POLITECNICO DI TORINO

DOCTORATE SCHOOL
PH.D. IN COMPUTER AND CONTROL ENGINEERING - XXVI CYCLE

PHD THESIS

INNOVATIVE TECHNIQUES FOR TESTING AND DIAGNOSING SOCS

ADVISOR:
PH.D. PAOLO BERNARDI

CANDIDATE:
M.SC. MAURICIO DE CARVALHO

FEBRUARY, 2015

Abstract

We rely upon the continued functioning of many electronic devices for our everyday welfare, usually embedding integrated circuits that are becoming even cheaper and smaller with improved features. Nowadays, microelectronics can integrate a working computer with CPU, memories, and even GPUs on a single die, namely *System-On-Chip* (SoC). SoCs are also employed on automotive safety-critical applications, but need to be tested thoroughly to comply with reliability standards, in particular the ISO26262 functional safety for road vehicles.

The goal of this PhD. thesis is to improve SoC reliability by proposing innovative techniques for testing and diagnosing its internal modules: CPUs, memories, peripherals, and GPUs. The proposed approaches in the sequence appearing in this thesis are described as follows:

1. ***Embedded Memory Diagnosis***: Memories are dense and complex circuits which are susceptible to design and manufacturing errors. Hence, it is important to understand the fault occurrence in the memory array. In practice, the logical and physical array representation differs due to an optimized design which adds enhancements to the device, namely *scrambling*. This part proposes an accurate memory diagnosis by showing the efforts of a software tool able to analyze test results, unscramble the memory array, map failing syndromes to cell locations, elaborate cumulative analysis, and elaborate a final fault model hypothesis. Several SRAM memory failing syndromes were analyzed as case studies gathered on an industrial automotive 32-bit SoC developed by STMicroelectronics. The tool displayed defects virtually, and results were confirmed by real photos taken from a microscope.

2. ***Functional Test Pattern Generation***: The key for a successful test is the pattern applied to the device. They can be structural or functional; the former usually benefits from embedded test modules targeting manufacturing errors and is only effective before shipping the component to the client. The latter, on the other hand, can be applied during mission minimally impacting on performance but is penalized due to high generation time. However, functional test patterns may benefit for having different goals in functional mission mode. Part III of this PhD thesis proposes three different functional test pattern generation methods for CPU cores embedded in SoCs, targeting different test purposes, described as follows:

   a. ***Functional Stress Patterns***: Are suitable for optimizing functional stress during

Operational-life Tests and *Burn-in Screening* for an optimal device reliability characterization

b. ***Functional Power Hungry Patterns***: Are suitable for determining functional peak power for strictly limiting the power of structural patterns during manufacturing tests, thus reducing premature device over-kill while delivering high test coverage

c. ***Software-Based Self-Test Patterns***: Combines the potentiality of structural patterns with functional ones, allowing its execution periodically during mission. In addition, an external hardware communicating with a devised SBST was proposed. It helps increasing in 3% the fault coverage by testing critical *Hardly Functionally Testable Faults* not covered by conventional SBST patterns.

An automatic functional test pattern generation exploiting an evolutionary algorithm maximizing metrics related to stress, power, and fault coverage was employed in the above-mentioned approaches to quickly generate the desired patterns. The approaches were evaluated on two industrial cases developed by STMicroelectronics; 8051-based and a 32-bit Power Architecture SoCs. Results show that generation time was reduced upto 75% in comparison to older methodologies while increasing significantly the desired metrics.

3. ***Fault Injection in GPGPU***: Fault injection mechanisms in semiconductor devices are suitable for generating structural patterns, testing and activating mitigation techniques, and validating robust hardware and software applications. GPGPUs are known for fast parallel computation used in high performance computing and *advanced driver assistance* where reliability is the key point. Moreover, GPGPU manufacturers do not provide design description code due to content secrecy. Therefore, commercial fault injectors using the GPGPU model is unfeasible, making radiation tests the only resource available, but are costly. In the last part of this thesis, we propose a software implemented fault injector able to inject bit-flip in memory elements of a real GPGPU. It exploits a software debugger tool and combines the C-CUDA grammar to wisely determine fault spots and apply bit-flip operations in program variables. The goal is to validate robust parallel algorithms by studying fault propagation or activating redundancy mechanisms they possibly embed. The effectiveness of the tool was evaluated on two robust applications: redundant parallel matrix multiplication and floating point *Fast Fourier Transform*.

Acknowledgements

Firt of all, I'd like to acknowledge the support of my advisor, PhD Paolo Bernardi, who has guided me from the beginning of this PhD Thesis and also for creating collaboration opportunities with valuable companies and research institutions. I'd also like to deeply express my sincere gratitude to the CAD group leader, Professor Matteo Sonza Reorda, who has generously spared his valuable time to give me support with his expertise, not only guiding me promptly through this research, but also for motivating me to finish this thesis at Politecnico di Torino.

I would also like to thank all of the CAD group members with whom I had the honour to work: Ernesto Sanchez, Michelangelo Grosso, Giovanni Squillero, Luca Sterpone, Lyl, Niccolò, Davide, Fabio, Marco Desogus, Marco Gaudesi, Salvatore, Riccardo, Gabriele, and many other assistant professors and students.

To Alberto Bosio, Patrick Girard, and all of the staff and PhD students at *Laboratoire d'Informatique, Robotique et Microelectronique de Montpellier* (LIRMM), I'm grateful for the collaboration opportunity in 2011 at this important research institution that contributed with important knowledge for part of this PhD thesis.

To Paolo Rech, who has shown to be a sincere friend during the bilateral collaboration projects at LIRMM in 2011 and at *Universidade Federal do Rio Grande do Sul* (UFRGS) in 2013.

To my friends Joaquim Detoni, Pedro Gusmão, and Balbina Mrozkiewicz, who have also undertaken the challenge of studying and living in Italy. I'm grateful for their friendship that helped me face this challenge with perseverance during 7 years living in Turin.

I'm thankful for all the sincere friends I made in Italy during my Master and PhD studies, that shared my suffering from the difficulty of living abroad, sometimes taking me to hospitals or lending money whenever needed, and also sharing pleasant times.

Last but not least, my immense gratitude for all of those that waited selflessly patient while I was unavailable working on my PhD: my mother Cleci, my father Robson, my brother Felipe, my girlfriend Camila, my aunts, my cousins, and all of my undergraduation and high school friends that were always helping me in some form.

Contents

List of Figures

List of Tables

List of Publications

1. **De Carvalho M.**, Sabena D., Sonza Reorda M., Sterpone L., Rech P., Carro L. (2014) *Fault Injection in GPGPU Cores to Validate and Debug Robust Parallel Applications*. In: IEEE 20th International On-Line Testing Symposium (IOLTS), Platja d'Aro, July 7 - 9, 2014. pp. 210-211

2. **M. De Carvalho**,P. Bernardi,E. Sanchez,M. Sonza Reorda,O. Ballan (2014) *Increasing the Fault Coverage of Processor Devices during the Operational Phase Functional Test*. In: JOURNAL OF ELECTRONIC TESTING THEORY AND APPLICATIONS, April, 2014 vol. 30, pp. 317-328. - ISSN 0923-8174

3. **M. De Carvalho**, P. Bernardi,E. Sanchez,M. Sonza Reorda,A. Bosio,L. Dilillo,M. Valka,P. Girard (2013) *Fast Power Evaluation for Effective Generation of Test Programs Maximizing Peak Power Consumption*. In: JOURNAL OF LOW POWER ELECTRONICS, vol. 9, pp. 253-263. - ISSN 1546-1998

4. **M. De Carvalho**, P. Bernardi,E. Sanchez,M. Sonza Reorda,O. Ballan (2013) *Increasing fault coverage during functional test in the operational phase*. IEEE 19th International On-Line Testing Symposium (IOLTS) 2013. pp. 43-48

5. P. Bernardi, L. Ciganda, **M. De Carvalho**, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, O. Ballan (2012) *On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors*. In: IEE 17th European Test Symposium (ETS).

6. P. Bernardi,**M. De Carvalho**,E. Sanchez,M. Sonza Reorda,A. Bosio,L. Dilillo,P. Girard,M. Valka (2012) *Peak Power Estimation: A Case Study on CPU Cores*. In: IEEE 21st Asian Test Symposium. pp. 167-172

7. M. Valka, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, A. Virazel, E. Sanchez, **M. De Carvalho**, M. Sonza Reorda, (2011)*A Functional Power Evaluation Flow for Defining Test Power Limits during At-Speed Delay Testing*. In: IEEE 16th European Test Symposium (ETS). pp. 153-158

8. **M. De Carvalho**, P. Bernardi, M. Sonza Reorda, N. Campanelli, T. Kerekes, D. Appello, M. Barone, V. Tancorre, M. Terzi (2011). *Optimized embedded memory diagnosis*. In: IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS). pp. 347-352

9. Campanelli N., Kekeres T., Bernardi P., **De Carvalho M.**, Panariti A., Sonza Reorda M., Appello D., Barrone M. (2010) *Cumulative embedded memory failure bitmap display & analysis*. In: 2010 IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Vienna (Autria), 14-16 April 2010. pp. 255-260

10. **De Carvalho M.**, Bernardi P., Sanchez E., Sonza Reorda M. (2010) *An enhanced strategy for functional stress pattern generation for system-on-chip reliability characterization*. In: 11th International Workshop on Microprocessor Test and Verification 2010, Austin (USA), 13-15 December 2010.

Introduction

Everyday we rely on many different technological appliances for our well-being, ranging from mobility (cars, planes, etc) to commodities (cellphones, tablets, etc). From time to time, the capitalist market demand even smaller devices with enhanced features outperforming its ancestor. In order to be time and cost effective attending the urgent demand, it is more convenient to constantly decrease semiconductor technology, allowing integrating macro discrete components on a single micro device made of silicon, namely *microchip*.

Since the first manufactured microchip in 1958 [1], semiconductor manufacturing technology (capability of producing the smallest conducting line width) has scaled according to Moore's Law, from 600nm to 22nm (2012) [2]. Moore predicted that the number of transistors (the smallest working component in digital chips) doubled every 18 or 24 months. Nowadays, *Very Large Scale Integration* (VLSI) technology can place upto *billions* of transistors on a single die, only achievable thanks to the standard cell library design used by the VLSI method. In practice, it can integrate a full working computer composed of one or more *Central Processing Units* (CPU cores), many peripherals, different memory cores (SRAM,DRAM,ROM,FLASH), and other processing units like: *Digital Signal Processing Units* (DSP) and *Graphic Processing Units* (GPU). Often, these embedded modules are bought from specific *Intellectual Property* (IP) vendors that rarely provide the design hardware description code, thus, increasing even more test complexity; test pattern generation and application. Restrict embedded modules have restrict specific test application parameters, which makes silicon validation a challenging task for test engineers that must elaborate a correct and effective testing scheme in a suitable time. Testing comprises not only on developing reliability tools, but also programming creative and effective test patterns and guaranteeing the device's functions integrity according to specifications, often in off-line and on-line modes. Therefore, after 60 years of technology scaling, chips reaching to technology dimension limits tied to architecture complexity, *Integrated Circuit* (IC) manufacturers observed a reliability loss in their products. In such context, test and diagnosis has been thoroughly studied to improve test coverage and application as well as fault analysis and consequently aiding on manufacturing processes for improving product quality, yield, and reducing material costs.

Moreover, the development cost per transistor decreased over the years due to technology scaling. On the other hand, architecture complexity of semiconductor devices increased overall test cost. More specifically, complex miniaturized devices requires more

test time, specific hardware and software tools, qualified test engineers, and specialized equipments, i.e. Automatic Test Equipments (ATE) contributing on test costs. In order to be cost effective, IC manufacturers and research institutes proposed embedded test modules in the device itself. Rather than increasing the ATEs functions and performance, it was more convenient increasing chip area by introducing test structures capable of improving test coverage and performance during manufacturing tests. Several *Design-for-Testability* (DfT) on-silicon structures have become IC industry standards, like: *Scan-chain design*, *Software-Based Self-Test*, *Built-in Self Test*, *Memory Built-in Self-Test*, and *Boundary Scan design*. They basically aid on test digital modules, since analog designs are still crafted manually by back-end designers and do not benefit from standard methods.

The invasive effect of embedding IC's on everyday appliances also reached safety-critical automotive and aerospace applications. However, several automotive catastrophic failures occurred as an immediate counter-effect of this phenomenon, concerning authorities. As a response to these incidents, car manufacturers proposed the ISO26262 functional safety standard for road vehicles [3] demanding a number of auditing E/E processes increasing test strictness for improving reliability. In practice, all the ICs controlling critical parts (i.e. *Anti-Block Brake System* (ABS), *Electronic Power Steering* (EPS) controller, and *Suplemental Restraint System* controller (SRS) the popular Airbag) need to be strictly tested during manufacturing tests and constantly tested during on-line tests, in order guarantee reliability and consequently mission safety.

Reliability is the keyword that motivates the development of this PhD. thesis because technology is rapidly diffused throughout the world in a fast competitive market where failures are unacceptable, especially for systems employed in safety critical missions like automotive, medical, and aerospace applications. Therefore, reliability engineering was developed to provide tools to understand and correct mistakes in manufactured devices throughout all of their life phases:

1. Early-life: before shipping the component to the client
2. Operational useful life: when the component is being used in mission by a consumer
3. Ageing or Wear-out: when the physical structure is compromised by plastic stress carried out through operational life and defects start to appear causing failures

This PhD. thesis is made of four parts encompassing test and diagnosis techniques to attend high reliability levels of automotive SoCs and GPUs demanded by the ISO26262 standard. Initially, reliability concepts and state-of-the-art testing and diagnosis literature are described. Then, the proposed methods are thoroughly detailed: Embedded memory diagnosis, Functional Test Pattern Generation for embedded CPUs, and GPU Fault Injection tool for validating robust parallel applications.

This PhD. thesis is structured as follows:

Part I is composed of two chapters encompassing background on reliability concepts and state-of-the-art literature on testing and diagnosis of hardware modules embedded in SoCs:

In Chapter 1, the main concepts of "Reliability" applied to semiconductor devices are detailed. It mainly describes most of the mature testing and diagnosis procedures of microelectronic devices from early-life to end-of-life in order to achieve high reliability levels. Finally, functional safety standards for high reliable electronic components are described.

In Chapter 2, "State-of-the-Art test and diagnosis efforts" found in the literature are reported. This chapter allows understanding the knowledge barrier on new CMOS architectures testing methodologies, their weaknesses, and solution that need to be adopted in order to cover insurgent issues.

Part II composed of chapter 3 explains the "Embedded Memory diagnosis" proposed solutions aiming at improving failure analysis and locate root cause defects arising from manufacturing errors. Embedded memories are the most dense devices inside SoCs and usually present a high defect level rate. Nonetheless, they are used in the final product because there are several mechanisms for memory array correction, which is transparent to the functional mode. In this chapter, a proposed java tool is described. It has the capability of analyzing test data and applying two diagnosis techniques for better understanding the memory defects:

- Cumulative Memory Analysis; first reconstructing physical memory representation from the logical one, then visualizing defect locations in the memory array by analyzing test results
- Optimized Memory Analysis; from the test results (either full or partial), the method is able to map failing syndromes of known locations to fault models based on a cumulative fault dictionary

In Part III, this PhD. thesis proposes three methods for "Functional test pattern generation of embedded CPUs" in SoCs. Usually, functional pattern generation is a time consuming task, thus in order to be time efficient, we propose manual and automatic generation approaches aiming at:

- Chapter 4: Increasing workload stress for improving *Operational-Life Tests* and *Screening*
- Chapter 5: Increasing functional peak power for limiting power during structural testing, thus avoiding circuit over-killing
- Chapter 6: Increasing functional coverage during the functional test operational phase.

Part IV is composed of Chapter 7 describing the benefits of a proposed "Fault injection tool for validating robust *General Purpose Graphic Processing Units* (GPGPU) programs" used in safety-critical applications. Automotive GPGPU applications require the practices of software robustness even when running on robust hardware. However, validating robust GPGPU applications is a challenging task, since the application engineer does not own the GPGPU design and resorts to costly and time consuming radiation tests. Thus, in this chapter we propose a cheap and fast GPGPU fault injection mechanism able to corrupt applications, by inserting bit flips in strategic places to emulate *Single Event Transient* (SET) and *Single Event Upset* (SEU). In particular, we propose to use the tool to validate robust GPGPU applications, thus activating redundant techniques they possibly embed and observing fault propagation.

In Chapter 8, "Conclusions" are drawn.

In Chapter 9, "References" used in this thesis are listed.

Part I

Background

1. Reliability on Semiconductor Devices

This chapter describes reliability elements: reliability characterization, defects to failures, fault models, test types, test patterns, and finally the negative consequences of tests.

Reliability is the characteristic of an item, expressed by the probability that the item will perform its required function under given conditions for a stated time interval. From a qualitative point of view, reliability can be defined as the ability of an item to remain functional. Quantitatively, reliability specifies the probability that no operational interruptions will occur during a stated time interval. This does not mean that redundant parts may not fail, such parts can fail and be repaired (without operational interruption at system level). The concept of reliability thus applies to non-repairable as well as to repairable items. To make sense, a numerical statement of reliability must be accompanied by the definition of the required function, the operating conditions, and the mission duration. In general, it is also important to know whether or not the item can be considered new when the mission starts. An item is a functional or structural unit of arbitrary complexity that can be considered an entity for investigations. It may consist of hardware, software, or both and may also include human resources. Often, ideal human aspects and logistic support are assumed even if the term system is used instead of technical system.

For the microelectronics area, the reliable item must be a hardware system composed by countless different transistors whose transitions, when interleaved, perform a desired operation. This functional operation must be correct or corrected in time to be acceptable for a system user throughout a stated time interval under given conditions. Thus the whole system's reliability is dependent on its weakest component.

The proper definition of ***reliability*** stated by the IEEE standard dictionary [4] is:

"The ability of a system or component to perform its required functions under stated conditions for a specific period of time"

Reliability can be mathematically expressed as:

$$R(t) = e^{-\lambda t} \tag{1.1}$$

where: R is the reliability of a component working properly as a function of time t. λ is the constant failure rate extracted from a population of the same device.

From a statistical point of view, exponential distributions are best suited to represent the reliability of a device when the failure rate is constant [5]. However, the failure rate is not constant throughout different life phases. In this case, the reliability of a given device changes in three life phases: Early-life, useful-life, and end-of-life. The bathtub curve drawn in Figure 1.1 shows this variation: In Early-life, the failure rate λ is high but decreasing, resulting on a large amount of *infant-mortality* components. Then, on the useful-life phase, the failure rate is relatively low and constant, that's where equation 1 applies. In this phase, the component is being used in applications during operational mission mode. Finally, the last phase is where components start presenting problems due to wear-out, and λ is low but increasing. Analogously, these life phases happen on living organisms: they born, live, and die.

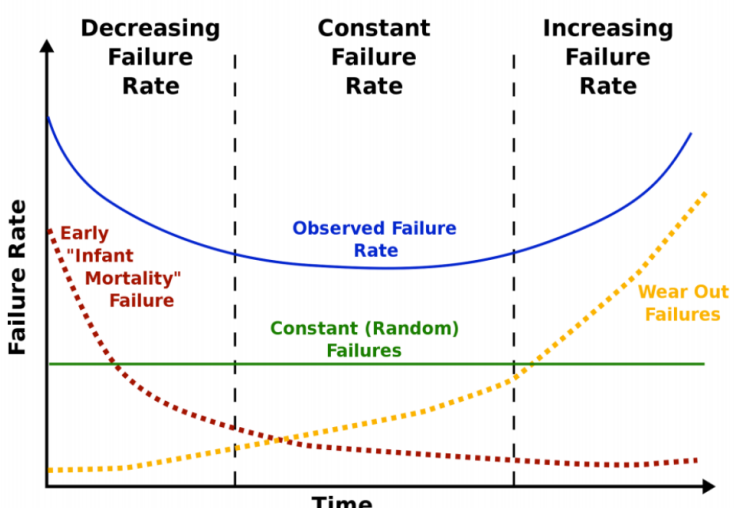

Figure 1.1: Bathtub curve

1.1 Reliability Characterization

Reliability characterization of semiconductor devices provides reliability and electrical measurements relevant to determine the physical failure rate throughout the component's entire life-cycle. It is fundamental to understand the accuracy and qualification of the advanced materials being used and the device's manufacturing processes adopted. This is very important on devices that work on critical environments like aerospace and automotive ones.

The purpose of ***reliability characterization*** is to develop methods and tools to evaluate and demonstrate reliability, maintainability, availability, and safety components, equipment, and systems, as well as to support development and production engineers in building these characteristics [6][7][8][9]. In order to be cost and time effective, reliability

engineering must be integrated in project activities, and support quality assurance and concurrent engineering efforts.

Thus, reliability characterization also targets the correct definition and reducing of failure rates in order to keep high quality in systems by keeping them reliable, maintainable, available, safe, and confidential.

1.2 Defect, Fault, Errors, and Failures

The design process VLSI semiconductor devices depend on the effort of many engineers and technicians. A design becomes a physical device after manufacturing processes are performed that accounts on several tasks and materials in order to become the final product. In general, in a manufacturing lot, a single design is replicated several times during the manufacturing process such that costs are reduced. In most cases, process variations and fabrication mistakes (i.e. air impurities, mask misalignment, bad process tuning, and etc) make twin devices different one from another. They can be slightly different producing the desired outputs or not. In the latter case, the final product had a defect introduced by some source that modified the physical structure, permanent or temporary, producing a fault that may be perceived as an error by other logic and possibly resulting into a system failure, not working as it should, if propagated to the primary outputs. So, the logic is: Defects produce faults that may or may not produce a function error that will propagate to the output provoking or not a system failure.

1.2.1 Defects

According to [10], **defect** in electronic systems is defined as:

"An unintended difference between the implemented hardware and its intended physical design."

A defect can be an impurity on the silicon structure connecting a wire to a voltage source, or ground as well as other wires resulting on a permanent erroneous behaviour. Usually, in VLSI devices, they cannot be mapped to a single fault, as they may produce different behaviours. For example, a microscopic misplaced metal particle connecting an output to a voltage source produces a stuck-at-1 (stuck-at-high), while if connected to ground, the output results on a stuck-at-0 (stuck-at low). Although it is the same defect, a misplaced metal fragment, it produces different situations resulting different faulty behaviors. For this reason, defects cannot be directly mapped into specific faults.

Moreover, there are specific malformed structures that are translated into transition delay defects. That is, cases like high metal resistivity or missing contacts may produce slow transitions, making a signal work as expected but slower. For this situation, there are

two types of behaviour stemmed from transition delay defects: Slow-to-rise and slow-to-fall. The signal is able to switch from one logic level to another, but in a longer time than specified. If the delay is high enough, then the logic reading the value can perceive as a stuck-at fault as mentioned above.

Another situation that may happen, defects may be induced by some other source coming from the environment like: radiation particles, magnetic fields, high or low temperatures, and physical stress. In this case, the defect is transient and solely modifies the physical structure (or its functioning) momentarily, just enough to produce a transient fault or not, as explained in the first paragraph of this section.

The most common defects are:

1. Process Defects: missing contact windows, parasitic transistors, oxide break-down, etc.
2. Material Defects: bulk defects (cracks, crystal imperfections), surface impurities, etc.
3. Age Defects: dielectric breakdown, electromigration, etc.
4. Package Defects: contact degradation, seal leaks, etc.
5. Environment-induced Defects: radiation particles, humidity, vibrations, etc.

1.2.2 Faults

Faults arise from defects and there is not a precise direct mapping from one to another, as explained in the last subsection. According to [10], the definition of **fault** is:

A representation of a defect at the abstracted function level

In addition, a small set of faults can represent a large set of defects. In practice, there are more defects than faults, thus a single fault can be represented by different types of defects. However, defects can only be detected but not accurately identified by analyzing the faulty behaviour of the logic. The possible listed faults observed in the past years has been classified by Avizienis et al. [6] and is shown in figure 1.2.

1.3 Fault Models

In engineering, complex projects are primarily modelled then simulated before actually being produced. Modelling refers to developing an approximation of a real item, whereas simulation is the action of stimulating the model with inputs (derived manually from specifications) and observe the functional behavior. Thus, simulating models allows correcting or improving the design before being actually manufactured.

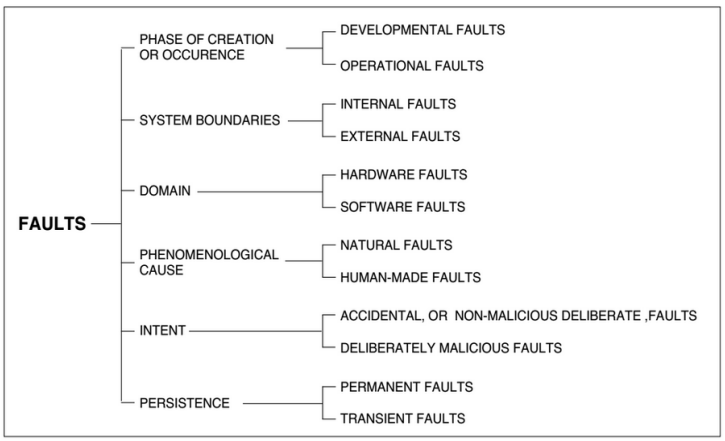

Figure 1.2: Fault classification

Models bridge the gap between the physical reality and mathematical abstraction [10]. They allow the development and application of analytical tools. They are thus essential in design. The most important models in testing are those of faults.

Fault models are engineering models of a failing transistors, circuits, wires, memory cells, that could manifest failure and is unable to make the design function properly. The fault models are used to predict consequences and localization of a particular fault, allowing improving the fabrication process accuracy [11].

This PhD thesis uses the fault models for circuits and memories listed in the following list:

a. Stuck at 0 (SA0): the wire/cell's content is always 0, even if a write 1 has been performed.

b. Stuck at 1(SA1): the wire/cell's content is always 1, even if a write 0 has been performed.

c. Rising Transition $\uparrow$ / 0 (TF1): the wire/cell fails to undergo a $0 \rightarrow 1$ transition.

d. Falling Transition $\downarrow$ / 1 (TF0): the wire/cell fails to undergo a $1 \rightarrow 0$ transition.

e. Coupling fault Inversion (CFin): a transition in one wire/cell inverts the contents of a second wire/cell.

f. Idempotent Coupling (CFid): a transition in one wire/cell forces the contents of a second wire/cell to a certain value (0 or 1).

g. Bridging (BF): this fault model involves any number of wires/cells and is caused by a logic level rather than a transition. It normally consists of a galvanic connection between two (or more) wire/cells, or lines and a state of a line.

h. State Coupling (SCF): a coupled wire/cell or line is forced to a certain value, when the coupling wire/cell or line is in a given state.

i. Active Neighbourhood Pattern Sensitive (ANPSF): the content of a wire/cell, or the ability to change its contents, is influenced by the contents of other wires/cells in the circuit/memory. ANPSF corresponds to the base wire/cell changing its contents due to a change in the values hold by the neighbourhood wires/cells.

j. Passive Neighbourhood Pattern Sensitive (PNPSF): the content of the base wire/cell cannot be changed (i.e., it cannot make a transition) due to a certain neighbourhood pattern.

1.4 Testing semiconductor devices

Before describing the different tests applied to silicon devices, it is necessary to highlight the difference between verification and test [10] [12] [13]:

- **Verification** is the process of testing the design functions according to specifications before product manufacturing
- **Test and Validation** is the process of testing the design hardware by identifying the localization of potential defects caused by manufacturing errors, environment-induced errors, design errors, and ageing effect errors. This process of testing is performed after product manufacturing.

Semiconductor tests aims at identifying defects caused by manufacturing errors, process variation, and ageing effects, verifying whether the device's physical structure is corresponding to the design model, and consequently the functions are compliant to the specifications. Deep sub-micron technologies has shown to be rather susceptible to all sorts of noise and process variations, thus often constant testing is a requirement.

To guarantee the reliability of manufactured microelectronic devices, especially on those employed in safety critical applications, distinct tests are applied to the component at different production/life stages. Each test has its own test pattern targeting specific goals. Table 1.1 lists the tests applied to the semiconductor at several different stages/locations during manufacturing and operational phases. Moreover, suitable test patterns are carefully selected/developed during test application at each stage, thus guaranteeing the components' quality in high reliable applications.

From table 1.1, this thesis explains the differences between *Functional* and *Structural Tests* that are performed during *Manufacturing* and *On-line testing*. The following subsections define these concepts using in the reliability characterization process.

Test #	Stage	Test Type	Applied Test Pattern
1	Foundry	Wafer level test	Structural patterns
2	Packaging company	Package level test	Structural patterns
3	Test company	Parametric test	Parametric patterns
4	Test company	Characterization test	Parametric and functional stress patterns followed by structural patterns
5	Supplier/Design and Test companies	Burn-in test	Parametric and functional stress patterns followed by Structural patterns
6	Application development company	Acceptance test / Incoming inspection	Structural and Functional patterns
7	Consumer	On-line test	Software-Based Self-Test (SBST) pattern during the operational mode

Table 1.1: Types of tests in semiconductor devices

1.4.1 Functional vs. Structural Tests

Functional tests uses the circuits functional inputs to stimulate internal logic and observe results at the functional outputs [10] [12]. Usually, test patterns are manually written by verification engineers from the specifications. However, there are Functional *Automatic Test Pattern Generators* (ATPG) that construct a set of test patterns to completely exercise the circuit functions, which can be very naive and inefficient when targeting fault detection [10]. For example, if we consider stimulating all of the functional inputs of a three-operand 64-bit adder (two inputs and one output), the result would be a functional test pattern containing 2^{n*2} patterns. Where *n* is the quantity of bits in each operand, in the example is 64, and $+64$ because we have 2 input operands. The number of functional input patterns is more than 3×10^{19} and would take an order of magnitude of about 10^{22} years to be applied using a fast 1*GHz Automatic Test Equipment* (ATE).

Nevertheless, functional test patterns can be very effective on CPU cores when targeting different test targets; for example, functional power consumption and stress, software related faults, activation of mitigation techniques used for *dependable* systems, and etc. The main drawback of functional testing is pattern generation time that scales exponentially with the number of inputs and the sequential depth of circuits, i.e the number of memory elements embedded in the *Design-Under-Test* (DUT) [10].

Structural test, on the other hand, only exercises the minimal set of stuck-at faults on each line of the circuit to test the physical structure of the device [10] [12]. Structural tests are much more efficient than functional ones when targeting physical defects. It is thus, an important test for proving whether circuit physical integrity is structurally correct

as designed. It does not prove that the circuit functions are designed correctly according to specifications [10]. Considering the previous 64-bit adder example, structural tests are so powerful towards fault detection that, after discarding equivalent faults, each bit-slice in the adder would only have 27 faults. In this case, we only need 1,728 test-patterns to test the adder. The 1 *GHz* ATE would apply these patterns in 0.000001728 s, and since this test pattern set covers all possible structural stuck-at faults in the adder, it achieves exactly the same fault coverage as the intractable functional test pattern set described above. Frequently, the circuit designer will provide a limited subset of the functional test patterns for the circuit, but those typically cover only 70 to 75% of the total number of faults. Testing for only 75% of the modelled failures is of limited value - it will catch only the most easy to cover defects. Thus, we see the importance of ATPG algorithms. The vectors they produce supplement the functional test vectors from the designer to raise the stuck-at fault coverage to 98% or higher levels.

In complex synchronous circuits containing many sequential elements (latches and FFs), structural tests are even more efficient than functional tests, since they benefit from *Design-for-Test* (DfT) structures implemented in the design specially for aiding structural tests. DfT adds short-cuts in the design by connecting the output of one flip-flop/latch to the input of another until all memory elements in the circuit are connected in a chain, forming a large shift register, namely *scan-chain design*. Then, special test input and output ports are added to introduce a test pattern and read results, respectively. This approach potentials the ATPG's work on generating structural test patterns and eases the ATE test application by reducing the number of inputs and test time, as well as increasing significantly test coverage with respect to functional tests, as explained before.

The main drawback of structural tests are the side-effects it produces. For example, just the test pattern shifting operation in the scan-chain of a CPU core ages the scan elements significantly, especially in deep sub-micron technologies [14][15]. Then, after shifting the pattern through the scan-chain, the actual structural test happens, testing the internal logic which increases switching activity and consequently power consumption in an irresponsible manner, such that premature damage happens even on the most strongest components. Thus making testing inefficient and costly.

1.4.2 Manufacturing, Incoming, and On-line Testing

Manufacturing testing uses deep knowledge of the design, equipments, materials, and communication protocols used in the fabrication process of a semiconductor device. Considering table 1.1, test from 1 to 5 are considered manufacturing tests. They comprise of all tests realized before shipping the component to the application company that will build a working application to satisfy a certain market niche with its product. The types of manufacturing tests are explained in the following paragraphs [10][12][13].

1. *Wafer level test* is the first test applied to the physical semiconductor design. In general terms, the wafer is where a population of components are printed using the manufac-

ture process and the first physical outcome of the design. The wafer is polished and cut into equal rectangular parts limiting boundaries of one device to another, called dies. The wafer test procedure uses a special ATE containing micro-probes that fit perfectly on the dies' micro-pads, allowing special test pattern application and result capturing. Usually, test patterns are functional and structural patterns with the minimal capability to separate dies containing many defects from defect-free ones. The former are marked with a sticker by the ATE identifying dies that needs to be removed, and the latter dies, namely *Known Good Dies*, go to the packaging process. This die separation process is also called *sorting*.

2. *Package level tests* are applied to known good dies after packaging. Packaging is the complicated task of bonding very thin wires to micro-pads of the die to pins that connect to the external world. This procedure is done either manually by packaging engineers or automatically by specialized equipments. Both cases are not error-free as they may have problems like: air impurity touching silicon surface, incorrect bonding, missing or open wires, connecting wires, die misplacement, *electrostatic discharge* (ESD) burning internal logic, corrosion due to high humidity, etc. Package level tests uses the ATE to inject test vectors (functional and structural) and determine good chips after packaging.

3. *Parametric tests* is the procedure of applying high/low voltages and currents stressing the chips pins and the logic, determining whether they work correctly according to electrical specifications. The key issue here is to provide correct limits of electrical stimuli to define if the chip and pins will work under certain conditions or not.

4. *Characterization tests* is usually applied to a random sample of a design production lot, by applying mission-like parametric stresses under high temperatures to quickly age the samples until all of them have failed. This procedure usually takes into account the Arrheinus chemical kinectic theory of semiconductor devices [16]. It is clear that using normal conditions for performing characterization tests would last for years and not days as proposed by the methodology. Characterization tests can also include *Operational Life Tests* where a phase of functional and parametric stresses are applied, followed by stress-free diagnostic tests which uses structural patterns to determine the amount of failed components [9]. This test diagnosis loop allows drawing the bathtub curve shown in figure 1.1 and determining the failure rate at each life phase to compute a more or less accurate reliability value of the production lot devices.

5. *Burn-in or Screening*: Burn-in, as the name suggests, is the process of heating the components at controlled higher temperatures inside an industrial oven for a stated time. It is used in OLTs, but during screening it uses characterization information acquired in the previous tests to age all of the devices of a production lot until they have reached the beginning of their useful-life phase. Then, final structural tests are performed to identify and remove infant mortality components from good ones. Finally, the good packaged devices are sold to the application development company. Most of

the times, the information about the electrical characteristics of the device is given to the application company.

Incoming inspection or Acceptance tests are applied by the application development company. If they are developing safety critical applications like automotive and aerospace, they must guarantee that the application will work according to reliability required standards. During component transportation there still exists a possibility that some components have been damaged internally either by ESD, vibrations, humidity and so forth. In this case, the application company can trigger limited structural tests embedded in the design whose programming has been given by the component manufacturer. The engineer programs a simple test routine to drive *Built-in Self-Test* (BIST) mechanisms and retrieve results for comparison. In other cases, this task is not possible due to manufacture secrecy and the application engineer must program manually functional patterns targeting functional error detection of the device only for the parts the application uses, otherwise test programming will be very time consuming. The failure rate in this phase should be low and constant as the device is matured in its useful life. Usually, the application company, client of the device manufacturer, can compare reliability figures obtained from manufacturing test information and the ones obtained by its own company during inspection tests.

On-line tests are applied to the device when already in mission, and all the manufacturing information is not accessible as well as embedded test structures like scan-chain, BIST, MBIST. DfT are mainly used during manufacturing testing, and besides being physically incorporated to the device, it is forcefully turned-off by the application company by soldering test pins to fixed values when in mission mode. This completely eliminates DfT structures from working in test mode. On the other hand, some international standards determine that a component controlling critical parts in safety critical applications should be constantly tested without disturbing normal operation during mission, as it can provoke catastrophic failures. Moreover, on-line testing allows retrieving information to the application company from activities acquired during mission mode and helps improving the application itself. Also, it allows warning end-users about imminent failures before they actually happen.

1.5 Reliability Requirement Standards for High Reliable Electronic Systems

The insurgence of even smaller and efficient microelectronic devices in safety-critical applications has concerned governments, authorities, and overall customers, due to possible reliability problems. Therefore, several standards were defined in order to guarantee mission safety by determining the usage of high reliable electronic components controlling critical parts in automotive and aerospace applications. Automotive critical parts can

be: air-bag, braking system, steering wheel, advanced assistance driving. In aerospace: take-off, landing, movement, and computation.

The International Electrotechnical Commission (IEC) develops and maintains international standards that provide systematic methods and tools for dependability assessment and management of equipment, services, and systems throughout their life cycles [17]. It defines dependability as:

"the collective term used to describe the availability performance and its influencing factors : reliability performance, maintainability performance and maintenance support performance" [18]

Reliability is directly correlated to safety, while the first tries to improve costs targeting more profit, the second assesses the safety quality demanded by standards. Usually, high reliability necessary attain high safety levels [8], thus ensuring protection against any hazard affecting the system, namely functional safety. IEC 61508 [19] defines specific rules for assessing functional safety in electrical/electronic safety-related systems.

Reliability has become a problem when dealing with new products in safety-critical applications, therefore, each field is adopting its own standard. In particular, car manufacturers developed the ISO 26262 standard [3], which is an adaptation of the IEC 61508 for Automotive electric and/or electronic (E/E) Systems in series production passenger cars. The standard addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems and their interactions. It demands a number of auditing processes during the whole product's lifecycle, to ensure high reliability and mission safety throughout the system useful life. Remarkably, in mission testing for error detection demands the adoption of on-line self-test technique as an essential test process in critical E/E vehicle parts.

Other examples of application specific standards dealing with Reliability, Availability, Maintainability and Safety (RAMS) issues are:

- Railway E/E equipment and programmable electronic systems: CENELEC EN 50126 [20]
- Aerospace hardware and software electronic systems: DO-254 [21], DO-178C [22] , ARP4754 [23]
- Nuclear Power Plants: IEC61513 [24]
- Medical Systems: IEC62353[25]

2. State-of-the-Art testing and diagnosis efforts

This chapter describes the state-of-the art testing and diagnosis techniques for reliable semiconductor devices, especially for SoCs. Initially, SoCs are described and then its internal modules; CPU, memories, and GPUs.

2.1 System-On-Chip

The need for even smaller, faster, power efficient, improved functions of semiconductor devices tied to technology scaling, led to the development of System-On-Chips. They include a full working computer with ***CPU***, ***memories***, peripherals, and ***image processing units*** on a single die. With the advent of SoCs, integrating many hardware functions on a single die consequently reduced testing costs by integrating standard design and test methodologies applicable for any internal logic.

Figure 2.1 shows the internal architecture of the 32-bit CPU LEON SoC currently owned by AEROFLEX, but originally developed by the European Space Research and Technology Centre (ESTEC) part of the European Space Agency (ESA). It has embedded a SPARC V8 RISC CPU core, memory cores (SRAM, DRAM, and Flash), communication and processing peripherals (USB, UART, DSP, etc), and a Graphics Processing Unit (GPU) accounting on many serial multiprocessors (SM) containing hundreds of processing elements for parallel computation. This device has been used in aerospace applications with radiation hardening (radHard) hardware. This SoC architecture presented is a **good example** for showing, on a single device, the target modules (in red) this PhD thesis focuses.

Although testing has many issues that need tackling, this PhD thesis concentrates on embedded memory diagnosis, functional pattern generation for testing CPU cores, and fault injection in GPGPUs. The following sections describe and list the state-of-the-art testing and diagnosis efforts realized in the past.

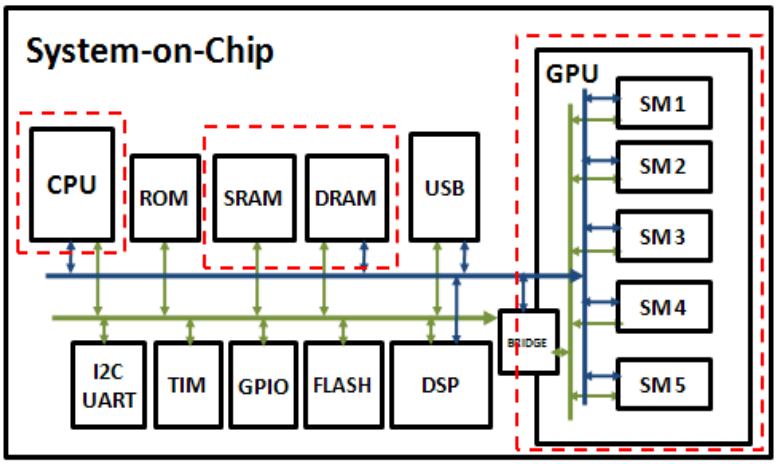

Figure 2.1: SoC Architecture

2.2 CPU

CPUs are the processing units responsible for controlling the SoC. It controls all the SoC peripherals by executing instructions previously programmed by engineers. Usually, instructions are placed in the instruction memory, and are decoded and executed by the CPU controlling the datapath to manipulate application data; whether external or internal information, finally restoring to the data memory as the final outcome. In general, SoCs have the potentiality to execute a *Real Time Operating System* (RTOS) to concurrently manage peripherals, software implemented algorithms and applications.

Testing digital CPU cores and peripherals can be rather simple during manufacturing tests by resorting to mature techniques (*Design-for-Testability*) developed throughout the years:

a. *Boundary and Scan-Chain Designs* [26] [27]: Joint Test Access Group Port is the common name for the industry standard IEEE 1149.1 and IEEE 1500. It adds a special test access port (TAP) and internal test modules to make structural testing easier, faster, and efficient and has the potentiality to exploit hardware and software debugging. Basically, the methodology exploits internal memory elements (Flip-Flops and Latch) to easily insert structural patterns for testing physical internal wires and logic gates during **manufacturing testing**. Flip Flops are slightly modified for being capable of operating in test and functional modes, by inserting a multiplexer on its primary input and adding a test mode selection. The multiplexer takes as inputs the functional inputs and the output of another FF. In test mode, all FFs are linked on a chain, where

the output of one element is the input of another. This sort of design is named scan-chain. It allows not only testing the CPU but also peripherals, however, the design must consider modifying the SoC such that test and functional modes do not interfere with one another, namely DfT design rules. Moreover, the standard defines boundary scan strategy able to test boundary pads as well as testing many devices in sequence.

b. *Built-In Self-Test (BIST)* [10] is a type of test that can be used for manufacturing, acceptance, and on-line tests. It employs a *Linear Feed-Back Shift Register* for generating random stimuli which is applied to the internal logic that produces execution results and stores them in a *Multiple Input Signature Register* (MISR). By analyzing the contents of the MISR, the test is capable of identifying possible failures of specific modules. In the literature, there are several works exploiting BIST during on-line and off-line modes [28][29][30].

c. *Software-Based Self-Test (SBST)* patterns are useful for testing the CPU cores embedded in a SoC. Usually, they correspond to a set of assembly routines able to apply the correct test vectors in the correct spots while the device is running in functional mode. SBST techniques for testing embedded CPU cores in SoCs during the operational phase may be preferred to hardware techniques like BIST (Built-In Self-Test) because they:

   - Are non-intrusive to the hardware and minimally intrusive for the application software
   - Run in the CPU core by using the instruction set, thus exercising it exactly in the same operational conditions of the normal applications
   - Provide higher defect coverage, since they run at-speed
   - Do not use DfT structures
   - Can be run in-mission by minimally disturbing the normal operation (provided that correct software isolation is guaranteed)
   - Do not impact significantly on performance since they do not add hardware in the critical path.

   In figure 2.2, a simple System-on-Chip (SoC) architecture and the relevant resources used by the SBST method are shown. The SBST routines capable of testing the CPU components for stuck-at faults are placed in the instruction memory. The CPU loads (1) and executes (2) the SBST routines which will stimulate each of the CPU components targeting stuck-at faults. Results produced by the internal logic will typically be written to registers inside the CPU and then saved into the data memory (3) at the end of the test. The final step is to compare the golden values with the results written by the test in the data memory. Such comparison determines whether the test was successful or not (i.e., some fault was detected). In the latter case, the SBST may also be able to determine which module is faulty.

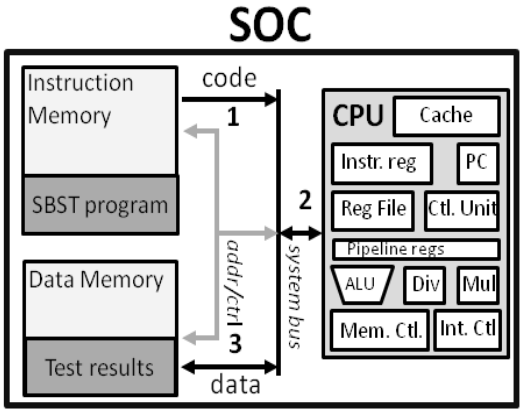

Figure 2.2: Embedded CPU under SBST testing

In [31] [32] the authors proposed some techniques to develop SBST routines which are effective during on-line testing providing a high level of fault coverage in a couple of case studies based on the MIPS architecture. They target most of the visible blocks that can be seen by the programmer. In some cases, they can also provide good results for the invisible blocks or the sparse logic that are essential for the correct functioning of the component. However, the proposed method is not able to cover all the faults of a CPU core embedded in a SoC. Detecting the untested faults sometimes may be crucial for safety-critical applications.

SBSTs have been used in industry and academy cases [33] [34] [35] [36] [37] for more than 30 years. The main drawback of SBSTs is the time required to generate them. Since they are functional patterns, it can be very difficult to stimulate the whole circuit targeting the controlability and observability of stuck-at faults. Moreover, it is practically impossible to navigate through all of the CPUs states aiming at test coverage. In this case, this state-of-the art methodology insurgent in the recent years [31] [32] cannot trigger corner cases in on-line mode itself. In chapter 6, we propose methods for quickly generating SBST patterns as well a specific methodology for testing corner cases.

2.3 Memories

Memories are the most dense semiconductor devices and are subject to many types of process failures, from design to manufacturing. Moreover, these particular devices are becoming increasingly used in miniaturized SoCs with enhanced capabilities. Design complexity and density tied with technology scaling results in some manufacturing error that sometimes are masked when tested. The efforts to make memories more efficient and compact lead to different routing schemes, namely scrambling, especially for SRAM memories. For example, bits of a word line are not placed adjacent to one another but

sparse in different memory locations. The reasons for memory scrambling second to [38] are:

- Geometry optimization introducing folding
- Address decoder optimization
- Cell area optimization by sharing contacts and well areas
- Speed and robustness optimization based on bitline twisting
- Yield optimization by introducing
- Achieving I/O pin compatibility utilizing address or data line swapping
- Data secrecy by natural encipherment introduced by scrambling schema

On the other hand, memory testing has become a challenge and testing techniques has been developed and employed in industry:

a. Memory Built-In Self-Test (MBIST): It is a wrapper module capable of directly manipulating the physical memory array and applying devised test patterns (march tests)[39]. Usually, an external computer communicates with the MBIST to lauch test execution and retrieve test results for diagnosis as proposed in an industrial case [40] .

b. Software implemented Memory Built-In Self-Test (SwMBIST): denote a test solution targeting memories embedded in a SoC, based on performing the test through a suitable program executed by a processor inside the SoC: the program is in charge of executing the sequence of accesses to memory (for both read and write purpose) mandated by a given test algorithm (e.g., corresponding to a March algorithm [39]). Several published works, including those from industry (e.g., [41][42][43]), underline the fact that various memory defects existing in new technologies require the test accesses to be performed at the maximum speed (or at least at-speed) in order to be detected.

Memory testing has become a challenge, since devised test patterns (i.e march elements) including data background [39] for an efficient test was developed to target the main faults, as introduced in chapter 1. Moreover, march tests for non-scrambled memories proposed in[39] and improved for complex memories in [44] cannot be directly applied since logical bits are not adjacent to each other on the physical memory array. An industrial case [40] the authors propose a memory test and diagnosis framework for embedded memories, but does not explore or cite the scrambling scheme nor mapping failing syndromes to improve diagnosis procedures.

In chapter 5, we contribute to the state-of-the-art by implementing a software tool developed in java for an efficient memory diagnosis and failure analysis. The tool first unscrambles the memory according to design parameters in order to obtain the physical

array structure. Secondly, the tool parsers test results and maps failing logical cells to physical ones, allowing understanding error patterns in the physical array. Finally, the tool can map failing syndromes to fault models by utilizing a fault dictionary.

2.4 Graphic Processing Units

General Purpose Graphic Processing Units (GPGPUs) have become popular for processing parallel data more efficiently than CPUs in several domains, from desktop computing to embedded and High Performance Computing (HPC) applications. Unfortunately, GPGPUs have shown to be rather sensible to radiation [45]. Hence, several software mitigation techniques, as well as robust algorithms, are being developed to overcome reliability problems. Although, many GPGPUs embed Error Correction Code (ECC) in all memory elements, they are still prone to radiation effects.

In [45][46] show that many types of radiation may affect GPGPU logic in operation due to *Single Event Transient* (SET), corrupting the logic outcome and producing an erroneous result, and eventually being placed into the memory, thus very similar to a *Single Event Upset* (SEU). The difference amongst both events is that the former affects logic circuits and the latter memory elements. In safety-critical applications soft-data corruption may impact severly on the mission.

In [47] [48] [49] propose software based mitigation techniques for GPGPUs for device availability and safety. The technique is coded in software and is tested in software as well. In [50] [51], the research present software hardening by implementing duplication and redundancy at different levels: thread and time.

On the other hand, the validation of the mentioned techniques is also performed on software and is run on the GPGPU with the application itself. For example, the code is slightly tweaked with the desired error, run on the GPGPU, and results are obtained [48].

Usually, radiation tests are the best form of validating an application since they accurately mimic the application's real environment. In [45] [51] beam experiments on GPGPUs have been performed showing the corruption of GPGPU applications even with radiation hardened hardware.

In chapter 7, this PhD thesis proposes a fault injection mechanism to evaluate the resiliency of an application running on a GPGPU and to validate software hardening techniques it possibly embeds. The proposed approach is based on the usage of existing software debug facilities and is not invasive to the application. Moreover, the developed fault injector has a low application cost, injection time is rather fast, and it uses the actual GPGPU hardware instead of using models. We report some experimental results gathered on selected cases of study to show the proposed approach advantages and limitations.

2.5 Other literature

2.5.1 Functional Test Pattern Generation

Functional test pattern generation can be very easy when the goal is to exercise all the modules' functional inputs and outputs. In this case, as reported in chapter 1, they can be very time expensive not only for generation but mainly for test application. In recent years, the main efforts for generating functional patterns of semiconductor devices is the automatic approach, which exploits heuristic-based algorithms [52] [53] [54] [55]. In particular, an Evolutionary Algorithm-based [EA] functional test pattern generation has been proposed in [56] .

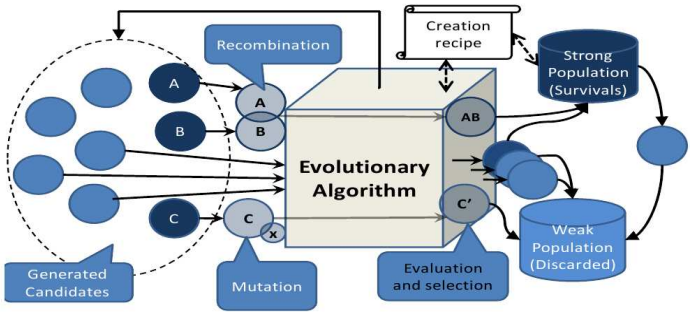

Figure 2.3: EA: Optimization of a population

It employs an EA, the basis of the exploited EA are presented in [57]. Such approach optimizes a population according to a specific metric by imitating the natural process of biological evolution, as shown in Figure 2.3. Following this perspective, a device's functional input data (or assembly program in case of CPU cores) program is an individual of a population which is handled by the tool. The initial population, a set of random functional patterns is iteratively refined mimicking the Darwinian Theory: new individuals are generated by mutation (an individual is slightly modified) or by recombination (two or more individuals are mixed in some way); the best performing individuals are selected for survival. The process is halted after a certain number of steps, called generations, or when a steady state is reached. The best individual is provided as the output. Each individual is characterized by a quality value called fitness, and it is computed at each generation step by an evaluator.

Figure 2.4 shows a possible framework configuration for automatic functional pattern generation for digital semiconductor devices. This framework has been exploited for generating assembly programs for embedded CPU cores, which the program is an individual composed of many assembly instructions. The framework is iterated for several generations mixing assembly instructions to maximize a certain metric. First, the EA generates random assembly programs based on a set constraints that determines the way individuals

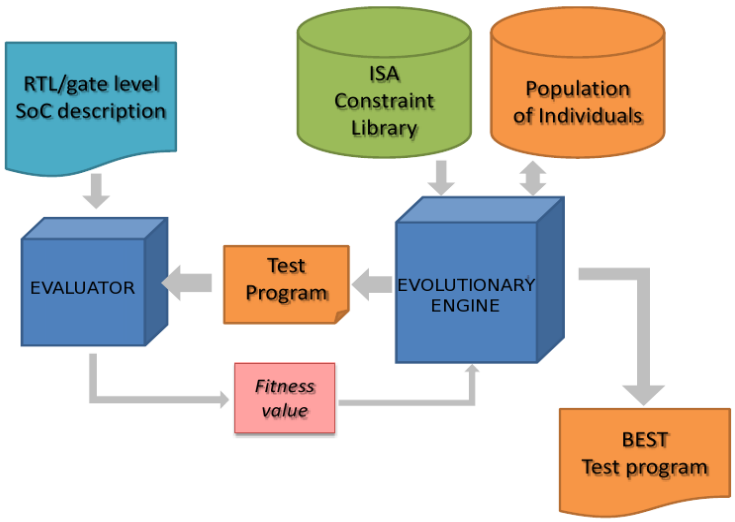

Figure 2.4: Feed-back framework for Automatic Functional Pattern Generation

must be created for a proper CPU stimulation. For example, each individual needs an initial fixed setup for loading the program into the CPU core memory and saving results into memory elements, returning to normal operation. Secondly, the generated program must be evaluated; this is performed by loading the hardware description (HDL) and the generated pattern into the HDL simulator and simulating the device. Then, the evaluator (manually written by the test engineer) extract results from a target metric that the EA must optimize, called fitness. Finally, the EA receives the fitness value and ranks it according to previously evaluated fitness values. By ranking the generated individuals, the EA can then perform optimization using the Darwin theory of evolution: improving good individuals and discarding weak ones. The better the evaluator is written and constraints defined, the quicker and efficient the evolution occurs.

The presented state-of-the-art methodology still needs fast evaluators and constraints refinement to quickly optimize metrics and consequently generate optimal functional patterns towards certain goals. As stated in the Introduction, this PhD thesis improves evaluations and constraints of the above-mentioned tool to quickly generate functional patterns for the proposed goals: functional stress (chapter 5), functional peak power (chapter 6), and fault coverage (chapter 7).

2.5.2 Power Consumption of Embedded CPU cores during Testing

High power consumption during test may lead to yield loss and premature ageing. In particular, excessive peak power during at-speed delay fault testing represents an important issue. In the literature, [14][58] several techniques have been proposed to reduce peak power consumption during at-speed LOC or LOS delay testing. On the other hand, some experiments have proved that too much test power reduction might lead to test escape and

reliability problems. So, in order to avoid any yield loss and test escape due to power issues during test, test power has to map the power consumed during functional mode. In literature, some techniques have been proposed to apply test vectors that mimic functional operation from the switching activity point of view. The process consists of shifting-in a test vector (at low speed) and then applying several successive at-speed clock cycles before capturing the test response.

In [15], the authors propose several flows for evaluating the power consumption when a CPU cores are under test. They use LOC and LOS, pseudo-functional patterns, and functional patterns running on CPU cores and compare their results on different technologies. Results of a couple of test CPU cases synthesized in 90nm and 65nm shows that test power is 14% and 60% higher than functional power, respectively. However, the set of functional patterns used in the evaluation was not developed for augmenting power, but for improving fault coverage and increasing functional stress.

In Chapter 6, this PhD thesis contributes to the state-of-the-art by proposing a methodology for quickly generating functional power hungry patterns for limiting manufacturing tests power consumption. From the generated patterns, we can derive upper and lower bounds of test power consumption such that power is not excessive while tackling *at-speed* performance testing faults.

Part II

Embedded Memory Diagnosis

3. Proposed Memory Diagnosis

This chapter details the benefits of a proposed software tool efficient for diagnosing memories embedded in reliable SoCs used in automotive applications. The component takes a fundamental part on controlling the steering wheel, braking system, and airbag control, and therefore, it shall not present any defects stemmed from manufacturing errors to ageing effects. We present two methods encompassing memory diagnosis for an optimal fault analysis:

1. Cumulative memory analysis technique that takes into account the memory topology and the executed memory test, and returns both syndrome and shape-based failure statistics. Furthermore, it allows the cumulative analysis over many memory cuts inside a die, a wafer or a lot.

2. Optimized memory analysis technique that exploits many levels of knowledge to produce accurate failure hypothesis. The proposed post-processing analysis flow is composed of a many steps investigating failure shapes as well as cell fail syndromes, and includes advanced techniques to tackle incomplete data possibly due to tester noise and/or by faults showing intermittent effects.

The effectiveness of both techniques are demonstrated on an automotive-oriented SoC manufactured in a 90nm technology by STMicroelectronics, which includes embedded *SRAM* memory cores tested using a programmable *BIST*. Unscrambled BITMAPS gives a visual feedback leading to quick physical defect identification.

3.1 Cumulative Memory Analysis

An effective silicon debug and diagnosis process has to be supported by on-chip hardware structures, suitable stimulation equipments and software tools for analysis.

By means of activating the capabilities of on-chip Built-In Self-Test (BIST) modules, a memory can be tested by applying several test algorithms; the memory test algorithm

selection mainly depends on the defects to be tackled; most silicon-debug- and test- oriented BIST engines are programmed accordingly to this decision, possibly also providing diagnostic information [59].

Test algorithm programming, launching, and monitoring during the test process are managed through the test equipment that is finally getting a set of information related to test execution. At the end of the test such information is downloaded to a host machine and then elaborated to isolate failure and to possibly provide their explanation.

Such an off-line analysis has to be performed by software tools taking into consideration the major needs for memory silicon debug and diagnosis, which are the capability of:

- relating the set of extracted faults to the physical topology of the investigated memory to ease the failure analysis
- associating each failing location to the set of test steps producing an error, and recognizing the shapes of the failing areas, such as rows, columns, pairs, clusters, to work out preliminary fault model hypothesis
- enabling the cumulative analysis of a population of embedded memory cuts in order to identify systematic marginality due to imperfections in the design, manufacturing process, or even due to component handling during test and packaging phases.

The following sections details the features that a software tool has to include to tackle the aforementioned issues. In summary, it illustrates how to parametrize the memory physical characteristics, store a minimal amount of information regarding the fault set produced by running a memory test, and which fault grouping strategy is significant to achieve a meaningful fault investigation for a single or a set of devices.

Examples and results reported in the following paragraphs are related to embedded SRAM memories, and a tool for the analysis of memories integrated into the System-on-Chip is described. Results gathered on an industrial case study demonstrate the effectiveness of the tool in a real scenario.

3.1.1 Memory Diagnosis Background

As briefly introduced, a memory diagnosis process involves several components. Figure 3.1 illustrates a complete flow for memory failure retrieval and analysis.

At SoC level, BIST modules enable at-speed memory test application; test selection may be fixed [60] or programmable [61]; BISTs functionality can take into account diagnosis [62] by storing a complete, or partial, set of observed failure behaviours. BIST functionality are controlled by the Tester; it stores a test program that can be constituted by a simple set of commands letting the BIST running or by a more complicated test

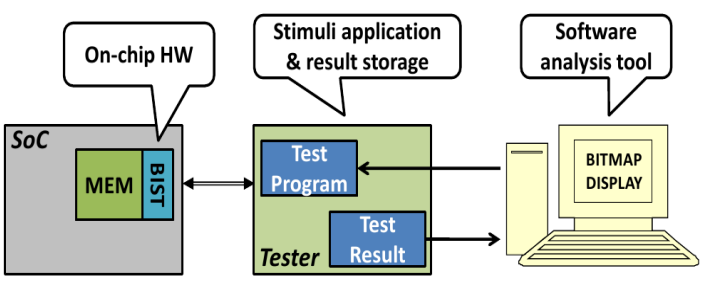

Figure 3.1: Memory diagnosis flow components

description including memory test code upload on BIST in case it is programmable and execution parameter manipulation. In this latter case the diagnosis is performed by the application of consecutive steps, eventually determined by the current observed result. The test is therefore also in charge of the temporarily storage of the test results until it is not completed.

The results collected on the tester are finally retrieved by the host computer to permit their analysis by test engineers and failure analysts. This analysis is crucial to individuation of the failure mechanisms; bitmap tools showing where and how failures appeared are an absolute need for quickly align over suspect memory zone, possibly providing also an initial hypothesis about the fault model.

To be effective towards these issues, bitmap analysis tool should be able to join all the information related to the SoC and Tester characteristics. Figure 3.2 shows the conceptual view for a memory analysis bitmap display tool.

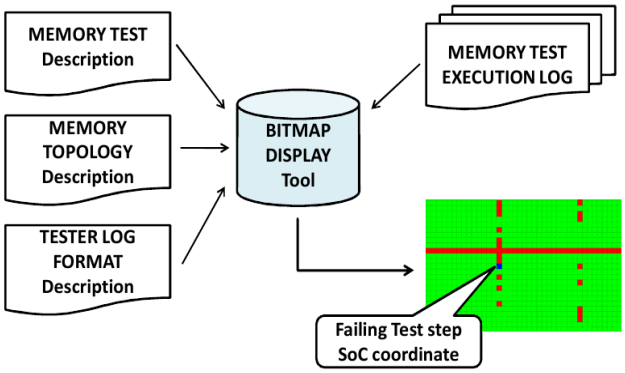

Figure 3.2: Conceptual view for Bitmap Display tool

3.1.2 Proposed Analysis Flow

The proposed methodology for an effective memory failure analysis consists in a set of steps finally allowing displaying the topology of the failing data and providing keys for the following steps of the failure analysis. To achieve this goal, the software environment elaborates the information returned by the tester, basing on a set of parameters describing the physical memory organization and the executed march test.

The implemented data structures minimizes the processing effort and the amount of required memory; such a characteristic permits to quickly elaborate fail statistics over a large base of faulty devices.

Memory representation

The first outcome of an effective tool for memory analysis is to represent the memory under analysis as it is physically structured and positioned. For this reason, the logical vision (usually appearing as a very long cell array whose width corresponds to the word parallelism) has to be translated into the physical one, which is an optimized map whose dimension is determined by a set of parameters constituting the so-called scrambling schema.

Other than the basic information such as the number of memory words (WORDS value) and the number of bits per word (BITS value), a scrambling schema is usually characterized by:

- MUX value: indicates the number of logical words included in a single physical row
- BIT ORDER sequence: describes how the bits of the logical words are organized in a physical row
- MIRROR values: whereas a memory presents a regular structure, memory portions can be horizontally and/or vertically repeated, eventually with an alternate order.

Example 1 - In a typical embedded SRAM scenario, in case the MUX value is N, every physical row is composed of BITS blocks, each one including N bits; block i commonly includes all the ith bits of the N words composing the row. Furthermore, bits in a block are normally organized according to the word address. Figure 3.3 illustrates this basic BIT ORDER sequence information. Additionally, physical row composition may be completed by a couple of SRAM specific mirror parameters (BLOCK MIRROR and BIT MIRROR). The former specifies if any block sequence order inversion is physically implemented, the other concerns the sequence of bits inside a block. Figure 3.4 shows an example with BLOCK MIRROR and BIT MIRROR different than 1.

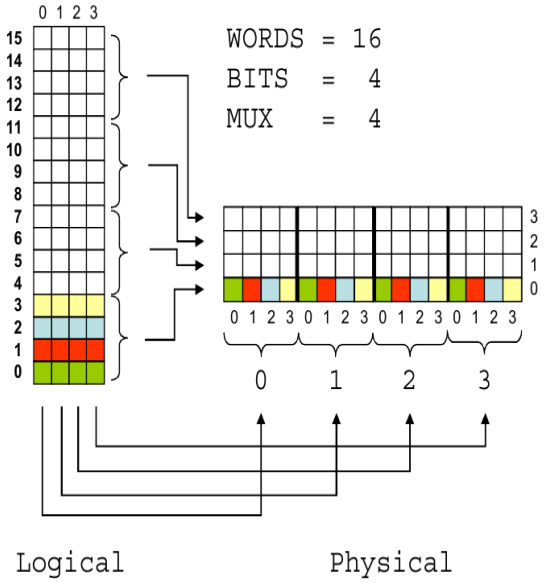

Figure 3.3: Basic mapping between logical and physical SRAM view

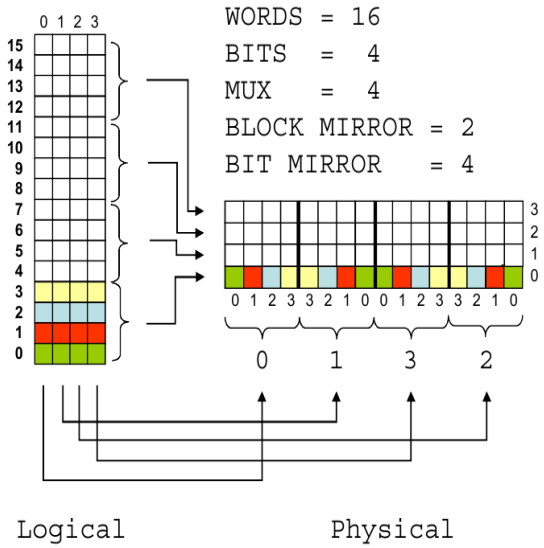

Figure 3.4: Memory view with block and bit mirrors

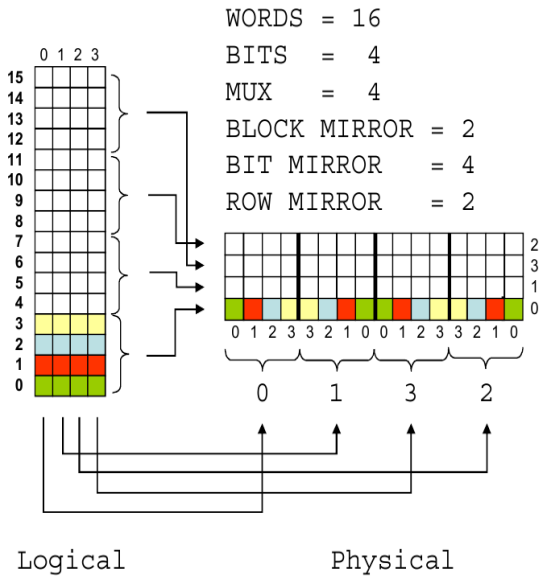

Figure 3.5: Memory logical to physical view with all mirrors

Another mirror parameter that can be considered for embedded SRAMs is related to physical row ordering. This parameter may be called ROW MIRROR. An example is shown in figure 3.5.

To succeed in the conversion from the logical view to the physical one, the software environment has to include an Unscrambler module able to translate a logical address into a set of physical locations by taking into consideration the Memory and Scrambling parameters. This capability permits to easily focus the attention on a specific point in the memory map during a failure analysis process.

Based on this goal, another crucial capability that the tool should include is the memory positioning information; this aspect is particularly important in the SoC since embedded memories may be rotated (+90° , -90°) and/or flipped bottom-up. Figure 3.6 shows some of the most significant positioning configurations starting from the embedded SRAM shown in figure 3.5.

The conceptual schema of an Unscrambler software module is reported in figure 3.7. Having such a module enables avoiding the memorization of the full memory such as a matrix.

Failure Bitmap storage

The Failure Bitmap is the representation of the memory failures on the physical map of the analyzed memory cut. To build such a graphical vision of a faulty component, two major problems have to be considered:

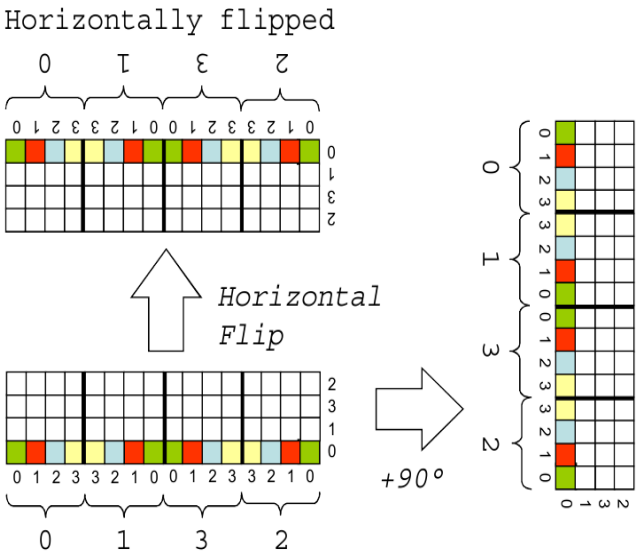

Figure 3.6: Rotated and flipped SRAM configurations

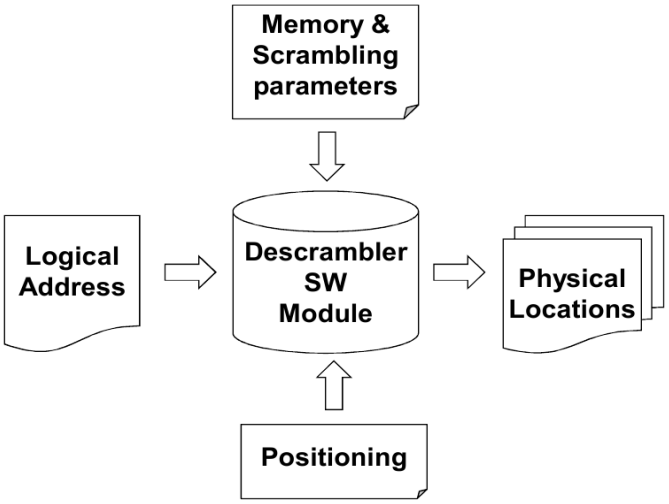

Figure 3.7: Address translation process

1. The access to the memory is usually performed in a logical manner, therefore returned faulty addresses have to be translated into the physical ones.

2. In many cases, during memory test execution more than one information is returned for a single faulty location.

The former issue can be partly solved by SW modules such as the Unscrambling one, but more information may be needed about the applied algorithm. In addition, the latter point requires the knowledge of the memory algorithm, thus allowing the compaction of the information retrieved along its execution which relate to a single location.

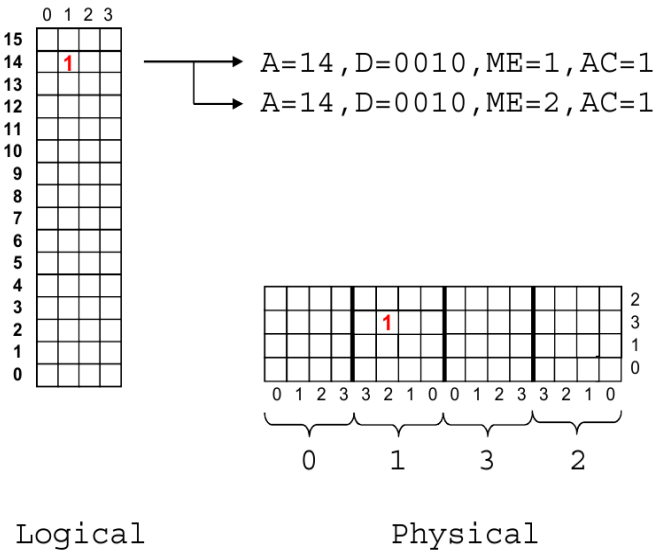

Figure 3.8: BIST responses and logical-physical fault location

Example 2 - Let consider the embedded memory drawn in figure 3.5. In case the physical location at Block 1, Bit 2, and Row 3 is stuck to 1 and the memory test executed is the March test [11]: (w0); (R0,W1, R1, W0); (R0)

Two failure information items will be returned, one related to the execution of the second March element (ME) at the first memory access (AC), the other related to the third one, again at the first memory access. The extracted logical address is in both cases 13 with read data 0010. This particular faulty case is illustrated in figure 3.8.

A suitable tool structure solving these problems is illustrated in figure 3.9; it includes the Unscrambler, complemented by an Expected Data Calculator module.

The combined work of both modules results into a list of physical locations, each one enriched with a collection of information concerning the failing memory test steps. This data structure is reported in figure 3.8.

Such a representation of the failing location has a double benefit. It is oriented to the physical memory structure, thus easing the drawing of the failure bitmap; furthermore, it is a compact representation that occupies a smaller space on disk and in RAM during display with respect to the memorization of the complete memory matrix.

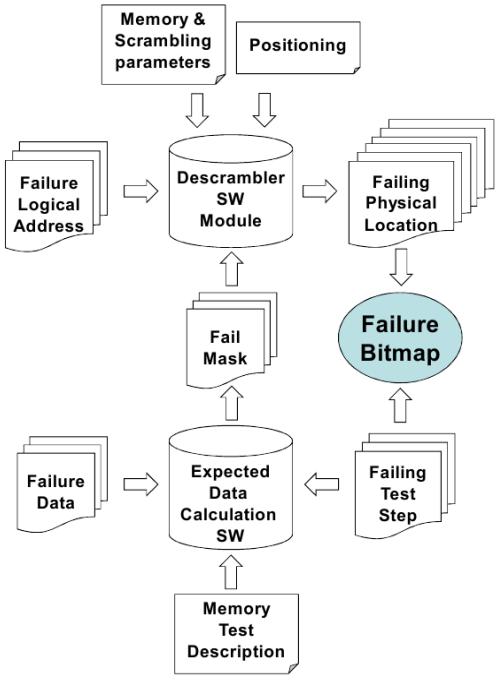

Figure 3.9: Conceptual working principle of the proposed framework

Such a structure is also suitable to perform two types of analysis (and their combination).

- *Syndrome grouping analysis*: it consists in considering the failing mechanism of each single cell, and grouping those showing the same set of failing test steps. This analysis consists in reordering the Faulty Physical Location list according to the list of Test Steps; the cost is in the order of O(N2) if the number of test steps is negligible with respect to the number N of failing locations.
- *Shape-based analysis*: it consists in recognizing a set of predefined shapes in the Failure Bitmap, such as failing rows (partial or complete), columns, pairs, clusters, etc.

Most of the classification can be achieved by simply ordering the Faulty Physical Location list according to the physical row/column address and then scanning it. The cost of this operation is of order O(2N+log2(N+1)). It is also possible to merge the two types of analysis by partitioning the failure bitmap by shape first, then by syndrome.

Example 3 - Let consider the embedded memory drawn in figure 3.5 and the March test used in the example 2. A more complex fault scenario is reported in figure 3.11,

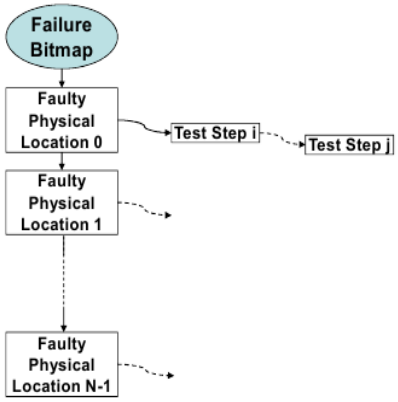

Figure 3.10: Failure bitmap data structure

including a pair of faulty cells both stuck at 1 and a column defect with all the cells stuck at 0.

In this example, the logical and physical representations are shown; the BIST response is referred to the logical addressing mode, therefore its output has to be converted as shown in the following failure bitmap obtained for the observed faulty configuration.

In this case the failure bitmap is ordered by column, therefore allowing the identification of the failing column. The two bits constituting the pair are not immediately classified, but they will after reordering the failure bitmap by row.

Cumulative Analysis

When tackling effective failure analysis, displaying and grouping failures inside a single memory is not sufficient any more, and the analysis must be extended to a larger set of memory devices.

Useful inspections working out systematic problems in the manufacturing process require observing a population of failing memories. Effective population compositions are:

- **Horizontal population:** it corresponds to accumulating information about the failures of all the memories included in the same wafer; eventually, the inspection can be reduced to accumulate failures in a wafer region.
- **Vertical population:** it corresponds to accumulating the information about failures over many wafers once a wafer coordinate has been selected; eventually, the vertical inspection can be done on a set of wafer coordinates.

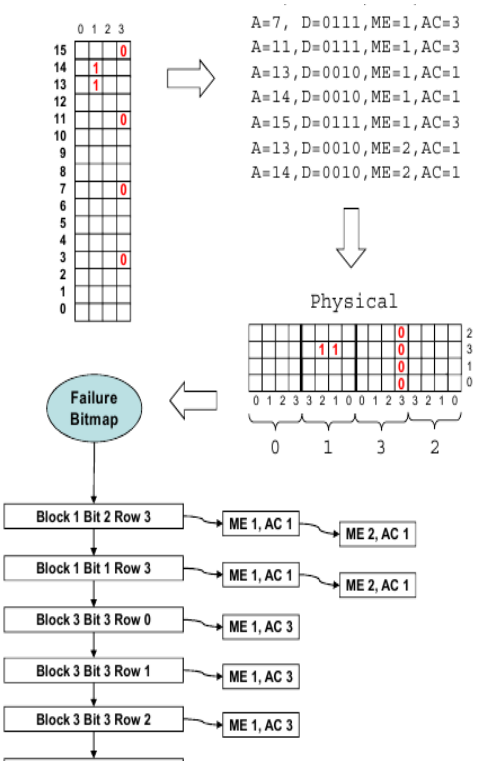

Figure 3.11: Example with pair and rows

Figure 3.12 illustrates the population composition. While horizontal inspection enables identifying systematic design weaknesses, vertical inspection provides also information about equipment induced failures such as those due to component handling or misalignments during mask and metal deposition process. Cumulative analysis can be done at wafer and singular component level, assuming in the latter case that traceability information is available to retrieve the wafer and the die position for each single component.

In case of accumulated information, shape-based analysis has to be performed separately on every single failure bitmap.

3.1.3 Experimental Results

A tool for the front-end analysis of embedded SRAM include in SoCs was implemented in JAVA, following the aforementioned concepts.

Such a tool is able to load information about the memory topology and SoC composition. It considers the output of a programmable BIST engine executing March tests, which functions are activated by a tester saving results in the Motorola SRecord format. Figure 3.13 shows the classification tree built after reading wafer test results; In order to

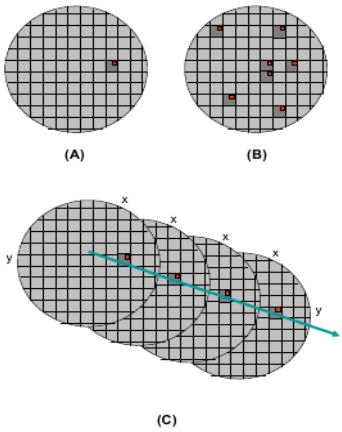

Figure 3.12: (A) single (B) horizontal and (C) vertical cumulative analysis

evaluate its performance and capabilities, the tool was used to analyze a population of SoCs belonging to 2 wafers.

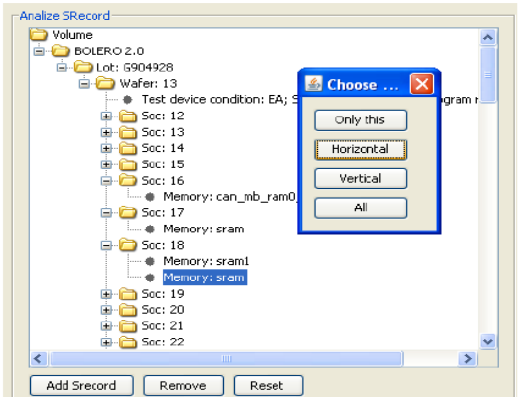

Figure 3.13: Memory classification tree

In terms of elapsed CPU time, visualizing the cumulative failure bitmap of 58 instances of a 20KB sized embedded SRAM including about 12,000 faults required about 800ms, while statistics calculation required about 1,5s. The amount of main PC memory required to maintain the failure is about 10MB.

In terms of the resulting bitmap display is shown in figure 3.14; the cumulative analysis for the SRAM memory core visualizes here a set of recurrent shapes, including partial and jeopardized failing row. The optical inspection leaded by this graphical hint enabled individuating the recurrent defect corresponding to a short metal connection, which is shown in figure 3.15.

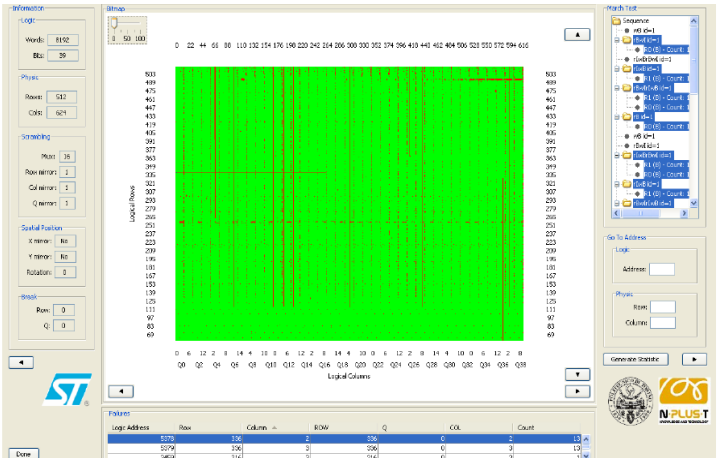

Figure 3.14: Failure bitmap display window

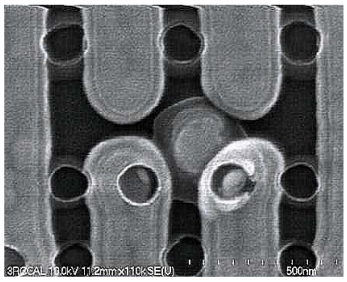

Figure 3.15: Identified defective mechanism, a stuck-open fault

3.2 Optimized Memory Analysis

Silicon debug and diagnosis processes are one of the most challenging tasks in the semiconductor industries, especially for newly designed SoC architectures. Effective diagnosis can be achieved via the combination of many techniques including insertion of on-chip design for testability hardware modules, test equipments applying suitable circuit stimulation, and software analysis tools to post process the retrieved data.

In the SoC arena, the test and diagnosis of embedded memory cores is usually based on Built-In-Self-Test (BIST)[59] [62] modules, especially optimized to quickly achieve accurate and reliable results. Since the diagnosis of failing memory cores is one of the major issues, BIST solutions offering diagnosis features (DBIST [60] [61]) are popular, and permit to extract from the silicon a comprehensible and accurate report including the unexpected test responses (called failure syndrome). DBIST modules are often designed to implement various testing procedures, e.g., activating several test algorithms or special walking modes.

The BIST procedure is managed by automatic test equipment (ATE) or tester which is responsible for launching and monitoring the test algorithms applied to the memories.

The ATE has to guarantee flexibility and must have enough intelligence to produce stimulation that may change depending on the encountered faults. Results stored in the ATE memory are finally sent out in the form of different files (eventually encapsulated into standard frames such as dictated by the SRECORD and STDF standard formats) to be further analyzed.

Software analysis tools primary outcome is to pinpoint failure modes and to indicate probable fault locations of failing chips. This is crucial to speed-up the failure analysis process which gives precious feedback to memory designers in a closed loop. To perform such a post-processing analysis, software tools have to put together all the test environment information, encompassing embedded memories characteristics, such as scrambling parameters, BIST features, applied test algorithms, and eventual limitation due for instance to the available ATE memory.

This section describes an affective technique to obtain meaningful information about location and type of embedded memory physical defects. The methodology is based on a visualization technique for failure bitmaps [62], whose output is augmented by analyzing failure shapes to restrict physical research efforts, since many logical failures are due to a single defect putting them in relation with fault models, joined with a confidence score.

Furthermore, the proposed technique tries to cope with noise in the testing environment, often observed when performing diagnosis at high temperatures, and intermittent failing behaviours. This is a very serious problem that to the best of our knowledge has never been mentioned before in the literature on memory diagnosis.

Examples and results reported in the following sections are related to SRAM memories embedded into a 90nm automotive SoC by STMicroelectronics. Results gathered

on such an industrial case study demonstrate the effectiveness of the approach in a real scenario.

Fault models

The set of classical fault models, presented in the *Reliability on Semiconductors* chapter, can be extended including technology-oriented fault models, and may consider also intermittent fault effects. In fact, it is quite common to observe incomplete fault effects; as an example, it is sometimes observed that, even in the case of faults identified as SA faults using optical inspection methods, the returned syndromes is not perfectly matching with the expected syndrome for this kind of fault.

The most used strategy to assign a faulty behaviour to a fault model bases on dictionaries, which store the set of possible syndromes that a faulty memory core can produce. To perform a correlation between the retrieved syndrome and the fault model causing it, the dictionary is accessed to select the most probable failure type hypothesis.

3.2.1 Proposed Optimized Flow

In this section we propose to extend the usual methods of performing memory diagnosis (presented on the previous section) by jointly using a set of failure information. In particular we propose the adoption of a software environment able to take into account the behaviour of all the relevant cells in the memory, and not only that of the single cells.

The optimized diagnosis flow results in the display of an augmented failure bitmap which is built by performing three logic analysis steps on the results returned by the ATE:

1. Failure shape recognition over the memory array, with data correction
2. Single failing cell fault model recognition through fault dictionaries
3. Final fault hypothesis is obtained by mapping and eventually correcting single cell information over failing shapes.

This flow (summarized in figure 3.16) is essential for an effective diagnosis, especially because of the inconsistency that may affect test procedures due to the following factors:

a. while the automatic test equipment is working, probable field stresses can introduce noise causing erroneous data registration and possibly resulting in false positives values

a. datalog truncation, that is shortening the test, may be done by engineers to reduce overall diagnosis time on the tester

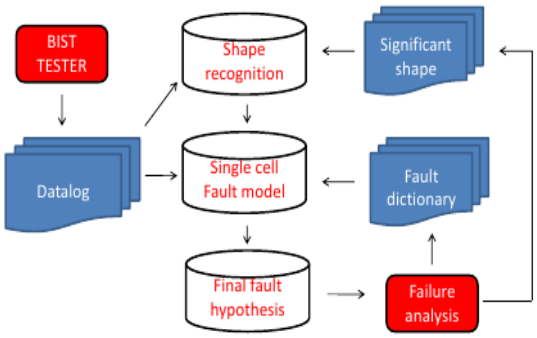

Figure 3.16: Proposed diagnosis flow

a. intermittent fault behaviour may hide the real nature of the fault that could finally become permanent when inspection is performed at high temperature or after the burn-in process.

The final result obtained by the software analysis tool which implements the aforementioned computation is the identification of failing memory areas joined with a preliminary fault type guess that is suitable to drive the efforts during the following failure analysis. In this way we can feedback more than a single piece of information about the location, but we can also try to understand the failing mode caused by the physical defect. This is particularly important to quickly close the loop within testing, failure analysis and design.

The next subsections detail the components of the proposed technique including some significant examples.

3.2.2 Failure shape recognition and completion

The initial step corresponds to identifying failures occurring in the memory cells, independently of their nature and organizing them into shapes (e.g., failing row, failing column, failing cluster).

In this phase, it does not matter the number of times the cell has shown a failing behaviour along the test algorithm execution, but just if it can be included into a particular shape taking also into account other faults appearing in the memory array.

Typically, the list of interesting shapes includes:

1. Columns and rows
    - complete
    - partial

- jeopardized

2. Pairs
3. Clusters
    - regular
    - irregular
4. Spots

Additionally, any regular failing schema in the memory array may be considered as caused by a single physical defect.

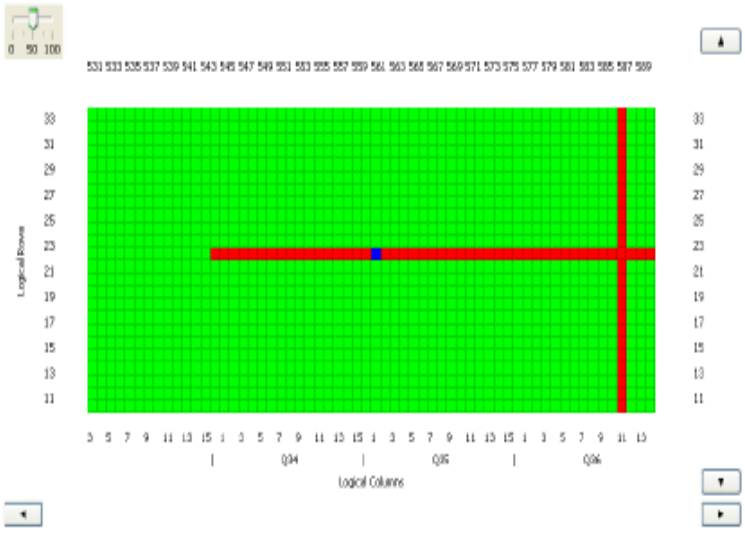

Figure 3.17: Bitmap with a partial row and a column shaped failures

In figure 3.17, a failure bitmap chunk that includes a couple of significant failing shapes is shown. One is a partial row failure, which starts close to the right side of the embedded memory and involves a relatively low number of cells. The starting location is the first candidate to investigate. Anyway, it is useful to go through a fault model investigation in order to obtain more reasons to attack this side of the failure.

The second is a complete column; this case is more difficult and not enough information is returned at this knowledge level. Furthermore, the failure shape is quite large since it involves all the cells from top to bottom of the memory array. This is not a problem at this level, but may become a factor to take into consideration when a fault model hypothesis is formulated.

It is sometimes useful to apply some corrections to the received data. As stated in section III, noise may interfere in the test flow, causing misleading and often random

unexpected behaviours. At the shape recognition level, the possibility exists that some cells which are expected to fail in a regular failing sequence are not detected as faulty. This is a suspect behaviour and may sometimes be corrected. Figure 3.18.a reports a column shaped defect whose datalog was corrupted at the BIST or ATE level.

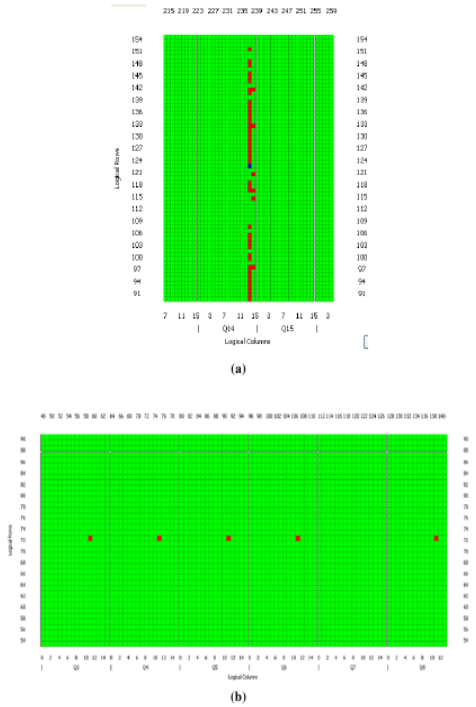

Figure 3.18: (a) failure bitmap corrupted by noise and (b) intermittent fault effect.

Another unexpected behaviour that could manifest itself during memory diagnosis is the intermittent fault effect. In many cases, a regular and clean sequence of failing cells is showing some holes in the sequence. Figure 3.18.b shows a case of jeopardized row where an expected to fail cell is not identified as faulty by the test procedure.

3.2.3 Single cell fault model recognition

The second step of the proposed test result analysis is intended to assign a fault model to each cell in an identified failure shape. This activity is crucial to reduce the manual efforts during physical inspections, since it returns indications about what kind of defect to search.

The fault model recognition step is based on accessing a dictionary including the syndromes of the most common fault models. The responses of failing cells in a shape are compared with known fault effects to produce a preliminary and completely logical hypothesis about the physical defect. This concept is illustrated in figure 3.19.

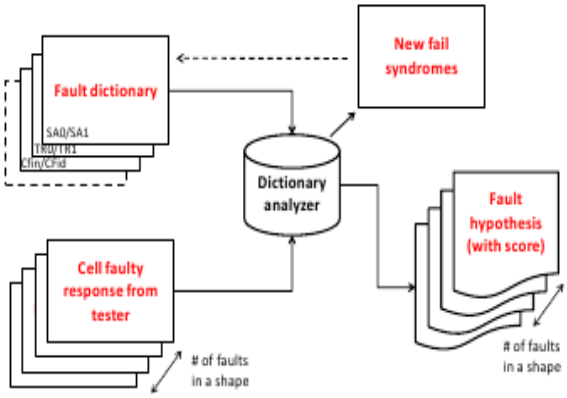

Figure 3.19: logical hypothesis produced by comparing the gathered syndrome with known fault type responses

It is quite usual to have unknown responses with respect to the traditional fault models. For this reason, it makes sense to add new fail syndromes when it is verified that it is produced by a well isolated physical defect. This step appears to be trivial, but it is not. In this scenario, at least 2 problematic situations may appear:

- Information is affected by noise, like in the shape recognition phase
- Datalog is truncated because test was stopped before retrieving all failure information.

The former situation may slightly deviate the analysis from the right diagnosis. To cope with this undesired effect, in this phase a score is given, corresponding to the matching percentage of the cell response with respect to a fault dictionary. As described in the next paragraph, this information will be completed and confirmed/changed in the final step of the technique.

3.20 illustrates an example of noisy context. In this particular case, the memory includes a failing cluster which is identified to be almost completely composed of cells affected by SA0 faults. Anyway, some cells show a haltered behaviour with respect to the responses that are not belonging to the expected SA0 syndrome.

In this explanatory scenario, cells showing a behaviour like cell B are labelled as SA1 with confidence of the 83.3% since there are 5 dictionary values out of 6 that are matching.

Concerning the data truncation, it usually intervenes when the physical defect affects a large set of cells failing many times; a typical scenario for truncation is the case of an entire column or a large cluster failing in a SA manner. In this unfortunate situation,

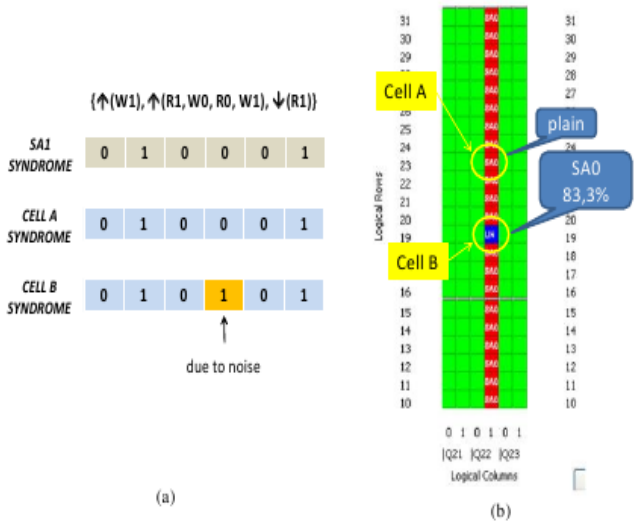

Figure 3.20: Noisy test environment affecting fault model selection: in (a) the March test executed and a comparison within expected and obtained syndromes, in (b) the shape of the failure including fault model hypothesis.

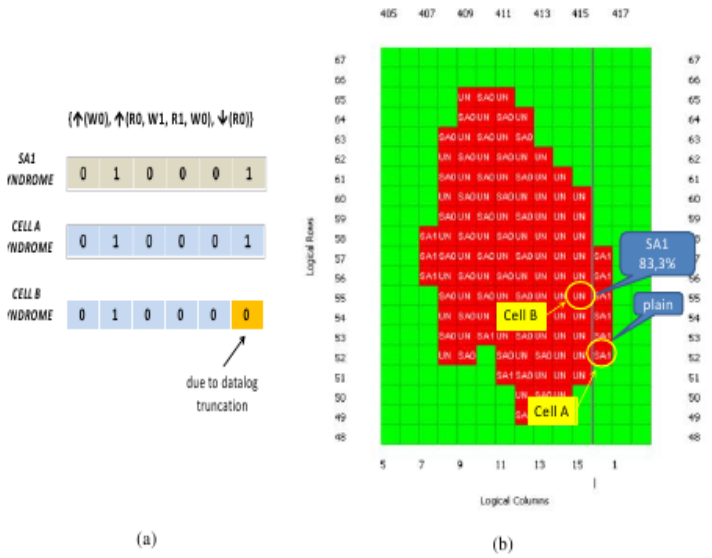

Figure 3.21: Effect of data truncation; in (a) the March test executed and a comparison within expected and obtained syndromes, in (b) the shape of the failure including fault model hypothesis.

the tester and/or the BIST impose a maximum number of fails to be recorded. As a consequence, it is possible that the result retrieval is incomplete. Figure 3.21 illustrates a possible context affected by data truncation; the diagnostic data retrieval is stopped in the middle of the column after the maximum number of failing information is reached, thus leading to differently categorization of failures belonging to different parts of the defective area. In fact, we notice a set of plain SA1 and a set of SA1 with a 83.3% score (5 out of 6 values are matching)

Similarly to noise and data truncation effects, also intermittent faults are possibly haltering the test results. As well, they are tackled by assigning a confidence score which is then taken into consideration in the final step.

3.2.4 Final fault hypothesis

In the final fault hypothesis step, all the information related to a shape are put together to provide a final useful logical identification of a step. The following sequence of data manipulation operations are performed

1. all faults incompletely identified are possibly assigned to a plain fault model; this operation is done whereas the given score is higher than a given threshold (80% is usually a reasonable value)
2. missing data in the shapes are filled, reporting a plain fault model indication whereas the rest of the shape is providing a regular failing schema.

Let us consider the examples given in figure 3.18 .a, 3.18 .b, 3.20 and 3.21. They were finally assigned to the following failure categories, respectively:

- 3.18.a: SA0 full failing column
- 3.18.b: Address fault affecting all bit 13 cells in a row
- 3.20: SA0 full failing column
- 3.21: SA1/SA0 cluster.

The user is only observing the final results, therefore it is not requested to do any manual activity to isolate the fault.

Furthermore, with the frequent usage of the tool, new significant shapes and fault models can be added to the software environment to refine the diagnostic results.

At this point, failure analysis can proceed. As demonstrated by the cases of study included in section IV, the approach is suitable to perform a fast identification of the likely causes for a failure.

3.2.5 Cases of Study and Results

Our cases of study consist of embedded SRAM memories of a 90nm SoC developed by STMicroelectronics.

These cores are tested with the support of a diagnostic BIST which can be programmed with an appropriate microcode describing march-like and non-march test algorithms. Usually the BIST is asked to reproduce a 36 elements memory test using several 2 databackground values and 4 walking combination (solid, checkerboard, row and column stripes).

The BIST functions are activated both at wafer sort level measuring at several temperature, and at packaged level by using the Teradyne J750 Personal tester. The data log capability is actually limited to 500 failure responses; the results are dump to SRecord standard compliant files to be further analyzed.

The software analysis tool implementing the methodology described in section III and owning all the functions needed to graphically display augmented failure bitmaps, is a java application providing many hint to isolate physical defects and to evaluate their incidence on the entire population of failing chips, such as through cumulative analysis. It is implemented and released by NplusT Semiconductor Application Center and by the Politecnico di Torino.

In the following, we report 4 cases of successful failure analysis driven by a preliminary logical analysis performed using the devised software tool.

Case 1 - Neighboring cells

In this case, the applied sequence of March elements allowed isolating a failure whose shape is a pair involving two neighboring cells. Figure 3.22 is showing the logical and the physical view of the failure.

According to the illustrated flow and without any correction of the identified fault model, the top cell was modelled by the tool as a Stuck-at-1, while its neighbour as a coupling inversion fault. The photography of the memory physical structure was taken by a microscope and an unexpected impure element was observed which is causing the mentioned faulty effect. The top cell, which is always at high logic level, also acts as an aggressor cell to the lower victim one that suffers a coupling inversion fault.

Case 2 - Irregular Cluster Shape

In this second experimental case study, an irregular shape was considered. This shape is composite of 2 SA1 faults (got syndrome is perfectly matching the expected for SA1), a

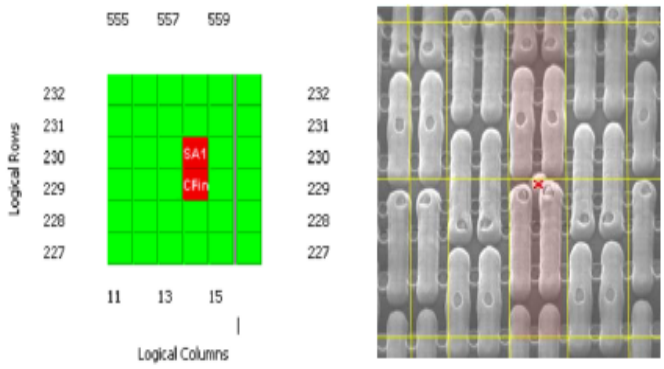

Figure 3.22: Pair defect: logical and physical appearance of the faulty array According to the illustrated flow and without any correction

SA0 (corrected after observing a 94% score) and a SA1 (corrected after 86% score) which is highlighted in blue.

From the physical defect inspection point of view, the failing zone was quickly identified. The various nature of faults logically pinpointed by the software analysis tool permitted to isolate the whole set of manufacturing imperfections. It can be seen here a macro defect in the middle of the cell group which is a residual producing an unwanted contact, but also some missing contacts that were "hidden" beyond the major impurity. For this second group of failures, their investigation was dictated by the response of the tool, while a manual analysis could even have missed them.

Case 3 - Regular Cluster Shape

The case shown in figure 3.24 reports a regular cluster shape defect. Anyway, it has to be noticed that the cluster is not fully modelled as a SA1, but it also include a SA0 fault. This hint was used to better investigate the behaviour of this particular cell which is showing an interrupted contact more than other cells in the cluster.

Case 4 - Irregular Cluster Shape

The case in figure 3.25 reports an irregular cluster shape defect, including both SA0 and SA1 behaviours. The physical defect observed during the failure analysis highlighted a large defect involving many paths to active area.

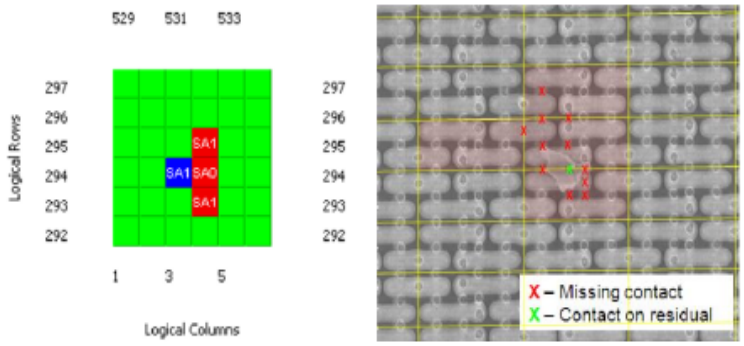

Figure 3.23: An irregular cluster showing many physical defects.

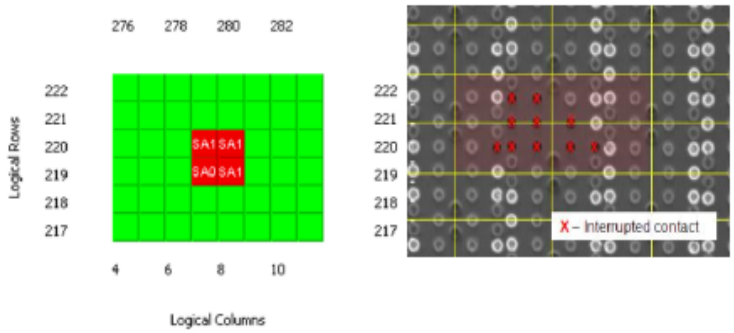

Figure 3.24: A regular cluster showing a composite logical hypothesis.

One of the most interesting reasoning that can be done on this picture is related to the cell at coordinate 492/122. Apparently, this cell is not affected by a defect when it is observed using a passive voltage contrast technique and could be lost if not pinpointed by the software analysis.

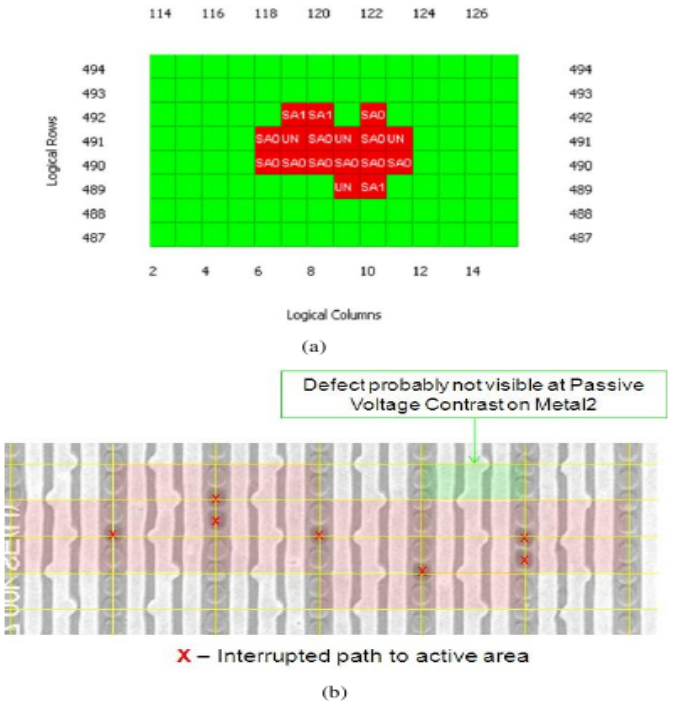

Figure 3.25: An irregular cluster showing a composite logical hypothesis.

3.3 Remarks on Memory Diagnosis

This chapter illustrated the software tool characteristics needed to obtain an effective memory silicon debug and diagnosis process. The described features were implemented in a tool developed in JAVA, suitable for the analysis of embedded SRAMs included in a SoC and tested via a BIST engine at wafer level. The proposed tool and methodology were able to categorize the failing results stemming from a memory test and to provide both information about the physical location of the defect and, possibly, a preliminary explanation of the corruption performed on the memory array. Some case study test results from SRAMs embedded in an industrial SoC developed by STMicroelectronics were analyzed by the tool. The diagnosis potentiality was proved by comparing the final results with real microscope images, pointing out the exact defect shape and a fault model hypothesis.

Part III

Functional Test Pattern Generation

4. Proposed Functional Stress Pattern Generation

Reliability characterization performed on a population of semiconductor devices has the goal to provide electrical and reliability measurements useful to individuate and classify physical failure data models along the entire component life-cycle. This process, known as *Operating Life Test* (OLT) [9], turns out to be fundamental for understanding the characteristics of advanced materials and device manufacturing processes. The importance of OLT is even greater for devices working in critical environments, e.g., the automotive and the space ones, and it is expected to continue growing due to the predictable technology scaling for the next generations of VLSI circuits.

With respect to burn-in screening (BI) [63] [12], OLT is mainly performed on test chips and monitors useful life and wear-out failures as well. Beyond their different purposes, BI and OLT are differently implemented. BI requires the devices to be stimulated at high temperature for few hours, either monitoring their outputs (test during burn-in, or TDBI) [64] or performing a successive test with automatic test equipment (ATE). On the contrary, OLT consists of several interleaved stress and test/diagnosis phases [65]; application engineers managing OLT normally have to plan for very long runs that can reach up to thousands of hours of duration and have to deal with the selection of proper stimuli for effectively stress the considered device and have to carefully plan for test/diagnosis measurements.

Another significant difference among BI and OLT is in the application of voltage stress that is obtained by toggling circuit node values [44]. BI requires the application of stress patterns owning high stressing capability all over the inspected device in order to rapidly exacerbate the insurgence of latent failures. Conversely, patterns used to induce voltage stress during OLT have to mimic the mission-like behaviour of the component for inducing realistic failure modes related to the entire life of the inspected device.

In this chapter, we concentrate on generating stress patterns for an effective SoC stress phase during OLT phases; similarly to the previous work, such patterns are functional test programs executed by the programmable resources available on chip, which maximizes the switching activity while complaining with the IC's normal functions. In the approach, the quality of the stress program under generation is calculated using some suitable met-

rics which are strictly related to the toggle activity of the circuit; these metrics, already discussed in [56], suit to measure how strong and uniform is the applied functional stress.

We propose a generation framework based on an EA [56], accounting on two-phases is considered to optimize the overall process. First, the EA operates at the RTL hardware description to quickly produce programs with a sufficient stress quality; in this phase, the generated programs are evaluated according to high-level metrics based on information gathered at RTL. These values are known to be correlated with gate-level ones, and are much easier to compute than the latter; therefore, this preliminary process is fast. After achieving this initial result, the generated stress patterns are optimized by operating on the gate-level description, thus requiring a high computational effort.

The benefits in terms of time gained with respect to the previous approach in [66] are evaluated on a 90nm technology System-on-Chip including an 8051 microcontroller core. Experimental results show that the generation time is reduced significantly and high stress quality achieved in a short time.

4.1 Background on stress

Borrowing some concepts from physics it is possible to say that stress is the internal resistance, or counter-force, of a material to the distorting effects of an external force. This counter-force tends to return the atoms to their normal positions. The total resistance developed equals the external force applied. This resistance is known as stress.

A force or load applied to a material distorts it in some way, independently of its strength. If it is a light load, then the distortion will most probably disappear when load is removed. This distortion is known as strain. So, stress attempts to overcome the material's inter-atomic forces holding it together. As stress continues to increase, the material starts deteriorating proportionally to the applied stress. This limit of applied stress is defined as the material's elastic region. If even more stress is applied to the material, this external force produces irrecoverable damages and this corresponds to the material's plastic region of deformation. Such application occurs when sufficient energy has been incremented to overcome internal forces [67]. Although it is impossible to measure the intensity of this stress, the external load and area to which it is applied can be measured. For a semiconductor device, it can be observed that physical stress modifies the molecules of the poly- silicon crystalline structure as well as the metal interconnections. Thus, it may be susceptible to failures as it creates specific faster paths for electrons, whose velocity augments the electric field and causes atoms to be pushed together or apart from each other. This phenomenon is denominated electronmigration. These atoms' movements create open or short circuits on the integrated circuit interconnections and flaws appear as the malfunctioning of the device [68].

On the other hand, temperature stress uniformly accelerates the ageing of Integrated Circuits, and an effective stress phase requires a ***functional stress pattern***, which means

toggling circuit node values, even without monitoring the device responses.

4.1.1 Functional stress quality evaluation metrics

Generally speaking, a stress procedure capable of effectively stressing a device under test has to activate transitions on the circuit nodes [44] [63] in such a way that: the number of transitions per node is maximized in a given time slot all nodes show the same (or almost the same) number of transitions in a given time slot.

These considerations are valid for any considered circuit and for any kind of selected stress pattern (e.g., scan, BIST, functional, etc.) meaning that the applied stress should ideally be strong and uniform over the whole set of circuit nodes [44].

Let us define Ti as the number of transitions taking place on the circuit node i during test application and TD as the Toggle Distribution resulting from monitoring Ti over the set of all nets. Suitable metrics to measure the strength of a stress pattern are the TD average, AVG(TD), and the TD variance, VAR(TD). In order to comply with the underlined stress quality requirements, TD has to show:

- High AVG(TD), implying high stress effectiveness on the circuit components
- Low VAR(TD), enabling stressing the circuit components as uniformly as possible.

In the specific case of the SoC reliability characterization process, functional stress methods provides some significant advantages: in particular, with respect to scan based methods [44], the adoption of functional stress programs allows to reproduce a mission-like behaviour of the analyzed SoC, thus Inducing more realistic failure modes Intrinsically avoiding power consumption concerns deriving from excessive nodes activity.

In a functional stress approach, the executed stress program should ideally guarantee that:

a The number of activated gates is as high as possible Switching activity stress applied to each core is as intense as possible

b The cores composing the SoC are equally stressed.

In the SoC scenario, some constraints are imposed by the processor core running a functional program. In fact, some components inevitably show more activity than others. For example, control and decode units are continuously required to process instructions, while functional units are active only when the SoC processor executes specific instructions. For this reason, we refine the aforementioned metrics by separately considering the contribution of each module composing the SoC. Therefore, it is proposed to generate functional stress patterns for these SoC units that continuously process instructions.

Indeed, stress effectiveness over SoC and its units is obtained by running functional stress programs and it is maximized when obeying the following order: The percentage of activated gates inside each module j is maximized

- AVG(TDj) of each module j is maximized
- VAR(TDj) of each module j is minimized.

4.1.2 Functional Program Generation using Evolutionary Algorithm

We explore the EA framework presented in [56] to automatically generate functional stress patterns. As described in Chapter 2, the EA is capable of mimicking the Darwin theory of evolution, by combining individuals or mutating them. An individual is an assembly program generated by the EA, which tries to optimize program stress quality metrics as described previously. If an assembly program is improving stress quality, then the EA makes it survive for the following generations, otherwise eliminates it from the optimized population.

An EA based methodology for the automatic generation of functional stress patterns is introduced in [66]. Even though the method achieved good results, it is time consuming, since the evaluation step is based on slow gate-level simulations. In this case, fitness values proposed by the evaluator and used to screen individuals are:

The percentage of activated gates of each module j is maximized

- AVG(TDj) of each module j is **maximized**
- VAR(TDj) of each module j is **minimized.**

4.2 Proposed Approach

In this chapter, we concentrate on the stress phase of OLT procedures suitable for SoC reliability characterization. The main goal is to generate a set of functional programs executed by the SoC processor core able to stimulate the majority of its internal gates as intense and constant as possible. Such programs are referred as functional stress patterns. With respect to [66], where only gate-level simulations are employed, we propose an EA based methodology that achieves better results in a shorter time.

The new strategy is composed of the following phases both based on stress programs evolution through EA:

a. The first phase aims at quickly generating stress patterns by working at RTL until sufficient stress quality values are reached.

b. The second phase starts from the stress quality level acquired in phase 1, and aims at improving the stress pattern population by working at gate-level:

   a initially using the fitness order as in phase 1.

   b then inverting fitness priorities for further refinement of the stress quality.

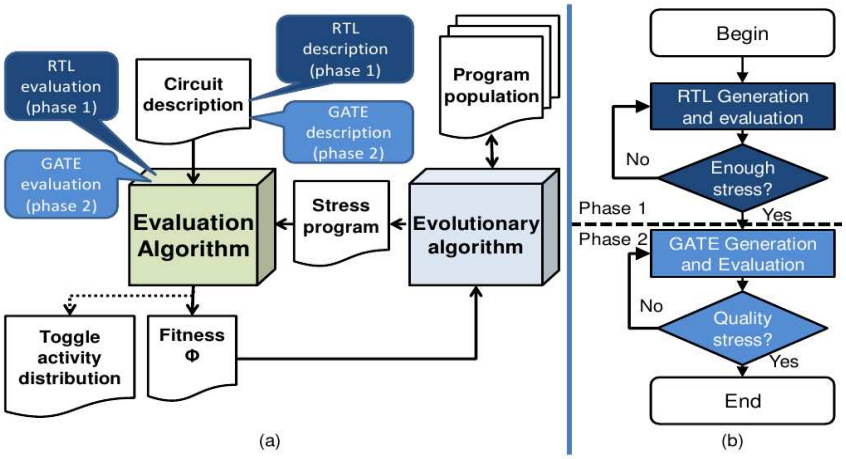

Figure 4.1: Proposed approach; (a) Generation flow, (b) Decision making process

The approach is illustrated in figure 4.1. Initially the framework in figure 4.1.a has the phase 1 setup configured (dark balloons), and it iterates until sufficient stress quality is achieved, flow of phase 1 shown in figure 4.1.b. Then phase 2 setup is configured (lighter balloons) and iterations continue until the functional stress patterns have achieved a desired quality level of stress within the program population. Figure 4.1.a represents the proposed framework for the generation flow and Figure 4.1.b the flow employed during each phase. The proposed methodology is feasible thanks to the strong correlation among RTL signals and gate-level nets. Thus, high switching activity of FFs' inputs and outputs of the sequential logic increases activity on the combinatorial one.

4.2.1 Phase 1 Fast stress pattern generation at RTL

This initial phase is intended to quickly generate stress patterns until a sufficient stress quality is met. The optimized methodology evaluates results based on the transitions of RTL signals gathered performing a logic simulation at RTL.

The timing optimization lies on the speed-up of the evaluation algorithm which simulates the SoC at RTL instead of performing the original task using the gate-level simulation, which is very slow.

The meaningfulness of this phase is granted because there is a correlation between the activities seen at RTL signals with the ones shown by the gate-level nets. In the detail, it is straightforward that signals in the RTL description are corresponding to Flip-Flops (FF) in the synthesized circuit; therefore, if signals are toggled, FFs will toggle accordingly. Moreover, the amount of activity of logic levels in the gate- level description is a direct consequence of the FFs transitions (i.e., if FFs move a lot, gates will show high activity). This correlation factor guarantees that the same stress program executed by the SoC at RTL produces proportional switching activity results to the SoC's gate- level activities.

However, information regarding the switching activity gathered at RTL tends to saturate after a while, partially losing switching activity correlation between RTL and gate-level. This is particularly true when dealing with complex circuits that are described at RTL with a few lines of code, as in the case of a logic multiplier.

Taking in consideration the above ideas, the first phase of the proposed approach is stopped when the first stress metric has progressed less than a given threshold K over a series of N consecutive generations.

4.2.2 Phase 2 Initial stress qualities refinement at gate- level

The second phase starts as soon as the stop condition is reached in phase 1. In this situation, to evaluate the stress metrics of RTL signals do not provide useful switching activity information to generate quality stress patterns, and so a thorough observation is needed to improve the current population. Therefore, the SoC RTL description is replaced by the gate-level one, allowing the evaluation of every single toggled gate in the design. In this new phase, the feedback parameters are also changed to gate-level values. The metrics priorities, based on a TD of gate-level nets, are described in the numbered list of the previous section.

During the second phase, simulations are slower since the SoC gate-level description is used. Consequently, the number of generations performed by the EA is smaller. However, this phase is very important to discover new instruction sequences able to stimulate gates that could not be observed in phase 1.

Once again, the stop condition depends on the saturation of all stress metrics values or the observation of constant stress quality. Since simulation is slow, a long period of time is usually required before reaching one of the stop conditions.

Final stress qualities refinement at gate-level At a certain level of evolution, the number of stimulated gates within a module is slowly increased and the tool starts generating programs corresponding to strong but non- uniform stress over the SoC, according to the list of fitness values described in the previous section. It means that few gates switch intensively while others are at all activated. Such characteristics are critical for the reliability characterization of a SoC. So, the feedback stress quality metrics 2 and 3, corresponding respectively to AVG(TDj) and VAR(TDj), are swapped to guarantee uniformity of toggles among gates. Thus the metrics priorities become:

1. The percentage (%) of stimulated gates (to be maximized)
2. VAR(TDj) (to be minimized)
3. AVG(TDj) (to be maximized) Where TDj is the toggle distribution of gate-level nets of module j.

4.3 Case study and results

In order to demonstrate the effectiveness of the approach, we present the results obtained on a SoC test-chip manufactured by STMicroelectronics in a 90nm technology; the reliability characterization flow principles described in the previous sections were implemented according to the requirements of a Massively Parallel Burn-In tester by ELES, the ART200 equipment. The test-chip design was developed with the aim of maximizing the technologic diversity and thus includes different types of functional cores:

- An 8 bit microcontroller
- A 64Kx8 bit SRAM storing instructions and data
- A combinational 16x16 parallel multiplier.

The test chip also includes an IEEE 1500 SECT based structure and a IEEE 1149.1 compliant TAP controller that enable transferring the stress program at low frequency and executing it at 40 MHz exploiting the BI equipment free- running clock commodity [69]. The internal structure of the chip is shown in Figure 4.2.

The stress program generation was performed by a tool called μGP3 that implements the EA exploiting a combination of Mentor Modeltech and an ad-hoc C++ tool for fitness computation. The elaboration provided a single functional program to be executed by the 8-bit microcontroller included in the chip.

For the sake of generating patterns according to the considerations of stress quality, we purposely applied the approach to the test-chip structure on the following components:

- The entire System-on-Chip (SoC)
- Single modules:
  - Microcontroller control unit (FSM)
  - Arithmetic Logic Unit (ALU)
  - RAM controller (RAM)
  - Memory controller (MEM)

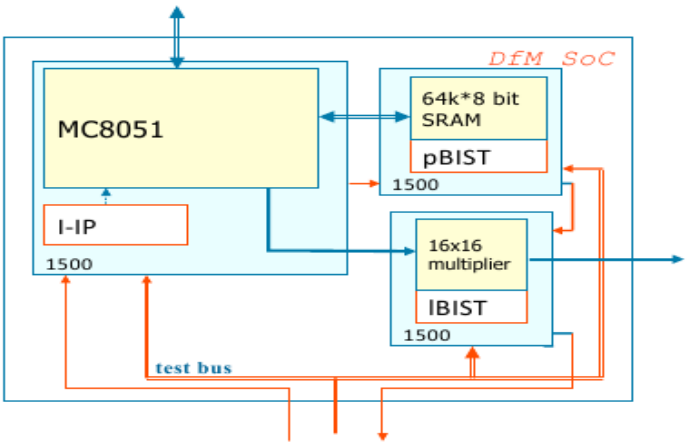

Figure 4.2: Block diagram of the test-chip

The approach applied to the memory controller (MEM) produced results useful to exemplify the behaviour of the tool during the two phases, even from a graphical point of view. Figure 4.3 shows the evolution on the number of stimulated signals/gates, in percentage, caused by the stress patterns executions at the core. During phase one, these programs were generated using the SoC RTL description framework (bottom curve) and evaluated on the SoC gate-level (top curve). In this figure, is noticeable the correlation among RTL signals and gate-level nets, since both curves have approximately the same behaviour during phase 1. Then, when the stop condition in phase 1 is reached (corresponding in our case to K=2% and N=30), the evolution is continued in phase 2.

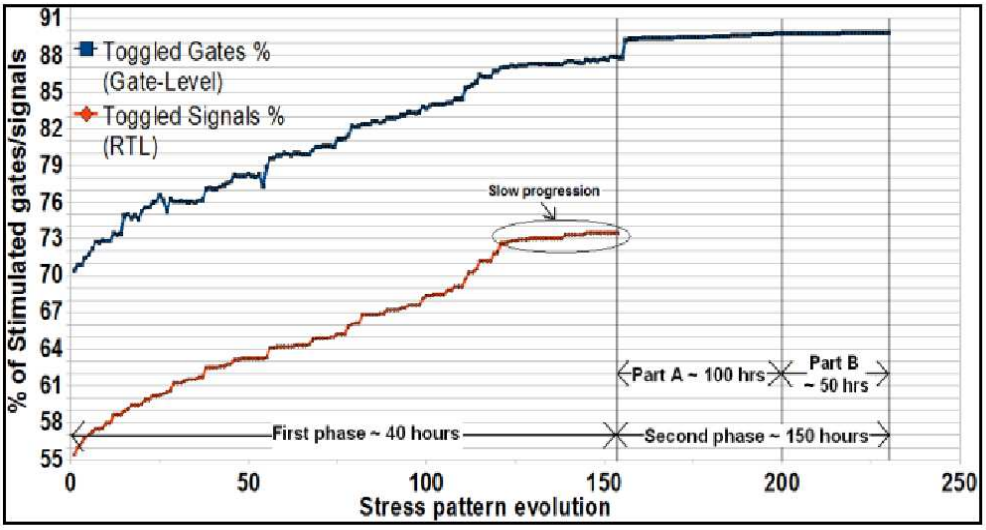

Figure 4.3: Stimulated gates in evolution of the MEM

In phase 2, the automatic generation was performed using SoC gate-level description, allowing refining the obtained stress quality up to this point. In part B of phase 2, the feedback stress quality parameter order is inverted (as discussed in the previous sections) to preserve and improve stress qualities. Figure 4.4 shows the AVG(TDMEM) evolution

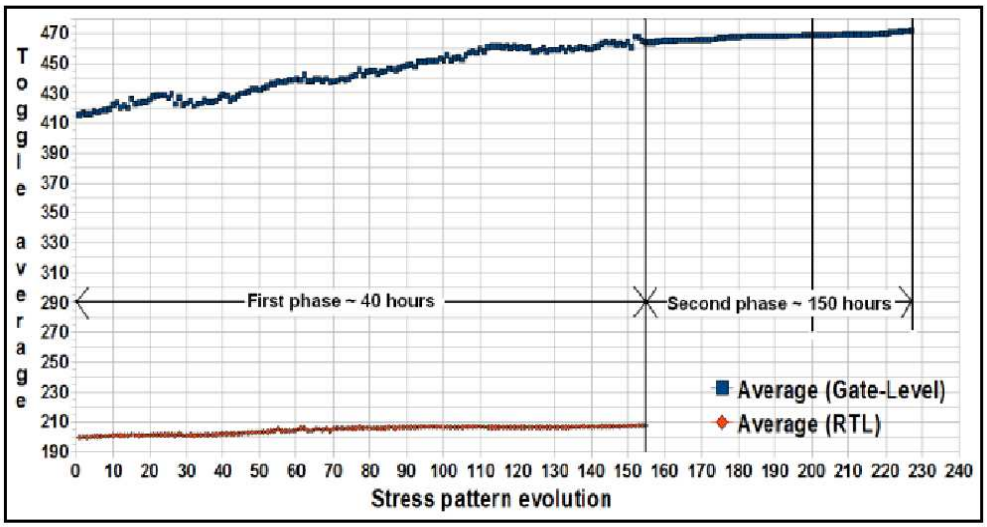

Figure 4.4: AVG(TDMEM) evolution

of the MEM module. Figure 4.5 shows the VAR(TDMEM) evolution of both phases and the effect produced by the metrics inversion.

Final results drawn in the second column of Table 4.1, shows the best stress patterns results of each SoC module, which were generated and evaluated by the two-phase strategy. In order to provide comparative results among the suggested approach and the previous methodology in [66], both procedures were launched at the same time. A preliminary stress quality level was reached by the proposed approach at around 190 hours of continuously pattern generation and evaluation, and then results of both methodologies were compared. This allowed measuring the stress quality disparity among both procedures within that period; After 190 hours the proposed approach reached high quality values, while the ones obtained by the previous methodology were still far from the best; The latter had to be executed up to 850 hours to reach comparable results. If we consider results produced on the FSM module by both approaches, it is noticeable that after 190 hours the new method was able to activate 98.94% of this module's gates. This value is better than the one obtained by the previous method [66] at the end of the same period (49.37%) and also after 850 hours (94.06%).

The proposed method was faster because a single generation cycle lasts 4 seconds when simulating the SoC RTL description, and about 45 seconds for the gate-level one. Since the previous method uses only the SoC gate-level description, simulations slowed the pattern generations.

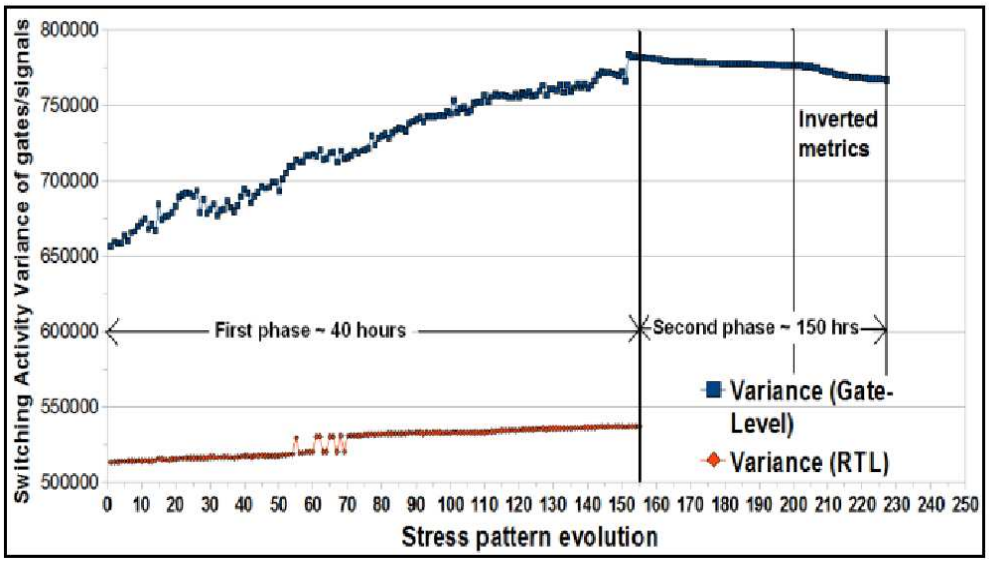

Figure 4.5: VAR(TDMEM) evolution

Module		Proposed Approach (190hs)	Previous Approach	
			190 hours	850 hours
SoC	%	76.16	71.70	75.91
	Avg	313	259	308
	Var	991,571	689,854	978,282
FSM	%	98.94	49.37	94.06
	Avg	978	898	966
	Var	1,838,778	1,497,723	1,786,832
ALU	%	97.69	91.98	98.14
	Avg	169	138	167
	Var	132,697	119,564	131,186
RAM	%	77.62	29.22	38.78
	Avg	395	178	192
	Var	1,349,509	549,755	575,106
MEM	%	89.91	69,41	93.33
	Avg	473	411	1,508
	Var	766,894	645,636	5,817,757

Table 4.1: Final results obtained for each SoC module.

Table I. Final results obtained for each SoC module.

4.4 Conclusions on Functional Stress Patterns

In this chapter a new method for automatic generation of functional stress patterns suitable for reliability characterization of SoC was introduced. We proposed enhancements to a previous methodology by adopting a new strategy; initially we use an RTL generation-based technique to speed-up the generation and quickly achieve improved results; secondly, we resort to a gate-level pattern generation approach to refine the results. It was demonstrated the feasibility of the improved methodology by proving the equivalence of gate-level and RTL approaches.

Results obtained by the proposed and original approaches were compared. The adopted strategy is much faster and achieves high level quality stress in a shorter time than the previous methodology.

5. Proposed Functional Power Hungry Pattern Generation

Nowadays, semiconductor product design and manufacturing are being affected by the continuous CMOS technology scaling. On the other side, high operation speed and high frequency are mandatory requests, while power consumption is one of the most significant constraints, not only due to the large diffusion of portable devices. This influences not only the design of devices, but also the choice of appropriate test schemes that have to deal with production yield, test quality and test cost.

Testing for performance, required to catch timing or delay faults, is mandatory, and it is often implemented through at-speed testing [14]. Usually, performance testing involves high frequencies and switching activity (SA), thus triggers significant power consumption. As a consequence, performance test may produce yield loss (due to over-stress), e.g., when a good chip is damaged while testing. In Ref. [70], the authors analyzed these phenomena and demonstrated that in some cases, yield loss may also occur when a good chip is declared as faulty during test (again due to over-stress). Hence, reduction of test power is mandatory to minimize the risk of yield loss; however, some experimental results have also proved that too much test power reduction might lead to test escape and create reliability problems because of the under-stress of the circuit under test (CUT) [14]

Hence, it is crucial to know which is the maximum peak power consumption achievable by the CUT under normal operational conditions. In this way the designer can check whether it can be tolerated by the technology and the test engineer can develop proper stimuli to force the CUT to work in these conditions.

In [15] the authors propose a flow able to precisely measure power consumption during test and functional modes of a processor. They evaluated several functional, pseudo-functional, and structural patterns and compared their power consumption, however none of these patterns were written to maximize functional peak power. The approach proposed in [15] uses an evolutionary algorithm able to create functional patterns (i.e., assembly programs) maximizing the functional overall power consumption. This novel strategy is effective but time consuming, since it uses slow simulations of the CUT in SPICE-like transistor-level description, where the final result may be obtained after weeks or even months depending on the desired level of details.

In another work [71] presented a detailed analysis describing the correlation among switching activity and power consumption obtained by feeding the CUT with the functional patterns. The results demonstrate that the overall switching activity strongly correlates with the overall power consumption, but one-to-one correlation among the identified peaks is weak. The method was able to identify the number of peaks of power by analyzing the switching activity occurring at different instants.

This Chapter proposes to speed-up the automatic functional pattern generation approach proposed in [72] by exploiting an intelligent and fast power estimator based on mimetic learning for increasing functional peak power of CPU cores. In [72] the approach employs an evolutionary algorithm (EA) that generates functional patterns and improves them based on their power consumption. However, power consumption evaluation is very slow, especially if we need to evaluate several hundred thousands of patterns. To speed-up the framework we propose a fast power estimator to be used in conjunction with the EA, thus reducing the evaluation time from weeks to days. In particular, we propose a method for fast power estimation based on neural networks whose inputs are the new switching activity per gate type and the output is the estimated power. In order to build the fast power estimator we need to perform the following tasks:

1. Analysis of the total switching activity within each clock cycle
2. Computation of a new SA (NSA) metric, by properly weighting transitions and glitches
3. Training a feed-forward neural network (FFNN) by using a 2-phase strategy which assigns proper weights to different gate types according to their NSA
   a. Generation phase: it increases the search space by generating proper patterns
   b. Training phase: it adapts the neural network for the newly generated patterns
4. Increasing the functional peak power by using the FFNN as a fast power estimator within the automatic functional pattern generation engine. The NSA per gate type is inserted in the FFNN and the power estimation is used as the feedback value for the EA.

In the proposed approach we are not trying to develop a new method for computing the exact power consumption, because power evaluators already exist by several vendors. Instead, we are proposing a method to quickly estimate peak power consumption; therefore, the method may sometimes miscalculate power, but, in general, it is able to correctly drive the EA to generate peak power effective patterns. The approach can reduce time from weeks to days providing a functional peak power consumption measure effective for mapping test power. The main novelty of the proposed strategy lies in its ability to improve the automatic functional pattern generation framework by inserting a FFNN-based external power evaluator which is faster than commercial ones.

The proposed methodology is validated on the Intel 8051-based SoC, synthesized using a 65 nm industrial technology. The training process of the FFNN required two

days. Also, the final generation using the trained FFNN reduced a single evaluation time by more than 60% with respect to commercial tools, while always individuating the test program points where the peak power was maximized.

5.1 Background on Power Consumption

Performance testing of low-power devices is crucial and challenging. In extremely small technologies, this type of test may reduce production yield due to high test power produced by the excess of toggle activity that may reach twice as much than functional mode [14] [70]. In scan-based tests, internal flip-flops are linked to form a shift register. Input test patterns are shifted in as quickly as possible in test mode, then several clock cycles are executed under functional mode, and finally output test values are shifted out. In some cases test data simultaneously stimulate parts of the circuit which would not have been stimulated under functional mode. If frequency and switching activity are increased during test so will the power consumption, as foreseen by formula 5.1. On the other side, we know that in low-power devices synthesized in small technologies the power consumption is an issue since it may very easily damage the component. The average dynamic power consumption of a CMOS circuit can be expressed as:

$$Power = \frac{1}{2} * f * vdd^2 \sum_{i=1}^{n} C_i * \alpha_i \tag{5.1}$$

where: f is the clock frequency, Vdd is the supply voltage, Ci and i are capacitance load and switching activity of node i, respectively. Most of evaluators use formula 5.1 for estimating power consumption in a CMOS circuit. for a well designed circuit, the power can be approximated by the switched-capacitance [73] [74].

The use of scan-chains for testing the circuit in a non functional mode may significantly increase the switched capacitances, leading to high power consumption. Several works investigate power modelling by approximating CMOS capacitances [75][76][74]. In [75], the authors model capacitances for several logic gates which allow estimating power at SPICE level. Results report that gates with larger capacitances have higher power consumption.

In [76], the authors propose modelling capacitances per blocks according to the type of logic used to estimate power. These works mainly allow the correct estimation of the average power consumption.

On the other side, in [71] the authors investigate peak power estimation of CPU cores using different tools. The functional peak activities and power consumption of the following levels were compared:

- register-transfer-level switching activity (RTL SA)

Method	Accuracy	Computational Cost
RTL SA	Very Low	1 X
GL SA	Low	2 X
WSA	Medium / High	10 X
TA	High	300 X
PLA	Very High	1,000 X

Table 5.1: Power evaluation methods

- gate-level switching activity (GL SA)
- weighed switching activity (WSA)
- transient analysis power evaluation (TA)
- post-layout analysis power evaluation (PLA).

from this comparison, it was verified that normal activities were very strongly correlated but peak activities did not correlate at all.

Table 5.1 qualitatively summarizes the characteristics of each method in terms of accuracy and computational requirements. As we can see from the table, RTL SA is the fastest evaluation we could perform, however it does not have a good level of detail and power cannot be evaluated with reasonable accuracy. As we go further down on the table we have more accurate results but the time for obtaining them increases significantly.

In [15] the authors consider structural tests reducing power consumption belonging to both launch-on-shift (LOS) and launch-on-capture (LOC) types and compare the power consumption during performance testing with the functional power during normal operations. They propose a power evaluation flow for CPU cores at SPICE level, which allows comparing LOS/LOC test power with functional power. Results show that test power is 14% more than functional power in 90 nm technology and almost 47% more under 65 nm. In addition, if technology scales and frequency increases during test the power can be even more harmful for particularly low-power devices.

In [71], maximum peak power estimation at TA by analyzing gate-level SA was proposed. Overall correlation indexes among the above-mentioned evaluation methods are strong, ranging from 0.8 to 0.9, although it was verified that one-to-one relationship at maximum peaks do not correlate. This is due to the fact that the impact on power consumption of glitches and valid transitions were considered equally.

In [77] the effects of glitches on the power consumption have been studied. They are shown to be responsible for 10% up to 60% of the overall power consumption, while the rest (from 40% to 90%) is due to valid transitions. This wide range depends on the circuit description, the used library, and the adopted synthesis parameters. If all these features have been carefully selected glitches minimally impact power consumption. The

peak power of functional patterns obtained in [78] and [79] was studied in Ref. [15]. The former work exploits an automatic functional pattern generation approach for maximizing stress by trying to activate every gate in the CPU core. The latter provides code snippets containing instructions able to increase power consumption. In both works a methodology for automatic functional pattern generation proposed in [72] was exploited. It employs an evolutionary algorithm able to generate functional patterns effective in maximizing a certain metric arising from specific circuit stimuli.

The maximum functional peak power value can be obtained by combining the automatic functional pattern generation with power evaluators (e.g., WSA, TA and PLA) for extracting peak values. However, the approach generates and evaluates hundreds of thousands of programs before converging to a desired maximum peak power value. Therefore, the evaluation phase must be very quick and able to generate effective programs in a reasonable time. Ideally, we would like to maximize metrics at RTL (i.e., peak SA in a clock cycle) and observe an increasing trend in PLA power consumption. Yet, this specific metric is still missed and alternative solutions shall be proposed.

5.2 Proposed Methodology

In this chapter we propose to speed-up the automatic functional pattern generation approach by developing an intelligent fast power estimator based on mimetic learning. The whole environment we refer to is shown in figure 5.1. Since our target is CPU cores, the input functional patterns are assembly programs. In 5.1, a power estimator tool is introduced in the evaluation phase to determine the maximum peak power and thus driving the EA for correctly generating functional patterns with maximum peak power consumption (survival of the fittest). for our objective it is an efficient tool because it randomly generates functional patterns for CPU cores, and evaluates them resorting to an external evaluator. It is then able to discard assembly programs or increase their strength according to the power result received from the evaluator. We resort to EA because it is able to maximize the power metric from an initial random population, discarding the weak ones and improving the stronger patterns. Random approaches are less effective than EA because generating functional patterns is an NP-hard problem where testing all combinations is computationally impossible. Therefore, EA is preferred since it can twitch good random individuals by applying genetic operators to quickly improve their characteristics.

Since the automatic generation approach typically requires the evaluation of hundreds of thousands of functional patterns, the speed of the evaluator (as well as its accuracy) is crucial for the effectiveness of the whole approach.

The proposed idea for power estimation is summarized in figure 5.2. We consider a bijective function where each element of the subset SA corresponds to an element in WSA, TA and PLA subsets when applying the functions f(SA), f'(SA) and f"(SA) respectively. However, it is difficult to discover these functions manually due the large quantity of data

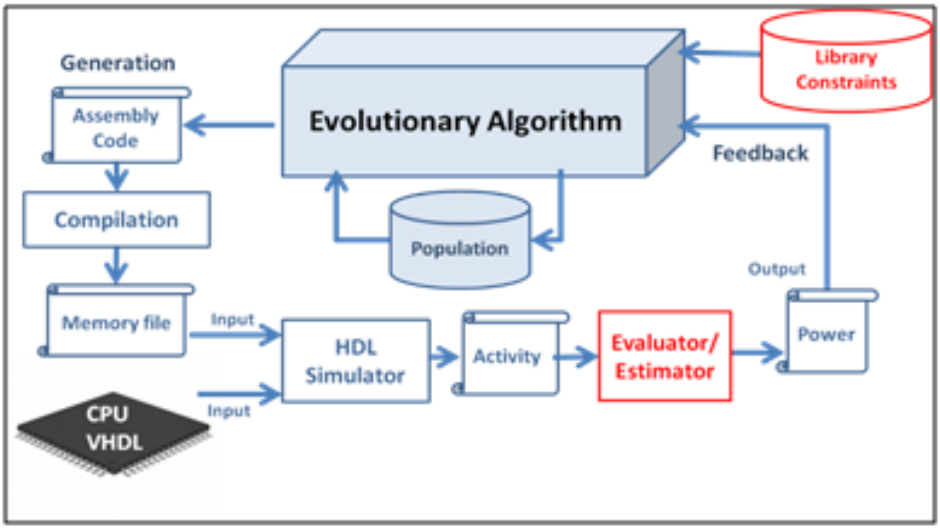

Figure 5.1: Automatic functional pattern generation framework

that must be analyzed. So, in order reduce complexity to develop these functions, we exploit a learning approach which matches SA to power consumption: initially the SA and TA subsets are elaborated from several functional patterns. The first is elaborated just by counting the SA using a simple ad-hoc program, while in the second, the power consumption is evaluated using a commercial tool. By integrating suitable learning mechanisms, that is a feed-forward neural network, the tool is able to understand the power evaluation behaviour and provide an f'(SA) which matches SA to TA subsets and thus correctly estimating power consumption. We resort to mimetic learning algorithms because it is able to estimate power also for unknown patterns, whereas curve fitting algorithms are able to fit known data from one subset to another. In order to construct the fast power estimator, we propose to perform some pre-processing of the CPU activity data by performing the following tasks:

1. The SA evaluation takes into account both effective transitions and glitches occurring within each clock-cycle

2. The NSA is computed by weighting glitches and effective transitions. We define k1 the weight for glitches and k2 the weight for valid transitions. By choosing the correct weights the correlation among NSA and power consumption improves, especially at the peaks.

Moreover, we propose the adoption of a fast power estimator, corresponding to a feed-forward neural network (FFNN). As discussed in the previous section, each gate type has its own properties [75]and usages [76] that impact accordingly on power consumption. The FFNN is able to mimic the power evaluation. Clearly, the quality of the estimation

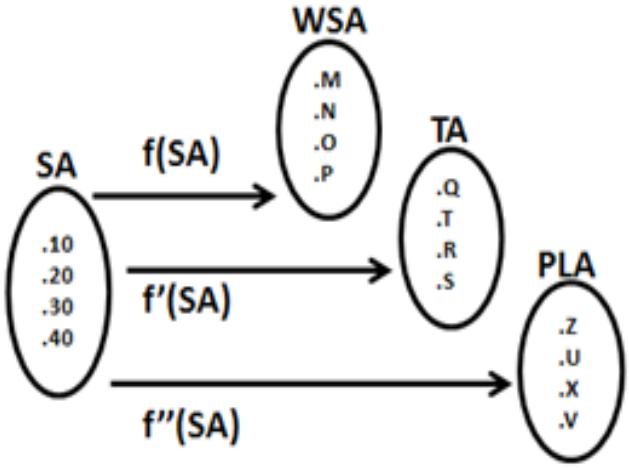

Figure 5.2: Matching SA pattern to WSA, TA and PLA power consumption.

by the FFNN depends on its appropriate training that is better detailed in the following subsection. To summarize, the proposed approach to increase functional peak power is composed of two parts:

1. Training strategy: correctly trains the FFNN for mimicking a commercial power evaluator and provide correct power estimation
2. Increasing functional peak power: exploits the EA and the trained FFNN to automatically generate functional programs containing the maximum peaks of power.

5.2.1 FFNN Training Strategy

In this subsection we initially discuss analysis of SA and NSA within the clock cycle, explaining that the latter is better to be used as input for the FFNN-based fast power evaluator. Then we describe the FFNN and how it can be used for estimating power consumption mimicking a commercial power evaluator. finally, we propose the training methodology which is able to correctly train the FFNN to effectively estimate power consumption. The proposed training methodology is composed of two phases that are iterated over time to achieve better results:

1. Generation phase: it creates functional patterns effective for training the FFNN
2. Training phase: it adapts the FFNN to correctly match NSA to its real power consumption and thus perform suitable power estimation. We will now better detail the key points in the above procedure.

5.2.1.1 Sum SA Within the Clock Cycle

In [71], the proposed approach is to match peak activities at gate-level SA to TA power consumption of CPU cores. The activities were analyzed at every 2 hundredth of the clock period, that is, per event. However, gate-level simulation assumes zero delay, meaning that gate and interconnect delays are not considered, like it is in TA and PLA power evaluations. In this way, it is extremely difficult to match peak activities. On the other hand, the activities within the clock cycle should be rather similar since the working models must respect the clock period. Therefore, we propose to sum gate-level SA, thus getting a metric that generally matches TA peak power within the clock cycle.

5.2.1.2 The NSA Value

The power consumption associated to a glitch is much smaller than the one of a valid transition, and sometimes considered negligible [77]. However, glitches may impact on power consumption as they are more frequent than valid transitions. Such assumption is particularly true when analyzing a CPU containing thousands of gates where the occurrence of glitches is much more frequent than the number of valid transitions. Though, a single glitch may not have a huge impact on power, but many of them may impact more than the sum of the valid transitions. Therefore, we propose to weigh glitches and valid transitions in such a way to improve the correlation index among estimations based on SA at logic level and real power behaviour. We propose the adoption of a new switching activity (NSA) value where glitches are multiplied by a constant k_1 and valid transitions by another constant $_{k2}$. Formula 7.2 shows how to compute the NSA value at clock-cycle i.

$$NSA_i = \sum_{g=1}^{G} (k_1 + k_2 VT_g) \tag{5.2}$$

where GL_g and VT_g are the number of glitches and valid transitions of gate g, respectively, while k_1 and k_2 are the multiplication constants of glitches and valid transitions, respectively. G is the total number of gate outputs where glitches and valid transitions are observed. Moreover, since a single glitch is less powerful than a single valid transition (as discussed in [77]), the following relationship must hold:

$$k2 > k1 \tag{5.3}$$

The k_1 and k_2 weighting coefficients are constant for the whole circuit and we do not estimate them for every single gate.

5.2.1.3 FFNN for Power Estimation

The main goal of *artificial intelligence* (AI) is to get knowledge from known input and output patterns. The ability of a computer system to match input to output patterns in an intelligent manner is called machine learning.

We propose a fast power evaluator based on a feed-forward neural network (FFNN). A FFNN is an AI-based tool corresponding to a directed graph with three layers: input, hidden and output.

Each layer has its own artificial neurons that sum the products of its inputs by weights and feeds the final result to the successive layer. In figure 5.3, a FFNN is drawn and formula 5.4 describes the computation of the output value y of a single neuron *k*. Originally, FFNNs were introduced for pattern recognition. An input pattern is fed at the input layer and the FFNN computes an estimated value characterizing the recognition or not for that input pattern. We propose to use the FFNN as power estimator by injecting at the input the NSA values for each gate type to compute an estimation of power consumption. The output yk of the k-th neuron is given by the formula

$$Y_k = \phi(\sum_{j=0}^{m} W_{kj} x_j) \tag{5.4}$$

where ϕ is the transfer function of the neuron; considering a neuron with *m* inputs, x_o through x_m are the input signals and wo through wm are the weights. This neuron model is not the most used but it is the best suited for our approach since NSA inputs must be correctly weighted resulting in an approximate but not strictly accurate value of power consumption for the specific gate type. It is also a more generic neuron model, and using it in an FFNN allows quickly computing the correct weights, thus reducing the training time.

The idea of using a FFNN to estimate power values comes from the concept of power estimation methods provided by [75] and [76]. In the former work, the authors propose to estimate capacitance and power models for each type of CMOS gate with an error of maximum 10%. Also, they verify that gates with higher capacitances present also higher power consumption. The latter work reinforces the idea of [75] and also proposes that power estimation must be performed on a block basis. The authors estimate the circuit capacitance by adding the combinatorial and sequential capacitances. Moreover, the average fan in, fan out and capacitance values are used to compute the power consumption. Assuming the above-mentioned concepts for power estimation, we propose to estimate power by assigning weights for each type of gate. The FFNN neuron weights are multiplied by the sum of the NSA for each gate type and the output result is the power estimation, almost accurate to the power estimation presented in formula 5.5 in respect to the example in figure 5.4. In figure 5.4 a circuit containing combinatorial and sequential elements is shown. There are 18 gates and 6 gate types: flip-flop, NAND, AND, OR,

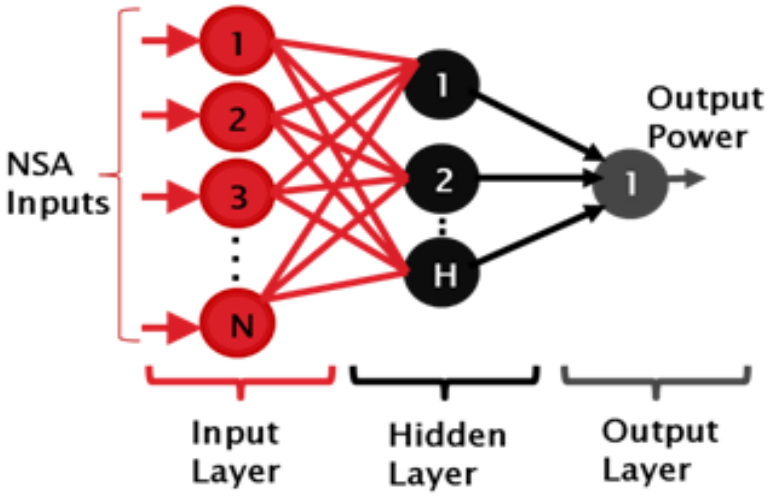

Figure 5.3: Artificial intelligence feed-forward neural network.

NOR, and inverter. The power estimation at every clock cycle k for this circuit is shown in formula 5.5.

$$P_k = \sum N_{kf}W_f + \sum N_{ki}W_i + \sum N_{kn}W_n + \sum N_{ko}W_o + \sum N_{kno}W_{fno} + \sum N_{ka}W_a \tag{5.5}$$

where Nkz is the NSA for the gate type z at the instant k, and w is the weight for each gate type (f corresponds to flip-flop, i to inverters, n to NANDs, o to ORs, no to NORs, and a to ANDS).

The trained FFNN uses the correct weights for each gate type, such that they assume the average characteristics of capacitance, fan-out, wiring resistance and other characteristics impacting on the power consumption. So, the power estimation provided by the FFNN and corresponding to an approximation of formula 5.5 must be very similar to the power evaluation provided by the commercial evaluation tool.

In order to assign the weights, it is necessary to train the FFNN; this means that the weights are adjusted to provide the correct power consumption estimation. This is performed by using a back-propagation training algorithm. This is an iterative training algorithm which initially assigns random weights to the FFNN and computes the error values (i.e., the difference between the computed and the desired output value) given a known set of input values. The back-propagation algorithm propagates back the error to the input nodes adjusting the FFNN weights, such that providing known input values to the FFNN will make it converge to the desired output ones. The iteration stops when a sufficient Mean Squared Error value is satisfied, that is when the Mean Squared Error between the difference of the computed output values and the desired output values are under a given threshold defining the FFNN's precision.

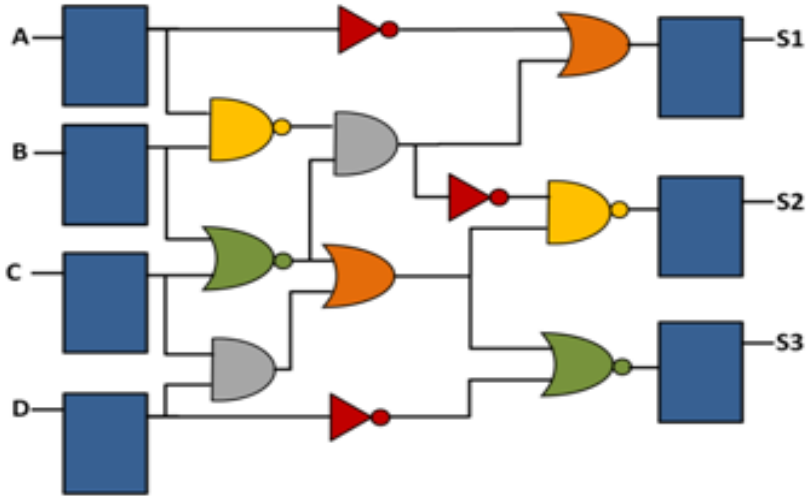

Figure 5.4: A sample circuit with different gate types.

5.2.1.4 FFNN Training Methodology

Using the FFNN for estimating power does not require too much effort. However, the difficulty in its development is to train the FFNN, i.e., assigning proper weights for the neurons of every layer. In addition, ideal functional patterns must be provided in such a way to cover most of the functional pattern space set. Hence, we propose a 2-phase training methodology: generation and training. figure 5.5 depicts both phases in a cooperation work to generate the correct functional patterns and to adapt the weights of the FFNN.

The first phase (generation phase) employs an evolutionary algorithm to generate functional patterns for the target circuit. The automatic functional pattern generation follows the strategy proposed by [72] and uses a framework similar to that of 5.1. This phase is iterated for several generations until enough useful individuals have been produced. Useful individuals are corner case patterns not covered by the FFNN. Hence, these patterns are used to adapt the FFNN in the training phase and consequently, improving the power estimation over runs on the training phase.

In figure 5.6 an evolution curve related to the output of the EA is presented. The EA evaluates a population of individuals and ranks the best and worst ones along the generations, whose typical behaviour is shown in the figure. Ideally, it is necessary to generate many individuals to discover corner cases not covered by the FFNN as explained in the previous paragraph.

In the second phase, power evaluation is performed on the functional patterns extracted from the evolution curve. Also, the NSA values per gate type of these individuals are extracted and matched to the evaluated peak power. Then, a back-propagation algorithm for training the FFNN is employed to improve weights and consequently the power estimation.

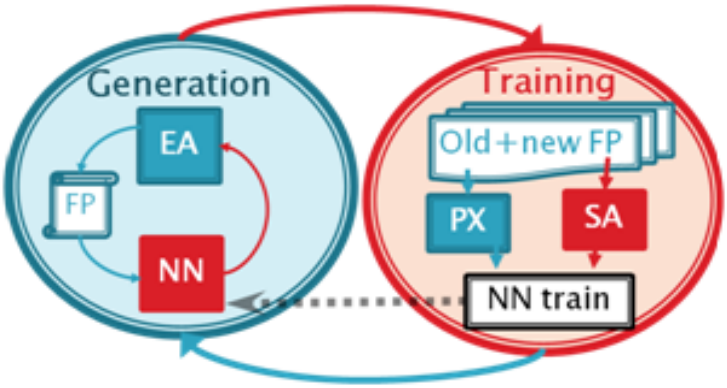

Figure 5.5: 2-phase strategy to train the FFNN.

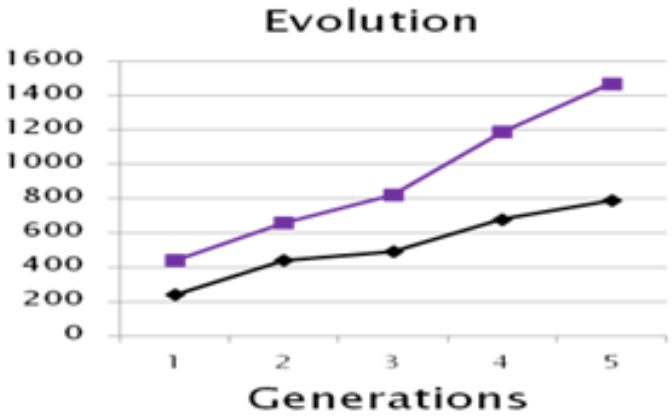

Figure 5.6: Evolution curve to extract individuals.

During the training phase we remove NSA which has already been logged to speed-up the training process. Usually, repeated sequences of assembly code throughout the individuals produce the same NSA, and therefore they should be excluded from the training data set.

5.2.2 Increasing functional Peak Power Using the Trained FFNN as the fast Power Estimator

We increase the functional peak power by performing an evolution exploiting the automatic functional generation approach using the FFNN as the fast power estimator and thus speeding up the overall procedure. The final procedure is iterated for several generations until the ending point is reached i.e., when evolution reaches a stable point. Then, some individuals of this curve are sampled and power evaluated in order to verify that there is an increasing trend in the real power consumption evaluation.

In figure 5.7, the final framework for increasing peak power consumption is presented. The EA generates a functional pattern that is compiled, simulated, and the NSA value per gate type is extracted to be inserted at the FFNN inputs where power estimation is performed. Then the estimated value is fed back to the EA so that it can generate better functional patterns.

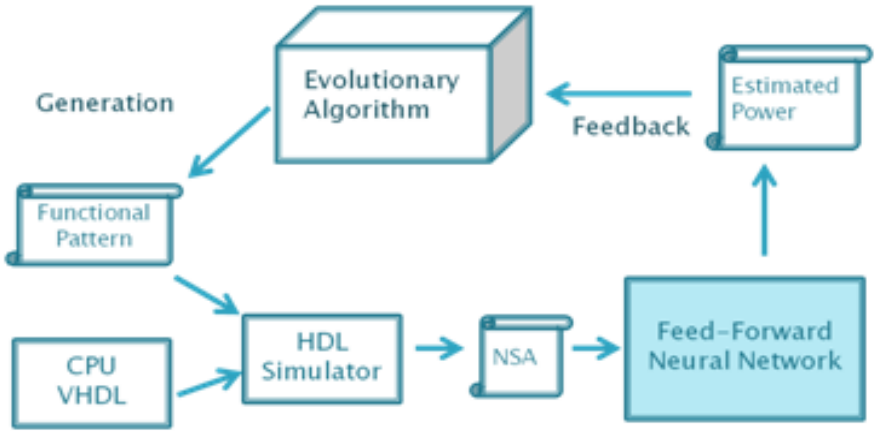

Figure 5.7: Final framework including the FFNN as the fast power estimator.

5.3 Case Study

In order to validate the proposed methodology we use an Intel 8051 CPU core synthesized in a 65 nm technology. The tool implementing the EA is the open source μGP3 software

[80], which was designed to automatically generate assembly programs for a desired CPU core under a set of constraints. The tool for simulating the hardware to extract the circuit activity is Modelsim 6.3b. The tool used to evaluate power consumption during the training phases is Synopsys Primepower. Lastly, the tool evaluating electrical signals is the Nanosim Synopsys tool suite, which allows precise measurement of the power consumption of functional patterns under a specific technology library. The neural network was trained using the Matlab NN toolbox.

5.3.1 Intel 8051

The selected case study is the Intel microcontroller MCS- 51(MC8051) synthesized in 65 nm industrial technology. The 8051 core represents a classical Harvard non-pipelined CISC architecture, with 8-bit ALU and 8-bit registers. Table 5.2 gives the details of the microcontroller synthesis in terms of number of primary inputs/outputs, flip flops, logical gates, and the number of gate types used in the circuit.

5.3.2 Glitch and Valid Transition Constant Assignment for NSA Computation

Several functional patterns with many different characteristics were analyzed. Glitches and valid transitions were multiplied by several k_1 and k_2 values. Then correlation indexes were computed among the NSA and power consumption. The best value found for k1 is 0.1 and for k2 is 0.9. In order to discover these values, a divide and conquer algorithm was employed. Initially, k_1 and k_2 were assigned the value of 0.5. Then, the value of $k_1/2$ was added to k_2 and subtracted from k_1 , in fact increasing the correlation index. If the correlation index increases, then the values of k_1 and k_2 are updated, otherwise they assume their previous values. The correlation index considered was the average value of 20 programs, and consumed less than 1 hour of computer effort to compute the values of k_1 and k_2 .

5.3.3 FFNN and Parameters

The FFNNs used in the proposed approach were implemented using the Matlab 2008b NN toolbox. Some training parameters were defined that allowed developing the correct FFNN:

a. Number of input nodes

b. Number of output nodes

c. Minimum and maximum values for each input

Primary inputs	65
Primary outputs	94
FFs	578
Gates	4,246
Gate types	97

Table 5.2: 8051 characteristics

d. Number of neurons and layers implementing the network

e. Neuron transfer function (ϕ) for each layer

f. Training type.

The FFNN for the 8051 CPU core for power estimation contains three layers: input layer, hidden layer and output layer. The input layer is composed of 97 input neurons with a hyperbolic tangent sigmoid transfer function. The NSA values per gate type are inserted in these neurons. The hidden layer is composed of 3 hidden neurons with a linear transfer function. The output layer has just one neuron with a linear transfer function; it sums altogether the intermediate values provided by the hidden layer and outputs the estimated power value. The interval of allowed NSA values in the input neurons is from 0 to 200 and the training method is back propagation. Also, the Mean Squared Error (MSE) training precision was stipulated as 0.005 Watts. This value could have been lower but training time would also increase. The result of the training is a FFNN which is able to estimate the power consumption of the case study circuit following the mechanism of formula 5.5.

5.4 Results

This section draws all the results we gathered to quantitatively characterize the proposed approach. The experiments were carried out on a 12 core Intel Xeon @ 2.67 GHz and 24 GB of RAM. We initially show the correlation index evolution and trends obtained by applying the 2-phase strategy for training the gate type weights in the FFNN. Then, we compare peak power with estimated peak power values to verify that the approach is feasible and that it is able to provide a certain gain in time. finally, we show the results in terms of performance and comparison measures using the proposed methodology containing the FFNN and the old methodology containing a commercial tool for evaluating power consumption described in [79].

Case	SA versus Power	NSA versus Power	FFNN Versus Power
Best	0.8533	0.9812	0.9943
Worst	0.8316	0.9333	0.9752
Average	0.8439	0.9582	0.9858

Table 5.3: Correlation indexes comparison.

5.4.1 Training Trend

In Table 5.3, correlation index results for the Intel 8051 CPU core are presented. We have analyzed 11 relevant programs and evaluated the SA, NSA, FFNN estimated power and TA power consumption. These programs were obtained by using the automatic generation approach in [79] that uses a commercial power evaluator. At the end of each generation the best pattern is extracted, and when peak power improvement over a given number of generations becomes lower than a given threshold, the evolution is stopped. In this way we obtained 11 programs coming from an evolution using a commercial tool. They have a wide range of peak power, ranging from a small peak to a high one, and each of the selected individuals, best describes the 1,000 programs of each generation. Lastly, we computed the correlation indexes among TA power consumption and the above-mentioned metrics. In the table, we can verify that the correlation values increases from left to right, and the use of the FFNN improves the correlation measures at the peaks, thus justifying its use for estimating the power consumption. figure 5.8 shows the correlation index evolution for the Intel 8051 CPU core by applying the 2-phase training procedure. We have iterated 8 times from generation phase to training phase and vice-versa. As we can see from the graph, the initial trained neural network has a correlation index ranging from 0.8 to over 0.9. In the literature, it is said that these values are strongly correlated, but in fact they are insufficient when we try to match peak values, and therefore still not useful for the final generation phase.

Also, it is possible to validate the correctness of the FFNN by computing the correlation index among the estimated value and the power consumption evaluated on a commercial tool of several functional patterns, as shown in figure 5.8. As we increase the number of training steps, the correlation indexes and their ranges become even more strongly correlated. The final correlation indexes obtained are presented in Table 5.3. We have noticed that not only the overall correlation index has increased but also that the test program generation procedure was able to find higher power consumption peaks. In addition, we have spotted some misleading results, because the trained neural network provides an approximation of the desired value which can be over or under estimated defined by an error threshold computed by the training methodology. However, we have also verified that the estimated peak power in about 95% of the cases matches the actual peak power.

Figure 5.9 shows the values of SA, NSA, FFNN estimation and power evaluations for 30 clock cycles for a sample functional pattern for the 8051. This functional pattern was not used for training the FFNN. The correlation indexes are reported in Table 5.3 for the

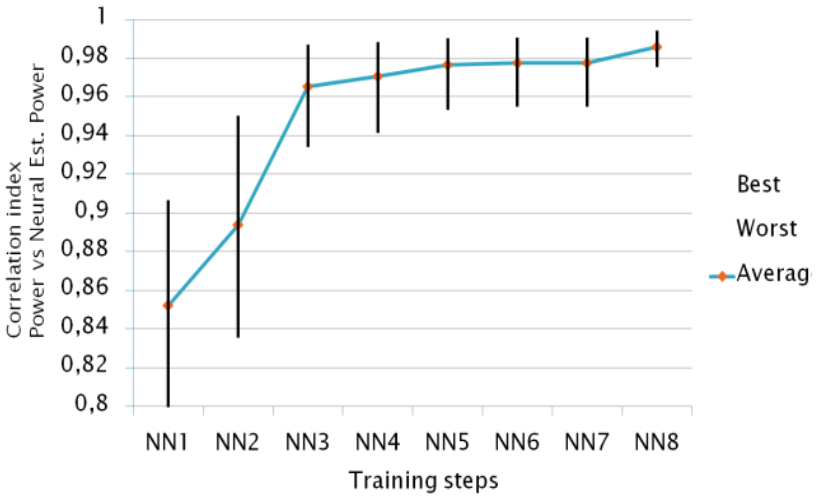

Figure 5.8: 8051 Training trend.

best case. In figure 5.9, the graphs have the resemblance with power consumption in an increasing order from SA to NSA and FFNN.

5.4.2 The Computed Gate Type Weights

In Table 5.4 the weights for each gate type computed by the back-propagation training method of the FFNN are reported. In the circuit there are 4,246 gates and the most used ones are reported. In the case study circuit there are 9 gate types responsible for 2,340 gates, whose weights are reported in the table. Also, their maximum capacitance values extracted from the spice library, average fan out extracted from the circuit and their number are reported.

5.4.3 Increasing functional Peak Power Evolution

In figure 5.10, the evolution of functional peak power consumption applied to the Intel 8051 CPU core is reported. The constant ascending curve (red) is the estimated peak power value computed by the FFNN-based fast power estimator. In the figure, we have extracted 10 functional patterns characterizing the evolution curve. In blue, the irregular ascending curve is the functional patterns' actual power value obtained using a TA power evaluator. There are two functional patterns in which disparity is high: 3 and 7. In the first case the FFNN has under estimated the power value while on the latter it over

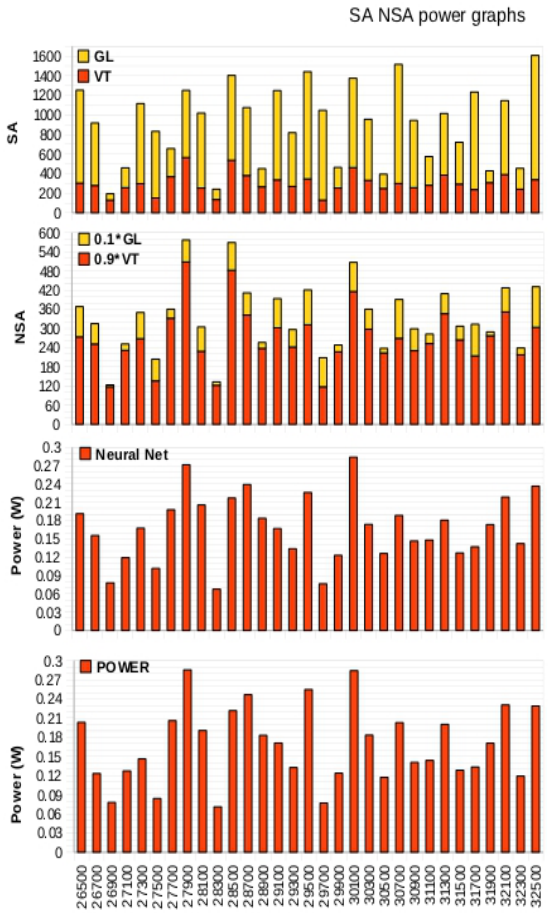

Figure 5.9: SA, NSA, FFNN power estimation and power evaluation of a sample program.

Gate type	Number	Capacitance (pF)	Average Fan Out	Computed Weight
D Type FF	429	0.16	6.8	0.0127
Inverter	404	0.035	2.41	0.0011
2-Input NOR	358	0.035	3.86	0.0053
2-Input NAND	349	0.053	3.72	0.0034
OR-AND inverted x 3	284	0.035	1.15	0.0043
OR-AND inverted x 2	185	0.035	1.01	0.0029
AND-OR inverted x 2	130	0.035	1.13	0.0009
AND-OR inverted	105	0.035	1.38	0.0005
4 Input MUX	99	0.07	1.22	0.0095

Table 5.4: Gate type characteristics and computed weights.

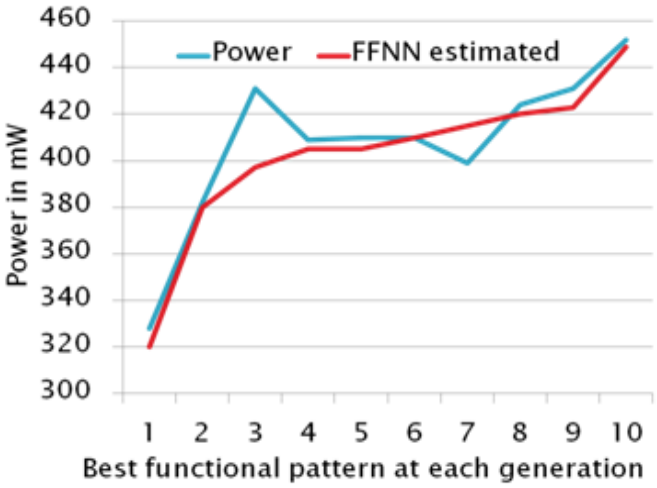

Figure 5.10: Peak power evolution.

Tool	Time(sec.)
Gate level simulation	90
Ad-hoc C NSA extraction	120
Power evaluation	400
FFNN back-propagation training	1,800
FFNN power estimation k	20

Table 5.5: Time consumption of each tool.

estimated it. The reason why FFNN incorrectly estimated these 2 values is because they provide corner cases which have not been used during training. This could have been fixed if the FFNN had been trained with better precision and used more sample patterns, although training time would increase significantly. Nonetheless, the evolution was not compromised by these miscalculated values. As we can observe, the TA power evaluation continues to increase irregularly but satisfactory to achieve our goal. In figure 5.10, only the first best individuals of 10 generations are shown, this is because the other successive 10 generations did not present improvements to this value, and therefore the approach was stopped because it arrived on a steady state.

5.4.4 Performance

This subsection describes the CPU requirements of the overall proposed approach. In Table 5.5, quantitative values for each tool used to generate, train and evaluate functional patterns in terms of time are reported.

CPU	Old method	Proposed approach
8051	430,000s (5 days)	190,000 s (2 days)

Table 5.6: CPU time requirements.

For the 2-phase training strategy, we have to separate the generation and training phase time consumption. In the first phase, we generated around 2,000 patterns, use the gate-level simulation, an ad-hoc C program for NSA extraction and for the usage of the FFNN.

The training phase uses the same tools used in the generation phase; additionally, the power evaluation and back- propagation training for the FFNN have to be executed.

The number of 2,000 patterns was obtained after achieving a 0.98 average correlation index between the FFNN estimated power and the commercial power evaluator.

In table 5.6, the final CPU time consumption of the proposed approach and the old approach are reported. The latter uses the automatic generation approach containing the EA as the program generation engine and the WSA power consumption evaluator that is provided in [79]. The time required to train the FFNN was around 2 days and was not included in the generation time shown in Table 5.6. The power evaluation using the trained FFNN requires 140 seconds which is around 60% faster than the commercial power evaluator used in the old approach [79]. The methodology applied to PLA power estimation could be much more time efficient, being twice as slow compared to RTL SA analysis but having an accurate measure (higher than 0.95 correlation index) compared to PLA power evaluation commercial tool.

5.5 Conclusions

In this chapter we have presented a methodology that automatically identifies functional test programs with maximal CPU core's peak power consumption; these programs could be useful to correctly define test power limits. We have performed an analysis of the switching activity and power consumption correlation measures. Then, we used a mix of neural networks and evolutionary computation to generate functional patterns data set to compute the correct weights for each gate type. After applying the training phase and iterating it over several training steps, we use the trained FFNN as a fast power estimator to be used in the automatic functional generation approach. The methodology has then been used for identifying functional test programs with maximal functional peak power consumption.

We have validated the proposed strategy on an Intel 8051 CPU core synthesized in a 65 nm industrial technology. The value of the correlation index computed during the 2-phase training strategy increases with the number of training steps. However, increasing too much the number of steps also makes the strategy and training slower. The results in fact show an evolution trend on the power consumption, although some irregularities

have been observed. Nevertheless, the evolution trend is also observed in the power consumption evaluation. Lastly, the proposed approach was able to reduce the time for power hungry test program generation from 5 to 2 days.

The proposed methodology is currently being experimented on the OR1200 CPU core, which is an open source architecture with a 4-stage pipeline: it has a more complex architecture (including caches) and for such reason the proposed approach needs some small adjustments to be applied: however, preliminary results seem to confirm the validity of the proposed approach.

6. Proposed SBST Pattern Generation

As already explained in the introduction of part 2, SBST patterns are useful for fault grading embedded CPU cores in automotive SoCs. During on-line testing, DFT mechanisms cannot be accessed because they may lead to system misbehaviors and consequently producing critical unrecoverable accidents, especially in safety critical applications. Therefore, structural testing must be performed through different mechanisms by using specific testing methodologies. The following sections in this thesis resorts to SBST patterns for on-line fault grading during the operational phase. This chapter is divided in two sections devised for on-line fault grading: the first proposes an automatic SBST pattern generation approach to quickly fault grade internal CPU modules and achieve a high fault coverage; the second proposes a manual SBST pattern in cooperation with a proposed external module for covering hardly functionally testable faults (HFTs) during on-line. Both approaches were evaluated on a 32-bit Power Architecture Automotive SoC developed by STMicroelectronics.

6.1 Automatic and Manual SBST Pattern Generation

The diffusion of electronic systems in the automotive field is increasing at a fast pace, and car makers constantly demand from electronic manufacturers for faster, less expensive, less power-consuming and more reliable devices. Microprocessor-based systems are employed in cars for a great variety of applications, ranging from infotainment to engine and vehicle dynamics control, including safety-related systems such as airbag and braking control.

The use of such devices in safety and mission-critical applications raises the need for total dependability. This requirement translates in a series of system audit processes that need to be applied throughout the product lifecycle. Some of these processes are common in today's industrial design and manufacturing flows, and include design verification and validation, performed from the early phases of product development, as well as various test operations during and at the end of manufacturing and assembly steps. More often, additional test operations need to be applied also during the product mission life, and may

include periodic on-line testing and/or concurrent error detection [81]. The reliability requirements need to be met by trading off fault/error coverage capabilities with admissible implementation costs of the selected solutions. Within the scope of microprocessor-based integrated systems, the on-line SBST approach has been addressed for a long time by the research community [82]. Compared to hardware-based test solutions, such as Built-In Self- Test (BIST) it presents many advantages, such as the possibility of autonomously testing both the microprocessor and the controllable peripherals in normal mode of operation, with no hardware modifications needed, and at-speed test application. Conversely, SBST methodologies raise some issues that have been limiting their application in industry throughout the years: those issues regard writing efficient and effective test programs and devising suitable methodologies for test application. Concerning in-field test application, a possible solution [36] relies in periodic test application during the system idle time (2.4). In simple words, the microprocessor is periodically (e.g., after each boot sequence at vehicle key-on) forced to execute a self-test code able to detect the possible occurrence of permanent faults in the processor core itself and the peripherals connected to it.

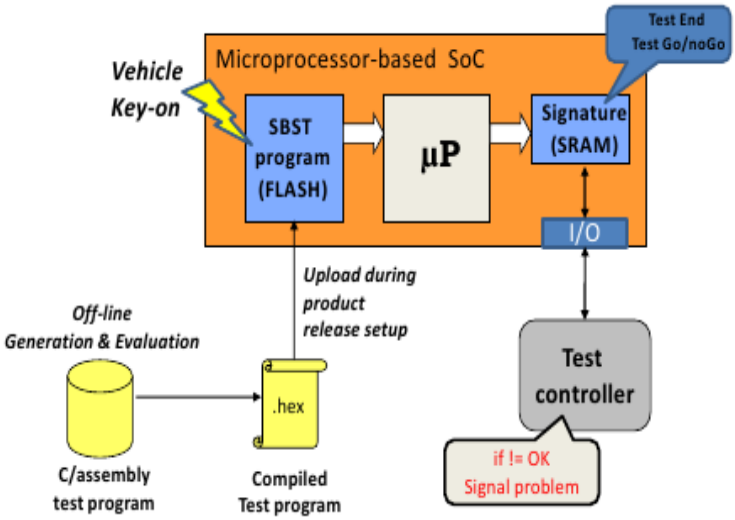

Figure 6.1: Conceptual representation of the in-field self-test procedure execution.

In the aforementioned approach, the SBST procedure is uploaded in a non-volatile (Flash) memory during the final device test; such procedure writes the self-test results in an available memory space, and then these are read out through a suitable serial port by an external Test Controller that inhibits the vehicle functional operations in case a faulty behavior is identified.

As far as it concerns test program development, many approaches can be found in the literature [83], employing manual or automated approaches, which are able to target different processor architectures and fault models. However, setting up an efficient SBST program generation framework within a consolidated industrial system development flow

involves a number of issues to be solved regarding the development and grading of the test programs.

This section presents an approach experimented by STMicrolectronics for the development and evaluation of SBST programs for microprocessors in safety-critical embedded systems. The pursued goal is to satisfy the reliability requirements given by emerging standards such as ISO/FDIS 26262 [3], which mandate a constant monitoring for the possible occurrence of permanent faults in the circuit. The actual test program generation is carried out employing manual and automated techniques, and an integrated framework has been setup for circuit analysis and partitioning and for fault grading of the generated programs.

6.1.1 Proposed SBST Program Generation Framework

Figure 6.2 presents a comprehensive view of the proposed test generation framework. The goal of this work is to obtain high fault coverage while satisfying constraints in terms of test code memory occupation and test duration. The first phase of the flow is circuit partitioning: this step is based on the analysis of the hierarchical RTL description in order to identify the main microprocessor modules, for which the test program generation development will be carried out following a divide et impera strategy so as to reduce the problem complexity.

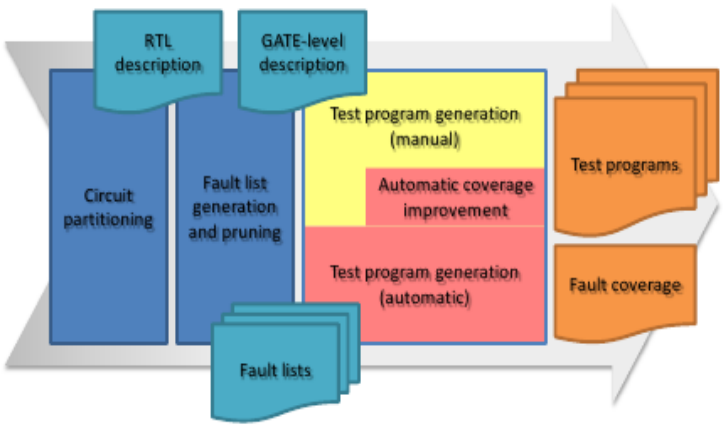

Figure 6.2: Conceptual view of the proposed integrated test generation framework

The gate-level description is then taken into account in the fault list generation and pruning phase, to generate the list of faults according to the previously identified module subdivision. In addition, untestable faults are removed to reduce the fault list size and to enable more accurate fault coverage computation in the following phase. Among the untestable faults that can be identified in this step there are the ones related to scan chains, which are usually implemented in the design and used for end-of-production testing, but

become useless at mission time. The following phase is the actual test program generation process. In the proposed framework, different strategies are employed to obtain effective test program, depending on the addressed microprocessor module. The different techniques are supported by a suitably developed fault grading environment [37], which provides a quantitative estimation of the effectiveness of the generated programs and useful information for their improvement. Further details about the fault grading process are given in the next subsection.

The manual test program generation process resorts to state-of-the-art methodologies that usually guarantee satisfying coverage results for the most common modules, such as the Arithmetic-Logic Unit (ALU) and the register file. Additionally, through manual generation of test programs, it is possible to explore particular processor conditions or activate specific corner cases, such as the operating modes dealing with the interrupt mechanism. An iteration flow guided by the outcome of the grading process is usually employed.

In parallel to the manual approach, an automated test program generation methodology is used, based on an evolutionary optimization engine [80]. Figure 6.3 shows the employed loop-based flow, presenting the three main blocks involved in an automatic run: an evolutionary engine, a constraint library, and an evaluator external to the evolutionary core (in this case, the fault grading environment). The constraint library stores suitable information about the microprocessor assembly language. The evolutionary engine, on the other hand, generates an initial set of random programs, or individuals, exploiting the information provided by the constraint library. Then, these individuals are cultivated following the Darwinian concepts of evolution, by applying mutations to the various individuals. The evaluation of the test programs is carried out through fault simulation, which evaluates different test metrics (fault coverage, memory occupation of the test code, execution runtime) and provides a fitness value which is used to guide the optimization process.

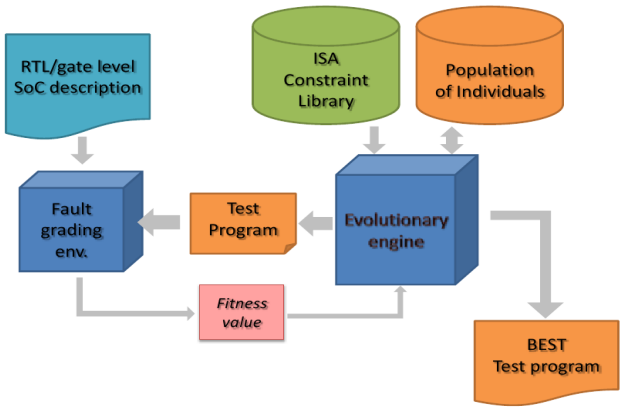

Figure 6.3: Automatic SBST Pattern Generation framework

The described evolutionary flow is also employed to increase the fault coverage figures

of test programs for microprocessor modules initially tackled through manual approaches, in the so-called automatic coverage improvement phase, when the algorithms employed in the manual flow are unable to provide satisfying coverage values. This is usually due to the peculiar architectures of the addressed devices or to specific synthesis choices. The obtained set of test programs still needs a manual refinement phase, to merge the obtained programs so as to build a comprehensive test procedure and, if needed, to cover additional corner cases or optimize memory occupation and test execution runtime. The obtained test procedure is finally prepared to be stored in the system code memory and run at the selected circumstances.

A. Fault grading environment

Fault grading is a extremely critical step of the test program generation flow, for different reasons. First of all, the output fault coverage figures need to provide a realistic information about the ability of the evaluated program in mission-like conditions. As an example, excessively optimistic results may lead to assume some faults covered by a test program, while in the field their effect may be masked during test application due to the lack of observability; conversely, overly pessimistic figures may uselessly make test program generation harder. In addition, since during the test program development a great number of programs usually need to be evaluated, the fault grading environment needs to be optimized so as to reduce the needed computational resources and time.

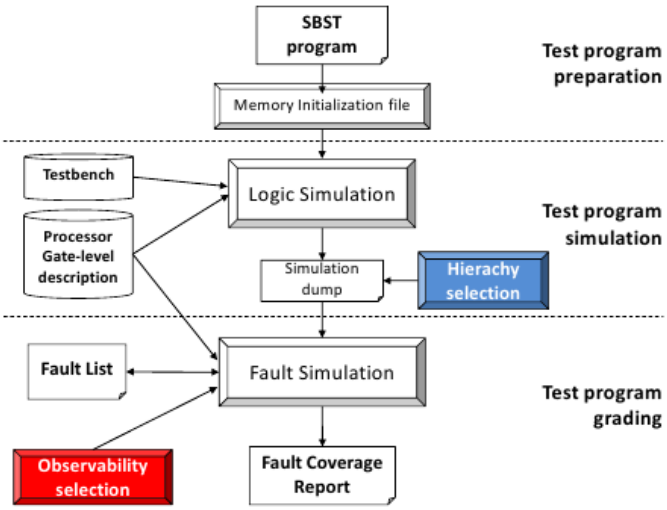

Figure 6.4: Fault Grading Environment

Figure 6.4 presents the adopted fault grading environment [37] . The main phases are

- *test program preparation*, where the developed high- level code is compiled and translated in a simulation- compliant format, and the code memory occupation is evaluated;

- *test program simulation*, during which the programs are run in the system using a suitable testbench, and a trace dump of the input/output signals of the addressed microprocessor module is produced. In addition, test application time is computed;
- *test program grading*, where the actual fault simulation is performed on the selected module, using as input the previously produced signal dump.

The most critical details to be taken into account are highlighted in red and blue in the figure and concern the selection of the circuit portion to be fault simulated and the signals to be observed to distinguish among correct and corrupted behavior. Hierarchy selection reflects the choices taken during circuit partitioning. A smaller circuit portion makes the fault simulation process faster, however, its ports may not be directly observable during test, and a higher hierarchy level may need to be employed in order to achieve fair fault coverage figures. Observability selection is another crucial point in the entire flow, since an inaccurate choice may lead to imbalanced measurements. If an internal hierarchy level has been selected during the simulation phases, all signals involved in the transport of observable fault effects outside the selected circuit zone have to be considered for observation, while the others must be left out. Furthermore, these signals have to be observed only during time instants when significant information is transmitted.

6.1.2 Case Study

The proposed test program generation framework is currently used to develop SBST program suites in STMicroelectronics microprocessor-based systems for the automotive field. The proposed test program generation framework has been built on the following tools: Cadence Incisive suite for fault-free simulation Synopsys TetraMAX for fault simulation μGP3 [80] as the evolutionary engine Ad-hoc programs, suitable developed to support the flow (fault list pruning, data format conversion, etc.) The experiments are run on a workstation featuring two Intel Xeon E5450 processors (8 processing cores running at 3 GHz) and 8 GB RAM.

As a case study, a SoC including a 32-bit pipelined MCU based on the Power ArchitectureTM is used for illustrating the proposed flow. It features a 576 KB Flash memory and 40 KB RAM, and is used for automotive chassis and safety applications, such as in airbag, ABS and EPS controllers (Figure 6.5). The focus of the program generation process was put on the SoC module called platform which includes the most computation-intensive and programmable components of the SoC. The addressed circuit modules include about 150K gates and 400K stuck- at faults. Figure 6.5 shows the SoC architecture and illustrates the hierarchical level chosen for fault simulation (in blue), the signals selected for observation (in red) and areas where untestable faults have been pruned (in green). The untestable faults removed from the fault lists include the embedded software debug infrastructure and the faults belonging exclusively to scan chain devoted components. The pruning operations were performed relying on ad-hoc tools working on the gate-level netlist in Verilog format.

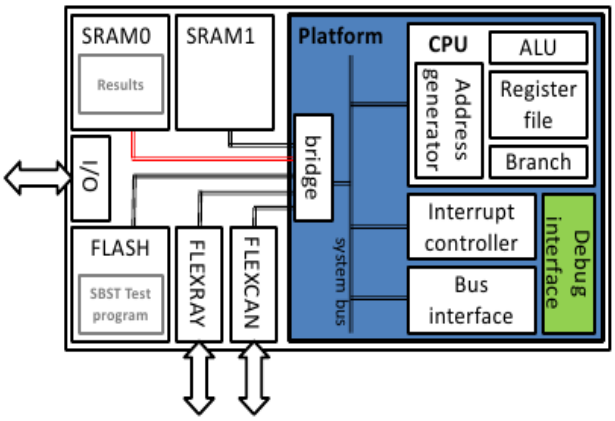

Figure 6.5: SoC architecture with hierarchy selection (in blue), observability selection (in red) and one of the untestable fault pruning zones (in green).

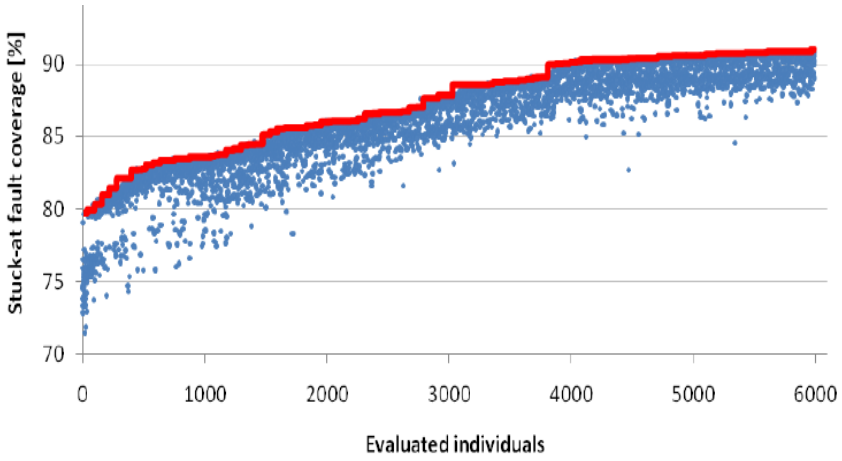

Figure 6.6: Evolution of test program fault coverage during automated program generation for the adder module; each blue point represents an evaluated individual, while the red line traces the generation best individuals.

Module	Generation methodology	Stuck-at faults [#]	Stuck-at fault coverage [%]	Development time	Clock cycles	Memory [Byte]
Register file	manual [83]	61,327	58.27	3 days	747	1500
Adder .	manual [83] aut. improvement [80]	. 4,188	56.67 91.79 (incr.)	3 days 10 days	9,013 1,384	160 838
Multiplier	manual [84]	31,809	95.25	3 days	376K	331

Table 6.1: Results of SBST program generation on some microprocessor modules.

Table 6.1 reports some preliminary figures regarding test program generation for some datapath components, obtained employing the automated and the manual test program generation flows. For the presented results, no methodologies were used to reduce test execution runtime; Concerning the adder module, the table presents coverage values for a manually generated program and for the following automatic program improvement, which yielded a relevant coverage increment.

Figure 6.6 illustrates the coverage evolution in time during the automatic improvement of test program for the adder module, starting from the coverage obtained with the manual flow.

6.2 On-line Fault Coverage of the Address Calculating Unit

The SBST methodology applied to automotive applications explained in the previous section sometimes has coverage issues when targeting specific CPU modules. For example, the Address Calculating Unit is responsible for computing the address of the next instruction to be executed. If an SBST tries to randomly stimulate this module, then the application and the test may be compromised due the inability of executing the correct instruction lines to save test results and return to normal application. Severe program flow issues can invalidade the SBST methodology presented. However, the unit needs to be tested by a devised SBST without compromising the test and mission while presenting a high fault coverage for the specific module.

This section presents a methodology exploiting manual and automatic SBST pattern generation for increasing significantly the fault coverage on the Address Calculating Unit without disturbing the mission. The unit is a specific adder which can only be stimulated by using *load* and *store* instructions. The rest of this section is composed of the proposed methodology, case study, results, and conclusions.

6.2.1 Proposed SBST Generation Approach

The proposed methodology is to introduce an effective strategy to generate SBST programs, or routines, suitable to be run periodically during the device mission to test the Adder Calculating Unit. The illustrated strategy falls into the non-concurrent on-line testing domain because the mission application is interrupted at regular intervals to let the self-test routines run. The address computation is performed by a dedicated adder, whose purpose is to sum an offset to a base value to address the RAM memory for reading or writing a value. Usually, this adder is not part of the processor ALU, and thus it does not perform any arithmetic computations required by instructions like *add* and *sub*. Testing an adder is often deemed as a trivial task, but in the case of the address generation module, controllability and observability are limited, and on-line requirements pose additional constraints. The criticalities in testing this module by using a software-based approach are mainly due to the type of instructions (load and store and all their variants) that activate the address calculation. In fact, a test program including many of such instructions may potentially induce some undesirable effects:

- Store instructions may corrupt the mission data and compromise the correct resuming of the system.
- Load instructions may retrieve data from memory zones (i.e., the parts containing the mission application variables) whose content can hardly be forecasted a priori, therefore compromising the signature generation, no matter how it is calculated.

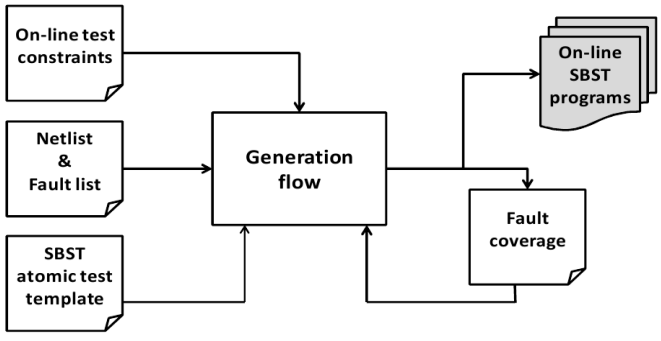

Figure 6.7: Conceptual view of the proposed generation approach

Taking these factors into consideration during on-line test generation is a must; careful planning of a SBST generation campaign should early consider memory access constraints. Figure 6.7 shows the conceptual view of the generation approach we propose. The main block in the scheme is called Generation flow. Other than usual inputs, such as the netlist of the circuit and the fault list of the tackled module, the Generation flow considers a set of constraints imposed by the coexistence of the test routines and the mission application, and leverages on a SBST atomic program template whose structure is tailored

to take the on-line constraints into account. The Generation flow is composed of a set of steps that are detailed in the next paragraphs; along the generation process, the fault coverage is a feedback measure that allows the program generation to proceed towards a set of routines guaranteeing high fault coverage. The final result of the generation process is a test suite mainly performing a sequence of load and store instructions, along with some arithmetic instructions devised to setup base and offset values for memory access.

A. On-line test constraints

The previous subsection outlined the general constraints that a self-test procedure has to respect. In the specific case of the address calculation module, limitations are even stricter:

- Store instructions have to write only in reserved memory locations, possibly contiguous in the addressable space
- Load instructions have to read from memory sections never varying in content, possibly reserved for this purpose
- Memory zones reserved for writing and reading should be separated, to avoid the aforementioned problems
- Both read and write zones need to be programmed with suitable initial values
  - Arbitrary values are suitable for reading zones, provided that they are diverse within the zone; in this way the method minimizes the aliasing potentially stemming from a compromised memory access (e.g., when the reading zone is all 0s, accessing a wrong address in the range may cause non-detection)
  - The all 0s value is fine for the writing zone, since values transferred to the memory are sourced from the reading zone, which is properly filled.

The setup of the initial memory content is a critical operation, since it may be affected by faults in the same logic for address calculation to be tested. To avoid undesired fault effects, direct memory access (DMA) driven initialization is suggested; for example, the reading zone may be filled by copying part of the test routine code which is invariant, while this is not guaranteed for the mission program code. The first issue arising from these limitations concerns the selection of memory zones. This selection directly reflects in the effectiveness of the generation process. Ideally, reserving a single small memory portion is desirable; however, a too small address range may prevent the achievement of a high coverage. For making an effective selection, it is useful to define first the acceptable amount of memory M that can be reserved for testing purposes (called Test Memory). This portion of memory can be eventually shared with test programs to store signatures; therefore, we suggest that the test of the address calculation module should be the first module considered in a SBST suite development plan. To select the range R defining the

starting and ending addresses of the Test Memory given the dimension M of the available memory, the following simple formula 6.1 can be used, where N is equal to the number of addressing bits; the formula assumes that the Test Memory is located in the middle of the address space of the whole memory:

$$R = [2^{N-1} - M/2, 2^{N-1} + M/2 - 1] \tag{6.1}$$

The effectiveness of this choice is due to the fact that in this way the Hamming distance between the extreme address values is maximum, i.e., it is equal to N, allowing the address calculation adder outputs to exhibit a large value variance.

For instance, let us suppose that the addressing space is 32 bits, but only with 128KB available within the address range $Rdata = [0x40000000, 0x40001FFF]$ corresponding to an addressable subspace of 17 bits. Supposing to have 1KB to be allocated to test purposes, the selected address range is: $R = [2^{16} - 512, 2^{16} + 512 - 1]$

Which is relative to the Rdata segment start address. Hexadecimal values better show the effect of this choice considering the Hamming distance in the range $R = [0x4000FE00, 0x400101FF]$ all 17 less significant bits are changing values. Figure 6.8 visualizes the effect of such address range over the address calculation adder. As a counter-example, let us locate the 1KB test data on top of the memory addressable space, in the range $R = [0x40000000, 0x400003FF]$ in this case, only 10 bits in the address range can be varied.

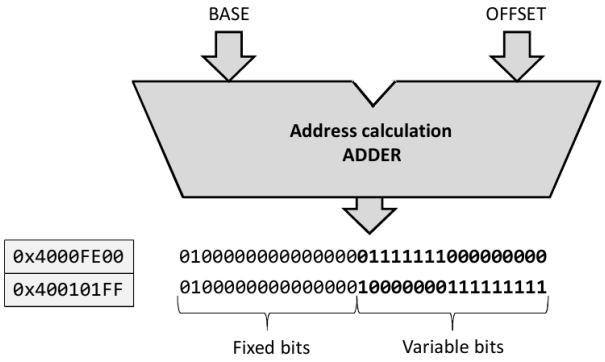

Figure 6.8: Effect of address range selection for address calculation adder test

The identified memory space can be used either for writing or reading by defining 2 sub-zones.

As it can be noticed, the test effectiveness of the address calculation adder is strongly limited by the available address range, since several adder output bits may be fixed to a constant value. To overcome this problem, a viable solution can be based (whenever possible) on accessing other chip resources located out of the Test Memory (i.e., 0xBFFF000 may be a suitable complement to the previously selected address range). As an example,

in most real cases the addresses reserved for peripheral core registers can be easily and safely accessed. Moreover, the selected complementary locations may be used for reading only, thus guaranteeing that their content is never changed.

The test program structure described in the next paragraph is suitably studied to cope with the identified requirements.

B. Atomic block structure

Roughly speaking, test programs targeting processing modules need to apply suitable values to the module inputs by means of controlling instructions; some instructions are used to setup the data to be elaborated by the tested module when a target instruction is executed, and then results are propagated to the processor outputs.

Testing the Address Calculation unit by using the SBST principles means reading and writing data from the memory at suitable locations. Let us consider the following generic memory access instructions (store and load):

```
Sd Rx, base (offset)
Ld Rx, base (offset)
```

When using such instructions, the actual address is calculated by adding the values provided as offset, and base, which usually correspond to two registers or to a register and an immediate value encoded in the instruction, respectively. The execution of this instruction excites the Address Calculation module by applying the base and offset values at the module inputs.

For testing sakes, base and offset have to hold suitable values to achieve fault coverage; in the on-line scenario, additional constraint must be considered and the resulting addresses must belong to a small memory range corresponding to the Test Memory.

To overcome the issue of performing effective sum operations while matching these strong constraints, we propose the usage of an atomic block devised to support the test generation process. The atomic block illustrated in the following can be used as a building element for the test program targeting the address calculation unit. The usage of similar blocks, called building blocks, was introduced in [85]; however, the authors did not consider the particularities of the address calculation modules of pipelined processors, neither the constraints regarding on-line testing.

In short, the proposed structure first loads a data value from a test memory location in the read reserved space; then this value is modified, and finally, it is saved again in a writable test memory zone. The pseudo-code of the atomic block is shown in Figure 6.9.

The first two instructions in Figure 6.9 (lines 1 and 2) load random values in the registers *rA and rB*. The value in *rA* is a constrained random address value (*rd* constrained)

selected within the readable Test Memory addressing range: a known value must be stored previously in memory at this address. On the other hand, the value placed in *rB* is purely random without any constraint. The addition and subtraction instructions of lines 3 and 4 prepare registers *rC* and *rD*, which activate the arithmetic adder; in addition, these instructions manipulates the registers involved in the load instruction placed at line 5, that accesses the memory location at the address in *rA*, during which the address calculation module performs the addition between *rB* and *rC*. The memory value is read in register *rE*. The instruction at line 6 manipulates *rE* by applying the function *f(rE, rD)*, which is aimed at merging the results of the memory read (line 5) and the arithmetic addition (line 4). During our experiments, the function *f(rE, rD)* was replaced by the logic *XOR* instruction on the registers *rE* and *rD*. The advantage introduced by this function is that both address calculation and arithmetic adder are tackled obtaining high levels of fault coverage without additional effort. The value in *rE* is later stored in memory completing the observability task of the atomic block for the test of both adders.

In the second part of the pseudo-code, (lines 7-9) new random values are loaded in *rA* and *rB*; both of them are constrained values (*wr* constrained), since they are used to calculate the destination address to store *rE* in line 9. Clearly, the addition of *rA* and *rB* must produce a value placed in the writable Test Memory. Finally, it can be noted that line 9 collects the information elaborated by the atomic block and uses a store instruction to send it to the appropriate location.

```
1.    rA ← RNDM (rd constrained) value
2.    rB ← RNDM (unconstrained) value
3.    rC ← rA - rB
4.    rD ← rB + rC
5.    ld_inst rE, M[rB,rC]
6.    rE ← f(rE, rD)
7.    rA ← RNDM (wr constrained) value
8.    rB ← RNDM (wr constrained) value
9.    sd_instr rE, M[rA, rB]
```

Figure 6.9: Atomic block pseudo-code

The structure proposed for the atomic block is as general as possible, so as to cope with different processor realizations. In the example above we supposed that the two elements involved in the memory access instructions correspond to a couple of registers. Clearly, depending on the targeted case, the adopted solution may include a couple of registers or an immediate value and a register.

C. Test program building flow

A suitable test program can be built by exploiting the atomic block introduced in the previous section. Its final form is a sequence of atomic blocks including selected values for exciting and propagating the address calculation adder faults. Possibly, no loops should be

included in the test program code since those constructs usually lead to a long execution time, even if they permit saving program code lines.

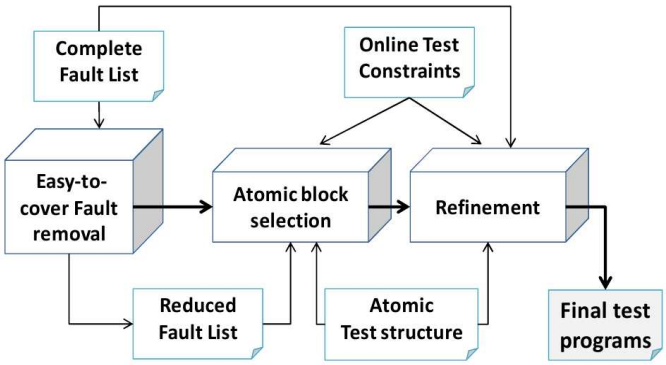

Figure 6.10: Proposed framework for on-line testing

To perform accurate and fast selection of the required values, we propose a three-step generation process, whose graphical view is reported in figure 6.11. The result of this flow is a program composed of a sequence of carefully selected atomic blocks to be sequentially executed. The three steps differ due to the set of faults they work on, and on the test generation method they adopt.

The proposed flow derives from this concept: in any circuit there are faults that are easy-to-cover, i.e., they are detected by a large set of patterns, while there are other faults that require very specific test sequences. Therefore, we propose:

1. to initially remove from the fault list a possibly large set of easy-to-cover faults, even using unconstrained test programs;
2. to produce focused atomic block values considering the remaining faults, in this phase the on-line constraints are taken into account; more in details
   a. when the required level of coverage is reached, the obtained test program is graded with respect to the whole fault list (including easy-to-cover faults previously removed)
   b. most of easy-to-cover faults will result covered also in this case; therefore
   c. the coverage figure will only slightly decrease;
3. to integrate some more atomic blocks with proper values to refine the test program.

The usual fault coverage trend observed along the generation process is shown in figure 5.

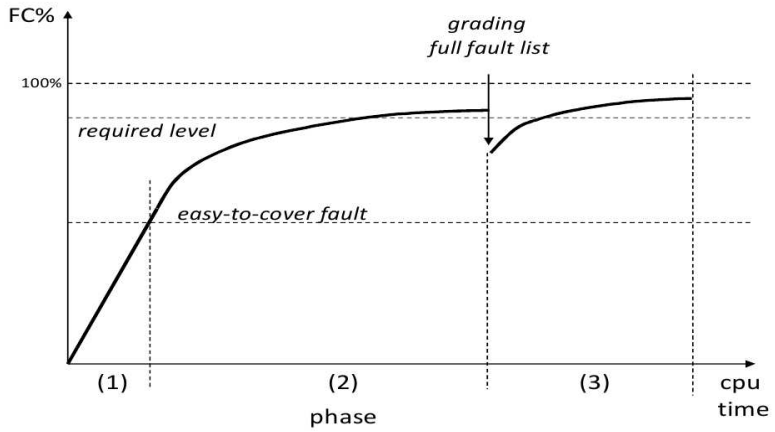

Figure 6.11: Fault coverage general trend along the generation process

1) Easy-to-cover Faults Removal

In this phase a preliminary test program is written and evaluated without considering any on-line test constraint. This program may be composed of few manually selected instructions exciting the address calculation unit; particular cases, such as 0 values for base and offset, and random values may be included. As a result, many faults will be covered, which are said to be easy-to-cover. These faults are usually related to the logic parts of the address calculation adder located close to its inputs and outputs. The coverage achieved by this process is usually quite high and up to 50% in some cases. The faults not covered are used as the input fault list for the next step.

2) Atomic block selection

Starting from the reduced fault list inherited from the first phase, the generation process strongly relies on the atomic test structure; such a generation process must comply with the constraints imposed by the on-line test environment. Therefore, in this phase the faults not belonging to the easy-to- cover category are considered, only; several test generation strategies can be adopted to identify the values to complete the atomic blocks and create a suitable program, as described in the next subsections. The goal is to reach the highest possible fault coverage, considering the cumulative effect of the test programs resulting from phases 1 and 2. As shown in the experimental results, the coverage on the complete fault list may reach up to more than 90%.

3) Refinement

At this point of the generation flow, the program obtained during phase 1 is removed. The coverage of the single program developed during phase 2 is evaluated; what is normally

observed is a slight decrease in the fault coverage level, but this is usually not substantial, since most of the easy- to-cover faults are detected by both programs. Starting from this new fault list, the refinement process is another generation step that tackles the few faults not yet detected by the on-line oriented program produced in step 2. We underline that the major novelty of the proposed approach lies in the introduction of the atomic block and in the adoption of a 3-steps test generation method, while the techniques used in each step are not crucial.

6.2.2 Case studies and Results

In order to verify the effectiveness of the proposed approach we performed our experiments on two 32-bit pipelined microprocessor case studies:

1. an automotive microcontroller by STMicroelectronics
2. a demonstrative case based on the miniMIPS core [86].

Following the flow described in the previous subsections, initial constraints were defined in both the presented cases. Firstly, it was assumed that the size of the memory devoted to allocate the on-line testing programs for the address calculation unit is 4KB (2KB for code and 2KB for data). In addition, realistic spaces for code and data memories were also defined for the specific test purpose of the considered unit.

Special considerations have to be done regarding the atomic block selection methodology, since it is possible to adopt different strategies to identify the number of atomic blocks composing the on-line test program, as well as the most suitable values for the operands involved in every atomic block. In the presented cases we adopted two strategies briefly described in the following:

- Evolutionary optimization tool: it is possible to use an evolutionary optimization tool such as the one described in [80]. The evolutionary tool can include the atomic blocks as a building element to assemble the test programs. In this case, the evolutionary optimizer can fine-tune the constrained and unconstrained random values in order to maximize every atomic block testing capacities.
- Random: a random strategy can utilize the proposed atomic block as an assembling structure where to put the constrained and unconstrained values. A test program can be generated out of a predefined number of atomic blocks.

Special care needs to be taken for fault coverage assessment. In [87] the authors describe the method used in this work to evaluate the effectiveness of a SBST test set: this step is very critical, since it has to provide a fair coverage evaluation while taking into account the observability constraints in the on-line test application context, and fault simulation can be a very intensive computational task.

The first case study is a SoC including a 32-bit pipelined microprocessor based on the Power Architecture TM. It is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers. This device contains over 2 million logic gates, including a 576 KB code Flash memory (2 KB of this memory are devoted to store the address calculation test program) and a 128 KB data RAM (a contiguous 2 KB space is reserved within it for the Test Memory). It also includes other modules, such as an interrupt controller, different buses and I/O interfaces, and a debug controller. Address calculation is performed within the ALU, where two parallel adders are included in the same unit; using different ports, they are in charge of performing both arithmetic operations and address calculations. This module counts 689 gates and 4,188 stuck-at faults; an additional difficulty is that faults in the module affecting arithmetic or address calculations cannot be distinguished.

We applied the generation flow described in the previous section including 1) an easy-to-cover fault removal consisting in a loop-based SBST strategy [82]] mainly devoted to cover arithmetic adder faults, 2) an evolutionary approach exploiting the μGP3 tool [87] that only includes a macro implementing the atomic block described in the previous sections, and 3) a refinement phase, again resorting to the same tool and considering corner cases and operating on the full fault list. The progression in the fault coverage values for stuck-at faults is shown in Table 6.2. Interestingly, the final test program counts only 31 atomic blocks, each composed of 13 instructions.

-	-	Test generation	Test generation	-
Case study	Easy-to-cover fault removal	Cumulative —— (phases 1+2)	Phase 2 only	Refinement
1	56.67	91.79	86.66	95.11
2	58.02 (59,51)	88.09 (90.72)	81.76 (82.85)	94.27 (96.38)

Table 6.2: Stuck-at Fault Coverage Percentual Obtained Along the Flow

The second case study is based on miniMIPS [86], a processor core based on the MIPS I ISA. It features 32-bit buses for data and addresses, and includes a 5-stage pipeline. It was synthesized using Synopsys Design Compiler and an in- house developed library, resulting in 33,117 logic gates. In this case, the adder performing the address calculation is a clearly separated unit within the execution stage, counting 342 logic gates and 1,988 stuck-at faults; both address calculation and arithmetic adders count 757 gates and 4,408 stuck-at faults. The same 3-step strategy was used in this case. However, we employed a random approach for the atomic block selection operation. The results are also shown in Table 6.2, where fault coverage values are reported for both adders as well as for the address calculator adder alone (in parenthesis). For the miniMIPS case, the atomic block was reduced to only 6 instructions, and the final program counts 65 atomic blocks. The proposed methodology is exploited resorting to two different generation strategies, an EA- and a random-based one. Remarkably, in both cases good coverage results were obtained, and the final test programs take about 1.6 KB, and require about 800 clock cycles to execute.

In order to corroborate the importance of the selection of the test memory placement, we performed a new experiment using a variable number of atomic blocks, and different code memory allocation constraints: the entire unconstrained processor addressing space, the 2KB space starting at address 0x00000000, and a configuration especially selected for on- line test, i.e., the address range 0x20007C00-0x200083FF. Table 6.3 reports the obtained coverage results. It can be observed that a careful selection of the memory space attains better coverage results, and that a small (2 KB)

Atomic blocks	entire memory	2K beginning	2K selected for on-line
8	80.58 %	69.57 %	77.11 %
16	82.24 %	72.03 %	80.68 %
24	82.95 %	72.59 %	81.54 %
32	83.15 %	73.26 %	82.85 %

Table 6.3: Stuck-at Fault Coverage for different atomic block/memory configurations for case study 2 (before refinement)

Test Memory allows achieving a fault coverage comparable to that achievable having the whole memory accessible (which is hardly the case for on-line test).

6.2.3 Conclusions

On-line test application poses a number of constraints to the SBST approach for microprocessor-based systems. The most critical aspects related to the problem were reviewed, and a new methodology for the generation of test programs for address calculation circuitry was proposed. The method is mainly based on the adoption of an atomic block, which can be replicated several times in the test programs; the choice of the parameters for each atomic block can be performed using different techniques. Experimental results obtained on both an industrial and a representative case study demonstrates the efficacy of the approach under on-line constraints.

6.3 Increasing Fault Coverage of CPUs During On-line Functional Test

Processor-based systems are increasingly used even in application areas where safety is crucial, such as automotive and aerospace. These systems become more and more complex, and every year technology scales, bringing smaller and more complex components. This increases circuit sensitiveness that may jeopardize safety-critical applications, which often rely on testing during the operational phase to detect possible faults and foresee their negative effects. Major concerns about ageing effects further push the need for effective on-line test techniques.

Standards and regulations define constraints aimed at better guaranteeing that a sufficient level of safety is achieved in several domains. As an example, the ISO 26262 standard for automotive systems mandates the application of a continuous sequence of tests on any safety-related system of a car (e.g., in the braking system, or in the airbag control), so that failures do not cause catastrophic events during the operational life, especially due to aging effects when reaching the end-of-life wear out. In order to match the above strict requirements different solutions can be used. Some of them are based on introducing some hardware redundancy, such as Triple Modular Redundancy, Duplication with Comparison (including Lock-step solutions), and monitoring with watchdog modules: these techniques effectively face not only permanent, but also temporary faults.

On the other side, software solutions based on introducing redundancy in the application or system code may also allow detecting faults. Both in the case of hardware and software solutions, key evaluation parameters are the achieved fault detection capability and the cost (in terms of additional hardware, additional code, decrease in performance, additional design cost, etc.).

When cost is a major concern, alternative solutions are considered, based on performing a test specifically addressing permanent faults. This test can exploit Design for Testability (DfT) solutions, such as BIST, or being based on a functional approach. In the case of processor-based systems, this approach is often based on forcing the processor to execute a suitable program, and then observing the produced results (e.g., in memory) and comparing them with the expected ones. This solution is known as Software-Based Self-Test, or SBST [31]. The SBST approach provides some advantages, especially in terms of flexibility (the test program can be easily changed), defect coverage (the system is tested exactly in the same configuration and at the same frequency used in the operational phase) and cost (no changes in the hardware are required). Moreover, this approach can in principle be tailored to identify faulty modules during mission providing on-line diagnosis facilities. On the other side, SBST adoption may be limited by the effort required to write the test program, especially when the test is developed by the system company, which often does not have access to detailed information about the internal device architecture. Moreover, SBST has some requirements in terms of memory (to store the test code and data) and time (to execute the test program) that may conflict with the application requirements.

In particular, when the SBST approach is adopted for the test during the operational phase, additional requirements may exist, in particular in terms of test duration and invasiveness [32] [88] [89]. Hybrid solutions resorting to the addition of an external module have also been proposed [90]. When required, the SBST approach can be extended to the test of communication [91] and system [92] peripherals. Recently [93], it was observed that some faults may result to be untestable during the operational phase functional test, for example because the circuitry they belong to was introduced for software debug or end-of-production test purposes, and it is not accessible any more in the operational phase.

These faults (called On-line Functionally Untestable Faults, or OLFUF) do not affect

the device behavior in the operational phase, and hence should not be the subject of any detection effort; hence, their identification may prove to be particularly important. In this chapter we focus on microcontrollers, and consider a further group of faults (that we define as Hardly Functionally Testable, or HFT, faults) that are hard to test during the operational phase when the functional approach is adopted. The difficulty in testing this particular group of faults mainly stems from the fact that they require some specific events occurring on an input signal that can hardly be triggered during a functional test. However, HFT faults are not untestable (hence, they do not belong to the OLFUF set), and their occurrence could severely limit the functionality of the system, despite their limited number. Examples of HFT faults are those in the logic managing the Non-Maskable Interrupt signal (NMI), or those in the circuitry for detecting errors in the bits received through a serial channel, or even power control management modules existing inside the microcontroller.

In order to make the test of HFT faults possible, this PhD thesis proposes the usage of a small hardware module to be added outside the microcontroller (hence, without changing anything inside); the module is connected to the bus as a memory-mapped peripheral component, and can be programmed using LOAD/STORE instructions in such a way that it can trigger some desired events on specific target input signals. In this way the test of the HFT faults becomes feasible even in a functional manner. The module could be implemented as an Infrastructure IP to be included in the design if we are addressing a System-On-Chip. On the other side, if the target system corresponds to a board the module may be implemented in an FPGA device close to the controller. We also describe how the corresponding SBST test program should be organized, and outline the phases which are required in order to allow suitable communication to take place in order to activate some desired external signal and thus detect the targeted HTF faults.

In order to experimentally evaluate the proposed method, we considered four cases of study: the first is the circuitry driven by the NMI signal; the second is the circuitry in charge of managing a check-stop state (corresponding to a non-recoverable state); the third one manages the power consumption navigating through the low power states; the last monitors the processor status and exception enabling. We focused on instances of these modules belonging to an industrial 32-bit Power architecture microcontroller with variable-length encoding provided by STMicroelectronics. We developed the proposed hardware module and wrote the test program to detect the target HFT faults and thus increase the fault coverage. We could then evaluate in all cases the advantages of the method (in terms of further faults that can be detected) and its cost (in terms of size of the hardware module). In order to optimize resources, we can integrate the stimuli for all case studies to increase fault coverage.

Although the method was evaluated on a few cases of study, it is general enough to be easily adopted for the test of HFT faults in several different blocks that can be found in today's SoCs. It is designed to be simply and easily portable to any other device.

A preliminary version of this chapter was published in [94], where a simplified version of the hardware module was proposed, and the set of considered benchmark cases

was significantly smaller. We also considered covering HFT faults of the *Universal Asynchronous Receiver Transmitter* (UART) peripheral.

6.3.1 Background

Testing processor cores embedded in SoCs during the operational phase is a hard task. In this case, DfT techniques cannot be used since the component can work in functional mode, only. In this scenario, even if some DfT structures exist in the device (such as scan or BIST), they are typically not accessible anymore, unless they have being designed for this purpose since the beginning. Also, the test techniques must be employed matching several constraints in order not to disturb the normal operational mode.

In this section we describe the state-of-the art microprocessor testing by using software-based self- test (SBST) approaches (see Chapter 2). Moreover, the test scheduling is described in order to devise and understand when it is the convenient moment to run specific SBST patterns for testing the different components inside the CPU. Then, we classify the on-line fault universe and the relationship among the different groups of faults.

Testing embedded CPU cores during the operational phase

Testing CPU cores during the operational phase within safety-critical applications is challenging since it must not interfere with the normal mode while providing sufficiently high defect coverage. Moreover, application and test programs must be correctly scheduled in such a way to provide mission safety during the operational phase.

In [32], the on-line periodic testing paradigm was defined such that test can be scheduled correctly without interfering with the application's correct execution. In figure 6.12, scheduling is organized in such a way that test schemes are scheduled according to their invasiveness and execution time: the figure show that test can be performed at power-on, at power-off, and during the operational phase. In the latter case, the test exploits the short idle times left by the application. The critical SBST patterns testing interruptions or even crucial processor states tackling difficult faults must be scheduled at power-on or power-off instants. They can also be scheduled when the application is in a safe situation during long idle times. In this way it is possible to schedule periodic tests without performance loss and with minimal impact on the system mission.

Fault classification

When testing a processor-based system-on-chip through a functional approach during the operational phase, it is usual that some resources used for design and silicon debug, and for manufacturing test are no more accessible. These structures can basically be classified into two groups:

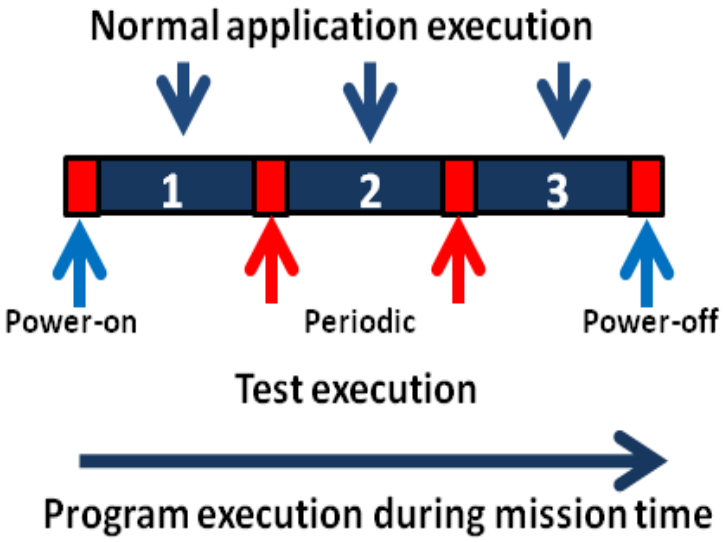

Figure 6.12: Test scheduling during the operational-phase.

- Design for Testability circuitries controlled directly on the boundary of the chip by a tester during manufacturing test, including:
  - Scan chains
  - Boundary scan and IEEE 1500 structures
  - Built-in self-test modules.
- Design for Debug units controlled by external controllers during silicon and software debug, such as Nexus-compliant modules.

When the device is mounted on the PCB board, the access ports of these modules are often soldered to ground or Vdd, or pulled to a fixed logic value when the system is put in its mission field. This is sometimes done for security reasons too, thus reducing the possibility to exploit these access points for supporting an illegal attack to the circuit.

The direct consequence of such a final configuration of the system is the appearance of a further set of untestable faults that we call On-Line Functionally Untestable Faults. Such faults are testable until the structures they are related to are used, but not in the final environment. There is another group denominated Hardly Functionally Testable Faults that cannot be easily detected during the operational phase by applying typical SBST patterns. These faults are linked to the logic attached to external pins (inputs and outputs) which cannot be easily and properly stimulated by functional approaches.

In figure 6.13, the fault universe is drawn. The rectangle represents all on-line related fault universe, whether testable or not. There are three groups of faults devised in the picture; Hardly Functionally Testable Faults; On-line Functionally Untestable Faults, and the

complement of these two groups w.r.t the fault universe, which are the on-line detectable faults. The group of Hardly Functionally Testable ones is a subset of the latter.

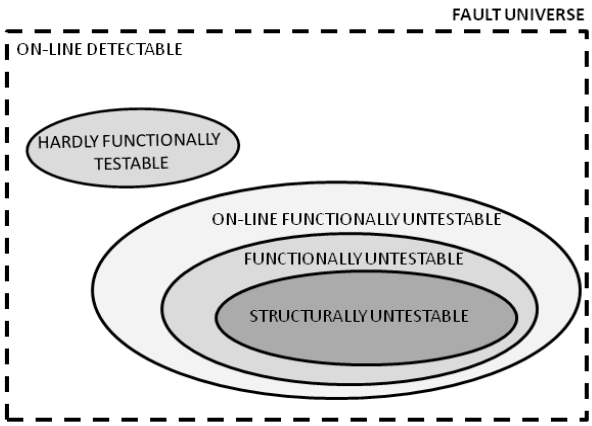

Figure 6.13: On-line fault universe and relationship among categories.

6.3.2 Definition of the Problem

In a typical microcontroller or (more in general) in a processor-based system it is possible to find blocks of logic that can hardly be tested during the operational phase using a functional approach. This is especially true for logic blocks that are only stimulated and/or observed if an external signal is properly triggered from an external source.

In fact, the functional approach is performed without switching the system to any special Test Mode, nor resorting to any Design for Testability feature: the test is performed by running some special test program, and looking to the results it produces.

Moreover, when functional test is used during the operational phase, the duration of the test is often a strong constraint, as well as its invasiveness (i.e., impact on the system resources): hence, the test is supposed to minimally disturb the system status. Therefore, to fully test the microcontroller it is necessary to develop several small test programs targeting the different modules, each having a short duration and suitable to be scheduled together with the application code, in such a way that their invasiveness is reduced to its minimum.

As an example, sometimes it is difficult or even impossible to activate during the test the external signals that are required to stimulate some specific piece of logic managing the exceptions through the corresponding input pin. A typical example of this situation is the logic driven by the Non-Maskable Interrupt signal (or NMI): this signal and the related logic are often very important for the system behavior (e.g., because they allow the management of critical situations, such as those related to power- down cases), but it is practically impossible to activate this signal by just working on the software side (as functional test does).

More formally, we could define HFT faults as those faults that require activating an input signal that cannot be activated resorting to the system software, only. Also, there are a number of HFT faults that are only testable if the CPU is forced to enter into a crashing state, thus requiring that an external reset is activated to return the microcontroller to the operational phase. It is important to underline that:

- HFT faults do not belong to the category of OLFUFs, as defined in [93], since the input signals they depend on are not physically tied to any fixed value, but are simply not controllable via software
- Although HFT faults are in general not so numerous, their effects may be particularly critical, and hence the importance of devising a solution for their test.

6.3.3 Proposed Solution

In this paper we aim at devising a solution able to stimulate external pins of a microcontroller and consequently to support the test of some corresponding logic blocks in order to increase the functional fault coverage that can be achieved during on-line testing. Our target is to cover the HFT faults that cannot be detected by using conventional SBST functional tests. For this purpose, we propose the introduction of a hardware module external to the CPU that can be programmed by an SBST program and can properly stimulate the external signals, thus suitably triggering the internal logic they drive. Additionally, there are some HFT faults which are only triggered when the CPU's output pin is asserted by its internal logic, either by the immediate reaction to the stimuli injected by an external source, or by forcing the CPU itself to assert the external output signal and consequently the logic attached to it. The latter is easily achievable by programming the CPU such that the contents of a general purpose register are copied or transferred directly to the I/O, for example.

This section is subdivided into several subsections explaining the proposed methodology, the software and hardware integration flow, the architecture of the proposed module, and the final working methodology to tackle the difficulty of detecting HFTs during the operational phase.

6.3.3.1 Proposed Framework

The framework we propose is based on adding an ad-hoc module able to overcome the above limitations. The proposed scenario and role in which the generic hardware module is used is shown in figure 6.14. The logic blocks we are targeting are those marked as "Logic i" and "Logic o".

The module is placed outside the microcontroller and is connected to the system bus, so that it can receive commands and send data from/to the SBST program running in the

CPU core and often stored in a Flash memory. Since it is attached to the bus (typically acting as a memory mapped peripheral), it is easy for the SBST software to program the module via the communication logic embedded in the module (e.g., through LOAD/STORE instructions inside the SBST program).

In addition, the microcontroller external input pins that have to be activated are driven by the output pins of the proposed module, thus being able to properly stimulate the processor inputs. Some of the module inputs may be connected to some of the processor outputs, so that the module can easily observe their value (tracing it for future usage) and exploit it for properly stimulating the processor itself.

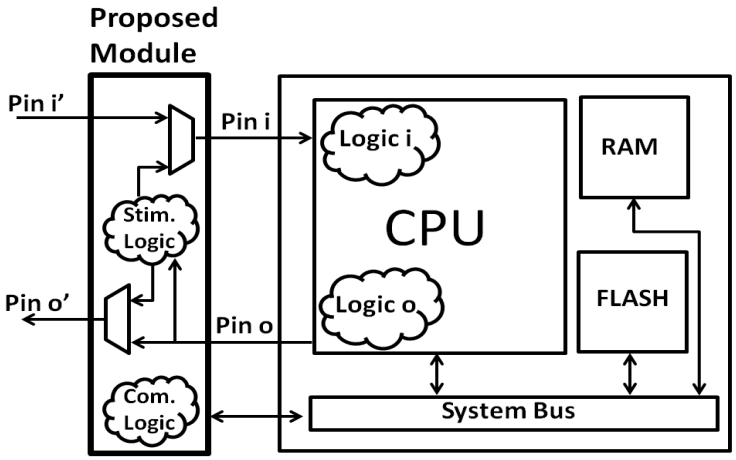

Figure 6.14: Proposed framework.

The required test stimuli depend on the function performed by the pin driving the target logic block.

In general, the situations supported by the proposed module are the following:

- The pin stimulating the microcontroller must be activated (driving it from 0 to 1, or vice versa) at a given specific time
- The pin stimulating the microcontroller must hold the same value of a given output pin driven by the processor (denoted as pin i in figure 6.14) apart from a given clock period, in which its value is complemented with respect to the value of pin i
- The reactive stimuli of a pin asserted by the microcontroller (pin o in figure 6.14), must be saved in order to be written back to the CPU memory for covering faults related to pin o
- The module may reset the microcontroller but must hold the CPU output values. In these cases, the logic in the CPU controlling these pins will be activated, allowing the test of the target logic block (either logic i or logic o, or both).

6.3.3.2 Hardware and Software Integration flow

Together with the hardware module, we proposed a hardware-software integration flow, as shown in figure 6.15. The instruction blocks of the SBST are on the left and the whole flow is composed of 3 phases, shown on the right: programming phase, stimulation phase and check phase. The flow is a combined integration work between the hardware and the software. In the programming phase, the SBST is setting up the behavior of the hardware (1). In the stimulation phase, the module is active (2), it then waits for a desired number of clock cycles to stimulate the CPU target pins for a defined number of clock cycles (3) and finally it saves the CPU output pins (4) if they have been asserted by the CPU. The last phase of the SBST is the check phase, when the CPU restores from the module the saved output pin values (5) and the test program compares the expected values with the observed ones. When the module has finished its job, it resets all the registers except the one saving the CPU's output stimuli (RESPONSE register).

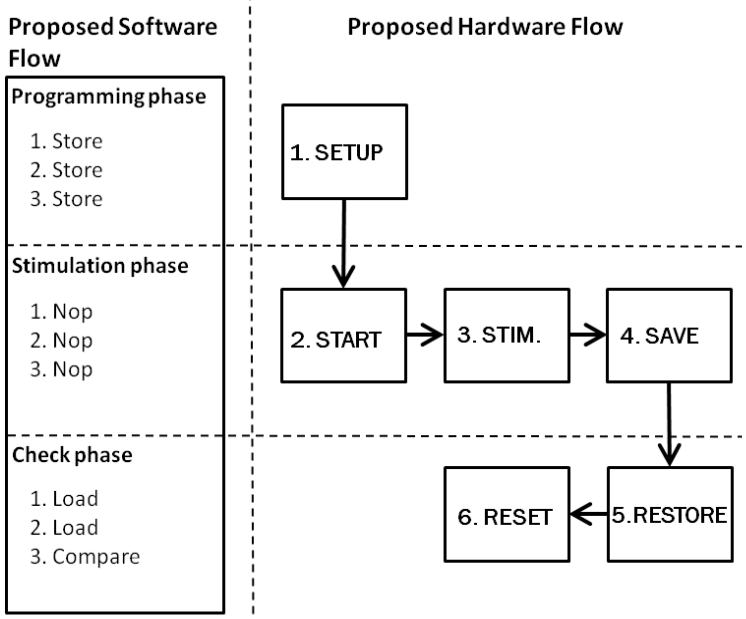

Figure 6.15: Proposed hardware-software integration flow.

In order to provide the proposed module with a common timing with respect to the CPU, we assume that the two share a clock signal.

6.3.3.3 Detailed hardware architecture

The proposed hardware can be used for any type of pins, and its basic architecture is shown in figure 6.16. During the programming phase, the SBST communicates with the module by writing commands in 3 registers that will drive the module:

- START
- ACTIVE
- MASK.

The START register tells the module to only activate a given pin when a number of clock cycles have passed. This number of clock cycles corresponds to the value stored in the START register. In figure 6.16, the counter 1 (Cnt1) decrements this value until cleared and then activates the stimulation logic.

The ACTIVE register configures the module to keep the given pin active for a number of clock cycles. Then, the pin is disabled after passing this quantity of clock cycles. In figure 6.16, the counter 2 (Cnt2) decrements the ACTIVE register value at every clock cycle until cleared, thus deactivating the pin.

The MASK register tells the module what are the CPU pins that are going to be stimulated. In some cases, the module may be configured to stimulate several different pins but at different time frames and controlled by different SBST programs. This register configures the module to mask pins that are not going to be stimulated when a specific SBST is testing a desired pin. In figure 6.16, a logical '1' at bit 0 of this register, for example, masks pin 1, and the stimulating logic for this pin is not activated.

Moreover, the MASK register may allow masking output pins (for example pin 3 in figure 6.16). This may be necessary when the sequence of stimuli applied to the processor produces an undesired event on some output pin. In this case if the output pin is not masked, it could send for example a signal to the application asking for a reset, and reset is not actually needed (nor tolerated) in the real application. Finally, the fourth register is called RESPONSE. The purpose of this register is to save the values of CPU output pins, like pin 3 in figure 6.16. Sometimes, the only way to observe and test the logic attached to pin 3 (logic o in figure 6.14) is to save this value and send it to the CPU through the bus.

The RESPONSE register has to be reset by the SBST pattern, because otherwise, there could be the risk of missing the observation of the CPU output stimuli for comparison. Some case study tests, that will be explained later on, rely on a global reset because the CPU crashes. However, before crashing, the CPU asserts some output signals that need to be observed and cannot because it enters the crashing state. Thus, the observation of these asserted CPU output stimuli occurs immediately after restarting the CPU at power-on by the SBST pattern.

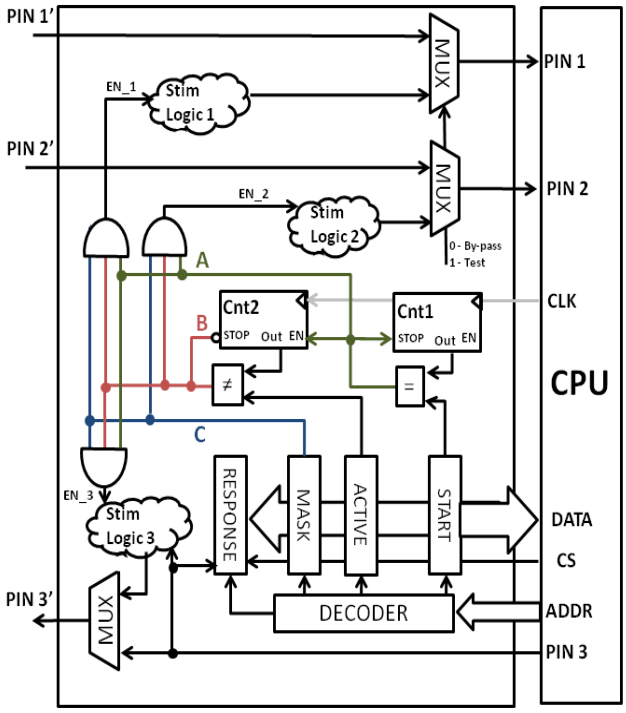

Figure 6.16: Proposed architecture.

The proposed module has two operational modes:

- By-pass mode
- Test mode.

In by-pass mode, the module is completely transparent, and does not interfere with the system operational mode. In Test mode the module works as described below.

6.3.3.4 Detailed Methodology

Description of the proposed In Test mode the module controls the microcontroller target input pins and applies to them suitable test patterns. In order to operate in test mode, the module receives commands from the SBST code (programming phase) through suitable STORE instructions. After all the necessary data have been written into all registers the test starts immediately, and the target input pins are suitably stimulated (stimulating phase). During this phase the SBST executes NOP instructions waiting for the module to finish its assigned job. Then, to verify that the logic has been properly tested, several status registers which have been modified during stimulation allow determining whether the logic passed the test or not. This part included in the SBST is called check phase and

was first proposed in [87]. When this test finishes, all the module registers (except the RESPONSE one) are reset and the module is switched back to by-pass mode.

It is important to highlight that the proposed module has two operational modes (test and by-pass), but the rest of the system always works in the normal operational mode, as mandated by the functional test paradigm.

In order to test logic i, it is necessary to correctly write the SBST program in such a way that the programming and stimulating phases are organized and synchronized in the proper manner. Assuming that the functional test is performed by exploiting some idle slots left by the system application, or by temporarily suspending it, the steps to be performed by the system are:

1. The system executes the system application
2. The system switches to functional test
3. The SBST program writes values to the START, ACTIVE and MASK registers of the proposed module (programming phase)
4. The module switches to TEST mode
5. The module applies stimuli to the target pin(s) specified by the MASK register, respecting the time to start and time to remain active (stimulating phase)
6. The target logic block(s) is (are) properly stimulated
7. The module captures any stimuli coming from the CPU and saves it into the RESPONSE register
8. The SBST then compares the obtained results with the expected ones (check phase)
9. The module returns to by-pass mode and the SBST code finishes, returning to the system application.

Clearly, the proposed approach has some drawbacks in terms of additional circuitry to be inserted in the system, which may also increase the logic on the critical path, thus potentially impacting on the performance under the normal operational mode. The practical evaluation reported in the next subsection shows the reduced impact of these limitations.

6.3.4 Case Studies and Results

In order to verify the feasibility of the proposed approach we considered four cases of study targeting different sets of HFT faults. These special faults belong to several logic blocks of an industrial Power Architecture-based microcontroller manufactured by STMicroelectronics and targeting automotive applications, whose architecture is shown in figure 6.17. The target HFT faults are located in the platform CPU core (logic i and logic o

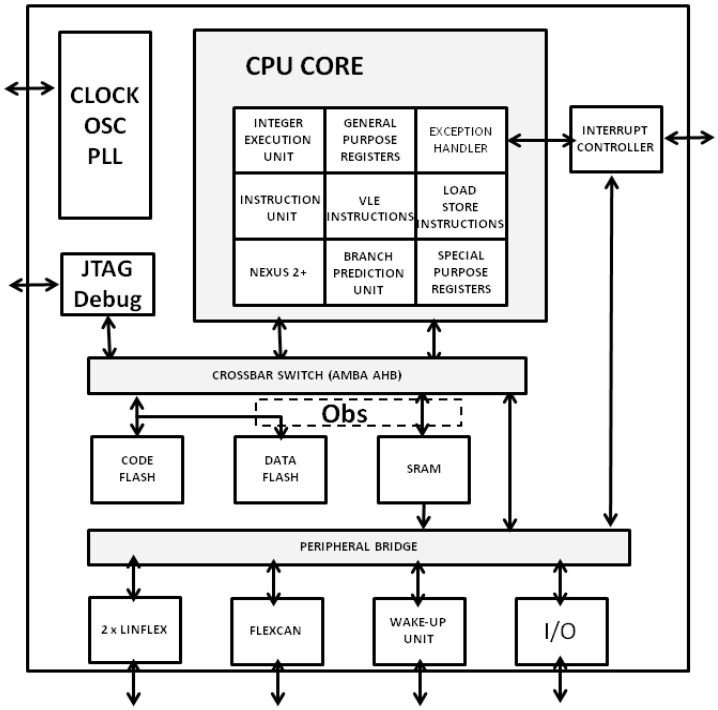

Figure 6.17: Target platform.

in figure 6.14). As explained before, these HFT faults are only observable when an input signal is properly stimulated by the proposed module or an output signal is asserted by the CPU itself and then saved in the module's RESPONSE register. During on-line functional testing the observability point is normally the data memory (either Flash or RAM), where results are written. This point is indicated by Obs in figure 6.17. It is important to highlight that the selected memory space for dumping results shall not interfere with the application memory. The ability of covering faults stems from the comparison between the expected results and the computed ones obtained by the execution of SBST test patterns.

The target external signals are shown in figure 6.18 and indicated by the letters from A to D. There are altogether 11 groups of I/O signals. However, only the logic controlling the HFT faults related to the following groups are used as case of study:

A. Interrupt Signals

B. Machine Check

C. Power Management

D. Processor Status and Exception Enables.

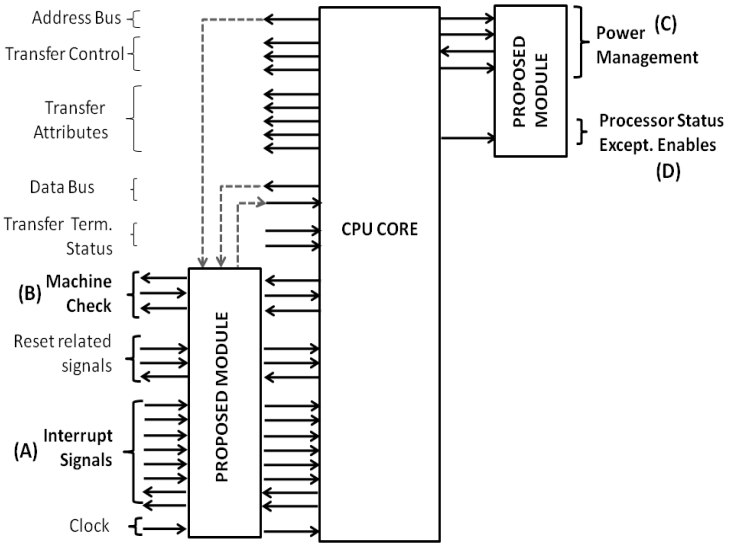

Figure 6.18: Detailed interconnection scheme between the target device and the proposed module.

Interrupt Signals - NMI case of study

In this case of study, we focus on the logic block managing the NMI pin. In order to test such a block, the NMI signal must undergo a 1 to 0 transition which generates an exception inside the CPU core and stimulates the NMI interrupt logic. Clearly, there is no way for activating such a signal in a purely functional manner. For the purpose of this case of study, the proposed module is simply in charge of activating the NMI signal based on the commands it receives during the programming phase.

The SBST program starts by configuring the proposed module, which means defining when to activate the NMI pin, how long the pin must be active and which pin to stimulate (in this case only NMI). Some padding instructions (NOP) are necessary to make the CPU waiting for the correct time to start computing the signature, and thus making the logic controlling the NMI pin observable. Hence, resorting to the proposed module an SBST program can stimulate the NMI-related logic, which represents a small but highly critical amount of logic. This small quantity of faults which can be covered by the SBST program in this way is very important to keep product robustness and provide a safe mission.

The SBST program is small, containing 3 load/store instructions, and a few padding instructions necessary to force the CPU to wait for the correct time to start observing the correct functioning of the interrupt logic triggered by the NMI pin. The SBST program lasts for around 35 clock cycles: 10 clock cycles for module configuration, 5 padding instructions (which can be variable), and around 20 clock cycles to check whether the obtained values for the special purpose registers (sprs) correspond to the expected ones, thus verifying the correctness of the logic. When a NMI occurs, the NMI bit of the sprs

machine check syndrome register (MCSR0) is asserted. Saving this register into a suitable test memory space allows checking whether the HFT faults related to the NMI were tested or not.

The proposed external module used in this study case has 8 ports, 210 nets and 265 cells. Its adoption allows detecting 291 further stuck-at faults in the NMI logic.

The SBST program using the proposed module includes 3 instructions to program the module, plus 3 other instructions to load the expected and computed values and compare them. This way it is possible to verify whether the corresponding exception procedure has been correctly activated or not.

Machine Check - Checkstop state

The second case study is to target the HFT faults related to a specific CPU state called checkstop state. When the processor is in the checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. To indicate that a checkstop state has been entered, the p_chkstop output pin is asserted whenever the CPU is in the checkstop state and cannot retrieve to its normal execution.

We consider two particular situations of this case of study; in the first CPU enters the checkstop state and it can then return to the normal operation mode through a correct programming of the interrupt handler. In the second situation the mcp_b pin is asserted, requesting a machine check, and then the CPU core enters the checkstop state, asserts the mcp_out and chkstop output pins requesting an asynchronous reset. This special case happens because the CPU has not been correctly configured during initialization. This has been done on purpose to activate the checkstop state. This state forces the microprocessor to crash and requires that an external controller resets the CPU core in order to return to normal operational mode. In both situations, the functional test must be executed during power-on or power-off only, since it causes the system to crash. Thus, the first task required by the SBST after the asynchronous reset is to verify whether the checkstop condition is working or not.

In the first machine check case, the SBST pattern includes a misaligned instruction of multiple words. The CPU program flow jumps to the interrupt handler and at the same time asserts the p_mcp_out pin that acknowledges that a machine check has occurred. In this first experiment, the interrupt handler is able to correct the misbehavior dodging the crashing state because it provides correct return address to continue the SBST execution and consequently to the normal application mode. The SBST pattern is composed of 3 STORE instructions for the programming phase, 5 NOP instructions for the stimulating phase, and 2 LOAD instructions followed by a compare instruction for the check phase. The program lasts for 20 clock cycles due to exception and its corresponding handler execution. This module consists of 265 cells and allowed detecting 749 stuck-at faults related to the asserted output pin.

The second machine check case requires that the machine check pin mcp_b is asserted externally to the chip. This exception puts the CPU into the checkstop state condition and at the same time it asserts p_mcp_out and p_chkstop. This situation is forcedly implemented and makes the CPU crashing after a couple of hundred clock cycles. Then, the module saves the stimulated pins values and resets asynchronously the CPU.

The SBST pattern is composed of an initial check by reading the external module's RESPONSE register and verifying if the checkstop state condition has been tested. If this register contains only zeroes, it means the condition has not been tested and should be tested at power-off. Therefore, the SBST pattern has an initial check phase of 3 instructions checking whether the test worked correctly by verifying that the saved values of p_mcp_out and p_chkstop pins were asserted. Then, if the respective pins were not tested, the SBST continues with the programming phase which includes 3 STORE instructions configuring the module, followed by the stimulating phase which leads to a CPU crash after a couple of hundred clock cycles.

The SBST pattern lasts for 230 clock cycles (excluding the CPU power-on after reset) and has 8 instructions. The proposed module contains 8 ports, 230 nets, 290 cells and allowed testing 1,034 stuck-at faults belonging to the HFT faults category. The superset of both cases mentioned in this case study was able to cover 1,352 stuck-at-faults.

Power Management Case Study

The third case of study is related to the CPU low power states introduced to minimize the overall power consumption. In order to achieve good fault coverage on the related logic, the CPU must enter and properly traverse this set of states. In particular, for this case of study there are two situations that allow covering most of the low power-related HFT faults: in the first we use suitable control instructions in the SBST. The second situation corresponds to suitably stimulating some input pins using an external source to initiate power management, thus entering the low power states and back to normal operation when needed. There are four power management states which the CPU enters and allow the test to cover the target HFT faults, as depicted in figure 6.19:

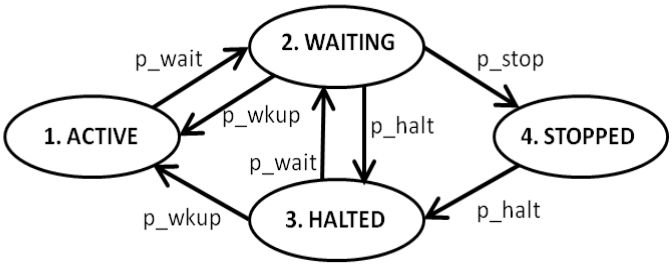

Figure 6.19: Power states

The active state is the default state for the CPU core in which all of its internal units are operating at the full processor clock speed. In this state, the CPU core still provides dynamic power management in which individual internal functional units may stop clocking automatically whenever they are idle. The waiting state is entered as a result of the execution of a wait instruction, the output p_waiting is asserted, and waits for an interrupt to continue normal execution. The halted state is entered to suspend the instruction execution and bus activities. However, the internal units remain active to quickly go back to the active state if needed. The stopped state stops all internal functional units except for the clock control state machine. This state consumes the minimum power while keeping internal values steady without having any data loss. The power in this state can be further reduced if the clock is stopped and finally reaches the power-off state. Whenever the processor changes the power state, some external pins that are asserted allows observing the corresponding logic in the CPU, and also the logic related to the CPU power state. In this case, this enables detecting not only the HFT faults related to the power management during the operational phase but also those in the logic related to the pins. In order to test correctly the CPU power states, the stimulating logic inside the proposed module includes a simple state machine asserting the CPU input pins then reading the CPU output pins. The module makes the CPU to navigate through active, waiting, halted, and stopped power states, and then back from one state to another until the active state again. At each state transition, the output pins stimuli are saved in the RESPONSE register (bit 0 for p_waiting, bit 1 for p_stopped, and bit 2 for p_halted) . The SBST pattern contains 3 STORE operations in the programming phase, 40 NOP instructions in the stimulation phase and 3 instructions in the check phase. This pattern lasts for 300 clock cycles and it is able to cover 1,780 stuck-at-faults.

5.4. Processor Status and Exception Enables The last case study is related to the processor status bits. There are 6 pins (pstat[0:5]) which indicate the processor status. The main difficulty in this case lies in the development of an SBST program able to navigate through the various processor statuses. To tackle this problem, the SBST includes randomly generated instructions, and includes also a manual effort to detect the HFT faults related to these pins. The processor statuses indicated by pstat[0:5] are subdivided into the following categories:

- ***Execution statuses*** determine whether the processor is executing any instruction and it is interrupted either by external or stall exceptions
- ***Processor states statuses*** tell the type of state the processor is in while it is idle waiting for an external acknowledge coming from the system
- ***Instruction completion statuses*** tell whether the processor has finished certain instructions like conditional branches and if they were taken or not
- ***Exception statuses*** inform when an exception has occurred and if their correct return to the normal application was successful.

In this case study, the proposed module is used several times during the execution of the state-of-the- art SBST patterns as described in [32] devised to target different stuck-at faults observable on-line at different CPU blocks at different time frames. What we did in this case of study was to include the state- of-the-art SBST program in the stimulation phase of the proposed SBST program for each time frame targeting in this case the HFT of processor statuses.

The programming phase includes instructions cleaning up the internal registers (especially the RESPONSE register), and to put the module in capture mode (where it does not stimulate any external pins, but remains ready to capture the CPU output pins stimuli to restore back to the memory through the peripheral bus). Such strategy does not affect SBST test execution or stuck-at fault already covered, but impacts minimally the final test execution. Also, at each test frame as described in the previous subsections, there is a check_phase to tell whether the HFT faults of the different processor statuses pins were covered correctly.

The results for this case study show that the processor status pins were stimulated correctly going through all the above mentioned statuses included in more than 20 state-of-the-art SBST patterns. The proposed SBST pattern included a latency of 10 clock cycles and 10 instructions per test execution time frame, summing up an overhead of more than 200 clock cycles and 200 instructions. This strategy detected more than 1,881 HFT stuck-at faults located in the logic block driving the processor status pins.

Proposed external module characteristics

We have developed external modules separately according to the case studies mentioned in the last subsections to demonstrate the effectiveness and the portability of the proposed approach. However, we have integrated all external hardware into a single one which allowed sharing parts of circuits (i.e. registers and communication logic) to reduce the area overhead. The final circuit contained 548 cells (240 common cells and 308 cells divided for each case study) and functioned correctly to detect the 5,304 HFT faults of the four case studies.

6.3.5 Remarks on SBST for HFT coverage

In this chapter, we first introduced the concept of Hardly Testable Functional faults, i.e., faults that may impact the behavior of a system, but can hardly be tested using a purely functional approach during the operational phase. In fact, these faults require some events on a given input signal that cannot be controlled via software. In order to allow the test of these faults, we propose the insertion of an external hardware module able to apply such stimuli in a programmable manner. Together with the module we also proposed an SBST structure to control the module and detect the HFT faults for target pins and CPU states. The module is memory mapped, and is able to receive commands from a SBST

program when on-line testing is performed, such that the module is only active when required. During the mission mode it is not active and it is transparent to the system, not affecting the normal operations. The proposed module is external to the Unit Under Test, thus minimizing the invasiveness of the method. Results gathered on case studies belonging to a 32-bit variable-length encoding Power Architecture- based CPU provided by STMicroelectronics show the feasibility of the approach and demonstrate that it can cover some faults that would remain undetected otherwise. The proposed method was able to increase the stuck-at fault coverage by about 2.7%. Even though it is not a significant number, these faults are crucial to mission safety. Thus, tackling groups of faults like the HFT ones existing in small portions allows increasing the overall coverage figures required by the standard ISO26262. Moreover, the reported results show that the proposed hardware placed outside of the CPU package has a small area and consequently a reduced cost. Currently, we are evaluating the effectiveness of our method by extending it to the test of other similar situations, like communication protocols widely used in the embedded systems in the automotive field. We are also devising a new version of the proposed module which can be used in a wider range of cases such that small modifications are minimal and easy to develop.

Part IV

Fault Injection in GPGPU cores

7. Proposed GPGPU Fault Injection tool

General Purpose Graphic Processing Units (GPGPUs) have become popular in the past years to deliver high end performance in many different areas, ranging from gaming to HPC [95] [96]. Additionally, GPGPUs are increasingly employed also in some safety-critical embedded domains, such as automotive, avionics, space, and biomedical. As an example, the Advanced Driver Assistance Systems (ADASs), which are increasingly common in cars, make an extensive usage of the images (or radar signals) coming from external cameras and sensors to detect possible obstacles requiring the automatic intervention of the breaking system [97].

One of the key issues about GPGPUs is their reliability. Depending on the environment and also on the GPGPU complexity, faults can occur due, for example, to the incidence of radiation particles, leading to silent faults or functional interruption. GPGPUs reliability issue can be unacceptable when dealing with safety-critical applications (e.g., in aerospace and automotive) or when a large number of GPGPUs is used for distributed computation (e.g., in HPC centers).

To increase the reliability of their products, GPGPU producers have provided novel GPGPUs with an ECC mechanism able to correct single errors and detect double errors in the available memory components. Nevertheless, ECC is not a definite solution for GPGPUs reliability. Radiation particles can still affect the operation of a logic gate leading to wrong results. Moreover, the corruption of critical resources like the scheduler, which is not protected by the ECC, may lead to multiple output errors [51][98][99]. As demonstrated in [51], even several errors still appear at the output even when the ECC protection is turned on. Moreover, the activation of the ECC mechanisms may results in significant performance degradation, which may not be acceptable in some cases.

The novel necessity to design reliable applications for GPGPUs has raised interest in the research community that has been putting efforts to produce mitigation techniques and more robust algorithms [48][100][101][50]. Fault Injection [102] is a common solution to validate the final GPGPU application code and check its detection or correction capabilities,. Fault Injection can be performed either exposing the GPGPU to accelerated particle beams, or by resorting to software methods. In this work, we propose an efficient fault injector for GPGPUS that does not require any information about GPGPU internal model, since this information is only viable for GPGPU producers. Fault injection solutions with low cost and invasiveness to GPGPU applications are clearly difficult to devise,

even because the GPGPU companies do not make available the test access infrastructures (e.g., the IEEE 1149 one). Despite the confidentially of the GPGPU architecture, some reduced intrusiveness techniques that allow injecting faults in GPGPUs have been developed [103][104]. In [103], the authors modify the source code to inject errors; clearly, the effects due to these faults are not the same of those generated by a real fault occurred in the GPGPU modules during the normal computation. In [104] the authors propose a fault injector based on software debugger. Substantially, they inject bit flips in random variables at random instants to evaluate the impact on non-fault tolerant algorithms.

In this chapter, we propose a novel fault injection platform based on the NVIDIA CUDA-gdb, which is able to automatically generate a fault list from a GPGPU application, to inject transient faults in several accessible memory components, to mimic faults affecting ALUs, FPUs, and L1 cache errors, and to classify their effects. The advantage of the proposed solution is the ability to inject faults in a real GPGPU hardware without being invasive to the application (i.e., without modifying the source code). With respect to [104], the tool is able to automatically generate the fault list, inject faults, and classify their effects.

To demonstrate the effectiveness and discuss the limitations of our approach we report some experimental results obtained using the proposed fault injection method on a commercial GPGPU device.

The rest of the chapter is organized as follows: first the chapter describes some background concepts about the GPGPU architecture, the effects of injected faults as observed through radiation tests, and the fault injection techniques for GPGPUs which have been previously proposed in the literature. Secondly, the proposed approach is decribed in details. Then, the proposed fault injection method is evaluated by reporting experimental results gathered on some benchmark applications. Finally, the last section draws some conclusions.

7.1 Background

GPGPU architectures raised a lot of interest in the recent years due to their high performance in parallel computation for delivering fast results in many different areas ranging from gaming to biomedical. Some of the GPGPU application areas (e.g., the automotive and aerospace ones) have safety constraints; moreover, the usage of GPGPUs in HPC also raises concerns about their reliability [105]. In order to assess the reliability of resilient applications and to validate the resiliency mechanisms they embed it is necessary first to analyze the GPGPU architecture.

In this section, we will introduce first the typical NVIDIA GPGPU architecture, and then we will outline some of the available solutions to increase the resiliency of GPGPU applications; moreover, we will overview the types of errors that may or may not produce misbehaviors, and the main fault injection methods proposed so far.

A. NVIDIA GPGPU architecture

The typical GPGPU architecture differs from the CPU one because it includes many serial multiprocessors (SMXs) to increase parallelism. In figure 7.1, a simplified GPGPU architecture, based on NVIDIA Fermi architecture, is presented. The Fermi GPGPU has an interface for communicating with the host processor (i.e., the CPU), global memory that store the data to be elaborated, a gigathread module for assigning instructions to the SMXs, several SMXs where the computation takes place, and an L2 cache.

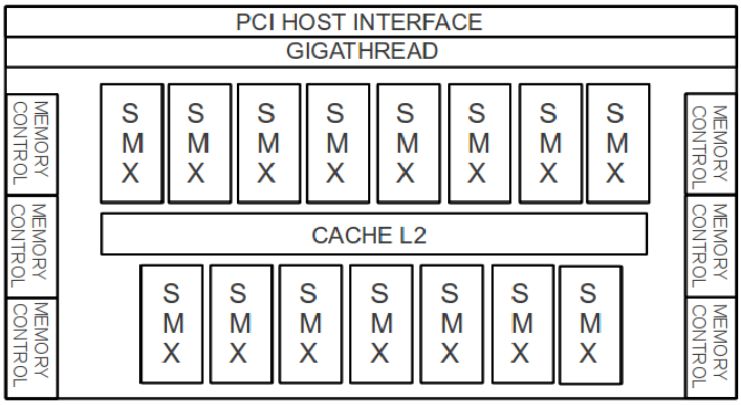

Figure 7.1: A Simplified GPGPU Architecture.

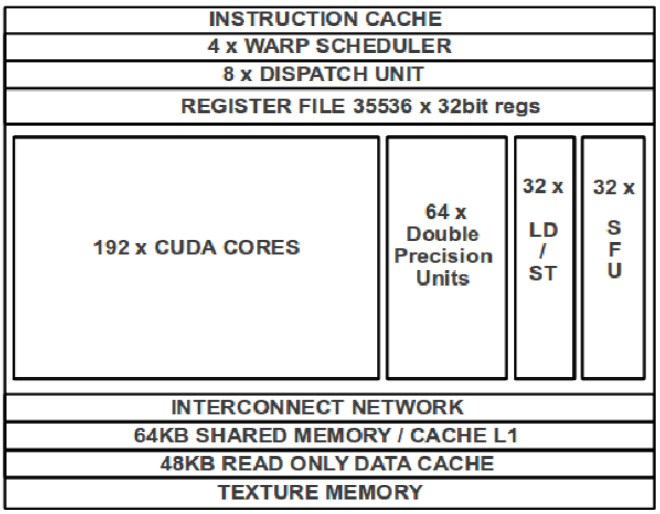

Figure 7.2: A Simplified SMX Architecture.

In figure 7.2, a simplified SMX architecture is presented. A SMX is typically able to execute several threads in parallel. The number of parallel processes executable in parallel

Error Classification	Explanation
No effect	The injected fault is masked, for example because the target variable is overwritten by another value before being read
Data Error	The injected fault affects some logic or memory elements and produce a wrong data in the output result
Timeout Error	The injected fault effects affect a control flow variable, thus forcing the program to enter an infinite loop; The fault can also cause the executing thread to finish earlier
Exception Error	The injected fault triggers an exception; the program returns without providing any results
Fault Tolerance Error (Error detection)	The injected fault is detected by some fault tolerance mechanism

Table 7.1: Classification of faults affecting GPGPUs

depends on the SMX capabilities and ranges from 32 to 192. In the specific sample case considered in this figure, it contains an instruction cache, a warp scheduler, a dispatch unit, a register file, 192 cores, 64 double precision units, 32 load store units, 32 special function units, a L1 cache, an interconnect network, and a texture memory.

B. Fault effects in GPGPU applications

There are several works (e.g., [51] [98] [99]) studying the effect of faults in GPGPUs caused by radiation. Also, process variations, aging, high temperatures, high current peaks, and low voltages may help jeopardize electronic devices, like GPGPUs, and may produce critical misbehaviors depending on the application.

In table 7.1, we listed the possible effects of errors provoked by injecting a fault in the GPGPU circuit and some explanation. Validating GPGPU applications reliability can be performed exposing the GPGPU to a controlled radiation beam. However, radiation tests, even is fast and effective, are is very expensive. Moreover, radiation test does not allow to fully understand where faults arise and which are the propagation mechanisms they activate. Therefore, alternative solutions based on fault injection are required.

C. Fault Injection in GPGPUs

In the hardware design industry there are mature methodologies and tools to test the architecture behavior by simulating internal components. However, this is only possible when a detailed model of the addressed architecture is available. In this case, faults can be in-

jected at any given time in any precise location of the architecture, giving the possibility to analyze their effects and validate the application.

Techniques for fault injection in GPGPUs have been proposed in several works in different ways. In [103] the authors propose a method for injecting faults by modifying the source code of the application. In [106] the authors propose the usage of a statistical fault injection tool based on a GPGPU simulator to analyze the architectural vulnerability when executing several parallel algorithms. However, in this case the test is not performed on the real hardware and the accuracy of the results depends on how strong the correlation is between the simulator and the real architecture. Lastly, in [104] a fault injection mechanism based on the software debugger (CUDA-gdb) [107] was proposed. This mechanism is able to modify variables of the executable code at runtime. With this tool the authors evaluated the resiliency of some parallel applications by applying random transient errors in random variables.

The works discussed above show that the fault injection tools for GPGPUs have the following advantages with respect to radiation test:

- More accurate and precise results
- Faster fault injection setup
- No risk of damaging the GPGPU
- Reduced cost.

On the other side, fault injection techniques have some drawbacks:

- They may be too slow to gather enough results to statistically assess the final reliability
- They may have difficulties in accessing some elements within the GPGPU
- They may require some modifications in the application code, thus providing results on a system different than the target one.

7.2 Proposed Approach

In this work we propose a GPGPU fault injector tool that forces bit flips in variables of GPGPU applications and observes their effects. The method exploits the CUDA-gdb software debugger, which provides the visibility of all runtime GPGPU variables, giving the possibility to inject faults at specific memory elements of the architecture.

In order to fully automate the fault injection campaign, the different phases typically implemented in a traditional fault injection experiments (fault list generation, fault injection, and result analysis) are all demanded to an ad-hoc tool which interacts when

necessary with the debugger via a proper command file. This guarantees a minimal effort to perform a GPU fault injection campaign and ensures that the developed tool is code independent. More in particular, the steps performed by the proposed fault injection method are the following:

- a CUDA code parser elaborates profiling information about the GPGPU variables and, based on the user inputs, generates a list of faults to be injected.
- The generated list and the profile information are used to create CUDA-gdb commands that will be executed while the program is running.
- The application is executed without any fault injection, thus recording its behavior (fault-free or gold execution).
- The application is executed with one single fault injected at a time.
- The values of some observable variables (selected by the user) are analyzed to check whether the fault-free and the faulty results match.

The architecture of the platform implementing the proposed approach is shown in 7.3.

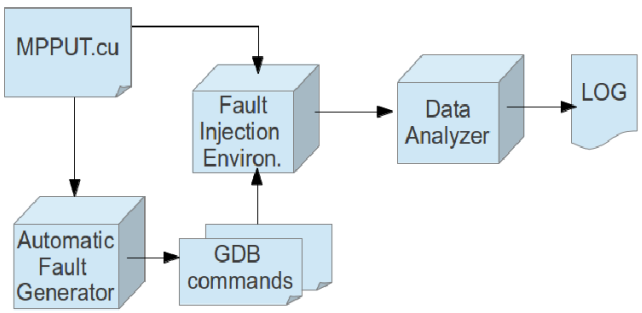

Figure 7.3: Architecture of the proposed fault injector

The proposed fault injector is composed of three blocks; the automatic fault generator, the fault injection environment, and the data analyzer.

The first module reads the program code and elaborates profiling information about the variables in which faults have to be injected. The automatic fault generator then translates each fault event (i.e., a bit flip in a selected variable) into CUDA-gdb commands. The fault injection environment manages the execution of the executable code and performs the fault injection experiment. The data analyzer compares bitwise the results acquired during the previous execution in terms of values of the output observable variables in the faulty-free and faulty executions.

The proposed methodology may increase slightly the simulation time, because the application needs to be compiled with debugging symbols needed for debugging. Also, execution time increases because the program needs to be halted at a specific point (e.g., when the program counter reaches a specific value) to force a bit flip in the target variable. Finally, the execution time is increased due to file access operations, because CUDA-gdb dumps the output values to a file to allow the comparison of faulty and fault-free results.

In some GPGPU architectures there are ECC mechanisms protecting memory components from transient and permanent faults originated by radiation, high temperature, voltage drops, and different kinds of stresses. On the other hand, other GPGPU components (e.g., FPUs and ALUs) may also be affected by faults, whose result is to have their operations corrupted, thus producing erroneous values that may sneak into the memory component. In this case, the ECC mechanisms will not provide any error correction.

The proposed fault injector has the ability to produce bit flips into runtime memory elements (global, local and shared memories) visible by the programmer to mimic transient and permanent faults in logic elements (ALUs and FPUs), memory elements (L1 caches and shared memory), scheduler (execution flow and thread ID errors), and kernel launch (exception generation). The bit flip operation differs for each data type (e.g., Integer and Floating point representations) as described in the following subsections.

A bit flip, resulting from the corruption of memory elements or reflecting a wrong computation, can be introduced in an integer number by simply using a single XOR operation between the variable and the user-defined bitmask, thus implementing the desired bit flip, as shown in 7.1.

$$op = op \oplus bitmask \tag{7.1}$$

where op is the variable affected by the bit flip.

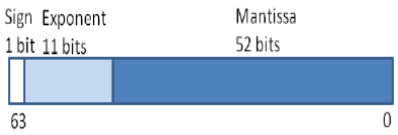

Figure 7.4: Representation of the 64-bit double floating point number

In FPUs the floating point variables comply with the IEEE754 standard, and contain three parts: sign, exponent and mantissa. Figure 7.4 shows the representation of a 64-bit floating point format. A 64-bit floating point number a is considered in scientific notation as:

$$a = m \times d^b \tag{7.2}$$

where *m* is the mantissa, is the magnitude and *b* is the exponent.

Considering the 64-bit representation, for each of the three parts of the floating point format there is a mathematical model to implement the bit flip operation, as described in [108]. Considering a in equation 7.2, the perturbed floating point number with a bit flip is expressed as:

$$\hat{a} = \begin{cases} a \pm 2^{j-52} d_a .....(a) \\ a 2^{\pm 2^j} .....(b) \\ -a .....(c) \end{cases} \tag{7.3}$$

a) Flip in the mantissa bits, b)Flip in the exponent bits, and c)Flip in the sign bit

a Fault Injection Environment

In figure 7.5, the proposed fault injection environment is presented; it is composed of 4 main steps. Initially (step 1) the Parallel Program Under Test (PPUT) is compiled with debugging capabilities for the target GPGPU: the obtained executable file contains the executable code plus the symbol table useful to debug the code itself. This GPGPU executable code is then manipulated by the cuda-gdb (step 2) according to a set of gdb commands previously defined, which in turn allows visualizing and modifying GPGPU runtime variables (step 3). Finally, in step 4, the produced results are analyzed.

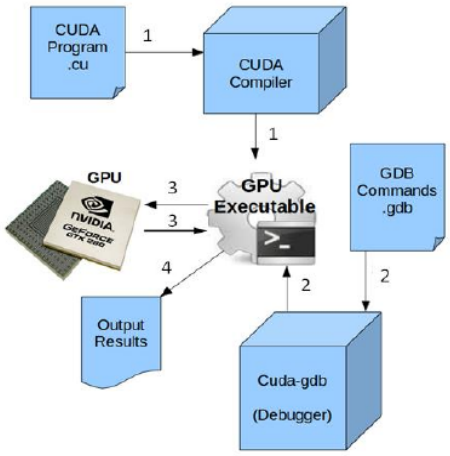

Figure 7.5: Fault Injection Environment.

The automatic fault list generator is a module that parses the GPGPU code and generates a look-up table to generate suitable CUDA-gdb commands triggering the fault simulation.

A simple CUDA code (GPGPU in red) embedded in a C code scheme (CPU in blue) is presented in figure 7.6 to demonstrate how the parser selects the variables where to inject faults. The parser is able to identify the fault injection points (numbers 2 to 8) which are the only the GPGPU variables accessible at runtime by CUDA-gdb as well as the number of blocks and threads involved in the GPGPU computation. By identifying these variables, the user is able to inject a single bit flip in one variable at a time per thread and study their effects on the final result.

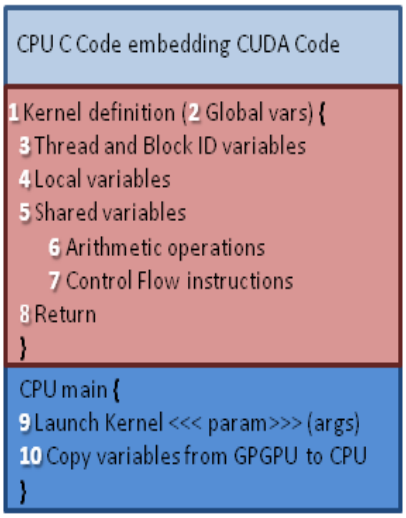

Figure 7.6: C code embedding a CUDA code and the possible fault injection points in the GPGPU application.

These variables are inserted in the fault list table which is organized to mimic a fault injection in the application. In this table there are some useful information needed to create the gdb commands to implement fault injection.

The table elaborated by the tool contains the data shown in table 7.2, their corresponding numbers in the code depicted in figure 7.6, and their given name abbreviation that will be later used to show how they are translated in cuda-gdb code.

The fault injection technique is only possible due to the CUDA-gdb debugger provided by NVIDIA. Although it is very powerful for debugging parallel software it is somewhat limited for our purposes. One of the limitations found is the inability to modify a variable at a random time even if located in the current context. In some cases, the bit flip operation cannot be performed because the thread that is being executed has already finished. Although the variable still exists in the GPUGPU it is not accessible by CUDA-gdb which is not allowed to modify it. . The second major problem is that watch points are not supported. However, by inserting strategic breakpoints for modifying the target variable and observing output variables, we were able to obtain the same effect.

In order to inject faults using the debugger, the fault injector translates the information parsed shown in table 7.2 into the CUDA-gdb code shown in code 1. Note that in line 11,

.	Corresponding no. in figure 7.6	Abbreviation
Kernel definition	1	KD
Target Variables where to inject a fault	2-5,8	VF
Where the target variable can be modified	Line number of 6-8	LTF
Bitmask used for the bit flip operation (defined by the user)	6-8	BM
How many executed lines containing the target variable before a fault is injected (user defined)	# of executed lines containing 6-8	#LH
Kernel Launch line number in the CPU code	9	KL
Output variables to detect errors	10	OBSV
Instant to check the output variable	Line number + 1 of 10	OBSL

Table 7.2: List of the parsed information needed to inject a fault

where the bit flip operation takes place, the CUDA-gdb commands depend on the variable type. Also, line 12 is optional and allows mimicking the effect of a transient fault. The corrupted variable (VF) is used in a CUDA instruction in the GPGPU application code and the contents with the fault is propagated to another destination variable. Then, at the next instruction of the application code, the fault is removed from the target variable which originated the fault.

The data analyzer is a module that compares the results obtained by the fault-free and the faulty executions. It makes a bitwise comparison showing the disparities observed in the output variables where we observe the error effects.

```
1. start
2. break KL
3. command 2
4. break LF
5. break OBSL
6. ignore 3 \# LH
7. delete 2
8. continue
9. end
10. command 3
11. Bitflip operation -- VF=VF\^BM
(12). Step; Reverse bitflip VF=VF\^BM (Optional)
13. delete 3
14. continue
15. end
16. command 4
17. set logging on KD.txt
18. print/t OBSV
```

```
19. set logging off
20. delete 4
21. continue
22. end
23. continue
24. quit
```

Code 1: CUDA-gdb command implementing the fault injection

7.3 Case Studies and Results

In order to evaluate the effectiveness of the approach we use 2 case studies with (robust) and without (plain) software hardening:

a. Parallel Matrix Multiplication

b. Parallel Fast Fourier Transform.

To demonstrate the features of the proposed fault injector, we inject a single bit flip in each kernel variable in each accessible code line for only one thread. Although the tool is able to consider all threads, we choose to corrupt only one thread to make the experiments simple. Then, we analyze the impact of each injected fault on the final result by evaluating the disparity between the faulty and the fault-free executions in terms of CUDA-gdb execution time and number of errors classified according to Table 7.1. We carried out the experiments in an NVIDIA GPU GeForce 9600M GS containing one serial multiprocessor including 32 cuda cores, 512MB of memory, and 100MHz clock frequency.

A. Parallel Matrix Multiplication

The first case study is a set of parallel matrix multiplication algorithms that use the duplication with comparison technique. The algorithms, proposed in [50], use thread and time redundancy. Moreover, each of these algorithms is implemented with data input duplication (full) or without data input duplication (partial). Table7.3 shows the results of this case study in its different versions. It presents the number of errors classified according to Table 7.1 of the fault-free and faulty executions. The fault-free version (plain) requires 12 ms to complete its computations, whereas the average faulty execution lasts for about 2 seconds. In the plain version, the most "dynamic" injected fault produced 140 data errors in the output variables. The robust versions were able to detect the errors and reproduce the execution, thus providing the correct results.

.	Injected Faults	No Effect	Data Error	Exception	Timeout
Plain	16	4	7	3	2
Full Thread	26	16	0	6	4
Partial Thread	21	15	0	4	2
Full Time	16	11	0	3	2
Partial Time	16	11	0	3	2

Table 7.3: Matrix Multiplication results

.	Injected Faults	No Effect	Data Error	Exception	Timeout	Fault Tolerance
Plain	74	16	49	6	3	-
Robust	137	82	0	24	3	28

Table 7.4: FFT results

B. Redundant Parallel FFT algorithm

The second case study is an FFT algorithm that uses the GPGPU. It includes a robust control flow technique that allows identifying if the program execution flow is correct or not [51]. If the technique detects the error in the execution flow, it runs the FFT algorithm again until there are no control flow errors. Moreover, the technique embeds error masking by executing several times the same kernel code in each thread. In table 7.4 the results of both the plain and robust implementation of the FFT algorithm are presented. The table shows the total number of injected faults and the number of injected faults producing errors (according to Table 7.1). In the plain version, the most "dynamic" fault propagated the error to 16,383 variables. The number of injected faults producing data errors is 49 for the plain version and 0 for the robust version, because they were masked or corrected by the fault tolerant techniques employed in the algorithm. The robust version, however, triggered more exception errors, because the control flow variables are vectors, whose indexes have been corrupted by the fault injection, generating exceptions. The average execution time increased from 45 ms (fault-free) to approximately 4s (faulty), due to the large amount of data to analyze.

7.4 Conclusions on GPGPU fault injection

In this work we presented a fault injection tool based on CUDA-gdb which is able to inject transient faults in GPGPU devices and can be used to validate robust parallel applications. Moreover, the proposed tool was able to mimic transient faults by applying

bit flip operations to corrupt the operation of ALUs, FPUs, and memory elements, thus allowing the evaluation of performance and robustness of parallel applications. The tool is non-invasive to the application and is fully automated to find variables and instructions in the parallel code where a fault can be injected. Faults were injected in two case studies resorting to plain and hardened software. The tool is being improved to mimic L1 cache and scheduling errors, and to support the possibility to inject faults on a probabilistic basis.

8. Conclusions

Everyday we rely on many different technological machinery and tools for our daily safety, security, economic welfare, and entertainment. In the following years, we expect to buy and use even smaller and cheaper semiconductor devices to improve our well-being, trusting that they will function as designed whenever used correctly under specified conditions. This property is called reliability. Usually, we trust our devices to work as expected, because they were subjected to a series of tests throughout its life, from manufacturing to end-user application, namely *Reliability Characterization*. In automotive applications, this procedure is thoroughly studied and applied to comply with insurgent demanded standards, in particular the ISO26262, as discussed in chapter 2.

This PhD thesis firstly described reliability concepts in details, narrowing down its usages specifically to semiconductor devices. Secondly, it detailed the state-of-the-art methodologies found in the literature for testing and diagnosing microelectronic modules; CPUs, memories, peripherals, and image processing units (GPUs), which compose top technology edge *System-On-Chips*, mainly used in automotive applications. Then, this thesis proposed three solutions contributing to the state-of-the-art testing and diagnosis of the above-mentioned embedded modules.

The first contribution to the state-of-the-art was a java tool exploiting an algorithm taking as input the memory design and test parameters, and test results making an accurate diagnosis. The tool was able to unscramble the memory array from a logical representation to the physical one and attribute test errors to specific cells according to test result errors. It also correctly identified failing shapes in a single memory analysis or even in cumulative analysis, that is, gathering all failing memories from specific wafer coordinates from different wafers. Then, the tool applied a fault model identification algorithm by comparing results with a fault dictionary. It accurately determined the fault location, shape, and provided a fault model hypothesis. The effectiveness of the tool was evaluated on five study cases defects found on a real industrial 32-bit Automotive Power Architecture SoC developed by STMicroelectronics.

The second contribution was the development of functional test patterns targeting three different test goals for embedded CPU cores: functional stress, functional peak power, and fault coverage. The main methodology used for automatically generating these patterns employs an Evolutionary Algorithm. The efforts in this part was the development of fast evaluators to quickly drive the EA to obtain a set of optimum test patterns.

1. Functional stress was quickly achieved by adopting a three-phase strategy by initially optimizing high-level metrics at RTL, then at gate-level, and finally optimization metrics order inversion. Evolution time was reduced from 850 hours to 190 hours, approximately, while achieving better transition activation coverage (about 30% higher), higher switching activity (about 15% higher), and lower variance (about 2% lower).

2. Functional peak power was achieved by refining the power evaluator employing a feed-forward neural network mimicking commercial off-the shelf power evaluators. We first studied the effects of switching activity on power consumption and derived other the new switching activity metric more effective for maximizing power. Then we proposed a methodology for increasing functional peak-power accounting on a two-phase strategy: training and evolution. During both phases the EA was employed, as mentioned earlier. In the first phase, the EA randomly generated functional patterns and evaluated them on the FFNN in simple runs of evolution. Then, the best and worst individuals were used for training the neural network, thus improving the correlation index with commercial off-the-shelf power evaluators. Training phase was stopped after achieving an average correlation index of 0.98. The second phase used the trained FFNN for the last evolution. The difference between both phases was the EA configuration; while in the first phase the EA was configured for generating few random and small programs over few evolutions, the second phases accounted on a EA generating many large programs over many generations. This optimization allowed the EA to quickly achieve functional peak power of over 400mW.

3. Functional fault coverage was achieved by generating automatically and manually Software-Based Self-Tests. The automatic generation allowed quickly increasing fault coverage on two CPU modules: adder and address calculation unit. The former was tackled by letting the EA work freely on all the instruction set achieving 91% of the module's fault coverage in over 10 days evolution time. The latter, on the other hand, had the EA configured to generate code module for the specific module. Thus achieving even better results (95% coverage) in a faster generation time. Moreover, a manual approach accounting on an SBST communicating with an external module was proposed. It allowed increasing the fault coverage during on-line testing by covering Hardly Functionally Testable Faults. Basically, the SBST routine programs an external module able to stimulate CPU logic attached to external pins (inputs and outputs) then retrieving the results for comparison. The overall CPU fault coverage increased in about 3% (out of about 290K faults) by tackling four case study pins: non-maskable interrupt, processor enables and statuses, check-stop state, and power-related pins and statuses.

   All three approaches were evaluated on an intel 8051-based SoC and on an automotive 32-bit Power Architecture TM SoC developed by STMicroelectronics and synthesized on 90nm technology.

The third contribution was the development of a fault injection mechanism for GPGPUs for validating robust parallel applications. With the insurgence of Advance Driving Assistance systems employed in the automotive field, the hardware and software safety-critical applications need to be fault free at all times. However, GPGPU manufacturers do not provide real design models for institutions in order to characterize the application's reliability under specific conditions. This is currently achieved by performing costly and time consuming radiation tests. In this part we proposed a GPGPU fault injection tool able to emulate Single-Event Transient and Single-Event Upset by injecting a bitflip in memory elements, thus allowing the validation of two robust parallel GPGPU applications: 16x16 matrix multiplication and 256 floating point FFT. The tool correctly activated the fault-tolerant embedded software mechanisms in both applications, pointing out their weaknesses and robustness. Also, the tool properly validated the robust applications showing that the absence of silent data corruption errors masked by the redundant technique employed in each algorithm.

New semiconductor technologies, architectures, tools, designs, communication protocols, and etc will always demand a testing scheme. In some cases, the new test methodologies can be slightly modified from older versions, but in others, a whole method needs to be developed from scratch. In addition, is important to underline that embedding test modules in SoCs is an essential practice not only to reduce ATE costs, but also to improve test completeness and validating new designs for any application it is employed. Thus, testing guarantees (in most cases) the reliability of our every day safety, security, mobility, economic welfare, and entertainment.

References

[1] John Kilby. *First Semiconductor Integrated Circuit (IC)*. IEEE Milestones, 1958.

[2] ITRS. *International Technology Roadmap for Semiconductors*. ITRS, 2012.

[3] 26262 Road vehicles Functional safety standard. *International Organization for Standarization Standard*. ISO, 2011.

[4] IEEE. Ieee standard computer dictionary. a compilation of ieee standard computer glossaries. *IEEE Computer Society Standard, ISBN 1-55937-079-3*, pages 610–1991, 1991.

[5] R. W Smith; L Dietrich Duane. The bathtub curve: An alternative explanation. *Reliability and Maintainability Symposium*, pages 241 – 247, 1994.

[6] A. Avizienis; J. C. Laprie; B. Randell. Fundamental concepts of dependability. *IEEE Information Survivability Workshop*, pages 7 – 12, 2000.

[7] A. Avizienis; J. C. Laprie; B. Randell; C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, pages 11 – 33, 2004.

[8] L. Ciganda. Phd thesis: New techniques for reliability characterization of electronic circuits. *Available [Online]: www.phd-dauin.polito.it/pdfs/Lyl%20CIGANDA thesis.pdf*, 2013.

[9] A. Birolini. *Reliability Engineering Theory and Practice 3rd edition*. Springer, 2003.

[10] Bushnell M.; Agrawal; Vishwani. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer, 2002.

[11] A.J. van de Goor. *Testing Semiconductor memories: Theory and Practice*. ComTex Publishing, Gouda, The Netherlands, 1998.

[12] L. Wang; C. Stroud; Nur A. Touba. *VLSI Test Principles and Architectures*. Morgan Kaufmann, Elsevier, 2006.

[13] L. Wang; C. Stroud; Nur A. Touba. *System-On-Chip Test Architectures*. Morgan Kaufmann, Elsevier, 2008.

[14] P. Girard; N. Nicolici; X. Wen. *Power-Aware Testing and Test Strategies for Low Power Devices*. Springer, 2009.

[15] M. Valka; A. Bosio; L. Dilillo; P. Girard; S. Pravossoudovitch; A. Virazel; E. Sanchez; M. De Carvalho; M. Sonza Reorda. A functional power evaluation flow for defining test power limits during at-speed delay testing. *IEEE European Test Symposium*, pages 153–158, 2011.

[16] S. Arrhenius. *Selected Readings in Chemical Kinetics*. Oxford,NY, 1967.

[17] Dependability management. *IEC 60300-1*. International Electrotechnical Commission Standard, 2003.

[18] International Electrotechnical Vocabulary. *60050 191-02-03*. International Electrotechnical Commission Standard, 1990.

[19] Functional safety of electrical/electronic/programmable electronic safety-related systems. *61508*. International Electrotechnical Commission Standard, 2010.

[20] The Specification, Maintainability Demonstration of Reliability, Availability, and Safety (RAMS). *CENELEC, prEN 50126-1:2012 - Railway applications*. European Committee for Electrotechnical Standarization Standard, 2012.

[21] Design Assurance Guidance For Airborne Electronic Hardware. *RTCA/DO-254*. Radio Technical Commission for Aeronautics, 2005.

[22] Software Considerations in Airborne Systems and Equipment Certification. *DO-178C*. Radio Technical Commission for Aeronautics, 2012.

[23] Guidelines For Development Of Civil Aircraft and Systems. *ARP4754 The Aerospace Recommended Practice*. SAE International - Society of Automotive/Aerospace Engineers, 2010.

[24] Safety instrumented systems for the Nuclear Industries. *IEC 61513 Functional safety*. International Electrotechnical Commission, 1997.

[25] Recurrent test and test after repair of medical electrical equipment. *IEC 62353 Medical electrical equipment*. International Electrotechnical Commission, 1997.

[26] Boundary Scan Architecture Standard Test Access Joint Test Access Port Group and Boundary Scan Architecture WG P1149.1. *1149.1-2013 - IEEE Standard for Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society, 2013.

[27] C/TT Test Technology. *IEEE 1500 Standard Testability Method for Embedded Core-based Integrated Circuits*. IEEE Computer Society, 205.

[28] M. Xiaoling Sun; Serra. On-line and off-line testing with shared resources: a new bist approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1045 – 1056, 1997.

[29] Steininger A.; Scherrer C. On the necessity of on-line-bist in safety-critical applications-a case-study. *International Symposium on Fault-Tolerant Computing*, pages 208 – 215, 1999.

[30] A. Manzone; P. Bernardi; M. Grosso; M. Rebaudengo; E. Sanchez; M. Reorda. Integrating bist techniques for on-line soc testing. *IEEE International On-Line Testing Symposium*, pages 235 – 240, 2005.

[31] M. Psarakis; D. Gizopoulos; E. Sanchez; M. Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27 n. 3:4–19, 2010.

[32] M. Psarakis; D. Gizopoulos; E. Sanchez; M. Sonza Reorda. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24 n. 1:88 – 89, 2005.

[33] L. Chen; S. Dey. Defuse: a deterministic functional self-test methodology for processors. *VLSI Test Symposium*, pages 5–262, 2000.

[34] M. Mayberry; J. Johnson; N. Shahriari; M. Tripp. Realizing the benefits of structural test for intel microprocessors. *International Test Conference*, pages 456–463, 2002.

[35] M. Tripp; S. Picano; B. Schnarch. Drive only at speed functional testing; one of the techniques intel is using to control test costs. *International Test Conference*, pages 136–143, 2005.

[36] O. Ballan; P. Bernardi; G. Fontana; M. Grosso; E. Sanchez. Fault grading of software-based self-test procedures for dependable automotive applications. *IEEE Design, Automation and Test in Europe Conference*, pages 88–99, 2011.

[37] O. Ballan; P. Bernardi; G. Fontana; M. Grosso; E. Sanchez. A fault grading methodology for software-based self-test programs in systems-on-chip. *IEEE International Workshop on Microprocessor Test and Verification*, pages 88–99, 2010.

[38] A. Avizienis; J. C. Laprie; B. Randell; C. Landwehr. Address and data scrambling: causes and impact on memory tests. *IEEE International Workshop on Electronic Design, Test and Applications*, pages 128 – 136, 2002.

[39] A.J. van de Goor. Using march tests to test srams. *IEEE Design & Test of Computers*, pages 8 – 14, 1993.

[40] D. Appello;V. Tancorre; P. Bernardi; M. Grosso; M. Rebaudengo; M. S. Reorda. Embedded memory diagnosis: An industrial workflow. *IEEE International Test Conference*, pages 1 – 9, 2006.

[41] A. J. van de Goor; S. Hamdioui; G. Gaydadjie. Using a cisc microcontroller to test embedded memories. *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 261–266, 2010.

[42] A. J. van de Goor; G. Gaydadjiev; S. Hamdioui. Memory testing with a risc microcontroller. *Design, Automation and Test in Europe*, pages 214–219, 2010.

[43] A. J. van de Goor; S. Hamdioui; H. Kukner. Generic, orthogonal and low-cost march element based memory bist. *International Test Conference*, pages 1–10, 2011.

[44] A. Benso; A. Bosio; S. Di Carlo; G. Di Natale; P. Prinetto. Atpg for dynamic burn-in test in full-scan circuits. *IEEE Asian Test Symposium*, pages 75–82, 2006.

[45] Oliveira D.A.G. ; Rech P. ; Pilla L.L. ; Navaux P.O.A. ; Carro L. Gpgpus ecc efficiency and efficacy. *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pages 209 – 215, 2014.

[46] Sabena D.; Sterpone L.; Carro L. ; Rech P. Reliability evaluation of embedded gpgpus for safety critical applications. *IEEE Transactions on Nuclear Science*, pages 3123 – 3129, 2014.

[47] Di Carlo S.; Gambardella G. ; Indaco M. ; Martella I. ; Prinetto P. ; Rolfo D. ; Trotta P. Increasing the robustness of cuda fermi gpu-based systems. *IEEE International On-Line Testing Symposium*, pages 234 – 235, 2013.

[48] S. Di Carlo; G. Gambardella; I. Martella ; P. Prinetto; D. Rolfo; P. Trotta. Fault mitigation strategies for cuda gpus. *IEEE International Test Conference*, pages 14–17, 2013.

[49] Ping Xiang; Yi Yang; Huiyang Zhou. Warp-level divergence in gpus: Characterization, impact, and mitigation. *IEEE International Symposium on High Performance Computer Architecture*, pages 284 – 295, 2014.

[50] D. Sabena; L. Sterpone; M. Sonza Reorda; P. Rech; L. Carro. On the evaluation of soft-error techniques for gpgpus. *IEEE International Design and Test Symposium*, pages 102 – 105, 2013.

[51] L. Pilla; P. Rech; F. Silvestri; C. Frost; P. O. A. Navaus; M. Sonza Reorda; L. Carro. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. *IEEE Transactions on Nuclear Science*, pages 1–7, 2014.

[52] Jar-Shone Ker; Tainan; Yau-Hwang Kuo; Bin-Da Liu. Functional test pattern generation for asynchronous circuits. *IEEE International Symposium on Circuits and Systems*, pages 1519 – 1522, 1993.

[53] I. Ghosh I. ; M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 402 – 415, 2001.

[54] S. Zeidler; Wolf C.; Krstic M. ; Kraemer R. Functional pattern generation for asynchronous designs in a test processor environment. *IEEE AsianTest Symposium*, pages 296 – 301, 2012.

[55] Wen C. ; Wang Li.; Kwang-Ting Cheng. Simulation-based functional test generation for embedded processors. *IEEE Transactions on Computers*, pages 1335 – 1343, 2006.

[56] F. Corno; et al. Automatic test program generation - a case study. *IEEE Design & Test of Computers*, 21 n.2:102–109, 2004.

[57] E. Sanchez; M. Schillaci; G. Squillero. *Evolutionary Optimization: the μGP toolkit*. Springer, 2009.

[58] A. Bosio; L. Dilillo; P. Girard; A. Todri; A. Virazel; K. Miyase; X. Wen. Power-aware test pattern generation for at-speed los testing. *Asian Test Symposium*, pages 506 – 510, 2011.

[59] D. Appello; P. Bernardi; A. Fudoli; M. Rebaudengo; M. Sonza Reorda; V. Tancorre; M. Violante. Exploiting programmable bist for the diagnosis of embedded memory cores. *IEEE International Test Conference*, pages 379–385, 2003.

[60] R. Treuer; V.K. Agrawal. Built-in self diagnosis for repairable embedded rams. *IEEE Design and Test of Computers*, pages 24–33, 1993.

[61] T. Bergfeld; D. Niggemeyer; E. Rudnick. Diagnostic testing of embedded memories using bist. *IEEE Conference on Design, Automation and Test in Europe*, pages 305–309, 2000.

[62] S. Boutobza; M. Nicolaidis; K.M. Lamara; A. Costa. Programmable memory bist. *IEEE International Test Conference*, pages 1155–1164, 2005.

[63] Huang; et al. Maximization of power dissipation under random excitation for burn-in testing. *IEEE International Test Conference*, pages 567–576, 1998.

[64] S. Bahukudumbi; K. Chakrabarty. Test-pattern ordering for wafer-level test-during-burn-in. *IEEE VLSI Test Symposium*, pages 193–198, 2008.

[65] D. Appello; et al. An innovative and low-cost industrial flow for reliability characterization of socs. *IEEE European Test Symposium*, pages 321–327, 2008.

[66] D. Appello; et al. Automatic functional stress pattern generation for soc reliability characterization. *IEEE European Test Symposium*, pages 93–98, 2009.

[67] Patrick dt; O'Connor; Wiley; Chichester. *Practical Reliability Engineering*. Wiley, 2002.

[68] John Wiley and Sons. *Wiley Encyclopedia of Electrical and Electronics Engineering - Life Testing*. J.Webster, 1999.

[69] P. Bernardi; M. Rebaudengo; M. Sonza Reorda. Using infrastructure ips to support sw-based self-test of processor cores. *IEEE International Workshop on Microprocessor Test and Verification*, pages 22–27, 2004.

[70] J. Saxena; K. M. Butler; V. B. Jayaram; S. Kundu; N. V. Arvind; P. Sreeprakash; and M. Hachinger. A case study of ir-drop in structured at-speed testing. *IEEE International Test Conference*, pages 1098–1104, 2003.

[71] P. Bernardi; M. De Carvalho; E. Sanchez; M. Sonza Reorda; A. Bosio; L. Dilillo; P. Girard; M. Valka. A functional power evaluation flow for defining test power limits during at-speed delay testing. *IEEE European Test Symposium*, pages 153–158, 2011.

[72] F. Corno; et al. Automatic test generation for verifying microprocessors. *IEEE Potentials 34*, 24:34 – 37, 2005.

[73] L. Glasser and D. Dobberpuhl. *The design and analysis of VLSI circuits*. Reading; Addison-Wesley; MA, 1985.

[74] F. Najm. A survey of power estimation techniques in vlsi circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, page 446, 1994.

[75] S. R. Vemuru; et al. Short-circuit power dissipation estimation for cmos logic gates. *IEEE Transactions on Circuits and Systems I*, pages 34 – 37, 1994.

[76] D. Liu; Svensson. Power consumption estimation in cmos vlsi chips. *IEEE Journal of Solid-State Circuits*, 29:663, 1994.

[77] D. Rabe; W. Nebel. Short-circuit power consumption of glitches. *International Symposium on Low Power Electronics and Design*, pages 125–128, 1996.

[78] M. De Carvalho; et al. An enhanced strategy for functional stress pattern generation for system-on-chip reliability characterization. *IEEE International Workshop on Microprocessor Test and Verification*, pages 29 – 34, 2010.

[79] A. Calimera; et al. Generating power-hungry test programs for power-aware validation of pipelined processors. *23rd Symposium on Integrated Circuits and System Design*, pages 61 – 66, 2010.

[80] E. Sanchez; M. Schillaci; G. Squillero. *Evolutionary Optimization: the GP toolkit*. Springer, 2011.

[81] M. Nicolaidis; Y. Zorian. On-line testing for vlsi - a compendium of approaches. *Journal of Electronic Testing: Theory & Applications*, 12:7–20, 1998.

[82] A. Paschalis; D. Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on CAD*, 24:88–99, 2005.

[83] O. Ballan; P. Bernardi; G. Fontana; M. Grosso; E. Sanchez. Software-based self-testing of embedded processors. *IEEE Transactions on Computers*, 54 n.4:461–475, 2005.

[84] O. Ballan; P. Bernardi; G. Fontana; M. Grosso; E. Sanchez. Built-in sequential fault self-testing of array multipliers. *IEEE Transactions on CAD*, 24 n.3:449–460, 2005.

[85] F. Corno; et al. Fully automatic test program generation for microprocessor cores. *Design, Automation and Test in Europe Conference and Exhibition*, pages 1006–1011, 2003.

[86] minimips available at http://opencores.org/project. *miniMIPS processor*. opencores, 2010.

[87] P. Bernardi; et al. Fault grading of software-based self-test procedures for dependable automotive applications. *Design, Automation and Test in Europe Conference and Exhibition*, pages 1–2, 2011.

[88] E. Sanchez; M. Sonza Reorda; G. Squillero. On the transformation of manufacturing test sets into on-line test sets for microprocessors. *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 494–502, 2005.

[89] V. Tantsios; K. McDonald-Maier N. Bartzoudis. Dynamic scheduling of test routines for efficient online self-testing of embedded microprocessors. *IEEE International On-Line Testing Symposium*, pages 185–187, 2008.

[90] P. Bernardi; L. Ciganda; E. Sanchez; M. Sonza Reorda. An effective methodology for on-line testing of embedded microprocessors. *IEEE 17th International On-line Testing Symposium*, pages 270–275, 2011.

[91] A. Apostolakis; D. Gizopoulos; M. Psarakis; D. Ravotto; M. Sonza Reorda. Test program generation for communication peripherals. *IEEE Design & Test of Computers*, 26 n. 2:52–63, 2009.

[92] M. Grosso; W.J. Perez Holguin; E. Sanchez; M. Sonza Reorda; A. Tonda; J. Velasco Medina. Software-based testing for system peripherals. *Journal of Electronic Testing Theory and Applications*, vol. 28 n. 2:189–200, 2012.

[93] P. Bernardi; E. Sanchez; M. Sonza Reorda; M. Bonazza; O. Ballan. On-line functionally untestable faults identification in embedded processor cores. *IEEE Design, Automation & Test in Europe*, pages 1462–1467, 2013.

[94] M. De Carvalho; P. Bernardi; E. Sanchez; M. Sonza Reorda; O. Ballan. Increasing fault coverage during functional test in the operational phase. *IEEE 19th International On-Line Testing Symposium*, pages 43–48, 2013.

[95] Zhe Fan; et al. Gpu cluster for high performance computing. *Proceedings of IEEE Computer Society, Washington, DC, USA*, pages 43–48, 2004.

[96] D. Luebke. Cuda: Scalable parallel programming for high performance scientific computing. *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 2008.

[97] B. Ranft; T. Schoenwald; B. Kitt. Parallel matching-based estimation: a case study on three different hardware architectures. *IV IEEE Intelligent Vehicles Symposium*, pages 1060–1067, 2011.

[98] L. Pilla; P. Rech; F. Silvestri; C. Frost; P. O. A. Navaus; M. Sonza Reorda; L. Carro. Gpus reliability dependence on degree of parallelism. *IEEE Radiation and its Effects on Components and Systems*, 2013.

[99] P. Rech; T. D. Fairbanks; H. M. Quinn; L. Carro. Threads distribution effects on graphics processing units neutron sensitivity. *IEEE Nuclear and Space Radiation Effects Conference*, pages 1–7, 2014.

[100] S. Di Carlo; G. Gambardella; M. Indaco; I. Martella; P. Prinetto; D. Rolfo; P. Trotta. A software-based self test of cuda fermi gpus. *IEEE European Test Symposium*, pages 33 – 38, 2013.

[101] S. Tselonis; V. Dimitsas; D. Gizopoulos. The functional and performance tolerance of gpus to permanent faults in registers. *IEEE 19th International On-Line Testing Symposium*, pages 236 – 239, 2013.

[102] A. Benso; P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer, 2003.

[103] J. Tan; N. Goswami; T. Li; X. Fu. Analyzing soft-error vulnerability on gpgpu microarchitecture. *IEEE International Symposium on Workload Characterization*, pages –, 2011.

[104] B. Fang; J. Wei; K. Pattabiraman; M. Ripeanu. Towards building error resilient gpgpu applications. *SC Companion: High Performance Computing, Networking Storage and Analysis*, pages –, 2012.

[105] L. Bautista Gomez; F. Cappello; L. Carro; N. DeBardeleben; B.Fang; S. Gurumurthi; K. Pattabiraman; P. Rech; M. Sonza Reorda. Gpgpus: How to combine high computational power with high reliability. *Design, Automation & Test in Europe Conference & Exhibition*, pages –, 2014.

[106] N. Farazmand; R. Ubal; D. Kaeli. Statistical fault injection-based avf analysis of a gpu architecure. *IEEE Workshop on Silicon Errors in Logic - System Effects*, pages –, 2012.

[107] NVidia. Cuda-gdb documentation. *http://docs.nvidia.com/cuda/cuda-gdb*, 2014.

[108] J. Elliott; F. Mueller; M. Stoyanov; C. Webster. Quantifying the impact of single bit flips on floating point arithmetic. *Tech. rep., Tech. Rep. ORNL/TM-2013/282, Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, 2013. 6, 9*, pages –, 2013.

[109] D. Appello; M. Barone; P. Bernardi; M. De Carvalho; A. Panariti; M. Sonza Reorda; N. Campanelli; T. Kerekes. Cumulative embedded memory failure bitmap display and analysis. *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 1155–1164, 2010.

[110] A. Vassighi; et al. Cmos ic technology scaling and its impact on burn-in. *IEEE Transactions on Device and Materials Reliability*, pages 208–221, 2004.

[111] M. Elbert; et al. Stress testing and reliability. *IEEE Southcon*, pages 357–362, 1994.

[112] K. Roy; et al. Stress testing of combinational vlsi circuits using existing test sets. *IEEE International Symposium on VLSI Technology*, pages 93–98, 1995.

[113] V. Dabholkar; S. Chakravarty; J. Najm; J. Patel. Cyclic stress tests for full scan circuits. *EEE VLSI Test Symposium*, pages 89–94, 1995.

[114] D. Gizopoulos; Y. Zorian; A. Paschalis. *Embedded Processor-Based Self-Test*. Springer-Verlag, New York (USA), 2004.

[115] Keun Soo Yim et. al. Lightweight silent data corruption error detector for gpgpu. *IEEE International Parallel & Distributed Processing Symposium*, pages –, 2011.

[116] T. Powell; A. Kumar; J. Rayhawk; N. Mukherjee. Chasing subtle embedded ram defects for nanometer technologies. *IEEE International Test Conference*, pages 850–858, 200.

[117] Z. Conroy; G. Richmond; X. Gu; B. Eklow. A practical perspective on reducing asic ntfs. *IEEE International Test Conference*, pages 349–355, 2005.

[118] J. B. Khare; A. B. Shah; A. Raman; G. Rayas. Embedded memory field returns - trials and tribulations. *IEEE International Test Conference*, pages 1–6, 2006.

[119] J. Crafts; et al. Testing the ibm power 7 tm 4 ghz eight core microprocessor. *IEEE International Test Conference*, pages 1–10, 2010.

[120] H. Al-Asaad; et al. Online bist for embedded systems. *IEEE Design & Test of Computers*, pages 17–24, 1988.

[121] A. Merentitis; et al. Directed random sbst generation for on-line testing of pipelined processors. *IEEE On-Line Testing Symposium*, pages 273–279, 2008.

[122] E. Sanchez; et al. On the transformation of manufacturing test sets into on-line test sets for microprocessors. *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 494– 502, 2005.

[123] J. Shen; A. Abraham. Synthesis of native mode self-test programs. *Journal of Electronic Testing: Theory and Applications*, pages 137–148, 1998.

[124] P. K. Parvathala; et al. Functional random instruction testing (frits) method for complex devices such as microprocessors. *United States Patent 6948096*, pages 137–148, 2005.

[125] I. Bayraktaroglu; et al. Cache resident functional microprocessor testing: Avoiding high speed io issues. *IEEE International Test Conference*, page 27.2, 2006.