translation of modified code and further resources necessary for the simulation. In this manner, the main structural element of the Cortex-A9 processor architecture modeled accurately is the register file, including 12 general purpose registers, plus stack and frame pointer.

Therefore, fault injection campaigns are limited only to the register file, which is a small portion of the complexity embedded in the A9 processor. However, is highly valuable the ability to analyze in an isolated manner the architectural vulnerability factor of the register file of a large-scale commercial processor.

B. Linux Operating System

The Operating systems is a reduced Linux v2.6 kernel. The codes are compiled using the standard g++ Gnu Compiler Collection and automatically run from the file system directory by a script. It is also possible to set application priority in the script, allowing behavior analysis under different OS scenarios.

One of the advantages of the Linux is the ability to natively detect exceptions by implementing dedicated kernel functions that monitors each application operation before execution. Kernel calls functions are generated via interrupts and manifest themselves in 4 ways: (1) an interruption issued to the processor by a hardware device indicating that it requires attention, (2) an exception indicated by the processor because of an error, (3) a kernel call or system call issued by an application, or (4) a kernel thread [12]. The activation of internal kernel functions is not defined only by the mentioned events, but also the current kernel state. In this work, we focus on the second and third items (kernel calls issued by an error in the processor or by an application), since the injected faults can result in both behaviors and are reflected similarly in the operating system mechanisms.

In this scope, a process may result in a hardware exception generation when, for example, it attempts to divide by zero or fails translating a virtual address. In a UNIX based operating system, this event automatically changes the processor context to start the execution of an exception handler in the kernel. In this cases kernel are able to manage the exception, like in a 'page fault' that happens when there is a mapped address page which is not loaded in fact, but the kernel can pre-identify the problem before an illegal access actually occur. As a result, the application exits, but if possible the page will be allocated, remapped and the application flow can follows.

When an exception is detected and cannot be handled by the kernel's internal mechanisms, them it is identified and a default trigger mechanism is called in order to handle these exceptions. This mechanism is called signals, where the system sends a notification to the application with the type of exception that occurred [13]. Examples of such exceptions that triggers default signals are:

- Division by zero: a division by zero error exception (SIGFPE) is generated by making the kernel sending this signal to the application.
- Segmentation Fault: access to a memory address out of virtual address space, making the kernel notify the application through a SIGSEGV signal, since he cannot know the right address.

By sending a signal back to the application indicating the exception, it may contain a handler to treat it. The existence of these handlers depends on the software developer, and if it does not exist, the operating system can terminate the application. In this work all benchmarks was edited by including a default signal handler able to capture any signal propagated by the Linux kernel with no corrective action, but only generating an external message to the OVPSim platform in order to provide error classification for further result gathering.

C. *Benchmark Applications*

The chosen applications are binary_search, bitManipulation, bubble, compress, crc, factorial, fdct, harm, matrixMult, mdc, peakSpeed. These applications comprises diverse data behavior, from a highly flow dependence, like crc algorithm, to data intensive, like matrixMult and bitManiulation. The applications choice was intentional in order to exploit the diversity of behavior and attempt to reach error detection classification variety.

Table I shows the number of instructions executed in bare metal and in Linux for the set of applications. The Linux boot executes 1176574110 instructions. Each application has a Linux boot before it starts. And at this time it is not used to inject faults. Note that the number of instructions of the application executed in bare metal and in Linux is very similar. The difference comes from the fact that for bare metal we use a compiler for the ARM Cortex-A9 architecture, while for Linux we use the default compiler, since the application is on top of the OS.

D. *Fault Injection Platform*

The ARM Cortex-A9 processor model implemented in OVPSim platform is functionally equivalent to the real processor, but since its execution is accurate in an instruction level, structural characteristics are not modeled in the same way as the actual architecture [14]. The only structural element of the Cortex-A9 processor architecture that is modeled accurately as the real architecture is the register file, which includes 12 general-purpose registers, stack and frame pointer registers.

TABLE I. NUMBER OF INSTRUCTIONS EXECUTED IN BARE METAL AND IN LINUX FOR THE SET OF APPLICATIONS

Applications	***Bare Metal # of Instructions***	***Linux # of Instructions***
binary_search	138,293	130,010
bitManipulation	305,211	293,520
bubble	236,548	236,164
compress	196,736	189,141
crc	201,812	223,448
factorial	310,343	310,061
fdct	507,696	537,709
harm	449,123	448,839
matrixMult	343,373	343,172
mdc	658,172	510,295
peakSpeed	19,859	19,574

The OVPSim platform is code translation, and the simulator is responsible to implement the just-in-time translation algorithm, memory allocation, back translation of modified code and further resources needed for the simulation. The ARM A9 model uses an API to convert the target processor code into native code. It builds an instruction decoder for the target processor instruction set providing the disassembly output and generating native code for the simulation without explicit reference to native instruction set.

The OVPSim platform setup consists of a single core ARM-cortexA9 processor connected by a bus to a memory model, both natives of OVPSim platform. OVPSim-FIM [1] is a module based on OVPSim technology to inject faults during the simulation in the process model register file. So in the case of the ARM-CortexA9, the 12 general-purpose registers, stack and frame pointer registers received random bit-flips during the fault injection campaign. OVPSim-FIM allows read and write access to all environment variables at any time during the simulation.

The fault injection flow is divided into 5 stages:

1) Gold execution: each application is executed with no fault injection intervention for reference parameters collecting, like instruction count and memory map results.

2) Fault creation: it randomly chooses a register and one of its 32-bit of that target register to flip. Also it determines the moment when this fault will be injected. The accuracy of time is at instruction level.

3) Fault injection: The following application executions are interrupted by the platform and the raffled bit-flip is applied.

4) Results collection: Each application under fault is compared with the golden run. If the normal control flow behavior or final data changes, it is considered an error.

5) Report creation: The platform collects the status of each run and classifies the errors accordingly to its effect. A final report is generated summarizing the campaign.

Due to Linux ability to handle errors, OVPSim-FIM classification is extended to include signals triggered by kernel categories. The signal handler added to each application code has the ability to write and classify in a reserved memory area with no code or application data each signal. During results collection phase, the fault injector can access this memory and classify the items together with default platform reports. The fault injection campaigns have shown that between operating system available signals, Linux was able to detect only segmentation fault, illegal access and floating point exception violations, and so, only this signals was included in results.

When executing an application in bare metal mode, all instructions in memory belong to the application and it is compiled directly for the architecture. Applications that run in Linux shall be compiled for the specific operating system and the resulting memory occupation may vary.

Additionally, in order to run the benchmark, the Linux shall be booted first, resulting in a much higher number of instructions than in bare metal version. This behavior generates an unfair comparison between platforms, since random fault injections are most likely to happen during Linux boot than in the application itself. Safe critical applications that use operating systems are commonly initialized before being exposed to the susceptible environment, thus booting should not be taken into account. Also, when performing the OS boot, all the latent errors are cleaned.

OVPSim-FIM was modified to identify application beginning and guaranty that fault injection happens only after application starts, but after that, Linux kernel is also exposed to faults, since it shares the processor resources. Fig. 1 presents the environment fault injection approach.

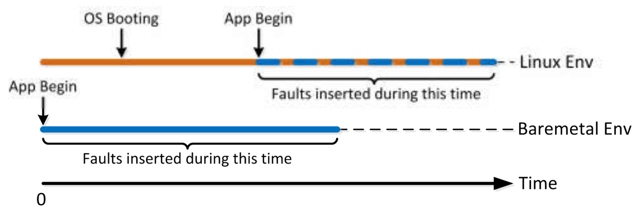

OS Booting
App Begin
Linux Env
Faults inserted during this time
App Begin
Baremetal Env
Faults inserted during this time
Time
0

Fig. 1. Fault injection localization in Bare Metal and Linux OS Environment.

IV. EXPERIMENTAL RESULTS

Each campaign consists on running each benchmark application 100,000 times for each software code case, i.e., bare metal and Linux OS, injecting one fault per run in a randomly sorted register of the register file.

A. Fault Classification

As mentioned, the error analysis compares each application running under fault injection with a gold reference to detect errors. An error would be reported when a mismatch is found in the application control flow execution or data results. The platform OVPSim along with the fault injector is able to report and classify errors, such as:

1) Unace: No error detected and the result memory is identical to golden execution;

2) Hang: The application or system hanged or crashed;

3) SDC: Means that application has finished but the data memory mismatch with the golden memory. Note that not only the application results stored in memory are compared but the entire memory space. This is important to analyze any latent error in the memory that may occur.

Fig. 2 shows the number of unace, hang errors and SDC errors for each bare metal application. Note that in general the error rate is slightly above 50 % of the total number of injected faults. The number of hangs ranges from 10% to 15%, showing that regardless of the application behavior, it tends to crash at a constant rate. This can be explained by the fact that this type of error is manifested due to a violation of the execution flow that causes an invalid state of the processor, not depending on the characteristics of the application itself. Also, the fact that ARM Cortex-A9 has a build in register renaming mechanism makes control variables to be equally distribute between registers, causing control variables stored in registers to be spread and consequently equally prone to errors. SDC errors in most benchmark cases range from 40 % to 47 % of total injected faults. But it is important to remark that the compress application shows to be less susceptible to SDC errors with 16.6 %. This may be due to the extensively use of external memory. The matrixMult shows a SDC error rate of 58.8 % because it abuses of 'for' functions, which is highly benefited from the register renaming mechanism that does loop unrolling, exposing more the data in the register file.

As mentioned, the Linux kernel has the capacity to manage failures before they happen due to memory virtualization and is able to pre evaluate the execution of an operation. The impact of this treatment is presented in Fig. 3. In Linux environment, the majority of the applications have an error rate ranging from 12 % to 25 %. Hangs in this case can also happens in OS kernel as it is exposed to faults during application execution, and so they prove to have a regular distribution, from 12 % to 18 % of total injected faults. It is important to mention that in this category, we also included hangs that the operating system was able to signalize. However two applications in this group, compress and matrixMult, draws attention by having 21 % and 32 % hangs, respectively, this may be due to the fact of the allocation of registers in Linux and for the application. In the SDC category, most applications stayed under 6 % of total injected errors. It happens because in Linux the application is statistically less likely to manifest an error, since the application shares the register file resources with operating system, i.e., part of the register file is allocated to Linux and part for the application, so the chances to hit a register file that causes a SDC in the application is reduced. As expected, matrixMult still has more SDC errors than the average due to hardware loop unrolling.

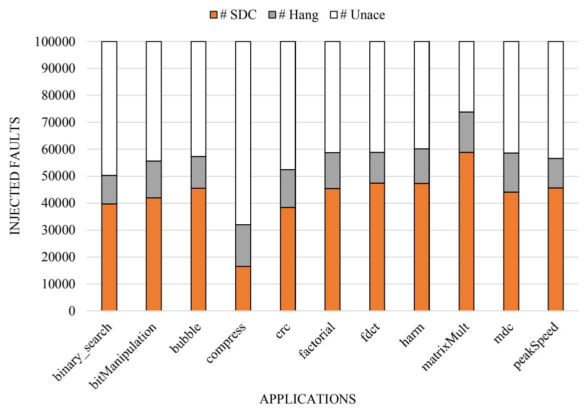


Fig. 2. Bare Metal Error Classification.

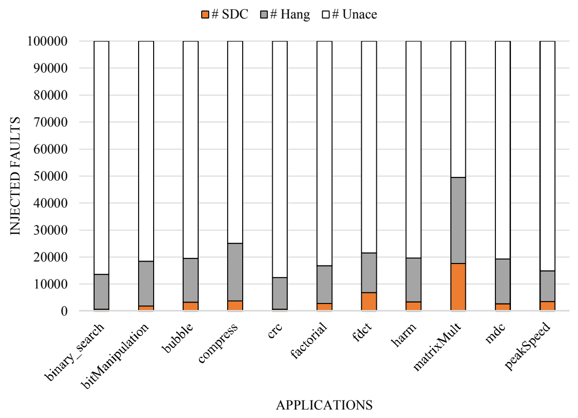


Fig. 3. Linux OS Error Classification.

The Linux campaign for all benchmarks was repeated by setting a different application priority, from low to high. Results showed to have less than 3 % of variation compared with normal priority, making them not conclusive, since they can be accredited to the randomness of the tests.

Error reports triggered by Linux kernel signals comprise a specific set. As detected and reported by the operating system they can be handled by functions added in the application code, allowing correction, discarding of unreliable results or even restart of the application. Exceptions detected by Linux are hangs that can be treated by the application and therefore should not be considered as errors. Fig. 4 depicts Linux Hang detection rate.

Linux error detection graph correlate the rate of identified soft errors by the operating system among total number of manifested Hang errors. Surprisingly, Linux shows to have a high rate of Hang perception, reaching 100% of detection in the case of the fdct application. As expected, analyzing the reports in details we found that around 90% of detected errors are Segmentation Faults, when the application tries to address a non-allocated memory area, making this more perceptive in applications that recursively access external data, like bit manipulation and matrix multiplication. In a regular distribution between applications, Illegal Access signal happened when the processor attempts to execute an illegal, malformed, unknown, or privileged instruction, an unsurprising behavior since we are modifying randomly internal registers values. Harm, fdct and mdc are the only applications with floating point operations in the benchmark, and so Linux triggered Floating Point Exception signal.

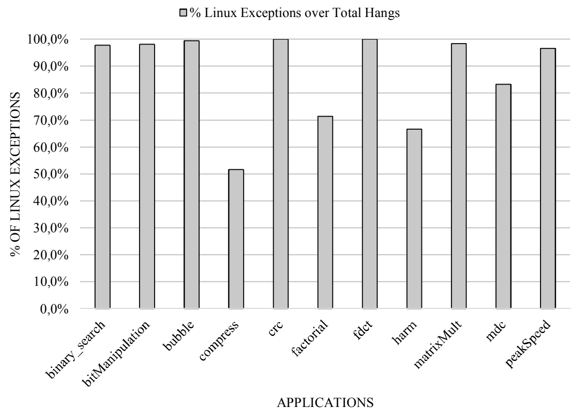


Fig. 4. Linux Error Detection.

Finally, considering the results presented, it is possible to determine the final error rate, which reflects the architectural vulnerability factor of the register file. The comparison between bare metal and Linux applications reflects the sum of SDC and hangs not detected by the operating system on both environments. Fig. 5 depicts the results.

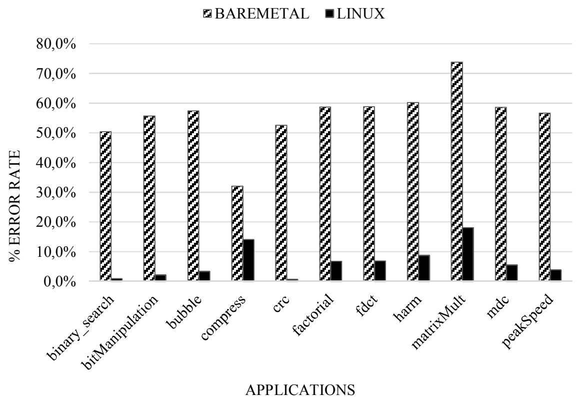


Fig. 5. Experimental Results Error Rate.

Results show clearly that applications running in bare metal environment are more prone to errors in general than when running in Linux OS version when faults are injected in the register file.

V. CONCLUSION

The paper analyzed in a comparative approach the susceptibility of several application under soft errors in two environments: bare metal and on the top of the Linux operating system. The implementation was conducted by simulation with OVPSim platform and the OVPSim-FIM fault injector to introduce bit-flips in the register file of the ARM Cortex-A9 processor. We classified the errors reported by the platform and extended the report including Linux signal reporting. The methodology promotes statistical results separating the main groups of errors between control flow, silent data corruption and Linux captured errors.

The goal is achieved by performing an analysis in terms of error and not failure domain, since location is not the focus, but the variety of errors that can arise in similar failures. By doing this, we have shown that application running bare metal increase significantly the overall error rate for all 11 benchmarks that comprises a diverse scenario of data utilization. Despite this, considering different groups of errors, we found that the operating system has considerably less silent data corruption than bare metal environment, having the majority of failures located in Hang errors group. By saying this we can conclude that in our experiment the Linux OS environment tend to hide more frequently soft errors, while bare metal is more prone to corrupt the data of the application. When comparing the results with [11] we faced a different behavior where SDC errors in Linux was almost the same as in bare metal; we understand and the paper itself points out that it happens mainly because in the neutron flux the faults were also injected in the cache, where the application data is always exposed, leading in data corruption even before the content reaches the register file. Another reason is that in [11] the application is restarted continuously without restarting the operating system, a process where bit-flips that are not expressed in a particular execution may be manifested in the future in others.

An additional category of error was analyzed by separating signals triggered by Linux kernel, where we found that the contribution of this group can be up to 100% of the total number of Hangs. Since the kernel reports back to the application the signal, it is possible to implement signals handlers in the code in order to deal with the exception and effectively mitigate the error effect.

As a future work, we propose to analyze the detailed error classification considering multiple applications running with Linux OS with the intention of proposing tolerant techniques focused on error distribution. Another proposal includes sharing processor and operating system resources by running multiple applications at the same time and evaluating soft errors masking effects with and without multicore platforms.

REFERENCES

[1] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, "A model of soft error effects in generic IP processors," in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, DFT 2005*, 2005, pp. 334–342.

[2] T.R. Oldham and F.B. McLean. "Total ionizing dose effects in MOS oxides and devices," in *IEEE Transactions on Nuclear Science*, 2003. pp. 483–499.

[3] F. Rosa, "A Fast and Scalable Fault Injection Framework to Evaluate Many-core Soft Error Reliability," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems Symposium. DFT 2015*, 2015, unpublished.

[4] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau, "Analysis of the effects of real and injected software faults: Linux as a case study," in *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings*, 2002, pp. 51–58.

[5] Z. Wang, C. Chen, and A. Chattopadhyay, "Fast reliability exploration for embedded processors via high-level fault injection," in *2013 14th International Symposium on Quality Electronic Design (ISQED)*, 2013, pp. 265–272.

[6] M. Rebaudengo, M. S. Reorda, and M. Violante, "An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor," in *Design, Automation and Test in Europe Conference and Exhibition,* 2003, pp. 602–607.

[7] M. Sonza Reorda, L. Sterpone, M. Violante, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, " ," in *Test Symposium, 2006. ETS '06. Eleventh IEEE European*, 2006, pp. 75–82.

[8] L. Sterpone and M. Violante, "An Analysis of SEU Effects in Embedded Operating Systems for Real-Time Applications," in *IEEE International Symposium on Industrial Electronics, 2007. ISIE 2007*, 2007, pp. 3345–3349.

[9] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat, "Assessment of COTS microkernels by fault injection," in *Dependable Computing for Critical Applications 7, 1999*, 1999, pp. 25–44.

[10] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid," in *Dependable Computing — EDCC-3*, J. Hlavička, E. Maehle, and A. Pataricza, Eds. Springer Berlin Heidelberg, 1999, pp. 143–160.

[11] T. Santini, P. Rech, F. R. Wagner and L. Carro, "Exploiting Cache Conflicts to Reduce Radiation Sensitivity of Operating Systems on Embedded Systems," in *International Conference on Compilers Architectures and Synthesis of Embedded Systems*, 2015, unpublished.

[12] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux As a Case Study: Its Extracted Software Architecture," in *Proceedings of the 21st International Conference on Software Engineering*, New York, NY, USA, 1999, pp. 555–563.

[13] "Linux.org." [Online]. Available: http://www.linux.org/. [Accessed: 05-Oct-2015].

[14] "ovpworld.org." [Online]. Available: http://www.ovpworld.org/. [Accessed: 05-Oct-2015].