ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Control and Computer Engineering (33rd cycle)

Self-Test Mechanisms for Automotive Multi-Processor System-on-Chips

Andrea Floridia
* * * * * *

Supervisor
Prof. Ernesto Sanchez, Supervisor

Doctoral Examination Committee:
Prof. Alberto Bosio, Referee, Ecole Centrale Lyon
Prof. Giorgio di Natale, Referee, TIMA Laboratories
Prof. Liviu Miclea, University of Cluj-Napoca
Prof. Haralampos Stratigopoulos, Sorbonne Universitè
Prof. Matteo Sonza Reorda, Politecnico di Torino

Politecnico di Torino
23 September 2021

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . .

Andrea Floridia

Turin, 23 September 2021

Acknowledgements

This work would not be possible without the support of many people. Foremost among them are, Ernesto Sanchez and Matteo Sonza Reorda, whom I thank for their encouragement and the opportunities they gave me.

To Aleksa and Esteban, just thank you (you know what I am referring to!).

To all the people in Lab3 and friends met in Turin in these five years, they have been a source of sanity in an endeavor that has occasionally been far from being sane.

To Roberta, that remembers me everyday how lucky I am.

Last but not least, my whole family, who have been enormously supportive even at 1500 kilometers away and always believed in me, even when no one else did.

Summary

While the technology for enabling fully autonomous self-driving cars is still ahead, today automobiles massively rely on electronics for a variety of functionalities. As these functionalities require more and more computational power, the embedded systems introduced in the automobiles had to evolve accordingly. As of today, Multi-processor System-on-Chips (MPSoCs) are commonly found for these applications. Such SoCs embeds two or more processor cores within the same silicon die, in conjunction with different peripherals and levels of memories (both static and non-volatile). It is known that due to the harsh environment in which they are deployed, physical malfunctions due hardware faults can manifest. Periodic in-field self-testing represents a common countermeasure against these threats. These mechanisms can be implemented both in hardware and software but, most of them were originally devised for simpler single-processor SoCs. While in recent years there has been a considerable effort in improving the hardware-based self-test mechanisms, the same is not true for the software-based ones. Software-based approaches mainly consist in the application of software self-test routines (or procedures) of a Software Test Library (STL). Over the years they have been shown to be valuable especially for the processor core, being the most critical portion of the system.

Therefore, the first contribution of this thesis was to study the applicability of STLs in a multi-processor context. Rather than the development of new Software-Based Self-test (SBST) methods, the main focus was the parallel execution of already developed STLs. This originates from the fact that automotive MPSoCs reuse the same IP processor cores that have been used (and extensively verified) for single-processor devices. In MPSoCs, it has been widely reported that the major hurdle of embedded software is its predictability in terms of execution time. In fact, when all the processor cores are active at the same time, the system bus activity considerably increases with respect to a single-core system. This higher activity (which induces the processor pipeline to stall) impacts the performances of each processor, since the accesses to the memory sub-system are delayed. As the STL is in practice a piece of embedded software, it is exposed to the same issues.

The most significant achievement under this perspective consisted in the development of a software scheduler. Such a scheduler differs from any other software

scheduler for embedded systems, since it is tailored for the needs of STLs in an automotive MPSoC. Through extensive experiments, it was proven to be a valid solution to fit the narrow test windows of industrial automotive MPSoCs (maximizing the overall system availability). At the same time, the proposed scheduler demonstrated a good execution time predictability. This is remarkable, since it was already mentioned that the embedded software execution time is hard to be predicted accurately.

Furthermore, the research developed in this thesis demonstrates for the first time through real industrial case study that this unpredictability is particular harmful for self-test procedures. Indeed, it was observed that such unpredictability not only alters self-test procedure execution time (when executed in parallel). Additionally, some self-test procedures intermittently fail when in field and/or produce a fluctuating fault coverage. For both cases, it was proposed a mitigation technique based on the usage of the inner most level of private cache memories. When the self-test procedures are executed from such private memories, with precautions, it is possible to isolate the self-test procedure execution form the rest of the system (thus achieving the required stability).

When dealing with self-test mechanisms, it exists a further category of mechanisms which are said to be hybrid. They indeed consist of both hardware and software cooperating together for implementing an efficient self-test. This thesis contributes to this new emerging self-test approach with a novel hybrid technique for checking the integrity of the comparators used in the lockstep configuration (commonly found in automotive MPSoCs).

The third contribution of this thesis consists of optimizing the fault grading methodologies to meet the necessities of the functional fault simulation. Functional fault simulation is an emerging approach for performing fault simulation in safety-critical applications when a processor executing software is involved. Recently, the Electronic Design Automation (EDA) companies are providing tools able to support these methodologies. However, different aspects must be taken into account, not previously considered with the traditional fault simulation approaches. It is worth noting that the applicability of these researches is not limited to processors of an automotive MPSoCs. One of the main applications of functional fault grading addressed in this thesis is in the STL development. The main bottleneck of the STL development remains the fault simulation. Therefore, part of the research efforts were directed towards the formulation of functional fault grading methodologies intended for STLs. The key concept behind these methodologies is the fault dropping which allows to considerably reduce the effort for the fault simulation.

The fourth contribution still concerns the functional fault grading, but in a rather different scenario. Due to the ever-increasing complexity of the newer devices, designers are shifting to emulation of the ASICs in order to speed up the verification process. The same emulators can be used as well for quickly evaluating the effectiveness of self-test mechanisms or general dependability analyses that require functional fault grading. This can be achieved by instrumenting the original netlist, in order to enable the injection of different faults (most often either Single-Event Transient or stuck-at). In this context, this thesis introduces a fault emulation platform to support dependability analyses of safety-critical Application-Specific Integrated Circuits (ASICs). Differently than existing works, the focus was the fault detection mechanism that allows to mimic the detection mechanisms of a fault simulator (including fault dropping introduced above). The proposed platform can be integrated in the already-existing IEEE 1149.1 JTAG infrastructure of the target ASIC. Therefore, it can be easily accessed with standard tools and perfectly compatible with the modern industrial emulators based on Field-Programmable Gate Arrays (FPGAs).

This thesis is organized as follow: Chapter 1 is devoted to the introduction. The motivations leading to this thesis are discussed in great detail. Moreover,the basic terminology used throughout the manuscript is provided. Then, the thesis is divided into two main parts. The first one addresses the description of the self-test mechanisms for automotive MPSoCs (Chapter 2, 3, 4).

The second part instead, focuses on the improvements in the functional fault grading methodologies (Chapter 5, 6). In Chapter 7 concludes the thesis. In that chapter, both the major achievements and future research directions are analyzed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays the automotive domain is one of the main drivers for the most recent technology advancements. The pursue over the last decade for fully autonomous (or self-driving) cars has radically changed the architecture of the embedded systems deployed in such domain. While originally relatively small System-on-Chips (SoCs) were deployed, today there is an opposite trend. Indeed, driven by the necessities of more complex software tasks required for implementing Advanced Driver Assistance Systems (ADAS), Multi-Processor SoCs (MPSoCs) can often be found in replacement of the simpler ones.

MPSoCs (or simply *multi-core systems*, the two terms are considered here as synonyms) are mixed-signal SoCs, since they include on the same silicon die both a digital (consisting of multiple processor cores, memories) and an analog portion (e.g. analog-to-digital converters). Since the focus of this thesis is on the digital portion, it is exclusively further discussed in the following. Before being adopted, the MPSoC must satisfy the requirements imposed by so-called functional safety standards. For the automotive field, the ISO 26262 [50] regulates the usage of electronics devices. The standard is an adaptation to automotive systems of the more general IEC 61508 [49]. The ISO 26262 covers the entire spectrum of the functional safety of electronics components for automotive applications, from the software to the hardware. Specifically to the hardware, the ultimate goal is to avoid that a failure in a given hardware block provoke a catastrophic consequence (e.g., damage to human beings). For avoiding such catastrophic consequences, the standard suggests different solutions. These are called *safety mechanisms*. A safety mechanism is a portion of the system intended for detecting faults and controlling system failures in order to achieve or maintain a safe state. Depending on the risk associated to the failure of the system, the standard defines the Automotive Safety Integrity Levels (ASILs). They are labeled with A, B, C and D. Systems, or more in general a hardware module, labeled as ASIL A signifies that the module is not safety relevant. On the other hand, modules labeled ASIL D are the most critical ones. For each of these levels, the standards defines the most appropriate safety mechanisms in order to meet some minimum reliability requirements. Failures are caused due to the occurrence of random hardware faults, which are distinguished in Single-Point Faults and Multi-Point Latent Faults. The former are immediately effective faults, meaning that when one of these faults occurs, the effects of its occurrence lead to a failure within few clock cycles. Instead, Multi-point Latent faults are faults within the safety mechanism. Their occurrence do not cause directly a failure. However, they might become dangerous when a second fault arise in the module guarded by the safety mechanism. Concerning these two types of faults, the ISO 26262 defines two metrics (directly related to the reliability requirments): The Single-Point Fault Metric (SPFM) and the Latent Fault Metric (LFM). Each of the above-mentioned ASILs has different requirements for these two metrics (Table 1.1). For assessing whether a given hardware module meets the required metrics for the targeted ASIL, the Failure Mode Effects and Diagnostic Analysis (FMEDA) [62, 73] is performed. It defines failure modes, failure rate and diagnostic capabilities for a given hardware module. In the context of this thesis, diagnostic capabilities refer to the ability of detecting specific faults (either single or multi-point faults). Together with the failure rate, they directly contribute to determine the SPFM. The failure rate is measured in FIT (Failure In Time) and strongly depends on the technology node used for manufacturing the device. The diagnostic capabilities are expressed with the Diagnostic Coverage (DC). The DC indicates the number of critical faults leading to a failure detected by the safety mechanism under analysis. It is normally computed resorting to fault injection campaigns. For these, two fault models are typically considered: the stuck-at and the Single-Event Upset or Transient (SEU and SET respectively). During early dependability analyses, the SPFM can be roughly approximated with the DC. Instead, the LFM indicates the number of latent faults covered in the safety mechanisms and it is computed considering exclusively stuck-at faults. Throughout this manuscript, the bare fault coverage of a given safety mechanism or self-test procedure is simply referred to as Fault Coverage. Withe Diagnostic Coverage instead the one considering the most critical faults only.

Table 1.1: SPFM and LFM Requirements for Safety-relevant Modules

ASIL	**SPFM/DC**	**LFM**
B	≥90%	≥60%
C	≥97%	≥80%
D	≥99%	≥90%

For these two metrics, the standard also defines the Fault Tolerant Time Interval (FTTI). The FTTI is defined as the time interval required for detecting a fault and then react accordingly. For example, for ASIL D devices the FTTI for single-point faults is in the range of 10-150ms. For the latent faults, the FTTI is expressed as a multiple of 10 hours.

In general, certifying an SoC for the highest ASILs (C or D) does not require deriving the SPFM and LFM metrics considering the SoC as a whole. Instead, it is followed the process known as ASIL decomposition. For each safety-relevant sub-module of the SoC, one or more safety mechanism is devised. Then, the final coverage achieved on the entire SoC is given by the combination of the coverages achieved in each sub-module. For example, considering the LFM for an ASIL D SoC, this implies in practice that it is not necessary to achieve a 90% coverage in each sub-module.

When considering the SPFM metric, the most commonly used safety mechanisms are based on redundancy. These are also called *primary safety mechanism*. When dealing with memories [38, 25, 63], Error Correction Codes (ECCs) are used to protect against bit flips or errors in the memory array. More recently, advanced schemes known as End-to-End ECCs [55] have been shown to be effective also to protect against faults arising in the data path from a processing element (e.g., a processor core) to the memory. As opposed with traditional approaches, End-to-End strategies computes the ECC when the data is actually leaving the processing element and it is send over the system bus to the memory. The ECC is send along with the data over the system bus. When the data reaches the memory, the ECC is computed again before being stored. In case the computed ECC differs with the one that was send, a fault occurred. In the most advanced schemes, the ECC is computed for control signals too.

For the processing elements as the processor cores, or safety-relevant hardware blocks as the Direct Memory Access (DMA), the most common strategy consists of replication [43]. For example, considering the processor core, the Dual-Core Lockstep (DCLS) [51] is the most commonly used. Two processor cores (main and checker) are paired together and fed with the same identical inputs. Their outputs are continuously monitored by a set of comparators that signal a mismatch due to the occurrence of a fault.

As already mentioned when introducing earlier the LFM metric, the additionally circuitry of these safety mechanisms is equally exposed to faults occurrence. This could invalidate the safety mechanism functionalities and might cause failures of the safety-relevant blocks to go undetected. Therefore, additional *diagnostic safety mechanisms* are required. They are exclusively intended for implementing in-field self-test mechanisms functionalities to avoid latent faults accumulation in the primary safety mechanisms. It is worth mentioning that the in-field test is more constrained in terms of resources and fault models compared to the end-of-manufacturing test. In general, the in-field test does not rely on an external tester (i.e., an ATE) and the test patterns must be internally generated/stored within the device itself. Moreover, since the test is performed when the device is already in the mission environment there exist also narrow test windows in which test must complete. For these reasons, exclusively stuck-at faults are in practice considered when devising self-test mechanisms for the LFM metric. Indeed, if one would consider also more advanced faults models such as cell-aware ones [46], the test time required for reaching an acceptable coverage would exceed the available test window.

When considering automotive devices (either single or multi-core), the in-field test is further distinguished into the Power-On Self-Test (POST) and the on-line self-test [82]. The former is the self-test performed when the device is turned on. The latter, is the in-field test performed concurrently with the mission software that the SoC is supposed to be running.

During the POST, the preferred self-test mechanisms are based on the Logic and Memory Built-In Self-Test principle (LBIST and MBIST respectively)[89, 66]. The former addresses permanent faults in the safety-related digital logic. The latter instead within the embedded memories. In the literature there exist MBIST strategies [74] that allow for a transparent execution with respect to the memory content. On the other hand, the LBIST requires a full system reset after its completion. Indeed, LBIST is based on the already existing scan logic and thus it produces non-functional stimuli. Therefore, its applicability is limited to the POST only.

This could become problematic if the time interval between two power-on events is too long, as in the case of several hours of continuous operations. Consequently, since latent faults must be checked even when the system is fully on-line, Software Test Libraries are increasingly becoming used. An STL consists of a set of software self-test procedures (or programs), and the main target are permanent faults within the processor core [83, 58, 76, 42, 4, 41, 56, 81]. This technique is also known in the literature as Software-Based Self-Test or SBST. The idea is relatively simple: when the target device corresponds to or includes a processor, we can force it to execute a suitable piece of code, possibly reading some input data and processing them in a carefully selected manner. The produced output data are accumulated to form a test signature. Such a signature is then used in filed to determine whether the test passed or failed. Normally, this kind of self-test exclusively relies on the already-existing on-chip resources.

In contrast with the non-functional approaches such as the LBIST, the STL produces exclusively pure functional stimuli. However, unlike the traditional functional tests, this approach relies on structural information of the processor core under test (i.e., a gate-level description). Therefore, it is possible to compute a test coverage with respect to a given fault model. Since the self-test is performed with the processor operating in mission mode, normally this self-test approach complements the non-functional stimuli produced by the LBIST [65, 45]. Indeed, it is often the case that due to test mode constraints or clock gating structures, some portions of the design are not fully accessible by the LBIST.

When used in automotive devices [13], the STL has two main roles depending on the targeted ASIL. For ASIL B devices, the STL is often used in conjunction with other techniques as primary safety mechanisms to detect the occurrence of single-point faults. For the most critical ones (namely ASIL C or D), the STL is used for the purpose of detecting latent faults to avoid their accumulation in the checker (or main) core of the DCLS configuration.

For both purposes, the STL is composed of two main portions: a boot-time and a run-time portion. The self-test programs belonging to the first category are executed during the namesake phase of the device, when the system is entering the on-line phase. In this phase, the processor initializes the different peripherals, the internal RAM blocks, the PLL and others. Before acting on these, the processor executes the boot-time self-test programs. The goal of this self-test is to check most of the processor functionalities. Self-test programs belonging to this category have complete access to the available hardware resources: the processor Special Purpose Registers, the Interrupt Vector Table. Additionally, they trigger exceptions and preemption is not allowed. For the sake of test purposes, they require to access specific addresses in the shared portion of the system RAM, outside the boundaries of the processor stack frame. Although there are in general fewer real-time constraints on the execution of these kind of self-test programs, there is typically an upper bound for the self-test to complete (also for sake of system availability). Indeed, the MPSoC is part of a lager Electronic Control Unit (ECU). Within an automobile, there are several ECUs that exchange information over a local network. At the key-on, all the ECUs execute their own self-test and acknowledge over the network the whether the test passed or failed. To avoid waiting endlessly for a control unit response (which might be stuck due to a fault), a timeout is typically set (which represents the aforementioned upper bound).

The self-test programs that belong to the second category are executed when the system is fully on-line. That is, the operating system and the application software are already running. They are conceived to coexist with the application source code and they target mainly the computational units (e.g., arithmetic units) within the processor core. Since they are executed in real time, they do not alter the processor status and can be interrupted in case higher priority tasks require to be executed. Differently than the boot-time self-test programs, the run-time ones do not access system RAM addresses outside the processor stack frame. This kind of self-test programs are designed to have minimal access to on-chip resources (i.e., minimal intrusiveness) due to the coexistence with the application software. Therefore, in general, the achievable coverage is lower compared to the boot-time ones.

The effectiveness of an STL (technically called fault grading) is normally assessed with fault simulators [14] and it is historically the most expensive part of the development flow. Additionally, the functional fault simulation is increasingly often required. This kind of fault simulation is an emerging approach that stems from the needs of the FMEDA analysis. Recently, it is being supported by the EDA vendors too. This methodology differs from the traditional one since it simulates in a unique model both the processor and the memories (data and code). Specifically, the stimuli are retrieved from the memories, unlike in the traditional one in which the fault simulator provides to the design under test (e.g., the processor) the input stimuli (usually stored in a textual file).

In general, fault simulation is a critical bottleneck in FMEDA analysis since it is required for several applications.

In order to address the growing complexity stemming from the usage of MPSoCs, there has been a considerable effort from the research community in order to improve the self-test capabilities. Most of these efforts were directed towards the improvement of the LBIST [69, 60, 68, 67]. Specifically, the lager becomes the circuit under test the higher is the pattern count for reaching the target LFM coverage. This implies that the test application time increases, because patterns are generated internally. Furthermore, the scan shift frequency is normally limited for sake of power consumption. Towards this end, the main research efforts considered test point insertion (often shared with same test compression logic used for end-of-manufacturing testing) for reaching the same coverage while reducing the pattern count. Additionally, new LBIST schemes have been introduced that considers the necessities of the automotive integrated circuits such as the LBIST with Observation Scan Technology (LBIST-OST) [70]. In that approach, a dedicated scan chain is inserted in the design with the only purpose of capturing data from the combinational logic during the shift cycle. This yield a significant advantage, since traditionally exclusively during the capture cycle the combinational logic is observed.

For the STL-based approaches, or more in general, those based on the SBST principle the same is not true. Indeed, most of the research efforts focused on the end-of-manufacturing testing only. In that field, several works proposed different solution aiming at reducing the test application time [6, 5, 37]. Since the in-field self-test presents different constraints with respect to the scenario of the end-of-manufacturing testing, these techniques are clearly not applicable.

The main problem to be faced when considering STLs for the in-field self-test of MPSoCs is still the test application time. However, the problem is not related to the number of stimuli to be provided. The processor cores used in MPSoC are often the same used in single-core SoCs for sake of shortening the design cycle and time-to-market for new devices. Therefore, the processor complexity is not altered and the already-developed self-test algorithms for testing the processor modules are re-used. The difference in MPSoCs is that all the available processor cores must execute the same STL in almost the same test time window of the single-core SoC. Ideally, a parallel execution of the STL would be the best solution. However, this is not always possible, as it is better detailed in the dedicated chapter. Indeed, other complications arise since the on-chip resources (in terms of available memory) for the STL are usually limited.

For the reasons listed above, rather than the development of new self-test algorithms, the thesis focuses on effective strategies for the parallel self-test when in field. The end goal is to reduce the test application time, while maintaining at the minimum the amount of required resources.

While there exist two main approaches for the self-test of latent faults (i.e., the software and the hardware ones), during the development of the thesis a further approach was explored: the hybrid one. This approach is based on a cooperation between software and hardware modules to implement the self test. Specifically to this thesis, a hybrid on-line self-test approach was investigated as countermeasure for latent faults in the comparators of a DCLS system. For this and similar purposes, the hybrid self-test approach is shown to be a valid alternative to the most expensive (in terms of physical resources) hardware-based approaches. At the same time, they inherit the flexibility of the software while overcoming the classical limitations of these approaches (i.e., they are able to produce functional stimuli only).

The second major focus of this thesis was on the fault grading process. On one hand the idea was to improve the fault simulation methodology for STL to meet the recent improvements concerning the functional fault simulation. Different techniques to alleviate the effort required for the functional fault simulation were proposed. Such techniques target the two different usages of the STL. That is, part of the techniques are for the STL used as primary safety mechanism. This is the case of single-core SoC or specific processor or co-processor within an MPSoC intended for security tasks. The other techniques are for the STL used against latent faults in a DCLS configuration. On the other hand, the fault grading process can be further improved when considering FPGA-based fault emulation. Historically, fault emulation was limited to transient fault models, most notably SET and SEU. This was due to the relative limited capacity of the emulators themselves. However, the recent improvements in the FPGA emulators capacity, now able to support several billions of logic gates, make the emulation of permanent faults affordable. Differently than the works present in the literature concerning fault emulation, in the scope of this thesis the focus was the observation mechanism. Such term refers to the mechanism through which a given fault can be marked as detected or not. The one developed in this thesis resembles the fault dropping mechanism implemented in modern fault simulators. This method allows to reduce the time required for each the fault simulation since faults are simulated until a difference is detected (with respect to a fault-free reference). The same principle was exploited in the proposed fault emulator too, allowing for a reduction of the effort required for the fault grading.

Part I

On-line self-test mechanisms for automotive MPSoCs

Chapter 2

Deterministic in-field parallel execution of self-test programs in MPSoCs

The purpose of this chapter is to introduce the problems to which self-test programs are exposed when executed in parallel in an MPSoC. These problems are always present independently from the selected scheduling approach. Indeed, embedded software executed in a multi-core context suffers of a limited timing predictability due to the higher system bus contention. When dealing with self-test procedures, this higher contention might lead to a fluctuating fault coverage or even the failure of some test programs when in field. To mitigate this issue, a cache-based strategy is proposed. The methodology does not require significant modifications of the already-existing algorithms and it does not introduce penalties from the memory footprint perspective. Along with these advantages, it does not require any additional on-chip resources.

This chapter is organized as follows: initially, a background section provides a clear problem statement and an overview of the related works already present in the literature. Then the proposed strategy is detailed. Finally, experimental results demonstrate that it is possible to achieve a stable execution while also improving the state-of-the-art approaches for the on-line testing of embedded microprocessors. The effectiveness of the methodology was exhaustively assessed on representative processor modules of an industrial MPSoC manufactured by STMicroelectronics.

This proposed approach and experimental results were published in [34].

2.1 Background

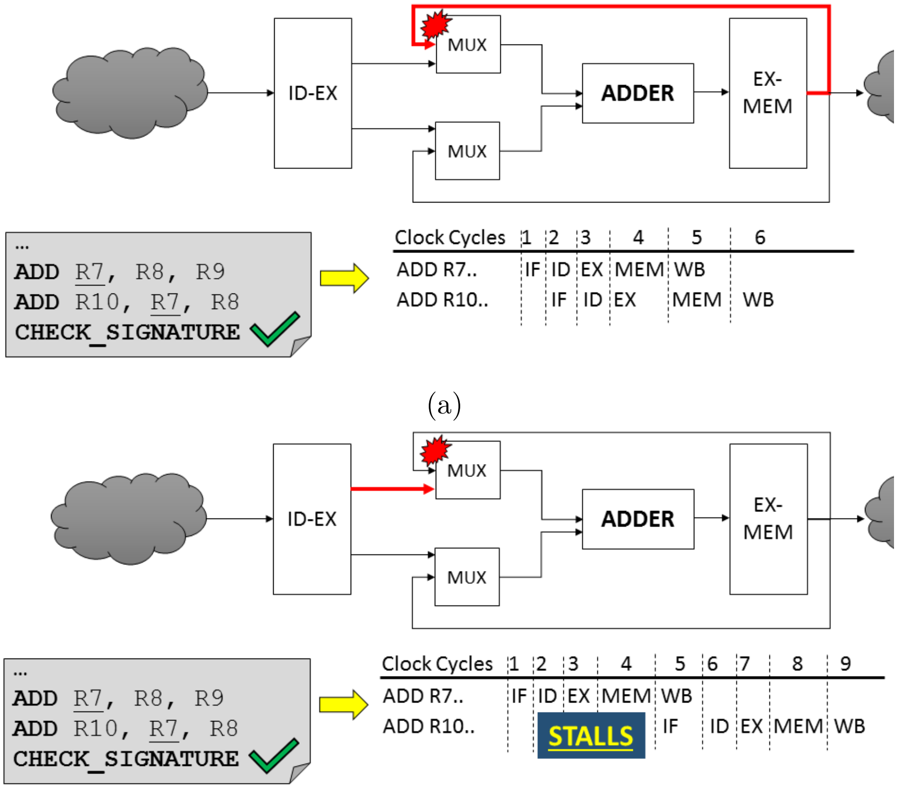
2.1.1 Problem statement

The main idea of the self-test via software is to convert test patterns into software instructions and accumulate their results to create a so-called test signature. Then, such a signature is compared with the expected test signature (obtained in a fault-free scenario, for example resorting to simulators) to determine whether the test passed or failed. When the test is executed in field, the test signature represents the only way to safely detect the occurrence of faults. When considering an MPSoC, a parallel execution of an STL is attractive for shortening the test application time. For the vast majority of the self-test programs composing an STL, this does not constitute a problem. However, some boot-time self-test programs, in order to be effective, require a proper sequence of instructions to be executed without any interruption. In MPSoC, this assumption cannot be guaranteed anymore. Indeed, the embedded software running in a multi-core context suffers of a limited timing predictability [61, 20], due to the higher system bus contention. These conflicts on the system bus block the processor pipeline when fetching instructions from the main memory. Due to these clock cycles of stall, the exact stream of instructions entering the pipeline cannot be determined in advance anymore. In an MPSoC, this has two important consequences on the self-test procedures requiring a specific sequence of instructions. The first one concerns the fault grading: the fault coverage is uncertain and it might vary depending on which portion of the processor is excited due to the system bus activity. Because of this, a given fault location might not be excited correctly and therefore remains undetected. This represents a serious concern, since the ISO 26262 standard imposes stringent quality requirements in terms of achieved fault (or diagnostic) coverage. The second one is related to the signature generated by the test program, which is now unstable. It means that the self-test procedure cannot safely identify whether the mismatch in the signature is due to the occurrence of a fault or due to an unexpected instructions stream.

For better clarify this problem, let us consider two examples: the forwarding and hazard detection mechanism of the classical 5-stage pipeline of the DLX processor [77] and two self-test routines for testing such mechanisms.

Let us consider the forwarding mechanism. The reported example considers forwarding among two consecutive instructions. It is important to note that the same reasoning is perfectly applicable even to more complex multiple-issue processors. The only difference is that the forwarding can also take place among two consecutive issue packets. Let us focus on the following forwarding path: the EX to EX path that fed with the first operand the processor adder. Figure 2.1a shows a portion of the assembly code testing the aforementioned path, along with its evolution across the pipeline stages in a single-core scenario.

In this case, the forwarding mechanism is excited correctly. The second add



```
...
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6
ADD R7..	IF	ID	EX	MEM	WB	
ADD R10..		IF	ID	EX	MEM	WB

(a)

```
...
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6	7	8	9
ADD R7..	IF	ID	EX	MEM	WB				
ADD R10..		STALLS			IF	ID	EX	MEM	WB

(b)

Figure 2.1: (a) Single-core execution of the program. (b) Multi-core execution of the program. In red the different paths activated by the same program.

instruction enters the pipeline exactly one clock cycle after the first one, since the memory subsystem has not produced any stall. Figure 2.1b represents still the same code fragment, but in a quite different scenario. It is assumed that the processor is part of a larger multi-core system, and the self-test procedure is executed in parallel by the other cores. As a result of the other processors' activities, the accesses to the memory subsystem are delayed. As depicted in that figure, the targeted forwarding path is not triggered at all. The second add enters the pipeline at the fifth clock cycle and can retrieve the content of R7 directly from the register file, without activating the forwarding path. This is a possible scenario that a self-test routine might encounter when executed in parallel in an MPSoC. In this context, the self-test procedure yield a correct signature. Indeed, the results produced by the two operations is still valid. However, the targeted forwarding path was not activated at all. Thus, although it returns a correct signature, the fault coverage is likely to be quite different since it varies according to the whole system activity.

When testing hazard detection mechanisms, in order to detect the occurrence of performance faults [48] the processor Performance Counters can be exploited (when available). Performance Counters that count the number of pipeline stalls are particularly useful since they could ease the detection of malfunctions in the hazard detection unit (e.g., stalls inserted between instructions when not needed). The self-test programs is the same seen before for the forwarding mechanisms, but this time the Performance Counters contribute to the signature. In this case, it is likely that the test program will produce an unstable signature. Considering the examples of Figure 2.1b the execution time is slightly increased due to the additional stalls, but yet enough for altering the values of the Performance Counters, that will report 3 additional stalls. Once again, these stalls are completely unpredictable and consequently also the signature.

Even though these phenomena have been described using as examples the forwarding and hazard detection units, they are applicable to all those self-test procedures that require a specific sequence of instructions to be executed without interruptions.

To mitigate these issues and achieve a deterministic execution, a cache-based approach has been explored. This well matches the requirements of safety-critical embedded software, for which predictability is mandatory and the memory resources are limited.

2.1.2 Related works

The usage of caches has been explored to store the self-test procedures intended for end-of-manufacturing testing of processors within a shared-memory multi-core system [6]. The purpose of that work was to reduce the test application time, avoiding off-chip memory accesses. The method is applicable exclusively for end-of-manufacturing, since it assumes that the self-test procedures are loaded into the caches through an external tester (which is not available when in field). Similarly, in [87] it was shown that a cache-aware test scheduler can take advantage of the memory hierarchy for speeding-up the run-time tests. Differently from these related works, the proposed approach deals with the in-field execution of boot-time procedures, and it uses caches for addressing the uncertainties introduced by a multi-core architecture.

A possible alternative to the proposed one consists in exploiting the processor Tightly-Coupled Memories (TCMs, also known as scratchpad memories) [7, 9]. This approach is typically adopted for the execution of real-time programs. Such programs are copied (during the system boot) and then executed from the instruction TCM when required. Conceptually, TCMs are similar to caches since they consist of a bank of SRAM local to each processor. Unlike caches, there is not the concept of cache miss or hit, since data or instructions have to be copied explicitly to these memories before being used. However, the fundamental drawback is that part of the TCM should be permanently reserved for test purposes (the amount of extra memory occupied is proportional to the size of the test program). Clearly, this impacts negatively on both portability and flexibility of the STL.

2.2 The cache-based execution approach

The vast majority of computer programs exhibit the so-called principle of locality [77]: that is, a given program will access a (relatively) small portion of the available address space. Two locality principles exist: temporal and spatial locality. The former states that if a given memory address is referenced, then it is likely that it will be referenced again soon. The latter stems from the observation that programs are generally executed sequentially and they seldom branch far from the actual program counter value. Additionally, data are often stored in contiguous memory locations: therefore, if a given memory location is accessed, then it is likely that the locations nearby will be accessed soon. Caches leverage these principles, by storing the content the most referenced addresses (i.e., data and instructions). In a multi-processor system, this provides isolation, considerably increasing the processor performances. Although these advantages, the caches are not deterministic since the actual increase in performance depends on the program length and organization, the cache size itself, and how often a context switch is performed. Therefore, issues could arise when using caches in conjunction with self-test procedures, since some of them require a precise timing.

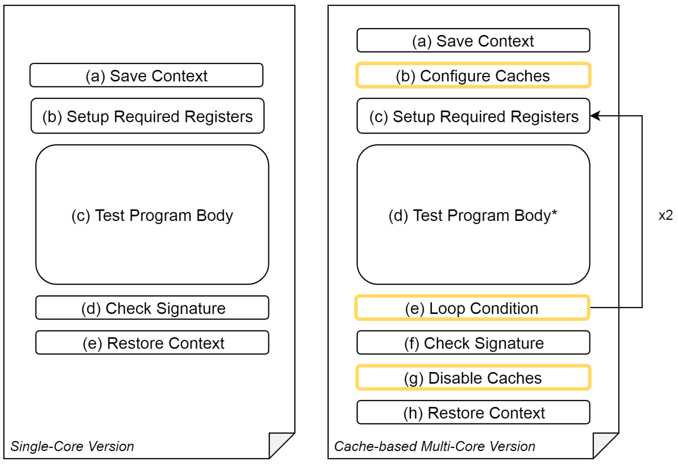
However, it is possible to achieve a deterministic cache-based execution if the test program:

- it is executed without any interruption;
- it exhibits strong temporal and spatial locality.

Given these two conditions, the idea is to move the self-test routine within the innermost level of caches (i.e., the ones private to each processor core), isolating its execution from the rest of the system.

From the above mentioned definitions of the locality principles, it is possible to derive a general structure, that embeds the single-core version of the self-test procedure. Given a generic boot-time test program, the few modifications required are:

1. The test program should be executed twice in a loop-based fashion. The body of the loop (blocks c and d in Figure 2.2b) is represented by the instructions intended for testing the processor which compose the single-core self-test procedure (Figure 2.2a, blocks b and c). This allows for a strong temporal locality, since all the addresses are referenced exactly twice. During the first iteration (hereinafter loading loop, Figure 2.3a), the test program is moved into the instruction cache. At the same time, the content of the data memory addresses referenced (if any) during this first iteration are moved within the



(a) (b)

Figure 2.2: The proposed Cache-based strategy. On the left-hand side the single-core version. On the right-hand side the modified multi-core test program version. In case of no-write allocate caches, the Test Program Body might be lightly modified.

data cache, assuming a write allocate cache memory. If this is not the case (i.e., a no-write allocate policy) each store operation must be followed by a dummy load operation to the same address. This will provoke a read cache miss, that in turn causes data to be moved within the data cache. Therefore, during the execution loop all the store operations will not generate a write miss, since they will find the proper data already in cache. It is important to note that during the loading loop the test program must not perform any check of the signature. Since the first execution might be still influenced by the other processors' activity, the computation of the signature is unreliable. Instead, the second iteration (the execution loop, Figure 2.3b) is the real test program execution. Since the program is executed entirely from the caches, the signature can be computed without the risk of being influenced by the rest of the system.

2. The entire test procedure code must be loaded in the instruction cache during the loading loop. This feature brings spatial locality and it avoids instruction cache misses during the execution loop that could potentially alter the signature. This condition implies that:

   2.1) Conditional branches that could potentially produce a different execution flow in the execution loop must be avoided. Exceptions are those conditional branches that intentionally alter the execution flow but due to the effect of a fault. Moreover, this does not preclude the applicability of the proposed methodology to loop-based test programs, as long as by the end of the test program all the possible branches are taken.

2.2) The size of the multi-core version (Figure 2.2b) of the self-test procedure must fit into the available cache memory. If the resulting test program is larger than the available cache size, it must be split into two or more smaller self-test procedures. It is important to note that this step is exclusively required if the cache memory is not large enough, and it does not compromise the fault coverage of the original single-core test procedure.

3. Both data and instruction caches should be initialized, by invalidating their content (Figure 2.2b, block b) prior the test program execution (Figure 2.2b, block c and d).

The proposed strategy based on cache memories achieves the requirements of both deterministic behavior and low resources usage since:

- Caches decouple the processor from the rest of the system. Therefore, the instruction stream is not altered by other processors' activity;

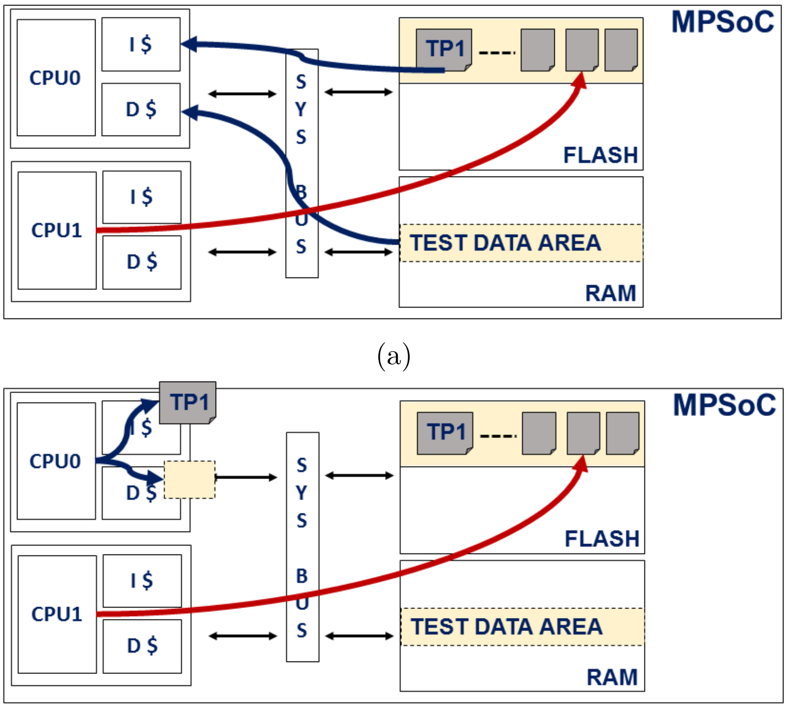
- The code is allocated in the cache memories, without altering the self-test routine memory footprint.

2.3 Experimental results

This Section is organized as follows: the first subsection describes the case study. The second, third and fourth subsections details the experimental results. These include the evidences of the issues presented in the previous sections and then the gathered results for the proposed methodology. Its effectiveness is also compared with the one of the TCM-based approach.

2.3.1 Case study and experimental setup

The device used this experimental part was an industrial heterogeneous triple-core System-on-Chip, manufactured by STMicroelectronics. The device is designed to be compliant to automotive safety-critical applications ranked as ASIL D. It embeds three dual-issue processor cores. Hereinafter, these cores will be labeled as cores A, B and C. The two cores A and B are the same 32-bit processor model, while the core C is different since it implements an extended instruction set able to deal with 64-bit operands. Each processor includes two Tightly-Coupled Memories modules (for data and instructions), along with private data (4 kB) and instruction



(a)

(b)

Figure 2.3: Example of the proposed cache-based execution. In an MPSoC two CPUs (labeled with 0 and 1) execute in parallel an STL. CPU0 executes a test program TP1 that uses the cache-based execution, while CPU1 executes another portion of the STL. In (a) is represented the loading loop. In (b) the execution loop. It is worth mentioning that normally in STLs the TEST DATA AREA is protected by access mechanisms. These are required for sake of determinism, for avoiding one core interfering with the execution of another.

(8 kB) caches. The caches support both write allocate and no-write allocate policies (configurable before being used).

For this device, two STLs were developed (core A and B share the same STL). Since the STLs are intended for latent faults (the processor cores are configured in lockstep), stuck-at faults were exclusively considered. Nevertheless, the applicability of the proposed methodology is not limited exclusively to this specific fault model. The total number of stuck-at faults of these processors varies from 643,209 (core C) to 473,052 (core B). It is worth noting that although core A and B are functionally identical they underwent different physical design processes (common practice for automotive designs). Therefore, from the testing viewpoint, they are quite different since the stuck-at fault lists are different.

For proving the effectiveness of the proposed cache-based strategy, the faults belonging to the Interrupt Control Unit and Hazard Detection Unit were exclusively considered. The self-test procedures developed for these units are significant examples of the complications that arise when considering a multi-core execution.

The problems related to the Hazard Detection Unit (which includes also the forwarding mechanism) were already presented earlier in this chapter in the Subsection 2.1.1. In the considered processors, the Hazard Detection Unit is composed of a Hazard Detection Control Unit and a Forwarding Logic. The former detects dependencies among issue packets, driving the forwarding paths and possibly stalls the pipeline if the forwarding is not possible. The latter is composed by the multiplexers that directly fed and collect the results produced by the different execution units of the processor.

The self-test program that addresses these mechanisms is based on the algorithm presented in [19]. The above-mentioned testing algorithm exhaustively test all the possible existing forwarding paths, both interpipeline (that is, dependencies between instructions of the same issue packet) and intrapipeline (dependencies between instructions of two consecutive issue packets). Moreover, it leverages performance counters for tracking the number of pipeline stalls in the processor during the self-test procedure execution (for detecting wrongly inserted stalls by the hazard control unit).

Concerning the Interrupt Control Unit, synchronous imprecise interrupts were examined. Such class of interrupts are still generated as consequence of a particular instruction being executed (i.e., synchronously) and from sources within the CPU. But, unlike precise interrupts [88], the imprecise ones are not recognized immediately, but only after that a variable number of instructions are executed beyond the interrupting instruction. The actual number of instructions depends on the instructions stream entering the pipeline, which is highly unpredictable in an MPSoC. Therefore, also the self-test procedures targeting these interrupts suffer of an unstable signature that varies depending on the other processors' activity. For testing this mechanism, a self-test procedure based on the strategy presented in [86] was implemented. The second column of Table 2.2 and the third of Table 2.3 report the number of faults within these units.

Finally, caches were configured with a write allocate policy: therefore, for both test programs, it was not required to insert additional load operations to avoid write misses in the execution loop (as explained in Section 2.2). Furthermore, for both test programs, it was not necessary to split them, since the instruction cache was large enough to contain the entire self-test procedure code.

2.3.2 Variability in MPSoCs

A first set of experiments consisted in analysing the behavior of the STL in an MPSoC. For this initial set of experiments, the test programs targeting imprecise interrupts and hazard detection unit were not included in the library and analyzed separately. The STLs were executed in parallel on the physical microcontroller. Their execution was tracked leveraging an external debugger, that monitored the number of clock cycles of stall due to the memory subsystem in each processor core. Table 2.1 reports the gathered measurements. As it can be noticed, when moving from a single-core scenario (in which all the other cores are completely turned off) to a triple-core scenario, the number of stalls in the system increased considerably. The major source of stalls is represented by the instruction fetch unit (second column of Table 2.1). This is a direct consequence of the higher bus contention: the instruction fetch operations are delayed due to the other processors requests, and as a consequence the pipeline is stalled. Moreover, it is worth noting that the figures in the second and third row of Table 2.1 represent average values gathered across several executions. The actual number of clock cycles of stall varies depending on the initial MPSoC configuration (and therefore it is not predictable).

Table 2.1: Parallel STLs Execution - Stalls (in Clock Cycles, CC) due to the Memory Subsystem

# Active Cores	**IF Stalls [CC]**	**MEM stalls [CC]**
1	200,679	117,965
2	717,538	305,801
3	1,878,336	663,386

2.3.3 Uncertain fault coverage

The experiments described in the subsection 2.3.2 confirm that the whole system activity heavily influences an STL execution, making its behavior in terms of execution time unpredictable. The second set of experiments focused on demonstrating the effects of these pipeline stalls on the self-test procedures. Specifically, these experiments involved the achievable fault coverage on the processor hazard detection unit. For performing these experiments, a simulation environment (which include also a fault simulator) was used with the SoC post-layout gate-level netlist. As extensively explained in Section 2.1.1 and demonstrated with the previous experiments (Table 2.1) Performance Counters (PCs) are unreliable in a multi-core scenario. When they contribute to the self-test procedure signature, they provoke its instability. Therefore, a straightforward solution might be removing the usage of PCs, sacrificing fault coverage. However, as depicted in Table 2.2 this is not enough for guaranteeing a deterministic fault coverage (referred as FC) of the forwarding logic. To prove these claims, the algorithm [19] was modified, removing the usage of PCs. Then, the obtained self-test procedure was executed in parallel on the different processors considering different scenarios: number of active cores (two or three), code position in code memory (low, mid and high Flash addresses) and different code alignment options (e.g., aligned at word, double-word or double double-word).

Table 2.2: Forwarding Logic Fault Simulation Results

Core	**# of Faults**	**min - max FC [%]** **no caches no PCs**	**FC [%]** **with caches no PCs**
A	53,298	64.14 - 75.19	79.61
B	57,506	63.61 - 79.59	82.08
C	113,212	56.24 - 66.48	68.79

Each of these logic simulations was then fault simulated, and the results are shown in third column of Table 2.2. As it can be observed, the fault coverage considerably oscillates: in the worst case, it was observed a difference of about 16%. It is important to note that the signature did not change during the logic simulations and yet the fault coverage varied significantly. These fluctuations depend on how many issue packets consecutively (namely in consecutive clock cycles, without any stall in between) enter the processor pipeline, activating different forwarding paths. On the contrary, when executing the self-test procedure embedded in proposed cache-based approach (fourth column of Table 2.2), the fault coverage significantly increased (about the 4% in the best case) while being stable across the different scenarios. The fault coverage obtained for core C is lower compared to the one of cores A and B because the multiplexers are 64-bit wide to support 64-bit operations. However, general purpose registers are still 32-bit wide. Therefore, the signature must be represented using 32 bits, which causes some faults effects to be masked. Nevertheless, improvements of the already existing algorithm for the forwarding logic would have been outside the scope of these experiments. For this reason, increasing further the fault coverage was not considered.

2.3.4 Unstable signature

A third set of experiments (Table 2.3) concerned Interrupt Control Unit and Hazard Detection Control Unit (ICU and HDCU respectively). For the HDCU, the complete algorithm of [19] was used (namely with performance counters). For the ICU, the aforementioned self-test procedure based on [86] was used. The fourth column of Table 2.3 represents the fault coverage figures when the self-test procedures were executed in the selected MPSoC in a single-core scenario (i.e., with the other cores switched off) without resorting to the proposed approach. In this scenario, the signatures produced by the test programs were stable as the fault coverage. However, when moving to a multi-core execution without using the caches the test procedures inevitably failed in any configuration. When introducing caches, the produced signatures become stable, and therefore the fault coverage can be computed correctly. It is worth mentioning that the achieved fault coverage in a multi-core execution is higher than in the single-core scenario. This lower fault coverage stems from the fact that the memory subsystem introduces 8 clock cycles of latency when fetching an issue packet from the Flash even in a single-core execution. Thus, it is not possible to fully excite all the forwarding paths or trigger correctly all the imprecise interrupts. Furthermore, while for the HDCU the coverage was similar over the three processors, the coverage for the ICU is about 10% higher in the core C. This arises from the implementation of the ICU itself. In details, the unit exposes some software-accessible registers for differentiating among the possible sources of interrupt. In the core A and B, different interrupt events are mapped to the same bits. As a result, even here some fault effects are masked (unlike core C).

Table 2.3: ICU and HDCU Fault Simulation Results

Core	**Module**	**# of Faults**	**FC Sigle-Core no caches [%]**	**FC Multi-Core with caches [%]**
A	ICU	14,230	46.57	51.36
	HDCU	16,096	62.53	70.37
B	ICU	13,149	46.39	50.97
	HDCU	15,783	63.84	70.12
C	ICU	13,888	54.94	60.91
	HDCU	19,931	65.66	68.09

2.3.5 Comparison with a TCM-based approach

This final section compares the cache-based approach to the TCM-based one described in Subsection 2.1.2.

Table 2.4 compares the two strategies for the self-test procedure targeting the imprecise interrupts.

Table 2.4: TCM-based versus Cache-based approaches for Imprecise Interrupts

Approach	**Overall Memory Overhead [B]**	**Execution Time [CC]**
TCM-based	2,874	16,463
Cache-based	0	18,043

Since both approaches require few additional instructions to be implemented, the flash overhead is negligible. The same reasoning applies also for the fault coverage, which does not vary. Concerning the TCM-based one, the execution time consists in the time required for copying the entire self-test procedure in the Instruction TCM and then execute from there the self-test program. For the cache-based one, it is the time required for executing as in Figure 2.2b. As it can be viewed, the cache-based approach does not increase the overall memory footprint of the self-test procedure (reported in Bytes, B), while it requires to be executed slightly more than 1,500 clock cycles compared to the TCM-based approach. These further clock cycles stem from the fact that the test program executes twice. However, it is worth noting that this overhead originates mainly from the low read access time of the flash memory and might be negligible when the STL is executed at-speed (8.25μs when the considered SoC operates at its maximum frequency of 180 MHz).

Chapter 3

Decentralized Schedulers for STLs in automotive MPSoCs

When dealing with automotive SoCs, the adopted STL is typically composed of two main software modules: one comprises the boot-time self-test programs, while the other the run-time ones. While the latter are handled by the embedded operating system, the boot-time tests execute immediately after the system concludes the power-on phase. In MPSoCs, the execution of boot-time self-test procedures is becoming a critical aspect since it might lead to longer-than-usual boot phases. Indeed, the STL must be executed on each processor core. In Section 2, the motivations and problem statement are extensively developed. Then, the most closely relevant existing works that address similar topics are discussed. Section 3 presents a set of decentralized software schedulers able to manage the concurrent execution of boot-time self-test procedures in an MPSoC. The schedulers represent a viable solution for modern automotive MPSoCs. Indeed, they guarantee a significant reduction in terms of test application time when in field, while maintaining the same fault coverage as in the single-core scenario. Simultaneously, all the schedulers yield minimum system resources usage managing the accesses to the shared resources (such as the system RAM). Finally, Section 4 reports experimental results on both dual and triple-core homogeneous/heterogeneous industrial multi-core automotive microcontrollers manufactured by STMicroelectronics.

Data and methodologies reported in this chapter were published in [35, 33].

3.1 Background

3.1.1 Problem statement

When dealing with STL for MPSoCs, a parallel testing solution is highly desirable in order to avoid situations as the one depicted in Figure 3.1. The user application might be delayed excessively due to a wrong test scheduling, decreasing the overall system availability. However, the parallel execution of an STL poses several issues. While the run-time tests (due to their design) do not constitute a problem, the same is not true for the boot-time ones.

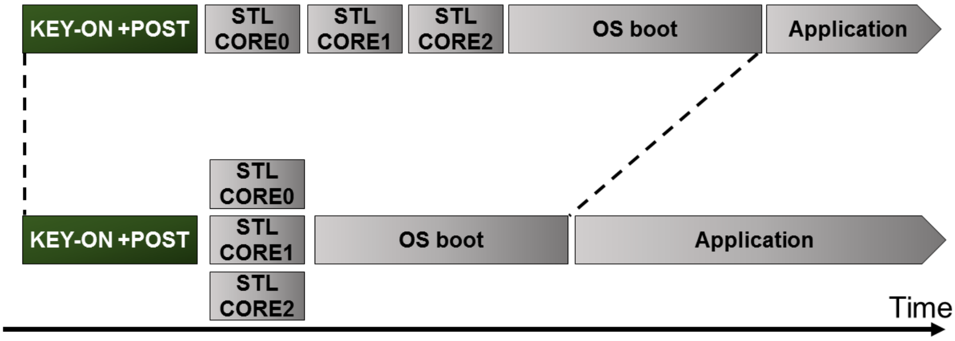


Figure 3.1: Serial versus parallel execution of an STL in a multi-core scenario.

Indeed, the boot phase of an automotive MPSoC is critical since as already mentioned in Chapter 1 the MPSoC is part of an ECU, which in turn is connected in a network to several other ECUs within a modern car. Each ECU executes a diagnostic composed of Logic and Memory BIST (Power-On Self-Test, POST in Figure 3.1) followed by the application of the boot-time portion of the STL. Reducing as much as possible the test application time of these self-test mechanisms is the major concern since the entire diagnostic must complete within few hundreds of milliseconds. In this context, one of the most important issues to be faced is related to the use of the system shared resources. Indeed, it is quite common that some of the boot-time self-test procedures require on-chip resources shared among the available processor cores. However, as shown in Figure 3.1 the boot-time STL is executed before the OS in place. This makes the adoption of already-existing methodologies difficult, since they rely on complex software structures which can be hardly implemented without an OS. In STLs, the primary shared resource is the system RAM. Usually, the RAM is divided into two main regions: the stack memory region (private for each core) and the shared memory region (in which the global variables reside). When dealing with an STL, there is a further memory region called Test Reserved Area. This memory region is available to be used by specific boot-time test programs. For sake of deterministic fault coverage, this portion of memory is fixed and allocated in the shared portion of the data memory [18]. When test programs accessing these addresses are executed concurrently by two (or more) cores, the outcome of the test becomes unpredictable. A possible solution to overcome this problem could consist in replication. Separate Test Reserved Areas can be used for each core. Then, the STL source code can be replicated multiple times in the code memory as well. However, this would increase the memory occupation by a factor of N, being N the number of cores in the system. Therefore, this is not feasible in real applications, since the memory resources in embedded systems are tailored and the space for the user application would progressively reduce as the number of cores increases. In addition, resources replication cannot be applicable to all the shared resources: indeed, system peripherals, such as the interrupt controller, cannot be replicated as there might exist exclusively one instance of a given peripheral in the entire SoC.

Consequently, it is evident that the best solution requires to resort to a suitable software scheduler able to manage the concurrent execution of boot-time tests in this scenario. Such scheduler aims at:

1. reduce the test application time when the STL is executed in parallel to comply with the ISO 26262 regulations. For achieving this, a lightweight software module that implements the scheduler is proposed;

2. maximizing the concurrency of the tests among the different cores while minimizing the system resources usage;

3. maintaining the same fault coverage of the STL as when executed in a single-core scenario;

4. not requiring additional hardware resources than the already existing ones;

5. guaranteeing a deterministic execution (i.e., predictable execution time), despite the fact that software executed in a multi-core scenario suffers a non-deterministic behavior.

3.1.2 Related works

As already discussed in Chapter 2, the embedded software executed in MPSoCs suffers of a non-deterministic behavior from the execution time view-point. This non-determinism originates from the higher system bus contention. One might erroneously think that the problem addressed in this chapter might be treated as a real-time scheduling problem. However in real-time scheduling, given a set of tasks $\Lambda = \{\lambda_0, \ldots, \lambda_m\}$ to be executed, and a set of processing units $P = \{p_0, \ldots, p_n\}$, the goal of a scheduler is to assign to each processing unit $p_j \in P$ a set of tasks $\Gamma \subseteq \Lambda$ so that each task completes by its timing deadline. When dealing with an STL, Λ corresponds to the set of self-test procedures composing the STL while P represents the different cores to be tested. The fundamental difference lies in the fact that each $\lambda_i \in \Lambda$ must be executed on each core $p_j \in P$. Therefore, in this scenario the scheduler should guarantee the execution of all the self-test procedures while avoiding conflicts due to common shared resources (e.g., the system RAM), so that the overall test execution time is minimized.

Other works that addressed STLs in a multi-core system are [87, 52, 59, 53, 54]. Yet, all of them assume an OS in which integrate a dedicated test scheduler. These solutions are more appropriate for the run-time self-test procedures but cannot be applied to boot-time self-test programs. As they execute during the system boot before the OS is running, they require a dedicated software driver. The same problems were addressed also in [6, 5, 37], although in a quite different testing scenario. In [6] a scheduling algorithm of self-test routines for shared-memory multi-processor system is proposed. The algorithm is based on an optimized usage of system caches and code replication in order to minimize the latency introduced by the memory subsystem. Instead in [5], it is shown how to exploit thread-level parallelism to improve the execution of self-test routines in each core of a multiprocessor chip. The same research group then presented in [37] an effective strategy for multi-threaded multi-core systems oriented at maximizing the execution parallelism of the self-test routines without affecting the fault coverage. All of these techniques are applicable exclusively to end-of-manufacturing testing, since it is assumed the availability of an external tester and full control over the system under test (clearly this is not the case in the in-field testing).

3.2 The boot-time tests schedulers

From the results presented in Chapter 2, it emerged that for STLs the flash memory represents the main bottleneck of an efficient parallel execution. Indeed, all the processors in the system eventually execute the same test programs, referencing the same addresses (when not using code replication). Conversely, in traditional multi-core application there is the opposite trend (i.e., distribute the workload among the different processing units). Therefore, the performances of a multi-core scheduler inevitably depend on the underlying architecture of the MPSoC (e.g., memory hierarchy, system interconnection network, available peripherals). Which means that it is hard to find a general solution that perfectly fits all the possible systems. In the following, it is assumed that the interconnection network is implemented using a crossbar switch (being widely adopted for multi-core SoCs [91]).

Concerning the memory hierarchy, it is not assumed any particular scheme. When dealing with boot-time test programs, cache memories are in most of the cases disabled, since they could alter the behavior of some test programs leading to unpredictable results (unless used with precautions as described always in Chapter

2). Moreover, cache memories are advantageous for test programs executed periodically, as is the case of the run-time programs. In contrast, boot-time test programs are executed exclusively during the power-on (and sometimes also at power-off). It is important to notice that this is not true for all the test programs executed in field. Indeed, as demonstrated in [87], the run-time ones can actually benefit from the caches.

This section is organized as follows: initially, the essential terminology used throughout the chapter is introduced. Then, the different decentralized architectures are described: starting from (for sake of a better comprehension) the description of an heterogeneous multi-resource scheduler, possible variants are analyzed (namely single-resource heterogeneous and multi-resource homogeneous).

3.2.1 Terminology and definitions

The smallest software unit considered in this paper is a Test Program. This term can be interchanged with self-test procedure or routine. From the scheduler perspective, each test program is seen as an indivisible block. This means that it cannot be interrupted (i.e., preemption not supported). A given test program might or might not require one or more shared resource for its execution.

A shared resource is an on-chip hardware resource (e.g., a peripheral, or a portion of memory) that offers some services to the test program in order to test the processor core. Considering again the system RAM, the memory is used to address fixed memory locations to better test specific portion of the processor (e.g., the address calculation unit, the branch prediction unit). In order to regulate the usage of shared resources and avoid race conditions, the proposed schedulers implement a synchronization mechanism based on semaphores.

Semaphores are one of the existing methods for achieving synchronization of software executed on different processors. A semaphore is an abstract data type (in most of the cases a shared variable) that regulates the access to a common resource by multiple processors. Depending on the number of processors allowed to access the common resource, the semaphores are distinguished in counting and binary semaphores. The latter are also called mutex, since exclusively one processor at a time can access the shared resource. When the processor is accessing the shared resource is said to be executing the code of the critical section. The basic idea is that the accesses to a shared resource are guarded by a suitable semaphore. Before entering the critical section, the processor checks the status of the semaphore. If the semaphore is available (i.e., unlocked), the processor tries to acquire the semaphore, locking the access to the shared resource. Independently from the low-level implementation, the mechanism used for checking and then acquiring the semaphore must guarantee that the two operations are executed atomically.

Considering an STL, the critical sections the mutexes are guarding correspond to the portion of code that invokes those self-test routines accessing the shared resources.

3.2.2 The common decentralized architecture

When implementing a multi-core execution of an STL, the major limitation is represented by the boot-time tests. Indeed, some of them require hardware resources which are shared by the different processors. One of the most fundamental observation of this research, is that very few test programs actually require such resources. On average, when considering an industrial STL, less than the 12% of the tests composing the STL require the usage of shared resources. The remaining tests exclusively use the the flash memory for fetching the instructions and accessing the stack region of the system RAM (private for each core). From this observation it is possible to conclude that only a minor portion of the tests cannot be executed in parallel. It means that, if one of the objectives is to improve the execution time, it is necessary to exploit better the parallelism offered by a multi-core system. Following this reasoning, it is possible to conceptually imagine a multi-core SoC as a distributed system, in which different modules communicate via a shared medium (i.e., the system bus). In the field of distributed systems, a popular scheduling approach is based on decentralized schedulers [24]. These solutions are valuable because they overcome the limitations of centralized approaches. In the latter, there is a high degree of control over the scheduling process, since all the requests are processed by a unique scheduler. Although it is conceptually easier to reason and design according to this paradigm, the drawback is that the scheduling is not so efficient in terms of performances. Instead, decentralized schedulers represent a more efficient solution, since they have an intrinsic distributed nature which takes full advantage of the underlying system. Typically, they are composed of a set of local schedulers (built upon a first come, first served policy) interacting each other. Therefore, there is less control over the scheduling itself. The proposed architecture for the scheduler falls into this category: specifically for an STL, each scheduler is local to each processor core. During the in-field execution, each local scheduler independently executes the test programs. The interactions with other schedulers are made when accessing a shared resource through a mutex (signalling whether the resource is busy or free). Additionally, the devised decentralized schedulers implement the *selfish* heuristic which is essential for providing deterministic execution time. Thus, they are named Decentralized Selfish Schedulers. Such heuristic, described more in detail in the following, was derived experimentally from the observation of the behavior of several industrial STLs.

3.2.3 Multi-resource heterogeneous scheduler

Heterogeneous SoCs are characterized by having N processor cores of T different types. Let us label each processor type with τ, $\tau \in [0, T)$. Each type τ might differ under several aspects: Instruction Set Architecture, micro-architecture and others. Therefore, each processor core type has a dedicated STL_τ, specifically devised for its features. Test programs composing a given STL_τ might or might not access at most Υ shared resources: each shared resource is denoted with υ, $\upsilon \in [0, \Upsilon)$.

Thus, since each local scheduler is in charge for the execution of the STL on the processor it targets, in this scenario there are T different Decentralized Selfish Schedulers (hereinafter each scheduler will be referred with DSS_τ) images loaded into the flash memory. The overall instances of DSSs are always equal to the number of processor cores N. It is worth noting that T also represents the minimum number of STLs allocated in the flash memory. In the following, it is described how the proposed solution does not require more than T different STLs.

Ideally, when dealing with shared resources, the most immediate solution is to serialize the access to the resource with a mutex. This solution works well when considering a single shared resource. However, it might become inefficient when considering multiple resources. Indeed, the system bus (when it is based on a crossbar switch) of a multi-core system normally allows multiple masters to access different slaves, given that there are not any conflicts. Namely, if a processor core requests to access a given resource (e.g., a portion of memory), it is allowed to access without being blocked by the activity of other processor cores (if it is the only one requesting that resource). Yet, as mentioned earlier, the flash memory represents the major issue to address when considering multi-core STL schedulers. Actually, the flash memory can be thought as a further shared resource, since only a limited number of reads can be performed in parallel. Although conceptually the same reasoning is applicable also to the RAM memory, it has been observed in the experiments in the next sections and in Chapter 2 that the impact of the flash memory is considerably higher than the one of the RAM. This stems from the fact that the vast majority of the test programs uses the system RAM most of the time for accessing the stack region. Furthermore, the amount of time spent in accessing the stack is limited compared to the overall activity of a generic test procedure. As an example, considering the test programs of the STLs used, less than the 2% of the total execution time is spent in stack-related operations (mainly for saving and restoring the context). For the remaining time, the computations are performed internally to the processor core.

Therefore, when considering a multi-resource scheduler, the maximum number of read ports available on the on-chip flash becomes a crucial aspect in order to achieve the best performances. Indeed, if there are enough read ports for each processor core, then it is possible for the test programs to access the shared resources in parallel. Otherwise, as it is detailed next in the experimental section, the overhead introduced by the flash memory worsens the performances.

Under the assumption of having enough read ports on the flash memory, each DSS_τ requires globally an array of Υ mutexes (one for each shared resource). Since the mutexes are the only way for the schedulers to communicate each others, they are allocated in the shared portion of the system RAM so that they are accessible by any DSS. Instead, locally (i.e., for each instance of the DSS) $\Upsilon + 2$ internal data structures are required: one Test Table, one Pending List and Υ Share Resource sets.

Considering a generic STL$_\tau$: the Test Table is an ordered set Λ_τ, containing the list of test programs (the i-th test program is denoted with $\lambda_{\tau,i}$). It defines the execution order for the test programs composing a given STL$_\tau$. It is important to note that the order specified in Λ_τ is identical for each processor core type τ.

The test programs of the STL$_\tau$ that cannot be executed in parallel due to conflicting accesses to a system shared resource υ are also present in the respective Share Resource set $\delta_{\tau,\upsilon}$. In general, for a given core τ there could be at most Υ Share Resource sets. The actual number of shared resources (and thus Share Resource sets) used by the test programs of a processor STL$_\tau$ is Υ_τ, $\Upsilon_\tau \in [0, \Upsilon)$. Let us denote with Δ_τ the class formed by the union of the *Share Resource* sets, namely:

$$\Delta_\tau = \bigcup_{\upsilon=0}^{\Upsilon_\tau} \delta_{\tau,\upsilon} \subseteq \Lambda_\tau$$

The sets composing Δ_τ are assumed to be disjointed, i.e., a given test program can use only one shared resource. This assumption should not be considered as restrictive, since the scheduling logic is oriented to promote the most frequent case. Indeed, from the observation of several industrial STLs this is actually the most common situation.

In the following, it is assumed that Λ_τ is ordered according to the selfish heuristic introduced beforehand. According to this heuristic, the test programs composing Δ_τ are the first ones in the Λ_τ sequence. The order of the different $\delta_{\tau,\upsilon}$ within Δ_τ does not affect the performances of the scheduler. The actual number of test programs executed is tracked by a different set, the Pending List Φ_τ. Differently than in Λ_τ, the order of the elements composing Φ_τ and the various $\delta_{\tau,\upsilon}$ is irrelevant.

Given these definitions, each processor core invokes one of the T different DSS$_\tau$ (whose pseudo-code is listed in Algorithm 1) depending on the processor type τ. Initially, $|\Phi_\tau| = |\Lambda_\tau|$.

The scheduler sequentially selects one test program λ_τ at a time from Test Table Λ_τ (coherently with the specified order, line 6), and it checks whether the selected test program is still present in Pending List Φ_τ (that is, not yet executed, see line 7). If so and at the same time the selected test program does not belong to any of the Υ_τ Share Resource sets (line 8), it can be executed. Once executed (line 24), it is also removed from the Pending List (line 25), reducing the cardinality of the set. Contrarily, if the test program is also present in one of the Share Resource sets (line 8), the test program can be executed given that the shared resource υ is not busy. This is achieved through a suitable mutex (line 10), which signals to each local scheduler whether another scheduler is currently executing another test program belonging to the same Share Resource set $\delta_{\tau,\upsilon}$. Therefore, the critical section corresponds to the portion of code executing tests belonging to Δ_τ (lines 10 to 18). If the resource υ is not free, the test program is skipped. Instead of performing busy waiting (i.e., waiting for the resource υ to be freed), the scheduler tries to execute another test program (line 20). If the resource is free, the test program is executed (line 11) and removed from Pending List (line 12). The STL execution completes when there are no more test programs to be executed, that is the pending list is empty (line 4).

As it can be noticed, the scheduler does not release the mutex (lines 14 to 18) as long as the next sequential test program still uses the same shared resource υ. This is the second feature of the selfish heuristic. In practice, the first local scheduler that acquires the mutex for a given shared resource υ, executes *consecutively* all the test programs within the same Shared Resource set $\delta_{\tau,\upsilon}$. It is noteworthy that this is done without freeing the shared resource υ. The experimental observations leading to this heuristic are discussed in the next section. The intent was the reduction of the number of conflicts (namely clock cycles of stalls) during the parallel execution of the library. Indeed, the faster software programs in a multi-core scenario are those with fewer number of clock cycles of stalls [71].

Actually, not releasing the mutex along with the Test Table ordering mentioned at the beginning assumes a crucial aspect for achieving determinism and efficiency. At the same time, it considerably reduces the clock cycles of stalls while executing the STL.

The scheduler falls into the category of the non-preemptive schedulers, as test programs are not interrupted during their execution. Instead, a given test program can be skipped. This means that the proposed scheduler does not alter the fault coverage of the boot-time tests. As stated in [13], as long as boot-time tests are not interrupted the fault coverage is not altered.

It is important to notice that, from the moment the test begins, each core is completely independent. This is one of the main characteristics of decentralized schedulers, which avoids complex centralized control mechanisms that would downgrade the overall efficiency. Moreover, in multi-core SoC such complex control mechanisms would be further counterproductive due to the high non-determinism of the system itself and the fact that the test is performed autonomously in field (without external testing facilities). Actually, the proposed decentralized scheduler is highly scalable while guaranteeing minimum resources overhead. This property stems from the fact that each instance of DSS_τ exclusively uses the private stack memory of each core (making the local scheduler itself reentrant). In embedded systems, the size of the stack region is determined in advance and it is large enough to accommodate the requests of the different functions invoked during the system lifetime. The most important aspect is that once a given function terminates its execution and returns to the caller, the stack memory reserved for that function is deallocated.

Algorithm 1: Heterogeneous Multi-resource DSS

```
// $\Lambda_\tau$ is the Test Table
// $\Phi_\tau$ is the Pending List
// $\delta_{\tau,\upsilon}$ is a generic Share Resource set
// $\Delta_\tau$ is the union of Share Resource sets
Function $DSS_\tau$ () is
    $\Phi_\tau \leftarrow \Lambda_\tau$;
    $MaintainMutex_\tau[\Upsilon_\tau] \leftarrow$ false;
    while $|\Phi_\tau| \neq |\emptyset|$ do
        $i \leftarrow 0$;
        for $\lambda_{\tau,i} \in \Lambda_\tau$ do
            if $\lambda_{\tau,i} \in \Phi_\tau$ then
                if $\lambda_{\tau,i} \in \Delta_\tau$ then
                    $\upsilon \leftarrow$ GetSharedResIndex($\lambda_{\tau,i}$);
                    if Acquire(Mutex[$\upsilon$]) is successful $\vee$ $MaintainMutex_\tau[\upsilon]$ is
                      true then
                        Execute $\lambda_{\tau,i}$;
                        $\Phi_\tau \leftarrow \Phi_\tau \setminus \{\lambda_{\tau,i}\}$;
                        $i \leftarrow i + 1$;
                        if $\lambda_{\tau,i} \in \delta_{\tau,\upsilon}$ then
                            $MaintainMutex_\tau[\upsilon] \leftarrow$ true;
                        else
                            $MaintainMutex_\tau[\upsilon] \leftarrow$ false;
                            Release(Mutex[$\upsilon$]);
                        end
                    else
                        $i \leftarrow i + 1$;
                    end
                else
                    Execute $\lambda_{\tau,i}$;
                    $\Phi_\tau \leftarrow \Phi_\tau \setminus \{\lambda_{\tau,i}\}$;
                    $i \leftarrow i + 1$;
                end
            end
        end
    end
    return;
end
```

This means that all the data structures required by each local scheduler no longer exists after the scheduler completes. The only data enduring after the end of the test are the global variables for implementing the mutexes, which requires a limited portion of the global memory region only. Finally, the scheduling solution presented above requires exactly T STLs memory images in the flash memory. The very same number of STLs is required for a single-core execution (i.e., each core not tested in parallel).

In the following subsection, the single-resource multi-core scheduler is first addressed. This case is particularly interesting since (as it is demonstrated in the experimental section) it can be useful even for implementing a multi-resource scheduler. When the on-chip flash memory has not enough read ports, the overhead generated by the parallel access to shared resources becomes unsustainable. Hence, it is worth to resort to a single-resource approach. The remaining cases for homogeneous systems are discussed as they can be derived from the heterogeneous one above by relieving some parameters.

3.2.4 Single-resource heterogeneous scheduler

A single-resource multi-core heterogeneous scheduler can be derived from the one in Algorithm 1. In this case, as there is a unique shared resource, the number of shared resources is $\Upsilon = 1$. Given that, there is also a unique shared resource set:

$$\Delta_\tau = \delta_{\tau,0} \subseteq \Lambda_\tau$$

Thus, all the vectored data variables (i.e., $\mathit{MaintainMutex}_\tau$ and *Mutex*) become a vector of one element (namely, scalar data variables). As already anticipated, this single-resource scheduler might be used also when dealing with multiple shared resources. In this case, all the shared resources present in the system can be ideally grouped in a single resource. It means that, even though the resources are physically separated, they are treated logically as a unique resource. The rationale is the same used for deriving the selfish heuristic: those test programs that generate more activity on the system bus are executed on only one core at a time. As demonstrated in the experimental section, in some cases this yield significant improvements when there are not enough read ports in the flash memory.

3.2.5 Multi/Single-resource homogeneous scheduler

Unlike the heterogeneous ones, the homogeneous SoCs have N processor cores but all of them belong to the same type ($T = 1$). The multi-resource scheduler is derived from Algorithm 1 by imposing $\tau = 0$. In this case, all the N processors execute the same STL_0 by invoking the same scheduler DDS_0. The same reasoning concerning the multi and single-resource scenario is also applicable to these systems (i.e., number of read ports on the flash memory). Finally, when $T = 1$ and $\Upsilon = 1$ the Algorithm 1 becomes a single-resource homogeneous scheduler.

3.2.6 Summary

Throughout this section different decentralized schedulers for STLs were introduced. The functional safety engineers can select the most appropriate scheduling solution depending on:

1. SoC type (i.e., heterogeneous or homogeneous);

2. the number of system shared resources accessed by the STLs;

3. the level of parallelism offered by the flash memory (i.e., number of read ports).

It is worth noting that the proposed solutions allow for a seamlessly integration of multi-resource and single-resource schedulers together. Indeed, as each data structure required by the scheduler is private, they do not influence each others. In practical terms, STLs with different number of accessed shared resources are fully supported.

For concluding this section, Figure 3.2 depicts a practical example.

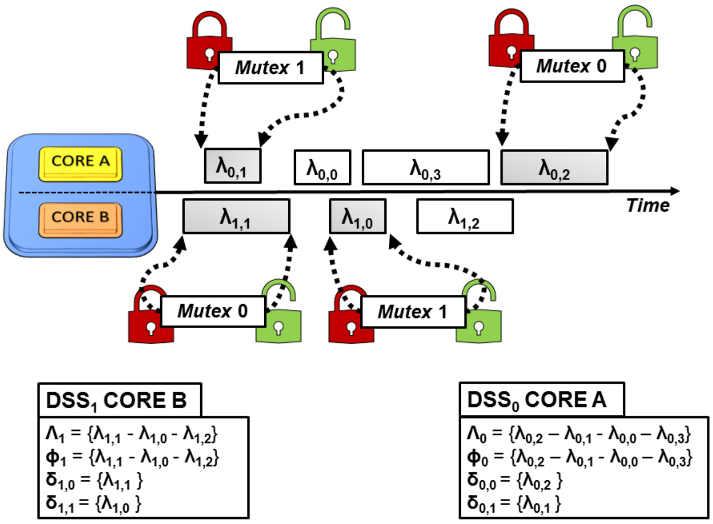

Mutex 1
Mutex 0
CORE A
CORE B
$\lambda_{0,1}$
$\lambda_{0,0}$
$\lambda_{0,3}$
$\lambda_{0,2}$
Time
$\lambda_{1,1}$
$\lambda_{1,0}$
$\lambda_{1,2}$
Mutex 0
Mutex 1
DSS$_1$ CORE B
$\Lambda_1 = \{\lambda_{1,1} - \lambda_{1,0} - \lambda_{1,2}\}$
$\Phi_1 = \{\lambda_{1,1} - \lambda_{1,0} - \lambda_{1,2}\}$
$\delta_{1,0} = \{\lambda_{1,1}\}$
$\delta_{1,1} = \{\lambda_{1,0}\}$
DSS$_0$ CORE A
$\Lambda_0 = \{\lambda_{0,2} - \lambda_{0,1} - \lambda_{0,0} - \lambda_{0,3}\}$
$\Phi_0 = \{\lambda_{0,2} - \lambda_{0,1} - \lambda_{0,0} - \lambda_{0,3}\}$
$\delta_{0,0} = \{\lambda_{0,2}\}$
$\delta_{0,1} = \{\lambda_{0,1}\}$

Figure 3.2: Temporal evolution of the execution of the two heterogeneous schedulers across the processor cores. The test programs in light gray indicates those accessing a shared resource.

Figure 3.2 considers a simplified scenario of an heterogeneous multi-core system with two cores (CORE A and CORE B) and with a dual-port flash memory. Thus, it is possible to use the scheduler discussed in subsection 3.2.3. As there are two different processor cores $T = 2$, CORE A is of type 0 (i.e., $\tau = 0$) and CORE B is of type 1 (i.e., $\tau = 1$). For CORE A the STL_0 (managed by the scheduler DDS_0) is composed by four test programs $(\lambda_{0,0}, \ldots, \lambda_{0,3})$. Whereas, for CORE B, the STL_1 (managed by the scheduler DDS_1) is composed by three test programs $(\lambda_{1,0}, \lambda_{1,1}, \lambda_{1,2})$. Test programs belonging to both STLs access two shared resources, thus $\Upsilon = \Upsilon_0 = \Upsilon_1 = 2$ (guarded by Mutex 0 and 1 respectively). For each STL_τ there are two shared resource sets $\delta_{\tau,0}$ and $\delta_{\tau,1}$. For the STL_0, $\lambda_{0,2}$ accesses the shared resource $\upsilon = 0$, while $\lambda_{0,1}$ the shared resource with $\upsilon = 1$. Instead, for the STL_1 the two programs are $\lambda_{1,1}$ and $\lambda_{1,0}$ respectively. Given these characteristics, the required data structure for each scheduler are filled as depicted in Figure 3.2. It is noteworthy that the test table Λ_0 and Λ_1 are ordered so that the test programs accessing the shared resource 0 are the first ones. Then, the test programs accessing the shared resource 1 followed by those not requiring any shared resource. At the beginning, both mutexes are free. Then, assuming that CORE B is the first one able to lock the Mutex 0, it can execute $\lambda_{1,1}$. At this point the other CORE A is not able to execute $\lambda_{0,2}$ (Mutex 0 is still locked) which is eventually executed later. Hence, it tries to lock Mutex 1 for executing $\lambda_{0,1}$ (the second program in its test table). As the shared resource 1 is not used by any processor core, the Mutex 1 is free and successfully locked by CORE A. After $\lambda_{0,1}$ completes, the Mutex 1 is released. Concurrently, CORE B frees the Mutex 0 since it is no longer required. CORE A proceeds with the sequential order specified in its test table executing those test programs not requiring any shared resource. CORE B does the same by locking successfully Mutex 1 and executing $\lambda_{1,0}$. Once this test program terminates, CORE B begins the execution of the last test. On the other hand, CORE A before completing has still a test program not executed, namely $\lambda_{0,2}$, which was previously skipped. Therefore, before terminating it locks Mutex 0 (now free) and it executes its last test program. After the execution of this test program, both schedulers have their Pending Lists empty. Thus, the execution can proceed with the normal boot sequence.

3.3 Experimental results

This section is organized as follows: initially, the two industrial designs (and its STLs) adopted for the experiments are described. Then the experiments considering the single-resource multi-core scheduler for homogeneous MPSoCs are reported. It is also reported the experimental evidences behind the selfish heuristic. In the successive subsection, the single-resource multi-core scheduler for heterogeneous MPSoCs is validated. Finally, the multi-resource scheduler for both homogeneous and heterogeneous MPSoC is discussed.

3.3.1 Case study and experimental setup

The two devices used to prove experimentally the effectiveness of the proposed schedulers are two triple-core (each core labeled with A, B and C) SoCs manufactured by STMicroelectronics, designed to meet automotive applications ranked as ASIL D. Hereinafter, these two designs are named D1 and D2. The former is an heterogeneous SoC, while the latter is homogeneous. The high-level architecture of these two devices is depicted in Figure 3.3. As it can be viewed, the global system interconnect is based on an AMBA® interconnect implemented as a crossbar switch, that connects the masters to the slaves (i.e., memories and peripherals). In case two masters compete for the same slave, the arbitration policy is the classical round robin. Nevertheless, the crossbar can be optionally configured to support schemes based on fixed priority (for hard real-time tasks). For both devices the memory hierarchy is composed as follows (from the innermost up to the outermost level): the closest (to each processor core) level of memories is represented by tightly-coupled SRAMs and private instruction and data caches. Then, there are the system RAM (608 KBytes for D1, 128 KBytes for D2) and the Flash memory (6 MBytes for both). The flash memory is organized so that there are two different read ports. Moreover, in the experiments, each core was allowed to use exactly no more than 2KBytes of system RAM as private stack memory. Focusing on D1, core A and B implement the same 32-bit instruction set, while core C implements an extended instruction set able to deal with 64-bit operands. For this device, the operating frequency ranges from 16MHz up to a maximum frequency of 180MHz.

For the design D2, being homogeneous, each processor core implements the same instruction set with 64-bit operands and additional instructions to implement signal processing features. Concerning the operating frequency, the minimum is 16MHz (as for D1), while the maximum is 200MHz.

Given these two devices, three STLs are required (two for D1, the other for D2). The most relevant characteristics of these STLs are reported in Table 3.1. All the three STLs require two shared resources: the system RAM and the Interrupt Controller (INTC). The latter is a clear example in which replication is not feasible, since each design has only one INTC. Such resource handles the external interrupts since the internal interrupts (i.e., exceptions) are handled by specific modules within each processor cores. Therefore, it is used by a specific test program that tests the capability of the processor core to react to external interrupts. For both D1 and D2 the execution time has been derived by using an operating frequency of 16MHz. This frequency was used also for all the performed experiments and gathered directly on the manufactured devices exploiting the available on-chip hardware timers. In general, the term Execution Time includes both initialization phase and the boot-time test phase. During the former, the processor internal

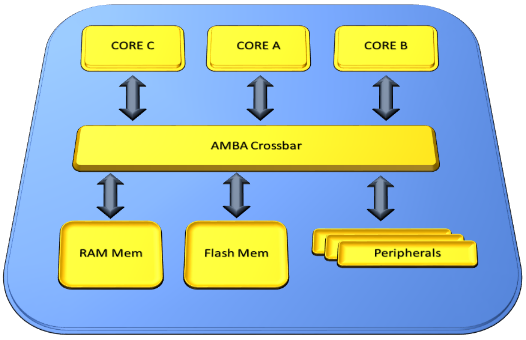


Figure 3.3: Internal architecture of the considered multi-core System-on-Chips.

state is initialized for the test (namely registers, interrupt service routines). In the latter, the STL is actually executed. In order to obtain the highest possible fault coverage, it is common practice to execute the entire library at the device startup (i.e., both boot-time and run-time tests). Then, the run-time tests are executed periodically when the system is on-line. To obtain a realistic use case of an STL in the automotive field, the STLs were configured in this way.

Table 3.1: Single-core characteristics of the three STLs for the designs D1 and D2.

Design	Target Core	Ex. Time @16MHz[ms]	Flash Occupation [KB]	# Boot-time Tests			# Run-time Tests	# Tot Tests
				Not sharing resources	Sharing resources			
					RAM	INTC		
D1	A, B	18.34	377	41	16	1	25	83
	C	24.1	403	58	17	1	47	123
D2	A, B, C	29.7	430	57	13	1	34	105

Moreover, when considering the multi-core scenario, the following setup was used:

1. to reproduce the worst scenario from the contention point of view, the measurements were gathered aligning the execution of the processor cores;

2. during the dual-core experiments, the third core was completely switched off for avoiding influencing the outcome of the measurement itself;

3. the execution time is computed from the moment all the processor cores are aligned to the moment the last processor core completes the test.

For the sake of experiments reproducibility, the following setup was used, avoiding features that would be too specific for the considered cases: for the crossbar, the default arbitration was adopted (round robin). The tightly-coupled SRAMs were not used, as they are not always available. Also the caches were disabled and possibly activated exclusively by those test programs that actually require them.

3.3.2 Single-resource homogeneous multi-core scheduler: serial scheduling

In the following exclusively the design D2 is considered. Since the analysis is for single-resource schedulers, the self-test program requiring the INTC is excluded (it is discussed later in the chapter when considering the multi-resource scheduler).

A preliminary set of experiments consisted in measuring the execution time of the STL when using a serial scheduler (i.e., the STL is executed serially on each core). These measurements are used as a reference point, being the upper bound of any multi-core scheduler. The measurements are shown in Table 3.2 and were gathered considering the STL executed on 2 and 3 cores.

Table 3.2: Performances of the serial scheduler @16MHz for D2

# Active Cores	Execution Time [ms]
2	57.15
3	88.01

3.3.3 Single-resource homogeneous multi-core scheduler: non-selfish decentralized schedulers

To prove that the proposed decentralized scheduler is valid, this set of experiments focused on analyzing the performances of different decentralized scheduling algorithms. They differ from the proposed one since they cannot be considered selfish (that is, the mutex is released even though the next sequential self-test procedures still belong to the Share Resource set). The considered schedulers are listed in Table 3.3.

Table 3.3: Performances of different decentralized schedulers @16MHz for D2

Decentralized Scheduler	Execution Time [ms]	
	2 Active Cores	3 Active Cores
DS1	40.18	71.93
DS2	49.31	79.89
DS3	39.87	66.80
DS4	41.12	71.42
DS5	41.81	79.80

Each of these decentralized schedulers considers different formats of Test Table: in DS1 the order of the self-test procedures is random. A total of 30 random orders were generated, and DS1 represents the best random ordering. It is worth to underline that the differences between the generated random orders were minimal.

Table 3.4: Performance counters values for the triple-core scenario for D2

Decentralized Scheduler	**Flash Memory Stalls [CC]**	**System RAM Stalls [CC]**
DS1	1,878,336	663,386
DS2	1,932,409	791,922
DS3	1,589,729	478,264
DS4	1,738,412	525,011
DS5	1,929,209	788,668

Therefore, for sake of conciseness, exclusively 30 random orders were generated and the best order is reported. DS2 orders the self-test procedures so that those included in Share Resource are executed first. The order of the self-test procedures within Share Resource is random. Differently, in DS3 the self-test procedures within Share Resource are ordered according to the duration of the self-test procedures themselves (with a descending order). For both DS2 and DS3 the remaining self-test procedures (i.e., those not included in Share Resource) are ordered randomly. Finally, DS4 and DS5 considers the self-test procedures still ordered according to their duration (descending and ascending order respectively), but independently from the fact that they could also belong to the Share Resource set. As it can be viewed in Table 3.3, DS3 is the decentralized scheduler yielding the best performances. This is justified by the fact that it is the scheduler that better reduces the memory access contention, as shown in Table 3.4.

Table 3.4 reports the values of the on-chip performance counters for the triple-core scenario (being the worst case from the access contention viewpoint). The first column reports the schedulers names. In the second column, the number of clock cycles corresponding to stalls due to the access contention for the Flash memory is reported. The third column shows the clock cycles corresponding to stalls due to access contention for the system RAM. It is worth noting that any multi-core scheduler is limited by the shared memory architecture: as an example, considering exclusively the Flash memory, when moving from a single-core implementation to a multi-core one, the total number of clock cycles stalls increased from 200,679 to 1,589,729 (with the DS3 scheduler). Hence, as confirmed by the values present in the second column of Table 3.4, the Flash memory represents the real bottleneck. The scheduler DS3 may seem the most promising one, since it reduces substantially the execution time in both dual-core and triple-core scenarios. However, further experiments showed that it could hardly be used in an industrial context. Although it meets some of the requirements (minimum system resources usage), it suffers from a non-deterministic execution time when the number of tests composing the Share Resource set varies.

The results of the third set of experiments are shown in the charts of Figures 3.4 and 3.5, for the dual and triple-core scenarios, respectively. In both cases, it was increased progressively the number of self-test procedures composing the Share Resource set, that is the set of procedures requiring the use of a shared resource. This was done adding one self-test procedure at a time, which was not originally part of the Share Resource set. It is noteworthy that this does not mean altering the total number of programs composing the STL.

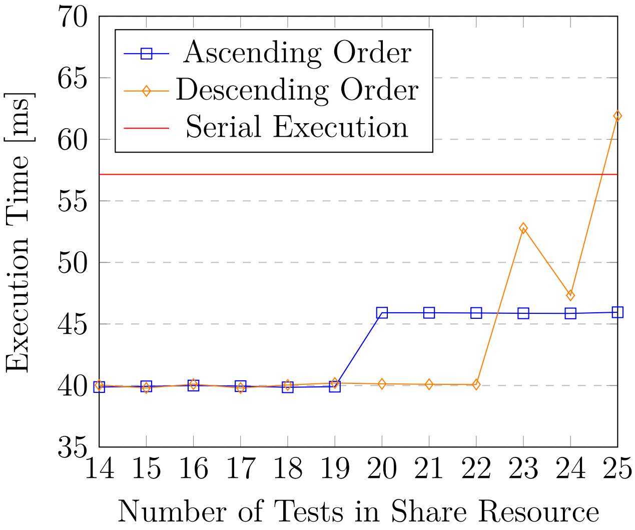


Figure 3.4: For design D2, the DS3 execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the Share Resource set in a dual-core scenario.

For both figures, the orange line represents the execution time when increasing the size of the Share Resource set, inserting in a descending order the self-test procedures starting with the longest test programs in terms of duration (their duration is in the range 10,000 to 23,000 clock cycles). The blue one instead represents the behavior of the scheduler when increasing the size of the set, inserting in an ascending order the self-test procedures starting with the shortest test programs (duration ranging from 500 to 1,000 clock cycles). The red line represents the threshold imposed by the serial execution. Since the purpose is to show the indeterminacy of these approaches, it was decided not to consider more than 25 self-test procedure composing Share Resource.

It can be noticed comparing the two charts that the execution time is not predictable and it presents non-negligible oscillations. By monitoring the STL execution on each processor core using an external debugger, it was observed a higher memory access contention. In particular, it emerged that these fluctuations are caused by the alternated execution of the test programs composing the Share Resource set with test programs not belonging to this set. Clearly, these fluctuations

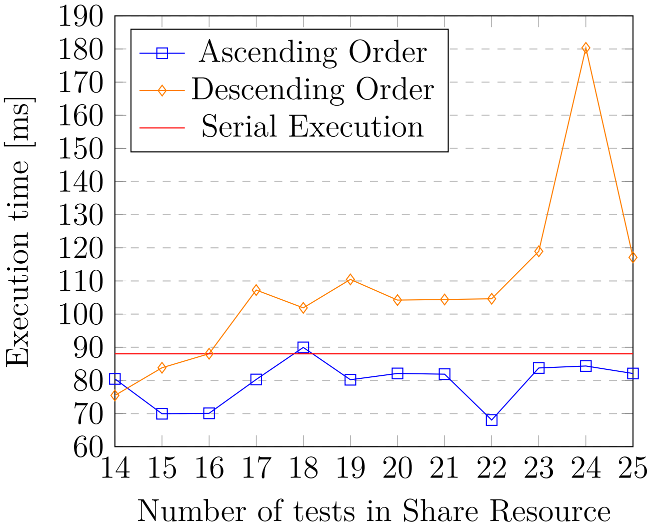


Figure 3.5: For design D2, the DS3 execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the Share Resource set in a triple-core scenario.

are not acceptable. Indeed, it is worth mentioning that industrial STLs are designed to be configurable: the final customer (i.e., the system company) decides whether some self-test routines are required or not. As an example, if the floating-point unit is not used, the customer may decide to not execute the self-test procedures targeting that unit.

Considering Figure 3.5, an example of fluctuations is when moving from 19 to 18 self-test procedures in the Share Resource set. This may well be the typical scenario in which a customer, using the STL, decides to disable one test program. With 19 tests the execution time is below the threshold of the serial execution, but when reducing the tests the execution time actually increases above the serial execution.

3.3.4 Single-resource homogeneous multi-core scheduler: the Decentralized Selfish Scheduler

The experiments of this subsection involves the proposed decentralized selfish scheduler described in the previous section. From the results presented in Table 3.4, it appears that DS3 outperforms the other schedulers since it reduces the memory access contention. The reason for this reduction (confirmed again by monitoring the schedulers execution) originates from the fact that if the longest tests are executed first, it is likely that they will keep the shared resource busy for a considerable amount of time. This forces the other local schedulers to execute the self-test procedures not included in Share Resource and then wait for the shared resource to be freed. This significantly reduces the bus activity, but as the experiments of Figure 3.4 and 3.5 confirmed, it depends on the actual duration of the self-test procedures composing the Share Resource set.

The proposed DSS enforces this condition with the devised order for Test Table and maintaining the resource busy until all the test programs in Share Resource are executed. The latter in particular avoids the alternated execution mentioned above, that causes the oscillations present in the scheduler DS3. Table 3.5 reports the comparisons among the serial scheduler, the DS3 scheduler and the proposed selfish scheduler when executing the STL with the original number of test programs in the Share Resource set (namely 13).

Table 3.5: Performances of the Decentralized Selfish Scheduler @16MHz for D2

# Active Cores	**Execution Time [ms]**		
	Serial Scheduler	**DS3**	**DSS**
2	57.15	39.87	38.27
3	88.01	66.80	57.18

Table 3.6: Overhead of the Decentralized Selfish Scheduler for D2

Overhead	**Single-Core STL**	**Triple-Core STL**
Memory Footprint [KB]	429	489
Execution Time [ms]	29.01	29.51

As it can be observed by comparing the second and the fourth column of Table 3.5, the proposed solution reduces considerably the execution time of about 33% and 35% (for the dual-core and triple-core scenarios respectively). This is significant, since especially in the triple-core scenario the execution time improves with respect to DS3 of about the 14% and it is comparable with a serial execution but in a dual-core scenario. It is worth noting that the order of the self-test procedures within Share Resource is now irrelevant: the test programs within Share Resource are executed as an unique block.

Clearly, having multiple copies of the STL in the code memory would be beneficial for any multi-core scheduler. However, this is normally not possible when dealing with in-field test of embedded systems since this means a flash memory occupation two to three times higher than in a single-core scenario. As an example, considering the STL under analysis, the single-core version of the library occupies 429 KBytes while the triple-core version is 489 KBytes. Therefore, having independent copies of the single-core version is not acceptable since it leads to an excessive memory usage. On the other hand, the overhead from a timing point of view of the proposed scheduler is also modest, since it accounts for about $0.5ms$. Table 3.6 summarizes the main characteristics of the proposed scheduler from a timing and memory footprint viewpoint for the triple-core scenario (being the worst possible in the considered case study).

The same experiments performed with DS3 were repeated and the results are shown in Figure 3.6 and 3.7 for the dual and triple-core scenario, respectively. The behavior illustrated in Figure 3.6 and 3.7 is now much more predictable compared to the charts depicted in Figure 3.4 and 3.5. Furthermore, it can be seen that the execution time in the dual-core scenario (Figure 3.6) is always lower than the serial execution, unlike the behavior of DS3 (Figure 3.4). It is important to note that in the triple-core scenario (Figure 3.7), the orange line after 21 self-test procedures in Share Resource crosses the red line. In more practical terms, it means that the execution time of the decentralized scheduler exceeded the serial scheduler. However, this represents an exaggerated case since the added test programs have a duration between 23,000 and 10,000 clock cycles (which is quite unrealistic in practical applications).

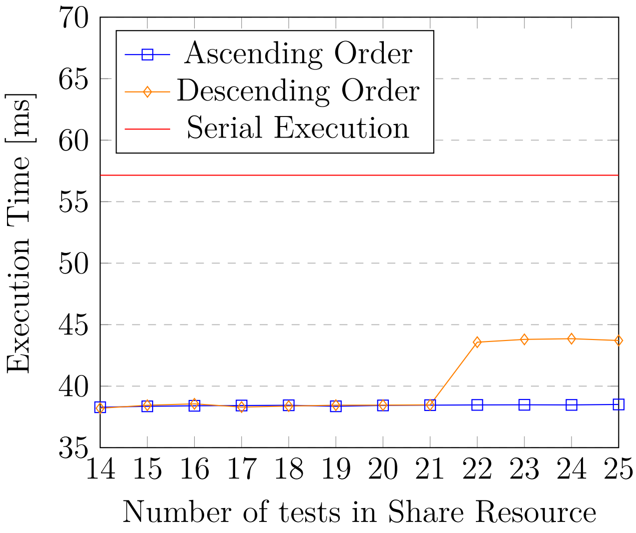


Figure 3.6: For design D2, the proposed DSS execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the Share Resource set in a dual-core scenario.

In order to asses the maximum achievable performances of the proposed scheduler, a further set of experiments focused on increasing the size of the Share Resource set, including progressively self-test procedures from Test Table not present originally in Share Resource. As in the experiments described in Figure 3.4, 3.5, 3.6 and 3.7, the self-test procedures were included starting from the shortest ones (in terms of duration) to the longest ones. However, the substantial difference with respect to the previous experiments is the fact that the aim is to increase the size of the Share Resource set as much as possible, well beyond the 25 self-test procedures of the aforementioned experiments. Figure 3.8 depicts the results of the

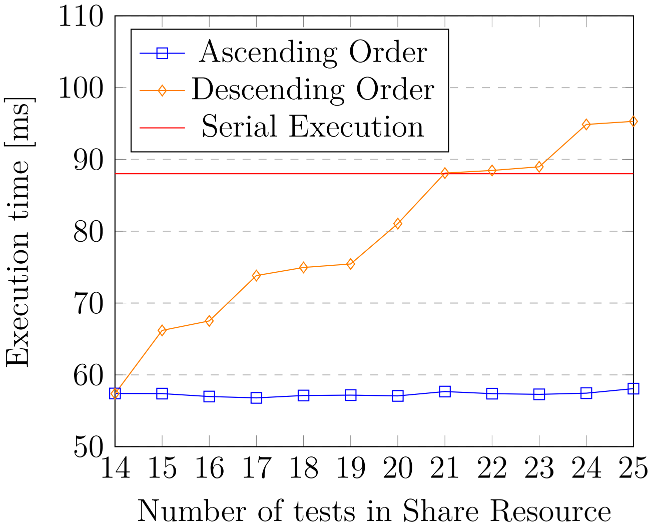


Figure 3.7: For design D2, the proposed DSS execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the Share Resource set in a triple-core scenario.

experiments for two and three cores.

For sake of generality, it is more convenient to express the results of Figure 3.8 using as x-axis the percent ratio between the duration of the Share Resource set and the total duration of the boot-time tests. It is important to underline that, in the considered STL, the tests labeled as boot-time are 70 out of 104. Therefore, 70 is also the maximum number of tests that can be included in the Share Resource set since the remaining 34 are run-time tests that are always present and by definition cannot be included in the Share Resource set. Figure 3.8 shows that up to a duration equal to 67% of the total duration of the boot-time tests, the execution time of the proposed decentralized scheduler is lower compared to a serial scheduler (for both dual-core and triple-core scenarios). It is noteworthy that a 67% figure corresponds to include in the Share Resource set 65 out of 70 self-test procedures. This means that it was possible to execute the vast majority of the test programs that can be labeled as boot-time tests. For completeness, the measurements corresponding to 90% and 100% of the boot-time duration were also gathered. In this case the execution time exceeded the one of the serial scheduler, since the last 5 test programs are the longest that can be included (each one requiring more than 15,000 clock cycles to execute). However, it is uncommon having such long programs accessing the system RAM for testing purpose. Typically, only few test programs require a shared portion of the system RAM for test purposes. Therefore, when considering a reasonable percent ratio of duration of the Share Resource set (namely 30-50%), the performances of the proposed decentralized scheduler are always superior compared to a serial scheduler.

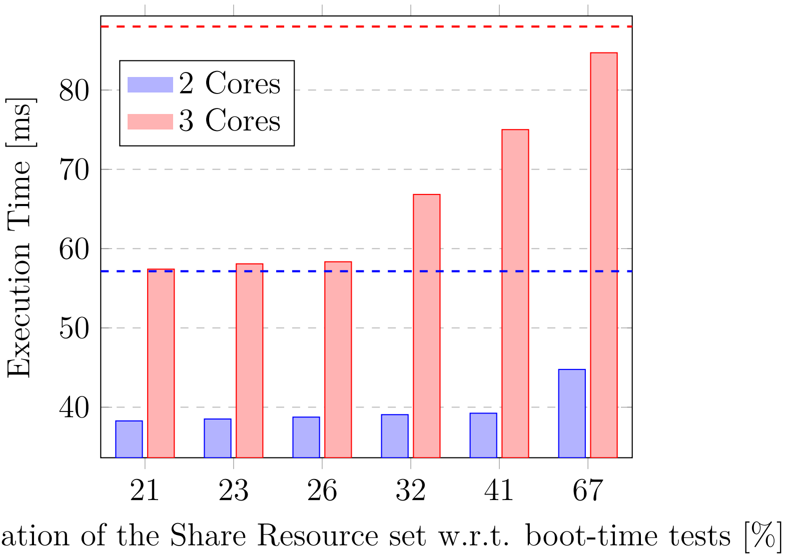


Figure 3.8: For design D2, the proposed DSS execution time (y-axis) at 16MHz when increasing the duration (x-axis) of the Share Resource set. The blue and red dotted lines represent the serial scheduler for dual-core and triple-core scenarios, respectively.

By comparing the results shown in Figure 3.6, 3.7 and 3.8 it can be observed that as the size of the Share Resource set increases, the execution time of the decentralized scheduler degrades faster in a triple-core scenario than in the dual-core. This depends mainly from the fact that three active processors generate considerably more activity in the system bus than two processors (taking into account also the parallelism of the flash memory).

3.3.5 Single-resource multi-core heterogeneous scheduler: the serial scheduler

The set of experiments described in the following considered the heterogeneous design D1. Again, the test programs using the INTC were momentarily excluded from the STLs since the focus is a single-resource scenario. Given this setup, the obtained results when using a serial scheduler are summarized in Table 3.7. The first two rows report the single-core execution time of the two libraries when executed without the test using the INTC. It is worth noting that the first row indicates the single-core execution of both core A and B (since they are of the same type, they execute they same STL).

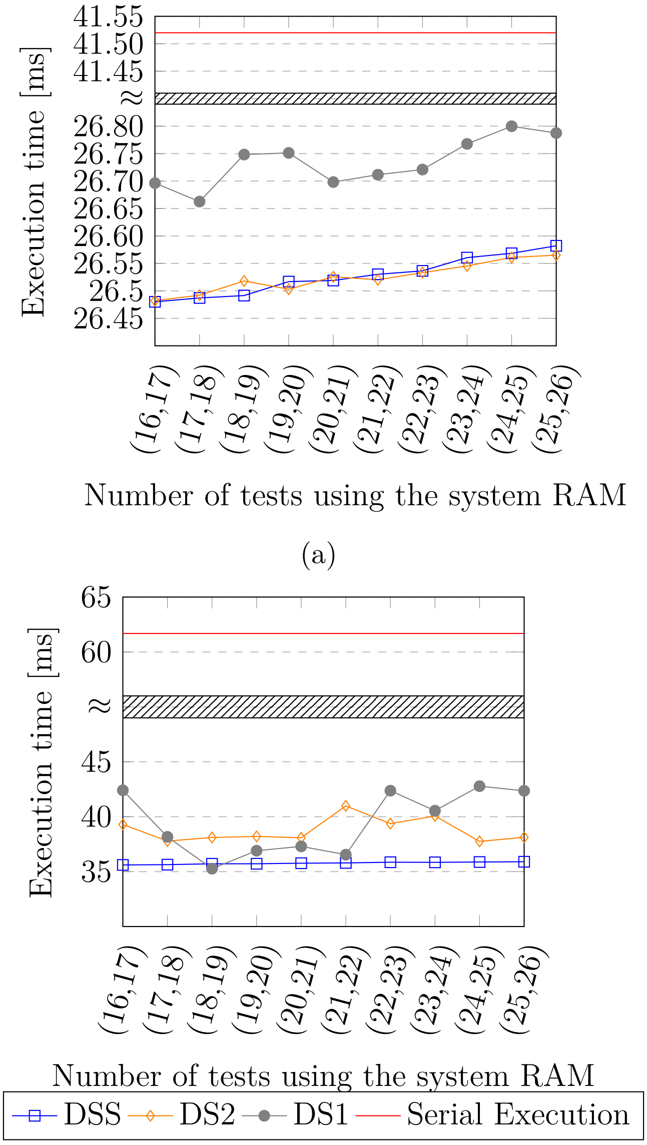
Table 3.7: The single-resource serial scheduler @16MHz for D1.

Active Cores	Execution Time [ms]
A (or B)	17.5
C	23.63
A (or B), C	41.52
A, B, C	61.68

3.3.6 Single-resource multi-core heterogeneous scheduler: the Decentralized Selfish Scheduler

To prove that the decentralized selfish scheduler is valid also in the heterogeneous scenario, the experiments described in the following focused on analyzing the performances of different decentralized schedulers. They differ from the proposed one since they cannot be considered selfish. For sake of conciseness, the two schedulers that achieve similar performances when compared to the proposed one were exclusively considered. Hereinafter, these schedulers are referred as DS1 and DS2. The proposed one is still named DSS. The scheduler DS1 is a non-selfish decentralized scheduler, and its test table is ordered randomly. Instead, DS2 maintains the same ordering as DSS, but it still does not implement the selfish heuristic. Although these two schedulers represent a valid alternative to the proposed one, again they all suffer of a non-deterministic execution time when the number of test programs using the shared resource varies. In these experiments multi-core configurations with 2 and 3 active cores were considered. For both scenarios, similarly to what was done in previous experiments, it was increased progressively (considering both a ascending and descending order) the number of test programs needing the shared resource (in this case, only the system RAM) of both STLs. Figures 3.9 and 3.10 depict the evolution of the four scheduling alternatives (the three decentralized with the serial scheduler) in the four aforementioned configurations (i.e, dual/triple-core and ascending/descending). The y-axis reports the execution time in milliseconds (always measured at 16MHz), while the x-axis the number of test programs using the system RAM in the two STLs (thus, represented by a pair of values). The dual-core scenario is analyzed in Figures 3.9a and 3.10a (ascending and descending order respectively), while the triple-core in 3.9b, 3.10b. When considering a dual-core execution (Figure 3.9a, 3.10a), all the three decentralized schedulers represent a valid solution as they considerably reduce the execution time compared to a serial approach. Notably, on average there was observed a execution time reduction with respect to a serial approach of the 36% when considering the ascending order, while a 28% with the descending order.

However, the situation radically changes when dealing with a triple-core system (Figure 3.9b, 3.10b). Concerning the ascending order in Figure 3.9b, there was observed a similar trend to the cases described beforehand with an execution time

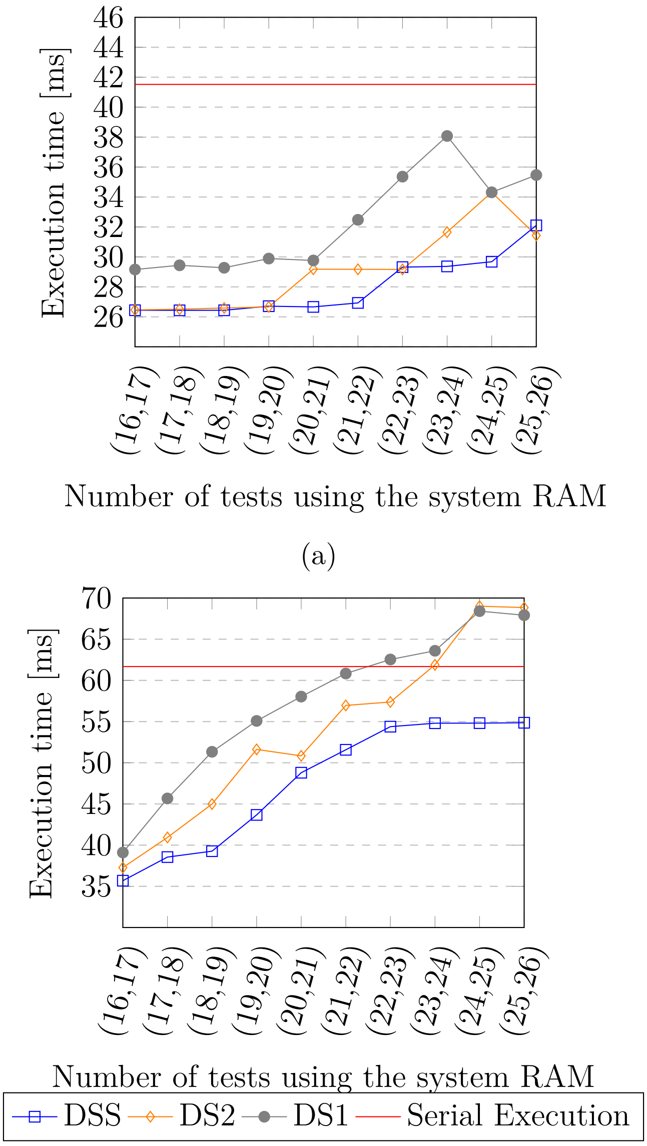


(a)

(b)

Figure 3.9: For D1, evolution of the execution time at 16MHz of different single-resource decentralized schedulers with respect to the serial scheduler when considering an ascending order. The dual-core scenario is shown in (a), the triple-core one in (b).

reduction of the 38%. Nevertheless, the same is not true when considering larger test programs (i.e., the descending order of Figure 3.10b). Coherently with the previous experiments, a parallel (i.e., decentralized) execution does not always yield an improvement in the execution time as one would expect. Specifically, the two non-selfish schedulers DS2 and DS1 eventually exceed the threshold imposed by



(a)

(b)

Figure 3.10: For D1, evolution of the execution time at 16MHz of different single-resource decentralized schedulers with respect to the serial scheduler when considering a descending order. The dual-core scenario is shown in (a), the triple-core one in (b).

the serial scheduler (in the worst case by 12%). Also in this scenario, the proposed one (DSS) always yield better performances than the other considered solutions. The only exception is Figure 3.9a in which DSS and DS2 have approximately the same behavior. Additionally, the DSS remains always below the threshold given by the serial solution, represented by the red straight line. Finally, also in these experiments the proposed selfish scheduler exhibits a predictable (i.e., deterministic) behavior compared to other decentralized solutions.

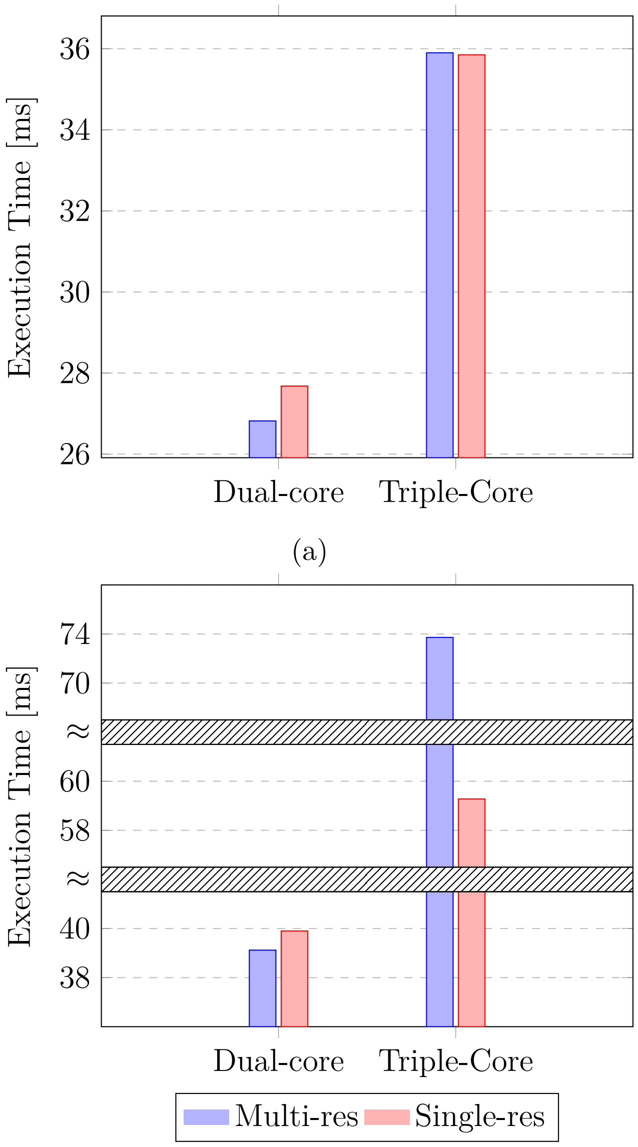
3.3.7 Multi-resource multi-core scheduler

In this last set of experiments, the full set of test programs were considered for both designs. The most relevant characteristics of the multi-resource scheduler in terms of memory footprint are reported in Table 3.8. It is important to notice that the figures reported in Table 3.8 are indicative as they might vary depending on the actual implementation. In this chapter, the scheduler was implemented in C, while the synchronization primitives were written in assembly. The data structures described in required by the scheduler were implemented as static arrays, since dynamic memory allocation is not allowed in safety-critical applications. As it can be noticed, the overhead of the shared portion of system RAM is negligible. Actually, most of the data structures resides in the stack memory region. The only data required by the scheduler to be allocated in shared portion of memory are the mutexes, which can be easily encoded in 1 byte each. Concerning the stack overhead, this has to be intended as the additional amount of stack memory required for each processor core with respect to a single-core STL. However, it is worth noting once again that when the test completes, these regions are completely de-allocated and can be fully used by the user application. Concerning the flash overhead, compared to a single-core implementation or serial implementation, each scheduler accounts for about 64KBytes. Considering D1, the fourth row indicates cumulative overhead of the two schedulers in memory. As the scheduler is written in C, the overhead of a single scheduler does not change with the design. Although, it is also important to say that in case of D1 the two schedulers share some parts of the code (e.g., synchronization primitives and other utility functions). As a result, the final overhead is less than the sum of the two libraries.

Table 3.8: Multi-resource scheduler overhead for D1 and D2.

Target Design	D1		D2
Target Core	A, B	C	A, B, C
Stack Overhead [B]	249	369	315
Flash Occupation [KB]	752		494
Shared RAM Overhead [B]	2		2

Concerning the performances, using a multi-resource scheduler does not always produce the desired results. Because of the limited parallelism of the flash memory, in some cases the single-resource selfish scheduler should be preferred. In order to use a single-resource scheduler, all the shared resources are logically grouped into



(a)

(b)

Figure 3.11: Multi-resource scenario: schedulers performances in dual and triple-core configuration. Picture (a) reports the figures for the design D1, whereas (b) for D2. The light blue bars represent the multi-resource scheduler. The red ones represent the single-resource alternative, with the shared resources logically grouped into a unique shared resource. In (b) the y-axis was split in different ranges to allow a better observation.

a single shared resource. Figure 3.11 shows the performances of the multi-resource scheduler against the single-resource one used for multiple shared resources. For both designs, the dual and triple-core configurations are reported. As it can be seen, when considering a dual-core configuration, the multi-resource scheduler performs better than the single-resource one. Indeed, the same trend was observed in both D1 and D2. This is due to the parallelism of the flash memory, which is able to sustain the concurrent access of the two active cores. The improvement of a multi-resource scheduler is not so evident because in the selected case study there exist exclusively two shared resources, which represents the most common situation that is found in modern industrial STLs. Moreover, the INTC is used by one test program only which limits the maximum achievable improvement. When the number of active cores increases, the multi-resource tend to be outperformed by the single-resource scheduler. This is shown in homogeneous designs as D2 (Figure 3.11b), when compared to heterogeneous D1 (Figure 3.11a). In D2 the system bus activity is higher because the three cores reference exactly the same addresses. As a consequence, they provoke more conflicts when accessing the memory subsystem.

3.3.8 Final Considerations

One important assumption of the proposed decentralized selfish schedulers is the fact that all the share resource sets $\delta_{\tau,\upsilon}$ are disjointed. In practical terms, a given test program can access exclusively one shared resource. It should not be considered a limitation, as test programs are devised to test a precise portion of the processor core. Notwithstanding, in case of a test program accessing two shared resources, the following solutions might be adopted:

1. If applicable, the test engineer should consider to split the test programs in two or more sub-programs;
2. Exclude the test programs from the STL, which is executed in parallel with the most appropriate scheduler. The excluded test programs are then executed serially on each core. By doing so, most of the library is executed in parallel, while only a few test programs are not.

In any case, the basic idea behind the effectiveness of these decentralized selfish schedulers is to make the common and most frequent case faster. Therefore, the proposed architecture was preferred to those general solutions (that adds complexity to the software) that accommodates all the possible existing scenarios but inevitably downgrade the achievable performances.

Additionally, according to the development flow presented in [13], test programs are independently developed and then fault graded. Their cumulative effect is later considered for computing the final fault coverage figures. As it is extensively discussed in the above-mentioned paper and also in Chapter 2, adopting the described development flow reduces the computational effort during the fault simulation process. Furthermore, another positive side effect stems from the fact that the computed fault coverage does not depend on the actual test program order. In particular, as long as the boot-time tests are not interrupted during their execution, the fault coverage is not altered. Therefore, since the proposed decentralized scheduler does not preempt the self-test procedures, the fault coverage is not altered.

Chapter 4

Hybrid on-line self-test mechanism for comparators of a DCLS processor

The Dual-Core Lockstep (DCLS) configuration is largely employed in safety-critical MPSoCs for the sake of compliance with ISO26262 to detect single-point faults. Such configuration includes two processor cores paired together, always fed with the same identical inputs and their outputs are continuously compared by a set of comparators. However, permanent latent faults affecting the comparators may invalidate the system functionalities, thus in-field self-test mechanisms are mandatory. This chapter is organized in three main section. In the background section, motivations, related works and other possible self-test solutions (with advantages and limitations) are discussed. Then, a hybrid hardware-software scheme for the on-line self-test of the lockstep logic is proposed. Such a solution leverages self-test programs developed according to the Software-Based Self-Test (SBST) approach, used in conjunction with a specialized hardware module. Finally in the third section the effectiveness of this approach was assessed on a modified version of the OpenRISC 1200 processor. Exhaustive experiments demonstrated that it is possible to achieve a fault coverage of stuck-at faults greater than 99%, while at the same time significantly reduce the area overhead of the hardware approaches.

This chapter is based on the results presented in [36, 31].

4.1 Background

4.1.1 Problem statement

The DCLS is the de-facto solution against single-point faults in processor cores of automotive MPSoCs when targeting the highest ASILs (namely C and D). Briefly, a system including DCLS consists of two identical processor cores, both initialized to the same initial state and fed with identical inputs. As a consequence, identical outputs should always be produced. A logic failure (due to permanent or transient faults) reaching the output in one of the two cores can be detected by continuously comparing their output. Once a failure is detected, the system reacts depending on the application requirements. One of the two cores is named the main core, which interacts with the SoC. The other is the checker core. The only purpose of the checker core is to confirm the correctness of the main core outputs, being fed with the very same instructions and data of the main core.

While extremely efficient to detect single-point faults, The DCLS configuration described above cannot detect failures that occur at the same point in both cores. These failures are normally called common mode failures, which cause comparators to produce a false match. A common technique for reducing such risk of failure is to provide temporal diversity to the two cores composing the system. This strategy consists of delaying the inputs fed to the checker core by means of a bank of shift registers. The outputs must be re-synchronized before being compared. This is achieved by delaying the main core outputs by the same amount of clock cycles of the checker core inputs. It is worth noting that the whole architecture is completely transparent from the application code perspective: indeed, the checker core does not have any direct access to the system resources.

The overall DCLS architecture is shown in Figure 4.1. For the sake of better comprehension, all the bits belonging to the same signal are grouped in a unique comparator (e.g., DATA RAM wires). The control signals (i.e., the system bus interface) are also grouped in a single comparator (CTRL CMP). All the comparators are then organized in a cluster, whom output is comparators' outputs OR-ed each other's.

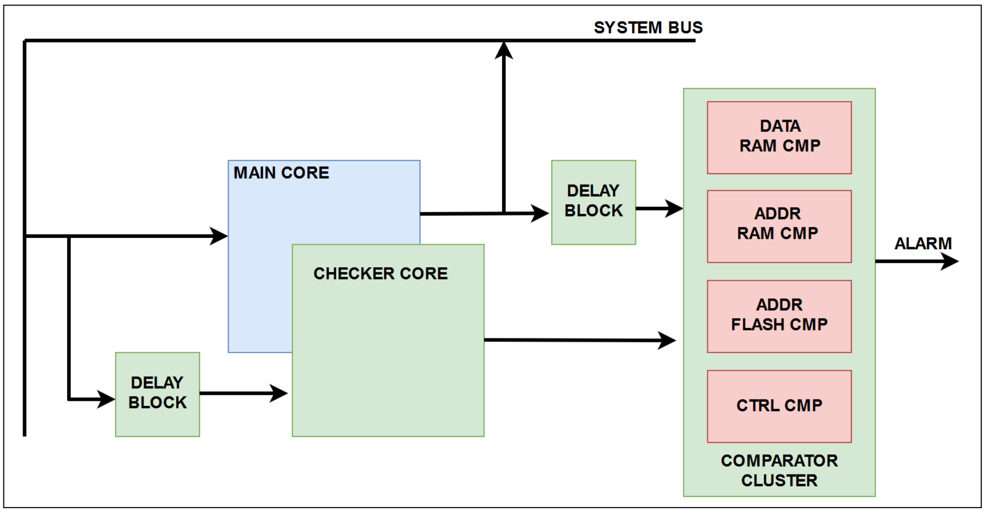


Figure 4.1: Typical delayed DCLS architecture.

However, this safety mechanism is exposed to latent faults accumulation. These must be detected during the power-on and during the on-line phase. STLs are adopted to mitigate this risk in both main and checker core at run time. It is important to highlight that the software approaches can produce pure functional stimuli, only. However, from the lockstep comparators standpoint, it means that some critical faults cannot be addressed with the support of this method only. Indeed, some latent faults might escape the test. Specifically permanent hardware faults in the comparators (Figure 4.2) could either cause a false alarm or an undetected critical failure. In the former, the ALARM signal is fired even though the two inputs match, while in the latter the signal is not fired even though the two inputs differ. Clearly, a false alarm is positive since it means that the hardware fault is detected. Instead, the second effect is potentially dangerous since a failure of the main core is not reported correctly, inhibiting the lockstep functionalities. Therefore, it is evident that in order to obtain complete system dependability, it is necessary to devise a suitable self-test mechanism for latent faults to be applied when on-line. While most of the latent faults are detected during the POST with the application of the LBIST, when on-line usually a specific circuitry is added to the comparators for implementing the self-test. This additional hardware has the penalty of additional system area to be devoted exclusively for test purposes.

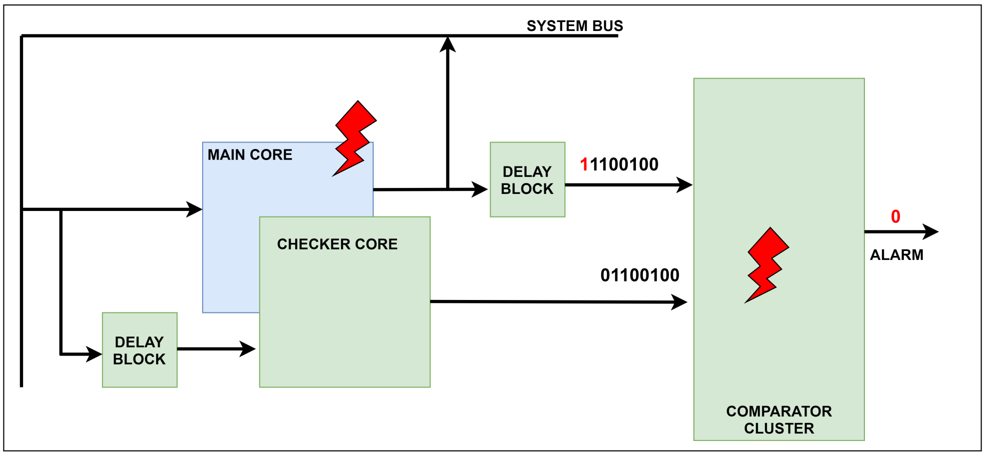


Figure 4.2: Example of latent fault causing a failure of the main core being masked.

4.1.2 Limitations of hardware and software self-test mechanisms

The basic elements composing a DCLS system are the two processor cores and a set of comparators. Authors of [44] already proposed an effective test strategy for the comparators. To fully test an m-bit wide comparator (i.e., two m-bit wide inputs), 2m+2 test patterns are required. As stated by the authors, the effectiveness of such patterns is independent on the low-level implementation of the comparators.

The 2m patterns generate a mismatch in only one bit at a time, and in practice this correspond to a walking 0 (or 1), starting from the MSB (or LSB) up to the LSB (or MSB). The last two patterns correspond to the case in which the two comparator inputs are equal, namely both inputs with all bits at 1 and then at 0. An example of the test patterns applied by this algorithm to a set of 4-bit comparators is shown in Table 4.1.

Table 4.1: Test algorithm for a 4-bit wide comparator

# pattern	**Input A**	**Input B**
1	0111	1111
2	1011	1111
3	1101	1111
4	1110	1111
5	1111	0111
6	1111	1011
7	1111	1101
8	1111	1110
9	1111	1111
10	0000	0000

Intuitively, when considering a pure software approach (e.g., execution of self-test procedures belonging to an STL), the aforementioned test stimuli can be hardly generated in the system under analysis. As the reported experimental results confirm, the structure of a DCLS configuration imposes a constraint on the generation of those test patterns. In particular, both the checker and the main cores are fed with the very same inputs and always produce the same outputs (unless a fault is present). As a consequence, it is not possible to create the difference needed by the 2m patterns. Moreover, a further challenge stems from the fact that the DCLS comparators do not only check for the integrity of data, but also all the processors' outputs. These include several control signals and addresses. Therefore, the previous algorithm must be translated in a proper sequence of instructions that force the processor to generate those values (which is not feasible with a pure software approach). On the other hand, a pure hardware self-test (e.g., [1]) guarantees the completeness of the test in terms of generated test patterns. Nevertheless, it suffers the problem of excessive additional hardware devoted exclusively for testing purposes.

4.1.3 Related works

Hybrid solutions for self-testing purposes are not new. In [15], a memory core storing an SBST-like test sequence is inserted in the system and connected to the system bus. During the test phase, the processor is forced to execute the instructions provided by this module. Also the authors of [47] proposed a solution which consists of a software-hardware-cooperated BIST as an attempt to reduce the test time for DRAM memories. Another hybrid solution that leverages the already-existing on-chip programmable resources was presented in [11]. That approach suggests the usage of the embedded DMA for RAM memory testing. It is also worth to mention that hybrid approaches have been extensively adopted also as hardening mechanism against transient faults [22, 12].

In [90] additional instructions are added to the ISA for testing purposes only (e.g., accessing to particular flip-flops within the processor), whereas in [72], observation points are inserted within the processor for increasing the effectiveness of self-test programs. As an attempt to mitigate the cost and performances penalties introduced by the safety mechanisms, in [64] the authors proposed a cooperation between hardware and software modules. In order to achieve the targeted safety level for a given SoC: for the main computational units (e.g., the CPU), they suggest the usage of hardware-based application-independent safety mechanisms. For the remaining of the system (e.g., peripherals) a combination of hardware-software mechanisms (application-specific).

4.2 Proposed hybrid self-test mechanism

4.2.1 The overall architecture

During the in-field test of the DCLS, a pure functional-based approach is only able to produce 2 out of the required 2m+2 test patterns, reaching in this way an insufficient fault coverage. Thus, in order to overcome this problem, the proposed test strategy is composed of two main elements: a set of self-test programs and a hardware module. The latter, called the Lockstep Self-test Management Unit (LSMU), supports the self-test programs during their execution. Figure 4.3 depicts the overall architecture.

The LSMU is composed of two parts, a control unit (CU) and a datapath (DP). The CU includes the bus interface logic, an FSM and a set of registers. The DP is made of a comparator, the Instruction Substitution Module (ISM) and the Control Signals Substitution Module (CSSM). The LSMU is directly connected to the system bus and it exclusively intercepts all the instructions that the checker core receives as input. At each clock cycle, it monitors whether a particular instruction (hereinafter target instruction) is going to be fed to the checker core. Whenever the instruction going to be fed to the checker core is the target instruction, the ISM replaces such instruction with a so-called substituted instruction. Similarly, the LSMU receives as input the output control signals of both the checker and main cores. When the CSSM is active and the target instruction is fed to the cores, the

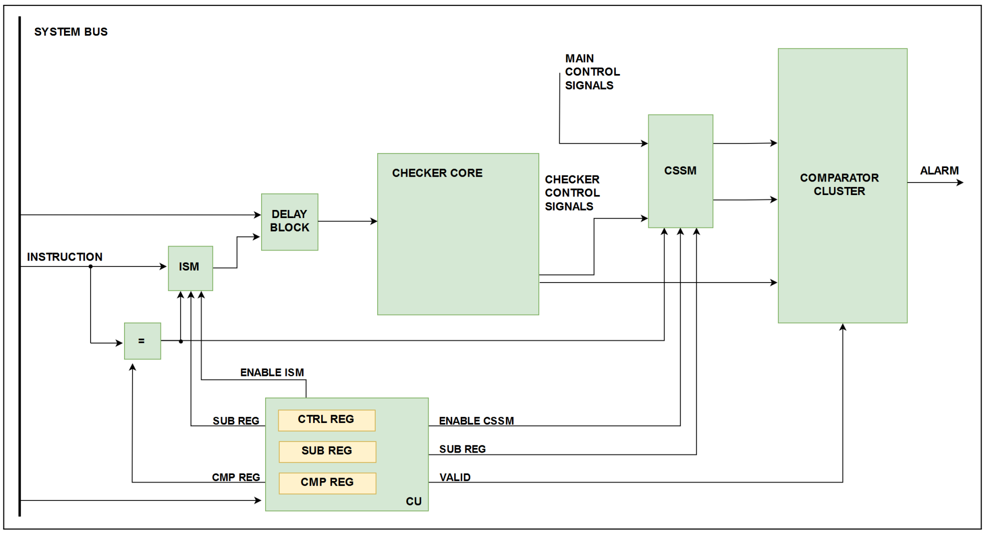


Figure 4.3: Architecture of the system including the LSMU. For simplicity, the main core is not depicted in this picture.

value of the control signals is substituted with a specific test pattern intended for testing the control comparators. As it is demonstrated later in this section, while for address and data comparators the ISM is enough for generating the required test patterns, the CSSM is required for control comparators since controlling the value of control signals is not always feasible via software. Relying exclusively on the ISM would yield a lower fault coverage, since the control comparators would not be correctly targeted. Moreover, it is worth noting that the output control signals are substituted right before being fed to the comparators. Thus, the SoC is transparent to this substitution, since the real output directed to the system (namely the main core outputs) are left unchanged.

Instructions, test patterns and other functionalities can be programmed by the main core via a set of registers at run time, as a standard memory-mapped peripheral. As mentioned before, testing the comparators requires to create a difference in just one of the bits fed to the DCLS comparators. This is not always possible since normally the comparators are grouped into a single cluster and the execution of different instructions by the two cores could lead to different control signals activated contemporary. More than one comparator active at the same time may cause masking problems, since their outputs are OR-ed together. For this reason, during the test phase, the main core can program the LSMU to disable the comparators that should not be tested (VALID signals in Figure 4.3) during the current self-test session so that they do not influence the targeted comparator. For example, let us assume that both the main and the checker cores execute a store byte but accessing to different addresses. As a consequence, a different byte should be selected which means that different control signals are activated, hence two different comparators are fired (CTRL CMP and ADDR RAM CMP in Figure 4.1).

For implementing the behavior described above, three registers are required:

- COMPARE REGISTER (CMP REG): containing the target instruction. The instruction written in this register must be encoded as in the program memory;
- SUBSTITUTE REGISTER (SUB REG): containing the substituted instruction or the test pattern to be applied to the control comparators;
- CONTROL REGISTER (CTRL REG): this register drives the behavior of the FSM and DCLS comparators to be disabled.

The bit-width of the first register depends on the considered processor architecture (i.e., 32/64-bit architecture), the second one should have a bit-width at least equal to the CMP REG (for fitting the substituted instruction). Possibly, if the number of control signals is higher, additional bits should be comprised in this register. The CTRL REG depends on the number of comparators. The first three bits of CTRL REG are dedicated to the substitution mode. To provide as much flexibility as possible, without bounding the self-test programs to any particular implementation, two substitution modes are provided:

- ONE-SHOT SUBSTITUTION: it substitutes the target instruction only once, and then the LSMU disables itself;
- CONTINUOUS SUBSTITUTION: the ISM continuously substitutes the target instruction, until the LSMU is disabled by the main core.

The remaining bits are dedicated to the TEST MODE, which allows disabling the comparators. The fourth bit specifies whether TEST MODE should be entered or not, while there are as many bits as the number of comparators to be disabled. During the normal operational phase, all the comparators are enabled. If the TEST MODE bit is set, then only those comparators for which the corresponding bit is set are enabled. Finally, one bit should be reserved for enabling the ISM or the CSSM. As an example, let us consider again the architecture of Figure 4.1. Given such architecture (once again, all the control signals are grouped in a single comparator), the CTRL REG has the structure shown in Figure 4.4. In the aforementioned figure, it is assumed that the ADDRESS FLASH comparators are enabled when the test mode is entered. Thus, it is not necessary to have a further dedicated bit within the CTRL REG.

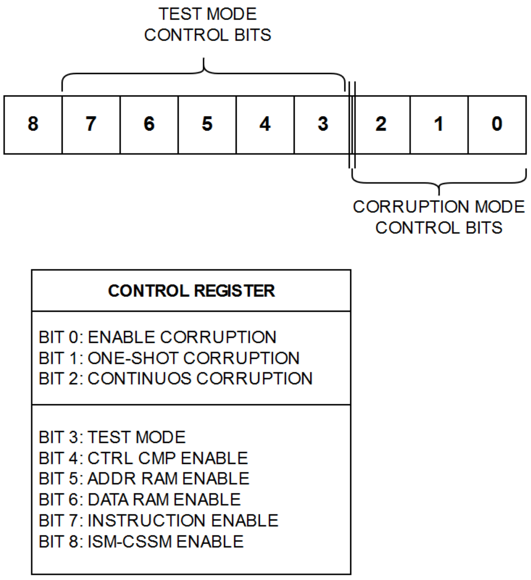
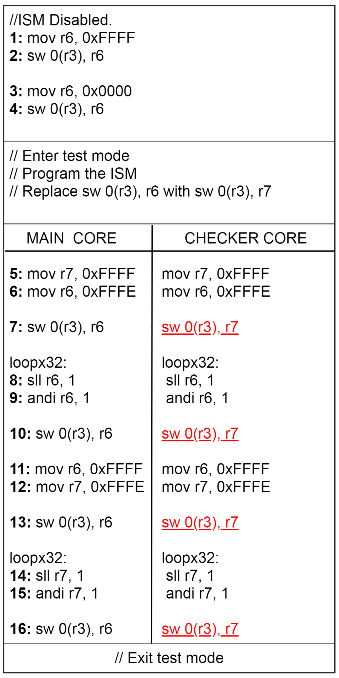


Figure 4.4: Control register (CTRL REG) description. The ADDRESS FLASH CMP are automatically enabled when the TEST MODE bit is set.

4.2.2 The on-line self-test flow

In order to test the DCLS comparators, it is necessary to execute a suitable self-test program while enabling the LSMU, providing in this way, the desired stimuli to the targeted hardware. Let us consider firs the case in which the test of DATA RAM comparator (each input is 32-bit wide) is performed. The self-test program structure is the one reported in Figure 4.5. For sake of compactness, it was assumed that the mov instructions support the sign-extension (as it happens with many modern RISC-based processors). Testing that comparator can be achieved by the execution of a sequence of store operations to a fixed address, each of these providing one of the 2m patterns. The first two store operations (lines 1 to 4 in Figure 4.5) correspond to apply the hexadecimal patterns 0xFFFFFFFF, 0x00000000. Since the same values should be applied to both inputs, the ISM is not active.

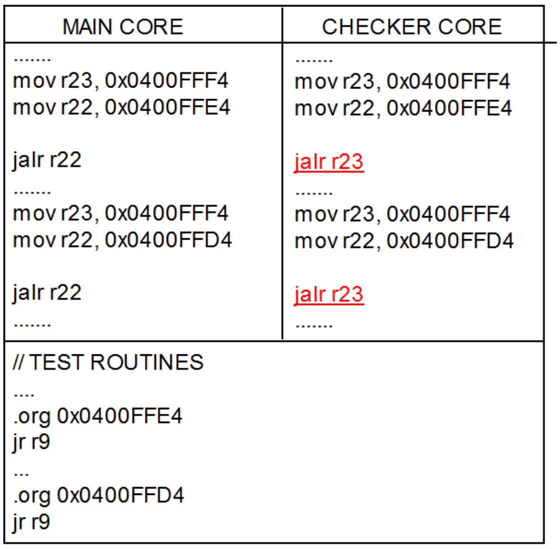
In the next step, since it is required to apply a difference to the comparator, the ISM must be activated. Clearly, the three registers are programmed depending on how the test patterns are generated by the self-test program. Assuming that the continuous substitution mode is enabled, the CTRL REG is programmed with the value 001001011. Considering the example, the CMP and SUB registers are programmed to replace the store operation with the store of a different value (i.e., register R7 instead of R6 as in line 7, 10, 13, 16). Then, the patterns are generated with a loop that implements the walking 0 followed by a store operation of the



```
//ISM Disabled.
1: mov r6, 0xFFFF
2: sw 0(r3), r6

3: mov r6, 0x0000
4: sw 0(r3), r6

// Enter test mode
// Program the ISM
// Replace sw 0(r3), r6 with sw 0(r3), r7

MAIN  CORE            CHECKER CORE

5: mov r7, 0xFFFF     mov r7, 0xFFFF
6: mov r6, 0xFFFE     mov r6, 0xFFFE

7: sw 0(r3), r6       sw 0(r3), r7

loopx32:              loopx32:
8: sll r6, 1           sll r6, 1
9: andi r6, 1          andi r6, 1

10: sw 0(r3), r6      sw 0(r3), r7

11: mov r6, 0xFFFF    mov r6, 0xFFFF
12: mov r7, 0xFFFE    mov r7, 0xFFFE

13: sw 0(r3), r6      sw 0(r3), r7

loopx32:              loopx32:
14: sll r7, 1          sll r7, 1
15: andi r7, 1         andi r7, 1

16: sw 0(r3), r6      sw 0(r3), r7

            // Exit test mode
```

Figure 4.5: A possible self-test program that leverages the ISM for testing the DATA RAM CMP. Before ISM activation, both cores execute the same instructions. After its activation, the instructions underlined in red are those substituted in the checker core.

newly generated pattern (lines 5 to 10). It is worth noting that one input must vary while the other one must be maintained unchanged; thus, the generation loop must be applied twice: first on R6, then on R7 (lines 11 to 16). Given the presence of the ISM, the final effect is to have the main core producing the first m patterns (walking 0 on the first input) and then the checker core producing the remaining m patterns (walking 0 on the second input). Once all the patterns are applied, the module is disabled, and the test routine terminates. The same reasoning applies to the ADDRESS RAM comparators. The self-test program structure is similar to the one discussed before, but the value stored is fixed while the addresses are generated resorting to a walking 0 strategy. In most of the cases, the two patterns 0xFFFFFFFF, 0x00000000 cannot be applied as the accessibility to these addresses depends on the particular memory map of the system under analysis. Indeed, normally RAM addresses are restricted to a particular range. Thus, to maximize the fault coverage, the two patterns correspond to storing data at the lowest possible address (with as many 0 as possible) and at the highest possible address (with as many 1 as possible). Then, starting from the highest address, walking 0 is applied. The same strategy pertains also for ADDRESS FLASH, but instead of a sequence of store operations, function calls are required. As depicted in Figure 4.6, each jump forces a different address (at each generated address, a valid function or piece of code must be present) so that all the required patterns are generated (clearly, within the address range of the program memory).



```
MAIN CORE                   CHECKER CORE
.......                     .......
mov r23, 0x0400FFF4         mov r23, 0x0400FFF4
mov r22, 0x0400FFE4         mov r22, 0x0400FFE4

jalr r22                    jalr r23
.......                     .......
mov r23, 0x0400FFF4         mov r23, 0x0400FFF4
mov r22, 0x0400FFD4         mov r22, 0x0400FFD4

jalr r22                    jalr r23
.......                     .......

// TEST ROUTINES
....
.org 0x0400FFE4
jr r9
...
.org 0x0400FFD4
jr r9
```

Figure 4.6: Fragment of self-test program testing the ADDRESS FLASH CMP.

It is important to note that forcing the main and the checker cores to jump at two different addresses in the program memory is totally safe, since in a DCLS system (Figure 4.1), the checker outputs are directed to the comparators only, while its inputs are driven by the main core. Therefore, it is always the main core that drives the execution flow in both cores. When dealing with the control comparators (CTRL CMP in Figure 4.1), the CSSM is required to be active. In this case, the self-test routine should:

1. setup the LSMU so that CSSM is active and all the other comparators but the CTRL CMP disabled;

2. configure the CMP REG with the target instruction;

3. configure SUB REG with the test pattern to be applied;

4. Execute the target instruction, which in this case behaves as a trigger for the substitution.

If the CONTINUOUS SUBSTITUTION mode is used, step 3 and 4 are repeated until all test patterns are applied. The LSMU allows for any test pattern to be applied, without any particular restriction. For obtaining a short and efficient test, they can be generated internally to the self-test routine following the walking bit strategy presented so far (e.g., as in Figure 4.5). Clearly, patterns can be stored in the memory Flash as constants. However, this would require additional space for the testing routine.

Finally, the adoption of the LSMU requires the insertion of the VALID signal (which acts as an enable) within the comparators. To achieve a complete fault coverage, it is required to test the hardware introduced for implementing such signals. For each comparator, the self-test routine should maintain the other comparators disabled and:

1. enable the target comparator and force the inputs such that they differ and then correspond;
2. disable the target comparator and force the inputs such that they differ and then correspond once again.

In the fault-free scenario, the outcome of step 2 should be independent from the value of the inputs. Step 1 is intrinsically implemented in the self-test algorithms presented in this section. Step 2 would instead require a custom routine.

4.2.3 Faults detection mechanism

Alarms raised by DCLS comparators are normally handled by a specific module integrated within the MPSoC, that reacts depending on the application requirements. Clearly, in order to adopt the strategy described above, such module must include a configuration setup that keeps track of the DCLS alarms raised during the on-line testing procedure and report any unexpected misbehavior. It is interesting to note that the hardware described in this chapter can be partially exploited in order to test the reaction capabilities of the top-level module handling the alarms. Although this is not the goal of this chapter, this can be achieved by adding suitable hardware to implement fault injection logic.

4.3 Experimental results

This section is organized as follows: the selected case study is first detailed. Then, the software and hardware self-test approaches are compared. Finally, the hybrid self-test mechanism is validated. Both overhead and achievable fault coverage are reported. Additionally, a detailed FMEDA is performed to asses the consequences of failure of the LSMU on the normal behavior of the SoC.

4.3.1 Case study

Experiments were conducted on an in-house modified version of the OpenRISC 1200 (OR1200) soft-core processor. It consists of a 32-bit pipelined RISC microarchitecture, with MMU and basic DSP functionalities. It includes also data and instruction caches. For the purpose of this work, caches and MMU were not included in the final synthesized version. The RTL source files are described in Verilog and are available from GitHub [2, 3]. Finally, the system includes also a behavioral description of a Flash and RAM memory. All the logic simulations were performed using Synopsys VCS, whereas Synopsys Design Compiler as logic synthesis tool. The effectiveness was assessed by means of fault simulation campaigns, using Z01X by Synopsys. Z01X is used widely for functional safety verification, providing an extremely flexible environment for the fault simulation. All the experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GBytes of RAM. A delayed DCLS configuration was implemented at the CPU level (i.e., all the logic within the CPU was duplicated), with a temporal diversity of two clock cycles between the two cores. A further bank of flip-flops was added for main and checker outputs to isolate any critical path from the comparators. Concerning the fault simulation campaigns, stuck-at faults were exclusively considered (1,374 faults), being the one used fault model for this kind of analyses in the ISO 26262 context. Nevertheless, the method is easily extensible to other fault models. Although the set of considered faults is relatively small, from the safety viewpoint they are extremely relevant. Indeed, the processor core is one of the main components of modern SoC and any failure of this unit is likely to affect the main application.

4.3.2 Evaluation of software self-test mechanisms

In the following, the fault coverage results of a set of software programs are analyzed. The programs used during the experiments fall into two categories:

- Application programs;
- Self-test programs of an STL targeting latent stuck-at faults within the CPU core.

For these experiments, stuck-at faults located within DATA RAM, ADDRESS RAM and ADDRESS FLASH comparators were considered. These three modules account for 936 stuck-at faults. In order to show the ineffectiveness of a software approach when addressing the type of faults mentioned above, four application programs were initially considered (Table 4.2). They implement simple applications: vector sorting (bubble_sort, quick_sort), minimum path identification in a graph (dijk) and random number generation (lfsr_32).

Table 4.2: Characteristics of applications programs

Programs	Duration [CC]	Size [B]	FC [%]
dijk	5,170	604	71.34
bubble_sort	1,184	192	70.84
quick_sort	3,278	932	71.34
lfsr_32	3,304	388	71.09

The DCLS modules are hard to be tested even when targeting latent faults within the CPU core via STL. Indeed, it was considered also a set of eight handcrafted self-test programs (Table 4.3), each of them was developed to test a specific part of the processor core. The fault coverage of the entire test suite against the whole processor stuck-at faults is 80.79%. For each program in Table 4.2 and 4.3 the duration (in clock cycles, CC), the size (expressed in Bytes) and the achievable fault coverage (FC) are reported. By comparing these two tables, one could immediately observe that for both sets, the fault coverage saturates at around 71%. The only self-test program that reaches a fault coverage of 72% is the one addressing processor Load Store Unit (LSU). Such test program generates a considerable activity on the memory interface. Thus, it is reasonable that the fault coverage is higher than any other program in the two sets.

Table 4.3: STL characteristics

Test Program	Duration [CC]	Size [B]	FC [%]
rf_test	1,502	3,508	71.84
cu_test	538	808	71.34
opmux_test	308	484	71.09
alu_test	10,820	3,448	71.84
mac_test	3,248	2,596	71.84
lsu_test	2,108	4,216	72.08
genpc_test	24,914	2,960	71.84
wbmux_test	538	808	71.34

4.3.3 Evaluation of hardware self-test mechanisms

As hardware self-test mechanism for the test of the comparators, the architecture described in [1] was implemented. For sake of a fair comparison, the hardware module was designed in order to generate the whole set of test patterns described in [44]. It is important to notice that the architecture described in [1] does not specify any test sequence to be used. The post-synthesis results are shown in Table

4.4. When applying patterns generated with this method, the overall fault coverage is 99.7% (obtained in about 500 clock cycles). The major drawback of this approach stems from the fact that the area overhead is directly proportional to the bit-width of the lockstep comparators (i.e., number of signals to be compared). The remaining faults not covered by this approach are related to the reset circuitry and as further explained in the next sub-section, they never produce a failure when in mission mode.

Table 4.4: Hardware Self-Test [1] Synthesis: Area Breakdown

Module	**Area [μm^2]**	**Total Area [%]**
DCLS CPU	140,891.39	100.0
Self-Test Module [1]	6,293.68	4.47

4.3.4 Evaluation of the hybrid self-test mechanism

The LSMU was designed in Verilog and included in the final RTL version. The overall system architecture is the same described in the previous section in Figure 4.3 and Figure 4.1, with all the control signals grouped in a single comparator. The entire system was synthesized and mapped to a 65nm CMOS technology library. The post-synthesis results are shown in Table 4.5.

Table 4.5: LSMU Synthesis: Area Breakdown

Module	**Area [μm^2]**	**Total Area [%]**
DCLS CPU	140,891.39	100.0
LSMU(ISM)	335.40	0.2
LSMU(CSSM)	204.36	0.1
LSMU(CU)	2,107.55	1.5

As it can be noticed, the LSMU accounts for the 1.88% of the of the entire DCLS CPU. The controller (CU) contains the CMP, SUB, CTRL registers and the FSM. The latter accounts for the 0.2%. It includes also the system bus interface, which represents the most expensive part (in terms of logic gates) of the block. It is worth noting that the ISM includes also the comparator for the target instruction shown in Figure 4.3. For avoiding performance degradation, the ISM was placed in between the two banks of flip-flops that delay the checker inputs.

In the following, the effectiveness of self-test programs leveraging the LSMU are analyzed. Four self-test programs were developed, each of them targets a specific comparator (including the test of the VALID signal). Table 4.6 summarizes the achieved fault coverage. While the data_ram achieves quite high fault coverage, the remaining programs were not so effective. Indeed, they are mainly limited from the fact that some fault locations are not accessible due to the system memory map. Therefore, an analysis of the fault list was performed in order to identify possible Safe Faults. This category of faults of the ISO 26262 can be seen as on-line functionally untestable faults, that do not produce any failure due to the application program, inputs data or in-field constraints [23]. Following the guidelines presented in [17], it was possible to remove faults due to the system memory map. It is important to notice that these are on-line functionally untestable faults which are application independent (i.e., they are not related to the application program) and therefore they can be individuated in any device. In the system under analysis, the following valid addresses exist:

- RAM from 0x0000_2000 to 0x001F_FFFF;
- Flash from 0x0400_0000 to 0x0400_FFFC.

Table 4.6: Characteristics of the self-test programs exploiting the LSMU

Test Program	**Duration** [CC]	**Size** [B]	**FC** [%]	**Safe FC** [%]
data_ram	1,392	818	99.03	100.0
addr_ram	1,626	1,038	91.35	100.0
addr_flash	1,132	1,738	75.32	99.12
ctrl_cmp	1,820	706	91.22	100.0

The reasoning behind this procedure stems from the fact that the processor is able to access only a portion of the available address space. As an example, according to the specifications mentioned above, when dealing with flash addresses the upper part of the address holds exclusively the value 0x0400. This causes having logic gates stuck to fixed value during the whole in-field behavior: as a consequence, it is not possible to change the value of some bits by executing any software. Such faults were identified resorting to TetraMax by Synopsys. The comparators inputs were connected to Vdd or ground. Then, given these constraints, the tool was instructed to perform a structural untestability analysis on the modified netlist. After this process, it was possible to remove about 20% of faults from ADDRESS RAM and ADDRESS FLASH comparators fault list. Then, further analyses were conducted on the remaining faults. Specifically, by using Inspect by Synopsys, it was possible to link the remaining faults to the reset signal of the comparators (that include flip-flops for breaking critical paths). Such faults would prevent the flip-flops from being initialized during the reset phase. However, when designing lockstep systems, it is common practice to invalidate comparators outputs for a certain number of clock cycles after the system leaves the power-on reset. This prevents receiving bogus data from both checker and main. As a consequence, those faults never provoke a failure (thus can be considered as safe) of the lockstep because when the comparators become active, they receive initialized data. The total percentage of safe faults removed is around 13% of the initial fault list. The final fault coverage after this pruning process is shown in the fifth column of Table 4.6. Using the hybrid test strategy described in previous section it was possible to achieve an overall fault coverage of 99.5% for all the cluster, with reasonable test duration and program size. Considering a clock period of 40ns (as in the performed experiments), the test programs were executed in 198.8μs, while their overall memory footprint is about 4KB. Lastly, it is noteworthy that the test program address_flash has a memory footprint almost double with respect to the other two test programs due to the additional portions of code needed by the test strategy for testing those comparators.

4.3.5 Failure Mode Effect and Diagnostic Analysis results

Normally, a complete FMEDA analysis involves also the computation of the failure rate. However, this depends on the technology used for the final implementation. Clearly, since the goal of the paper is to introduce a new architecture, these data are missing. For this reason, the focus of this analysis is mainly centered in determining the possible failure modes of the LSMU (and the related critical faults causing the failure), assessing their impact (i.e., whether they lead to a critical failure) and possible countermeasures against these faults. The possible failure modes (denoted as FM) affecting the LSMU (and its submodules) due to permanent faults are:

- FM1: the ISM is active, but is not able to correctly substitute the target instruction with the substituted instruction;
- FM2: The ISM is active, but is not able to recognize the target instruction (namely it does not perform the substitution at all);
- FM3: The CSSM is active, but is not able to substitute the control signals with the test pattern at all;
- FM4: The CSSM is active, but is not able to correctly substitute the control signals with the test pattern;
- FM5: The LSMU is not active, but the CU disables the lockstep comparators;
- FM6: The ISM is not active, but it performs a substitution of the instruction;
- FM7: The CSSM is not active, but it performs a substitution of the control signals.

Failure modes FM1-4 do not impact the safety of the application, since they become relevant during the test mode only. Thus, the only inconvenient is a lower fault coverage. FM7 is likely to be detected by the lockstep comparators and do not cause any critical failure since CSSM outputs are directly connected to those comparators. To further eases the detection of these faults, the reset value of the SUB REG could be set to a value such that only 1 bit differ (e.g., all zeros and a bit at one only). By doing so, it is possible to immediately detect the occurrence of faults causing this specific failure mode.

Failure modes FM5 and FM6 are quite critical instead, since directly impact the normal execution of any user application. For identifying critical faults that cause these failures, it was built a functional safety verification environment with Z01X. For increasing the truthfulness of the experiments, functional fault simulations (that is fault simulation of the entire SoC including the memories) were performed. The chosen fault model was still the single stuck-at, and fault were injected in the LSMU logic only (3,286 faults). When performing these evaluations, it is important to specify observation points and diagnostic points. The former are points of the design (namely, ports or internal wires) where to observe the effect of the faults. The latter instead are points of the design where to observe the reaction of the safety mechanism. Faults detected in both observation and diagnostic points can be labeled as dangerous detected. All the faults detected in an observation point but not in a diagnostic one are labeled as dangerous undetected. The remaining faults can be considered as safe faults.

For both failure modes, the same application programs used for the previous experiments were adopted. For FM6, using as reference Figure 4.3, the observation point was placed on the checker inputs. The FM6 causes a wrong instruction to be fed to checker, thus by observing its inputs, it is possible to identify faults leading to that failure mode. After running the fault simulations, the 5.17% of the total faults cause a wrong instruction to be fed to the checker. As safety mechanism (i.e., countermeasure) for these faults, it is possible to use the already existing comparators of the DCLS configuration. This can be done since it is assumed a single stuck-at fault only. The fault simulations were repeated, and the diagnostic point was placed on the ALARM signal. At the end of this campaign, all the 5.17% critical faults were detected also at the diagnostic point. Thus, they can be labeled as dangerous detected whereas the remaining 94.83% as safe since they do not cause this specific failure. The same procedure was performed for FM5 as well. Since this failure mode causes lockstep comparators to be disabled, the observation point was situated on the VALID signal, output of the LSMU. In this case, 5.42% of the total faults lead to this failure and they all belong to the FSM within the CU. As countermeasures, two options are possible: leverage the self-test routines of Table 4.6 or use a Triple Module Redundancy (TMR) configuration for the FSM within the LSMU

Considering the first option, by placing the diagnostic point on ALARM (namely the same when testing the lockstep comparators) 97.25% of the critical faults results being detected. This means that most of the faults are actually detected during the test of the DCLS comparators. Concerning the second option, using FSM in a TMR configuration on one hand increases the fault coverage, the detection time, but also silicon area of the LSMU. As already shown at the beginning of this section, the FSM accounts for the 0.2% of the total design. By using a lockstep variant, the overhead of this module increases up to the 0.66%. Although a TMR configuration is adopted, it is important to notice that few faults in the majority voter logic might still lead to a failure. However, such faults are less than the 0.2% and thus there is still compliance with the ISO 26262 standard. Table 4.7 summarizes the possible failure modes and the countermeasures, along with the achievable Diagnostic Coverage (indicated with DC, being the number of critical faults detected by the safety mechanism).

Table 4.7: FMEDA LSMU: Failure Modes and Countermeasures

Sub-Module	**Failure Mode**	**Critical**	**Safety Mechanism**	**DC [%]**
ISM	FM1	NO	NONE	-
ISM	FM2	NO	NONE	-
CSSM	FM3	NO	NONE	-
CSSM	FM4	NO	NONE	-
CU	FM5	YES	Self-Test Programs	97.25
CU	FM5	YES	TMR-FSM	99.99
ISSM	FM6	YES	DCLS	100.0
CSSM	FM7	YES	DCLS + SUB reg safe reset	100.0

4.3.6 Comparison of the hardware, software and hybrid self-test mechanisms

Table 4.8 summarizes the three self-test alternatives described in this section. The first important observation is that the STL solely does not detect all the possible latent faults within this safety machanism. Therefore, the STL must be necessarily complemented with either a pure hardware self-test module (e.g., [1]) or the proposed hybrid architecture. The former yields a complete fault coverage with a short test application time. Instead, the proposed one is still able to generate the required test stimuli, while mitigating the area overhead introduced by a pure hardware self-test approach (halving the area overhead). On the other hand, since now part of the stimuli are generated via software, the test duration is higher.

Unlike pure hardware strategies, in which the test patterns are hardwired, the test performed with the proposed approach is much more flexible: test patterns are not anymore fixed and can be updated on-the-fly. Finally, it is worth mentioning

Table 4.8: Area, fault coverage and test duration for the three approaches considered

Self-Test Approach	**Area w.r.t. DCLS [%]**	**FC [%]**	**Duration [CC]**
Hardware [1]	4.47	99.7	500
STL	0.00	72.0	43,976
LSMU	2.10	99.5	5,970

that the area reported in Table 4.8 for the proposed approach includes also the additional circuitry that mitigates the critical failures.

Part II

Improvements of functional fault grading methodologies

Chapter 5

Improved fault grading techniques for STLs

In order to assess the effectiveness of the STLs, functional fault simulation is performed, so that the achieved fault coverage (e.g., in terms of stuck-at faults) can be computed. This chapter explains the reasons why the functional fault simulation of the STLs represents a different problem with respect to the classical fault simulation of test stimuli (for which very effective algorithms and tools are available). Then, it provides evidences that this kind of fault simulation can be highly computationally expensive. Finally some solutions are proposed to reduce the computational cost taking into account the final usage of the STL (i.e., for single-point faults or for latent faults in a DCLS system) and possibly trade-off between results accuracy and cost. Motivations and the basic differences of functional fault simulation with respect to classical approaches are presented in Section 5.1. Then, Sections 5.2 and 5.3 describe the different fault grading approaches. All the proposed solutions are based on the usage of commercially available EDA tools, thus being easily adoptable by professionals in the field. With respect to similar works [14, 80], it is also provided a comprehensive and commented overview about the techniques that can be adopted to effectively perform the fault grading of STLs. Finally, all these techniques are experimentally validated in Section 5.4

The material found in this chapter was published in [32].

5.1 Background

5.1.1 Motivations

Computing the fault coverage achieved by a given test sequence is typically done resorting to fault simulation. In the past, a wide research effort led to the development of effective and highly optimized algorithms for fault simulation as for example [75]. Such algorithms have been widely adopted in commercial tools (e.g., stand-alone fault simulators and those integrated in ATPG tools) which are used in different commercial environments for generating and grading test patterns for digital circuits. Hereinafter, these techniques are called sequential circuit fault simulation (SC-FSIM) approaches. However, those tools and methodologies targeted a rather different problem than the one considered in fault simulation of STLs (hereinafter, self-test procedures fault simulation or STP-FSIM approaches). In fact, traditional SC-FSIM approaches are intended to compute the fault coverage provided by a sequence of input vectors applied to a generic sequential circuit. It is assumed that all the output signals can be continuously observed. This scenario is very similar to the one that can be found during the end-of-manufacturing testing. However, it is very different to the one related to in-field testing. When considering the effects stemming from the execution of a self-test procedure, there is a processor executing a program stored in a memory and producing some output results. These are also written in memory and observed at the end of the procedure execution (most often compacted in a unique test signature). This scenario is quite different than the previous one. For example, the sequence of inputs for the processor (e.g., the sequence of executed instructions) often changes due to the effects of faults. This frequently happens for example when a fault affects the logic implementing the instruction fetch mechanism. Moreover, the observability mechanism is completely different: a fault can be labeled as detected when it produces a wrong result in memory at the end of the self-test procedure execution.

The fault grading of STLs becomes relevant during FMEDA whenever they are used as self-test mechanism for the processor cores. To better fit in this scenario, EDA tool vendors introduced so-called functional fault simulators (e.g., Z01X by Synopsys, or as new features in the Incisive platform by Cadence). Such tools allow fault simulation of circuits at different abstraction levels (i.e., from RT to gate level), as often done during functional safety analyses.

5.1.2 Sequential circuit fault simulation (SC-FSIM)

With SC-FSIM the goal is to compute the fault coverage achieved when a given test sequence is applied to the Circuit Under Test (CUT). Normally it corresponds to a combinational or sequential digital circuit (Figure 5.1). Generally, this kind of test takes place at the end of the device manufacturing. In this scenario, the CUT is mounted on an ATE and the test sequence is applied. In most of the cases, the test of the CUT resorts to Design for Test (DfT) structures. The sequence of test patterns (i.e., values) applied to the CUT is fixed and does not depend on: the sequence of output values produced by the CUT itself during the test; nor on the effects of the faults. Moreover, the output signals are continuously monitored to detect possible fault effects. As soon as a difference on any output signal is produced by a fault, the fault can be labeled as detected. The fault simulation should accurately reproduce this scenario. However, the goal of the fault simulation is only to compute the number of detected faults. Since fault simulation is computationally intensive, once the fault is detected, it is not necessary anymore to simulate its effects in the following. This mechanism (known as fault dropping [39]) significantly reduces the computational cost of the fault simulation, since most of the faults are only simulated for a relatively limited period of time.

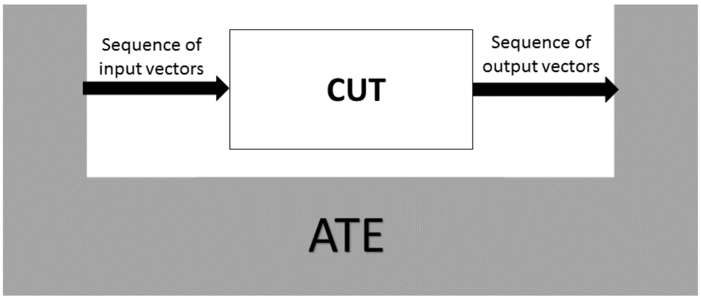


Figure 5.1: Test vectors-based end-of-manufacturing test scenario for a generic sequential CUT.

5.1.3 Self-test procedures fault simulation (STP-FSIM)

When the goal is to compute the fault coverage achieved by running one or more self-test procedures on a CPU, the scenario is rather different. This process is also known as fault grading [14] of self-test procedures. In particular, in this case the scenario to be considered is more complex than the previous one. Since the typical application of STLs is for in-field test (which is performed when the device is already in the operational phase) the self test does not rely on any external ATE. In this scenario the CPU is not fed with test patterns but with processor instructions and data read from the memory (or coming from input peripherals, if any). Actually, during the test the CPU executes a piece of code, and thus continuously interacts with the memory modules for instruction fetch and for data read/write operations. The CPU may also interact with peripheral modules, too. Therefore, the CUT to be fault simulated cannot be the CPU only, but all the modules it interacts with (Figure 5.2).

For the sake of simplicity, and since I/O operations are conceptually similar to memory operation, in the following I/O operations are neglected. During the in-field execution, to create the test signature, the self-test routines accumulate the produced results in a single register. The content of this signature register is then checked by the self-test code itself against a golden one (computed off-line in a fault-free scenario): a fault is considered as detected if the execution of the test code produced some wrong signature values in the signature register. At the end of the test, the self-test code itself performs the check on the signature. It then

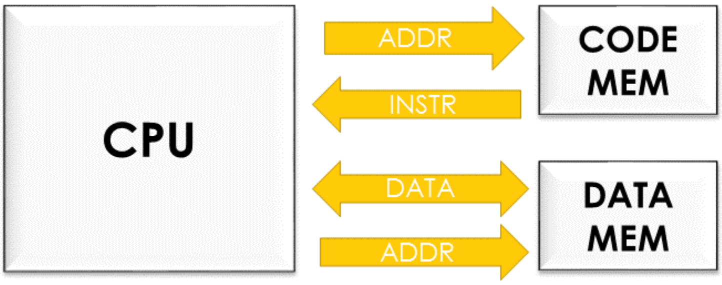


Figure 5.2: STL-based in-field test scenario for a CPU-based system.

returns to the caller by storing a flag in the memory. Such flag (i.e., a go/no-go binary value) states whether a fault was detected. Along with with this go/no-go flag, the self-test procedure can possibly return also the computer signature. It is worth noting that this flow is only one of the possible alternatives for checking the signature. Alternatively, the software invoking the self-test procedure might check the signature value (which is returned by the self-test procedure to the caller via a register or memory location).

In general, the following statements on STP-FSIM are true:

1. The CPU is stimulated with an input sequence corresponding to:

   - values coming from the code memory, corresponding to the codes of the instructions that the CPU execute;
   - values coming from the data memory, corresponding to the values of the accessed memory locations;

   In both cases, the input sequence corresponds to the content of the memory locations accessed by the CPU by outputting an address. Hence, the input sequence may change if a fault modifies any of the addresses generated by the CPU. SC-FSIM techniques (and tools) are hardly able to manage such a scenario.

2. A fault is detected when a specific condition is met at the end of the test code execution (e.g., a given memory location stores a given value, or a given value is returned by the self-test procedure). Again, SC-FSIM techniques (and tools) are hardly able to manage such a scenario too;

3. Since a fault can be labeled as detected only at the end of the test code execution, fault dropping cannot be performed. All faults must be simulated until the end of the test code execution, thus resulting in a significant increase of the computational cost.

As a conclusion, the fault simulation of self-test procedures composing an STL differs from the SC-FSIM, since it requires methodologies and tools able to:

1. efficiently fault simulate a CPU while executing a piece of code (i.e., interacting with the memories);
2. support more sophisticated fault detection strategies than simply detecting a difference on the output signals;
3. limit the computational cost, by taming the extra effort required by the absence of the fault dropping mechanism.

It is worth noting that the specifications for STP-FSIM maybe more complex than those outlined in this section. As an example, a fault may provoke effects which are different than simply producing a wrong value in memory at the end of the test procedure. Indeed, it is quite common that faults trigger an exception or force the processor to enter an endless loop. In both cases, the fault is typically categorized as detected, forcing the fault simulation tool to support fault detection mechanisms that can hardly be implemented by SC-FSIM techniques.

5.2 Basic fault grading techniques

In the following, possible approaches for STP-FSIM are described. The different approaches differ in terms of:

- fault detection mechanism;
- input stimuli.

In the context of this chapter, the fault detection mechanism defines which output signals (i.e., observation points) of the design to observe and when to observe them in order to determine whether a fault can be labeled as detected or not. The input stimuli instead refers how the instructions and data are fed to the CPU. In the following, without loss of generality, it is assumed that each self-test procedure stores the computed signature at the end of its execution in the data memory (as it often happens).

The analysis is focused initially on the fault grading of a single self-test procedure. Then, the reasoning is extended to the fault grading of the whole Software Test Library. This case requires a dedicated discussion since more than one self-test procedure must be evaluated, considerably increasing the computation effort. For each approach, advantages and drawbacks are presented. These are then validated in Section 5.4.

5.2.1 Fault grading of a single self-test procedure

For performing the fault grading of a single self-test procedure three main alternatives are possible.

1. STP-FSIM0: this approach is based on the traditional SC-FSIM. Techniques and tools belonging to that category can be adopted in this case, although being originally conceived for a rather different purpose. However, as discussed in the following, this may become quite inadequate and may not yield correct results. When dealing with this type of fault simulation, the CUT is the CPU only. Its inputs are fed with test patterns. In this case, test patterns correspond from one side to the sequence of encoded instructions composing the self-test procedure. These are fed sequentially to the CPU each time it performs a fetch operation. From the other side, to the data values the CPU reads from the memory each time it performs a read memory access. Since a traditional SC-FSIM method is considered, the input sequence is fixed. Hence, independently from the fact that a fault might change or delay the sequence of instruction/data addresses produced by the CPU, the sequence of fetched and executed instructions/data remains the same during the whole fault simulation. Moreover, during the fault simulation experiment, all CPU outputs are observed clock cycle per clock cycle. As soon as a difference is detected, the corresponding fault is labeled as detected (and dropped from the simulation). In this way, this approach could lead to wrong (i.e., larger than real) figures in terms of fault coverage. Indeed, faults might provoke a difference in one or more output signals, but in the real execution such a difference could later be masked. Thus, it could not be reflected in the final self-test procedure signature.

2. STP-FSIM1: The procedure described above can be improved to increase the correctness of the fault simulation results. In order to mimic the real-case scenario (that is, when the test is performed in field), the observability is limited to the signals directed to the Data Memory (signals on which the signature is supposed to transit when being stored). Moreover, the fault simulator should be instructed to observe such signals exclusively at the time the result is going to be written in memory. This situation is represented in Figure 5.3, where the offset time represents the initial period of time where the test program executes its instructions without writing the results into the memory. The main limitation of this approach remains the fact that it is not possible to reproduce the effects of faults that lead to a different execution flow of the test code. Additionally, fault dropping is not exploited: outputs are observed exclusively at the end of the self-test procedure. Thus, all faults must be simulated during the whole experiment, and the computational cost grows significantly.

3. STP-FSIM2: The limitations of the methodologies described above partly stem from the fact that the CUT is exclusively the CPU. In this model, the interactions with memories cannot be suitably modelled, especially in the faulty circuits. However, when resorting to a functional fault simulator these limitations can be overcome by simulating the entire system in which the CPU is integrated, including data and code memories (possibly with the peripherals). In this way, the exact Fault Coverage figure achieved by a self-test procedure can be computed. Normally, the CPU is described as a gate-level netlist as in the SPT-FSIM0 and STP-FSIM1 approaches. The other components (in which faults should not be injected) resorting to behavioral descriptions. The instructions (which represent the input stimuli for the CPU) are directly fetched from the instruction memory which is now part of the simulated model. Thus, it is possible to model the effect in which a fault forces a different execution flow. Moreover, it is also possible to model also the scenario in which different data (due to the effects of a fault) are retrieved or stored to or from the data memory throughout the program execution. Concerning the observability, the final content of the memory is directly observed, once the self-test procedure terminates. Clearly, it makes sense to check only those addresses in which the test program is supposed to write; otherwise, it becomes unfeasible to check the entire memory for a large design. However, since the simulated model is now much larger, the computational cost required with respect to the previous two approaches is significantly higher. Indeed, simulation of memories is computationally expensive. Additionally, since fault dropping is not performed, the fault simulation tends to be slow and memory intensive from the host (i.e., the computer in which the fault simulation is run) point of view.

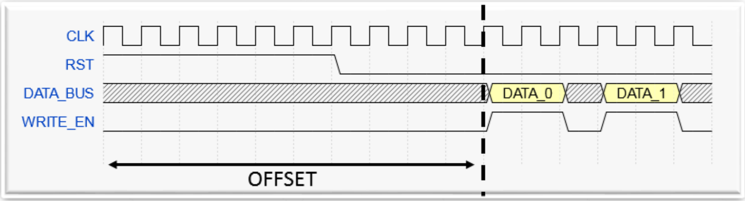


Figure 5.3: Graphical representation of the fault detection mechanism for STP-FSIM1.

5.2.2 Fault grading of a Software Test Library

Normally, the self-test procedures are developed according to a divide and conquer strategy. When the target module is a CPU, this is partitioned into sub-modules. For each sub-module, a specific self-test procedure is developed [13].

The methodologies described in the previous sub-section represent an acceptable solution for performing the fault grading of a single test procedure. During the development of the self-test procedure for the targeted module, only the faults it contains are considered. Nevertheless, during the global development flow it is quite common that a fault simulation of a set of self-test procedures on the entire CPU fault list is required. This step is essential to assess the overall fault coverage achieved up to that point and (during the STL development) to better guide the development to reach the target fault coverage. The rationale behind this relies on the fact that it is likely that a test program developed for a given sub-module can also detect faults present in different sub-modules. The most efficient strategy is the so-called incremental fault grading. The fault simulation of the whole STL is divided into different passes. At each pass, a different self-test procedure is fault simulated. Initially, all the faults present in the fault list are labeled as not detected. After the first pass, another test program is fault simulated. This second pass inherits from the initial one all the faults that are labeled as not detect. This process is repeated until all the self-test procedures are fault simulated. The main advantage of this approach relies in the fact that only the first self-test procedure is fault simulated against the full fault list. As passes are executed, the number of faults to be simulated progressively reduces and thus also the effort for the fault simulation itself. It is worth noting that the strategy described above can be applied to any fault simulation methodology without any loss in accuracy concerning the fault coverage. Besides, those that benefit the most from this approach are the fault simulation methodologies more computationally intensive, such as STP-FSIM2.

5.3 Optimized fault grading techniques

With this section the aim is to further extend the set of available techniques considering two additional strategies. They are built on the top of the STP-FSIM2, but they enable a faster fault simulation at the expenses of a limited loss in accuracy concerning the final fault coverage figure. The common observation behind both techniques is that, in STP-FSIM2 the fault dropping is not exploited at all. Hence, in the following it is discussed a suitable way to include such a mechanism.

5.3.1 STP-FSIM3

The main difference of this method (Figure 5.4) compared to basic STP-FSIM2 concerns the observability. Instead of observing exclusively the data memory at the end of the test program execution, the address signals towards the code memory are monitored at each clock cycle. The rationale for this choice originates from the fact that some faults can cause a different sequence of instructions to be fetched from the code memory. In most of the cases, this leads to a different execution flow of the test program, and thus to a different signature produced by the test procedure itself. In order to save fault simulation time, the idea is to quickly identify faults that force a different execution flow, by discarding them from the fault simulation as soon as possible (hence enabling the fault dropping). Obviously, not all the faults that cause a different execution flow finally produce a different memory content. As a consequence, a slightly different fault coverage is expected, higher than STP-FSIM2. On the other side, the experimental results show that the difference is normally small, while the savings in computational cost may be relevant.

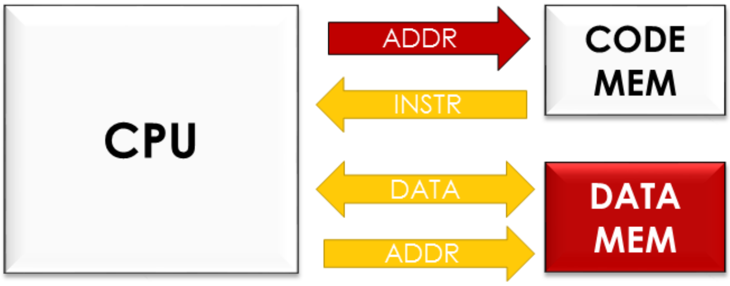


Figure 5.4: STP-FSIM3 scenario: the observability locations are marked in red.

5.3.2 STP-FSIM4

A further optimization (Figure 5.5), which can provide additional speed-up with respect to STP-FSIM2, consists in observing the CPU signals connected to the data memory (namely data and address signals) when the CPU performs a write operation to memory. A fault is marked as detected (and hence immediately dropped) if the address value produced by the CPU when a memory write operation performed is different than the expected, or when the data value written to memory at the same time is different. This strategy is clearly the most aggressive one since it leverages as much as possible the fault dropping. Once again, on one side this method allows for fault simulation time reduction, while sacrificing accuracy of the fault coverage. Theoretically, this method may lead to optimistic results since faults affecting memory operations could not be reflected in the signature. In practice, few memory operations are normally performed during the execution of the self-test procedures. These operations involve saving/restoring the previous context prior to the self-test program invocation and store/load operations to specific addresses for testing specific units. In the former case, any corruption of the stack frame due to faults irreversibly leads to the test failure. In the latter case, since normally the entire test is built upon these memory operations, any variation is reflected in the final signature.

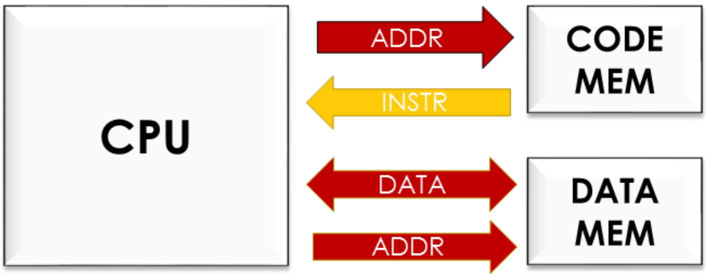


Figure 5.5: STP-FSIM4 scenario: the observability locations are marked in red.

5.3.3 Summary of the different techniques

Table 5.1 summarizes the fault simulation methodologies presented in this paper. For each method, all the relevant characteristics are reported. For STP-FSIM2-based approaches it is not required any input sequence since the stimuli required for the CPU are directly taken from the memories and they may vary depending on the fault effects.

Table 5.1: Fault simulation techniques comparison

Fault simulation technique	Simulated System	Input Sequence	Observed Outputs	Observation Instants	Fault Dropping	Accuracy
STP-FSIM0	CPU	Fixed	All CUT outputs	All Clock Cycles	Yes	Low
STP-FSIM1	CPU	Fixed	Data Memory Signals	When results are written to memory	No	Low
STP-FSIM2	CPU and Memories	None	Final Memory Content	End of Test Program	No	Complete
STP-FSIM3	CPU and Memories	None	Final Memory Content	End of Test Program	Yes	High
			Instruction Address Signals	All Clock Cycles		
STP-FSIM4	CPU and Memories	None	Data Memory Signals	Memory Write Operations	Yes	High
			Instruction Signals	All Clock Cycles		

5.4 Case study and experimental results

In this section, the following topics are covered:

- a brief overview of the flow used for assessing the fault coverage of a Software Test Library;
- the relevant characteristics of the self-test procedures;
- the fault simulation environment used to quantitatively evaluate the effectiveness of the different techniques.

The gathered experimental results are also presented and discussed.

5.4.1 Experimental setup and case study

Experiments were conducted on the same open source OpenRisc1200 (OR1200) processor already described in Chapter 4. The overall architecture of the processor is depicted in Figure 5.6. The OR1200 gate-level netlist (synthesized with Synopsys Design Compiler with a 65nm CMOS library) was inserted in a SoC, which comprises a WishBone interconnect, a Flash memory and a RAM. The Flash and RAM memories are 2MBytes each.

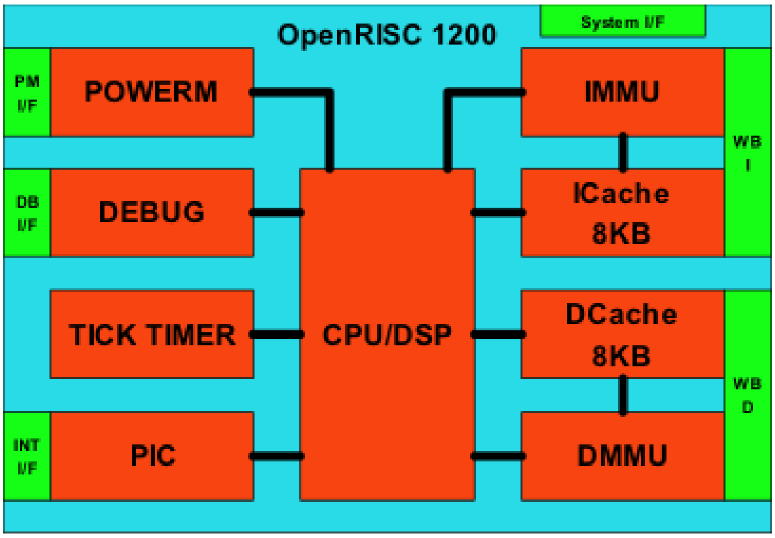


Figure 5.6: The OR1200 architecture.

The Software Test Library used for the experiments described in this Chapter is based on the one described in Chapter 4. However, with respect to that STL the one used in the following includes some minor modifications. The main features of these modified self-test programs are listed in Table 5.2. Each self-test procedure addresses stuck-at faults of a specific CPU sub-module. The self-test procedure also computes internally the result of the test (i.e., the test signature) and then this is written to a known memory location in the system RAM.

Concerning the fault simulation campaigns, it was adopted the same functional fault simulator used for the experiments of Chapter 4. Such tool allows to implement SC-FSIM and STP-FSIM techniques. For implementing both STP-FSIM0 and STP-FSIM1, the test patterns (i.e., the instructions) are provided to the CUT (that is, the OR1200) by means of a Value Change Dump (VCD) file, previously generated through a logic simulation of the self-test procedure on the gate-level netlist (for these experiments, leveraging Synopsys VCS). During the STP-FSIM0 fault simulations, the observation points were placed on all top-level ports of the OR1200 (namely the green boxes in Figure 5.6). For STP-FSIM1, the observation points were limited to the Data WishBone Interface (namely WB D in Figure 5.6).

Table 5.2: STL characteristics

Self-test procedure	**Duration [CC]**	**Size [B]**
rf_test	1,506	3,536
cu_test	542	836
opmux_test	312	512
alu_test	16,424	14,776
mac_test	3,252	2,624
lsu_test	2,112	4,244
genpc_test	24,904	2,960
wbmux_test	538	808

Moving to the STP-FSIM2, STP-FSIM3 and STP-FSIM4 experiments, the simulated model is a system composed of the OR1200 core and two memory modules of 2 MBytes each. During STP-FSIM3 the RAM memory content and the Instruction WishBone Interface (WB I in Figure 5.6) were exclusively observed. For STP-FSIM4 Data and Instruction WishBone Interface were observed. Among the other functionalities offered by the tool, there is also hyperfaults [40] detection. However, fault simulators do not always support this specific mechanism. Hence, for the sake of experiments reproducibility, this feature was disabled during the fault simulation campaigns. All the experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores and 256 GBytes of RAM.

Fault simulations were run leveraging just one of the available cores. For the sake of generality, the experiments were performed on the processor stuck-at fault list, without removing any kind of untestable faults. Finally, the fault simulations were performed with a zero-delay mode (i.e., all combinational and sequential delays were ignored).

Table 5.3: Fault simulation results

Fault simulation method	**Duration [hours]**	**FC [%]**
STP-FSIM0	0.6	83.25%
STP-FSIM1	7	75.91%
STP-FSIM2	41	81.01%
STP-FSIM3	18	81.26%
STP-FSIM4	13	80.67%

5.4.2 STP-FSIM methods

In Table 5.3 the gathered experimental results are reported for the methodologies presented in Section 5.2 and 5.3. The figures concerning the fault coverage were computed for the entire STL against a full CPU fault list (which accounts for about 98k stuck-at faults), using the incremental fault grading strategy. It is possible to observe that both STP-FSIM0 and STP-FSIM1 approaches are undoubtedly the fastest, although they are the ones producing the highest discrepancies concerning the fault coverage figures with respect to STP-FSIM2 (being the one yielding correct fault coverage results). STP-FSIM1 is slower compared to STP-FSIM0 due to the reduced observability that inhibits the fault dropping mechanism. Moving to the STP-FSIM2-based techniques, STP-FSIM2 yields the exact value of fault coverage, since it reproduces the same operational conditions as the ones when the SoC is deployed in field. The self-test procedures were designed so that their result is written in a single memory location. Therefore, at the end of the test program execution, only that memory location should be checked. Finally, the two optimized techniques (STP-FSIM3 and STP-FSIM4) exhibit a considerable speed-up compared to the base approach. STP-FSIM3 allows a fault simulation time reduction of almost 56%, while STP-FSIM4 goes even further, around 68%. This is significant, since the loss of accuracy of fault coverage is very reduced (0.3% for STP-FSIM3, 0.4% for STP-FSIM4, both with respect to STP-FSIM2). The reason for this minor difference between STP-FSIM3 and STP-FSIM4 mainly stems from the fact that in STP-FSIM4 there is a higher number of faults marked as potentially detected by the fault simulator. If these faults were counted as detected, the two methods would yield almost the same fault coverage figures.

Interestingly, after processing the fault lists of STP-FSIM2-3-4 with a Fault List Analysis Tool (FLAT, [10]), it emerged that the three approaches detect slightly different sets of faults. Such sets of faults are denoted with A, B, C, D, E, F and G (Figure 5.7). In Table 5.4 instead, it is detailed the number of faults within each set. As shown in Figure 5.7, the set of detected faults by each fault simulation approach can be expressed as a composition of these sets (for sake of conciseness, since E, F and G are empty in the considered case they are omitted in the following).

It can be observed from Table 5.4 that the set A is the largest one. Indeed, it is the one that contains the faults covered by all the three techniques. The set B is in common between STP-FSIM2 and STP-FSIM3, but not present in STP-FSIM4. It should be noticed that all the faults covered by STP-FSIM2 are included in the faults covered STP-FSIM3. The faults within the set C, that are only covered by STP-FSIM4 and STP-FSIM3 (marked as not detected in STP-FSIM2), mainly belong to the OR1200 modules genpc, if and ctrl. This is reasonable, since STP-FIM3 and STP-FSIM4 differ from STP-FSIM2 in the observation of the instruction bus: the modules genpc, if and ctrl are directly connected to that interface. The same reasoning applies to the set D, exclusively included in STP-FSIM4. Faults in

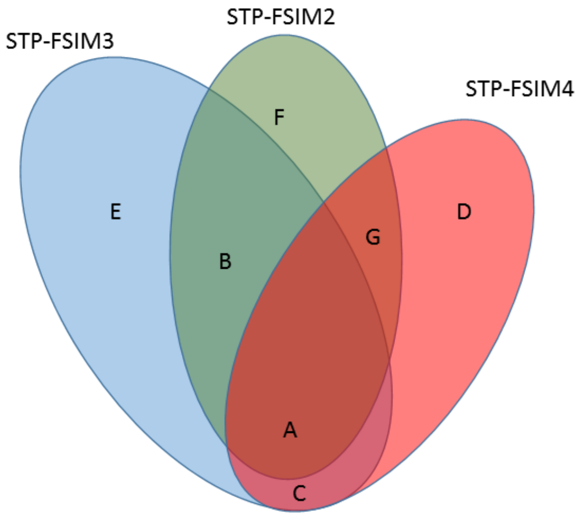


Figure 5.7: The detected faults by STP-FSIM2, STP-FSIM3, STP-FSIM4 and their possible intersections. In this case, E=F=G=0.

D are mainly related to the lsu and except units, which have a connection to the data bus, and STP-FSIM4 is the only technique that observes these signals.

Table 5.4: Size of faults sets

Set of Detected Faults	Number of Faults
A	78,583
B	643
C	251
D	168
E	0
F	0
G	0

Figure 5.8 summarizes the results produced by the presented methodologies. As it can be noticed, STP-FSIM0 and STP-FSIM1 approaches are the fastest concerning fault simulation time, although they provide quite inaccurate fault coverage metrics (with respect to the exact value, represented by the red line). On the other hand, STP-FSIM2-based approaches are more computationally intensive but yield the most accurate results. Specifically, given that STP-FSIM2 provides exact results, STP-FSIM0 is supposed to yield always higher values of fault coverage, since all outputs are observed continuously. Differently, STP-FSIM1 gives lower values of fault coverage, since it does not consider the effect of faults causing a different

execution flow. When dealing with STP-FSIM3 and STP-FSIM4, the fault coverage metrics are an acceptable approximation of the real coverage (as confirmed by the experiments).

It is noteworthy that with a careful development the self-test routines, the difference between the exact fault coverage figure and the one given by any of the approximate methods can be minimized.

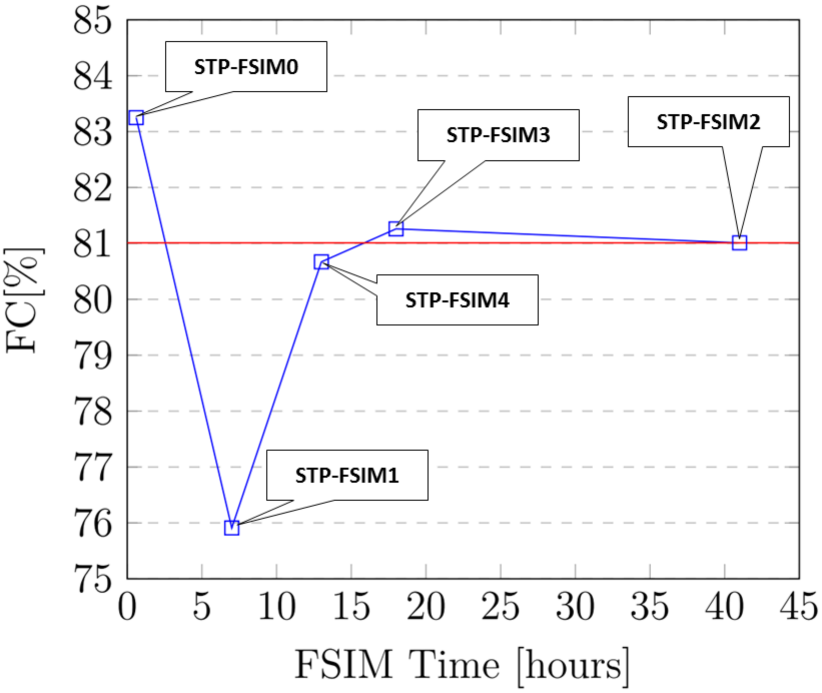


Figure 5.8: Fault simulation methodologies: fault coverage accuracy versus fault simulation time. The red line represents the exact fault coverage figure.

As an example, consider Table 5.5, that reports the fault coverage values on each module targeted by the specific self-test routine when using STP-FSIM0 and STP-FSIM2. For tests like rf_test, alu_test, and lsu_test the difference between the two techniques can be significant (e.g., 24.62% of difference for rf_test). This mainly stems from the fact that modules like the LSU are directly connected to the OR1200 top-level outputs (that is, the system bus interface) thus it is easier to detect faults effects when using STP-FSIM0 (in which all outputs are observed) rather than with STP-FSIM2 (which leverage exclusively the produced signature).

For RF and ALU instead, the discrepancies are due to the fact that the ALU is used for computing the effective address of load and store operation. Therefore, the ALU output is also connected to the system bus interface and it happens that during the test of RF (tested with a test algorithm like the one presented in [19]) and ALU some bogus data transit on that signal which are not reflected in the

Table 5.5: STP-FSIM2 VS. STP-FSIM0 fault grading

Self-Test Routine	**STP-FSIM2 FC[%]**	**STP-FSIM0 FC [%]**
genpc_test	53.78	57.04
cu_test	74.58	73.90
rf_test	57.93	82.55
opmux_test	86.65	87.32
alu_test	66.15	74.36
mac_test	88.30	89.59
lsu_test	72.69	84.68
wbmux_test	66.15	70.87

final signature. On the other hand, there exist modules deeply embedded in the processor core that do not suffer from these divergences. Indeed, MAC and Operand multiplexers do not have a direct connection with top-level signals, hence the results are quite similar with both approaches.

For the sake of completeness, the fault simulation time required for a monolithic (i.e., without incremental fault simulation) fault grading of the STL was computed for the STP-FSIM0, STP-FSIM1 and STP-FSIM2 approaches (Table 5.6). As it can be noticed, although it is still feasible for STP-FSIM0, for the other approaches the fault simulation time explodes, and it would become quite unfeasible for large designs.

Table 5.6: Monolithic VS. Incremental fault grading

Fault Simulation Method	**Duration of monolithic approach [hours]**	**Speed-up with incremental fault simulation [%]**
STP-FSIM2	316	87
STP-FSIM0	1	40
STP-FSIM1	76	90

5.4.3 Fault grading for DCLS-oriented STLs

From the experiments described so far, it is undeniable that STP-FSIM0 and STP-FSIM1 are not suitable for providing a final fault coverage figure that reflects what happens in the operational field. However, when dealing with STL generation for cores embedded in a DCLS-based SoC, the usage of the STP-FSIM0 could significantly improve the development time and yet provide correct results. Indeed, DCLS is normally adopted for meeting the ISO 26262 constraints of complex devices such as multi-core ones. In this context the STL are used for detecting latent faults affecting checker or main core. Clearly, adopting STP-FSIM2 for such huge designs could be problematic. However, as anticipated in the previous chapters the key idea of the DCLS configuration is to add an exact copy (from the functional viewpoint) of the main core, so that any fault effect that propagates up to some output signals can be immediately detected by a set of comparators. This principle holds as long as just one of the two replicas is affected by the faults.

For this reason, STLs are used against the latent faults, so that the possible occurrence of faults is forced to appear as a failure at the processor output. Since these outputs are continuously monitored by comparators, when this happens an alarm is triggered and the SoC reacts accordingly. Thus, there could be faults that manifest themselves before the final check of the signature by the self-test routines, due to the lockstep comparators. The scenario described above is conceptually similar to what happens in fault simulation, especially in STP-FSIM0, in which the output signals are monitored and compared with the golden values of a fault-free machine. Hence, STP-FSIM0 can be exploited (along with its advantages) when developing and grading STLs for DCLS-based SoC. Obviously, there is not a perfect equivalence as the experiments performed with the same DCLS version of the OR1200 used in Chapter 4 confirmed. The lockstep was applied to the lowest possible level, that is at the CPU level.

Table 5.7: STP-FSIM0 for DCLS-oriented STL grading

Self-Test Routine	**CPU STP-FSIM0 FC [%]**	**Comparators Output FC [%]**
cu_test	77.90	77.87
lsu_test	86.28	86.23
rf_test	90.88	90.87

Table 5.7 summarizes the results of the experiments. By observing the second column, it is worth noting that the self-test routines have fault coverage metrics slightly higher compared to those in Table 5.5. This is because for the results in Table 5.5 the observation points were placed on the output signals of the OR1200 that embeds the CPU, that is, one level of hierarchy higher with respect to the what was done in Table 5.7. This was necessary to have a fair comparison with the results when leveraging the lockstep mechanism only. The third column reports the fault coverage when observing the output signal of the lockstep comparators. As it can be noticed, the differences are present but negligible (0.03% in the worst case).

Chapter 6

JTAG-based fault emulation platform for processor-based ASICs

As extensively discussed in Chapter 5, the fault simulation step is to date one of the main bottlenecks of the STL development flow (especially when considering functional fault simulation). However, when dealing with safety critical applications functional fault simulation is required also for other purposes.

While in Chapter 5 the focus was on techniques to reduce the fault simulation time when dealing with STLs, in this chapter the intent is different. Indeed, a possible alternative to fault simulation is the fault emulation. The fault emulation is normally implemented resorting to programmable logic devices, such as FPGAs. These are interconnected via high-speed links in a cluster to form emulators.

Specifically, in this chapter an emulation platform based on the already-existing JTAG infrastructure is proposed. Differently than in the existing works that focus on the fault injection mechanism, the focus is on the fault detection mechanism that allows to mimic (including fault dropping) the one found in fault simulators.

It is worth noting that the discussed platform is applicable in all the steps that require fault simulation for dependability analyses (not only the FMEDA performed in the automotive domain). Indeed, it is actually an alternative to fault simulation when performing dependability analyses of processor-based ASICs intended for safety-critical applications.

In the following the discussion is developed considering as use case the typical dependability analysis that a processor-based ASICs undergo. The stuck-at fault models is considered, however the methodology is applicable to transient fault models too (being the two most popular fault models adopted for these analyses). Section 6.1 provides motivations, differences with respect to related works and then the adopted fault injection mechanism presented in [85] for stuck at faults (which, once again, is not the focus of this research). Section 6.2 introduces the proposed platform, while Section 6.3 the experimental results.

The content of this chapter was published in [30].

6.1 Background

6.1.1 Introduction

Safety-critical processor-based ASICs normally require long and expensive dependability analyses before being adopted. The goal is to ensure that the system will deliver the expected services even in presence of hardware faults. The most popular approaches used in industry resort to fault injection campaigns, performed with fault simulators. In these kinds of analyses, the fault simulation process consists of injecting one (sometimes even multiple) faults in the design (either memory element or logic gates) and observe the behaviour of the system while executing a given workload (i.e., an application program that resembles the final one when in field). Mainly stuck-at or transient fault models (SEU and SET) are considered.

The general flow consists of an initial fault simulation, which identifies the most critical faults that exist within the design. Successively, depending on the safety requirements, hardening mechanisms (both hardware and software) against such critical faults are devised. Eventually, the fault simulation is repeated to verify the effectiveness of the chosen hardening mechanisms. This process might be iterated several times until the requirements are met, which implies that several fault simulations are required. However, fault simulation slows down significantly when performing dependability analyses. For this reason, FPGA-based emulators are an attractive solution for overcoming such limitations. Design emulation is today part of the standard design process of complex ASICs. Most notably, it is often used for design verification, pre-silicon validation and early firmware development.

6.1.2 Related works and motivations

In the literature, there exist several approaches for implementing fault emulation. One category of these approaches consists of directly altering the FPGA bitstream [57, 21, 84]. These methods have the advantage that the entire FPGA is available for the design, therefore the hardware overhead is in practice null. On the other hand, each fault injection asks for reprogramming the device, which requires several tens of seconds. Another drawback stems from the fact that FPGA vendors do not disclose information concerning how a given design is mapped to the FPGA blocks and translated into the bitstream. With the above-mentioned approaches, trying to emulate the same fault models adopted for dependability analyses in ASICs may not be feasible through FPGA emulation. Indeed, it requires an additional overhead to understand how to map in the bitstream faults that resemble the actual ones. Therefore, altering the bitstream is unfeasible for complex designs and large emulators (which are made by multiple FPGAs wired together). Finally, these approaches do not allow to inject transient faults (e.g., SET or SEU) but only stuck-at permanent faults.

The other category is based on circuit instrumentation [85, 29, 26, 8, 28]. The original netlist is converted into a fault-injectable netlist. All the faults of interest are programmed at once in the FPGA: therefore, reconfiguration is not required. These approaches allow to inject stuck-at [85], SEU [28, 8, 26] and SET [29] while also begin compatible with the technology offered by the modern emulators.

The main drawback stemming from the adoption of these approaches is that the fault injection mechanism implies a non-negligible hardware overhead. Additionally, some pins are added to the design in order to control the injection.

Likewise with fault simulation, also when performing fault emulation, the fault detection mechanism is a crucial aspect. In this context, the fault detection mechanism refers to the method used by the fault simulator (or emulator) to determine whether a given fault can be labelled as detected or not.

The approaches presented in [85, 26] monitor the functional output or the internal flip-flop at each clock cycle. These approaches are not viable, since they require a huge amount of data to be transferred and stored outside the FPGA. In [29, 28, 8] the idea is to instrument the software executed by the processor during the injection. At the end of execution, the results of the program are internally checked by the system itself. There are three main problems with this approach:

1. the emulation of one fault requires the entire execution of the workload;
2. it might happen that there could be discrepancies (in terms of fault detected/not detected) with respect to the results of a fault simulator;
3. there could be a loss of accuracy since the produced data are exclusively observed (while completely missing internal signals activity).

For overcoming the above-mentioned limitations, the goal of this chapter is to propose a fault emulation platform to support dependability analysis of ASICs. However, rather than on the fault injection mechanism (well addressed in the aforementioned works), the focus is on the fault detection mechanism. The idea is to resemble the one existing in the modern fault simulators, including fault dropping capabilities described in Chapter 5. To avoid increasing the pincount of the design, the proposed fault emulation platform is based on the already-existing JTAG infrastructure. This also makes the whole platform easily applicable with modern emulators since it does not require dedicated hardware for the communication. The JTAG infrastructure for performing fault injection was already exploited by the authors of [79]. However, in that work the purpose was injecting transient faults (i.e., bit flips) in an already manufactured device exploiting the access to the internal scan chains. In this work, the idea is to integrate the fault injection mechanism in the JTAG infrastructure and communicate with it with the JTAG protocol.

6.1.3 The fault injection mechanism

The fault injection mechanism used in this chapter is based on the one presented in [85]. It is worth noting that the proposed platform is applicable to any fault injection mechanisms based on netlist instrumentation. The approach [85] consists of instrumenting the original gate-level netlist by means of Fault Injectable Elements (FIEs). Each FIE, placed right before the logic gate inputs and output, allows to inject one stuck-at fault at a time. The injection is controlled by the structure depicted in Figure 6.1. To reduce the hardware overhead, several pairs of counter-decoder and one demultiplexer are used. Each pair of counter-decoder controls the injection of a subset of faults. As it can be seen, the whole control structure requires two signals only, plus a separate reset line (not shown for simplicity) to be added to the design. A pulse in the Scan_Clock signal inject one fault in the design by activating one FIE control signal. As long as the control signal is active, the output is forced at the logic value 1. Contrarily, when control signal is inactive, the FIE is transparent and the input value is reflected at the output. A pulse in the Select_Clock selects another pair counter-decoder, which controls a different set of FIEs.

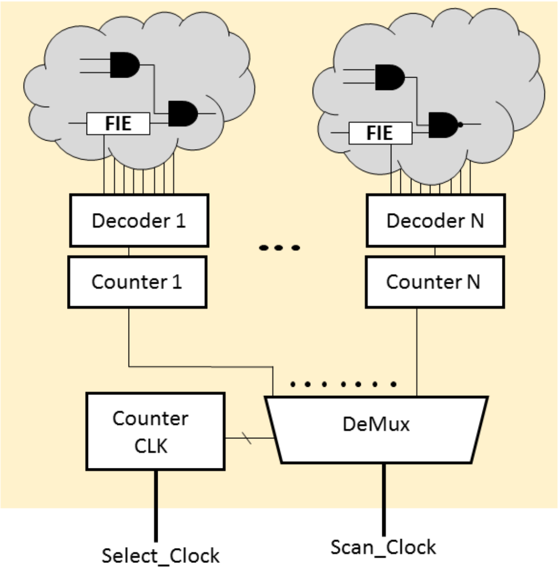


Figure 6.1: The fault injection mechanism described [85]. The other design inputs and outputs are not shown here for simplicity.

6.1.4 Terminology

In this subsection, the basic terminology already introduced in Chapter 5 is recalled. When performing fault simulation, the fault simulator is instructed to observe specific design signals (strobe points) at specific simulation time instants (strobe times). At the specified strobe times, the fault simulator compares the actual strobe points values with the one of the fault-free design. In case of a mismatch, the fault is no longer simulated (dropped). This method is known as fault dropping [39]. When correctly configured, it is possible to reduce the fault simulation time. For example, when assessing the dependability of the processor core, the system bus signals are normally monitored. For the strobe time, several options are possible. Ideally, the more often (e.g., at each clock cycle) the strobe points are checked, the higher will be the speed up of the fault simulation (as demonstrated in Chapter 5). However, this is not generally true. Indeed, checking the strobe points values requires a non-negligible amount of CPU time. During this time, the fault simulation is not proceeding. Therefore, unless faults manifest themselves often, a large amount of time is potentially wasted in checking strobe points values. In general, configuring correctly the fault dropping mechanisms requires engineers' expertise.

6.2 The fault emulation platform

6.2.1 The global architecture

The high-level schematic of the proposed fault emulation platform is shown in Figure 6.2. Apart from the clock and reset generation module and the TAP controller, it is possible to identify three main blocks. The ASIC domain is the fault-injectable gate-level netlist of the targeted ASIC. Considering the fault injection mechanism described earlier, in this domain the design, all the control structures and FIEs depicted in Figure 6.1 are contained. It is worth noting that the ASIC Domain is the only portion of the circuit that contains the fault injection elements (and thus, it is the only one affected by faults). The two control signals (Select_Clock and Scan_Clock) as well as other dedicated reset signals are driven by the logic contained in a second domain, the Fault Injection Manager. Finally, the third domain is the Observation Domain which is in charge for implementing the fault detection mechanism. Internal ASIC signals (i.e., the strobe points) are extracted and fed to this domain. As it can be seen, both ASIC and Observation Domains have the same clock and reset grids. The Fault Injection Manager runs on a separate (the one of the TAP controller) clock and reset grids.

Each of these domains is part of an IEEE 1149.1 JTAG network and accessible via dedicated JTAG instructions. Within each domain, there is a JTAG register able to receive data from the JTAG interface and interact with the internal logic. It is worth noting that in the case of the ASIC domain, these JTAG registers are the original ones for debugging and testing the design. Additionally, the only pins required to be controlled externally are the system clock and reset (CLK and

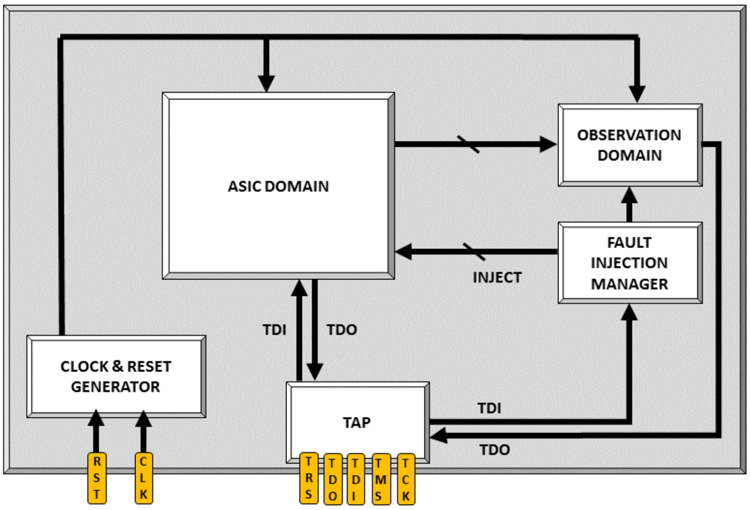


Figure 6.2: The proposed JTAG-based infrastructure. Only relevant pins are shown.

RST pins) plus the five JTAG wires (TRS, TDO, TDI, TMS, TCK). Given this infrastructure, the injection of faults is reduced to the programming of the JTAG register within the Fault Injection Manager with the sequence specified by the selected fault injection mechanisms. It is important to notice that this structure can be adapted to any fault injection mechanism requiring signals driven from outside the emulator.

6.2.2 The observation domain

Figure 6.3 represents the schematic of the Observation Domain. Ideally, fault simulators have the capability of observing strobe points at each clock cycle. In order to achieve the same behaviour, the strobe points (Observed Signals in Figure 6.3) values are compacted with a time (or infinite memory) compactor. The compaction is implemented with a dedicated Multiple Input Signature Register (MISR) [92]. MISRs are normally used for compacting test responses of BIST mechanisms. The most important feature of a MISR is the ability to compact several hundreds of thousands of clock cycles of data into a small signature (e.g., 32-bit long). In case a fault effect is propagated up to the observed signals, it is latched in the compactor and will remain there until another error erase it (aliasing). However, when properly implemented (i.e., when using primitive polynomial compactors) the probability of aliasing is null. With this approach is possible to have a clock cycle resolution of the circuit activity during the fault emulation but without moving huge amount of data outside the emulator. Since an error injected in the compactor will persist in the signature, it is possible to implement fault dropping with a Repetitive Timer and a FIFO memory. The timer is configured through the lower portion of the internal JTAG register (which exclusively drive the internal logic) to expire every k clock cycles.

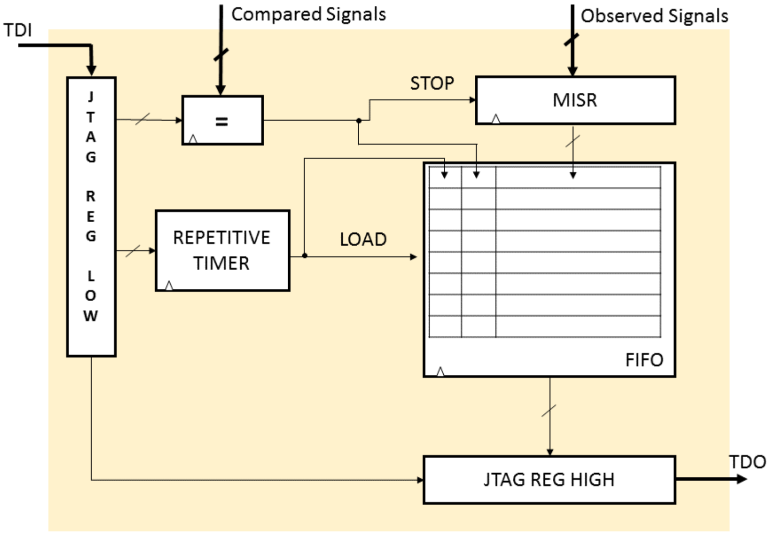


Figure 6.3: Internal structure of the Observation Domain.

Whenever it expires, it generates the LOAD signal that captures a new partial signature into the FIFO. It is worth noting that the partial signature reflects the circuit status at the clock cycle in which it is captured. As it can be seen, the LOAD signal is stored in the FIFO along with the MISR value. During the fault emulation, the external interface continuously read the higher portion of this JTAG register (which exclusively observe the internal logic). Whenever the LOAD bit is set, the partial signature is valid and can be compared with the golden one. The FIFO also maintains the correct sequence in the signatures produced and read, avoiding losing signatures due to a slow external interface. Furthermore, in the FIFO an additional bit is stored, the STOP bit. This bit is generated with a comparator and it is used for signalling the end of the emulation. The end of emulation condition is programmed via JTAG as well, and it could be any value appearing on Compared Signals. For example, in case of a microprocessor executing software, the Compared Signals could be the processor Program Counter. When a specific address is produced (i.e., a specific instruction is going to be fetched) the STOP is generated and the MISR stops recording the Observed Signals.

6.2.3 The fault emulation flow

Given this architecture described in Figure 6.2 and 6.3, the procedure for performing a full fault emulation campaign is shown in Listing 6.2. Preliminary, a

Golden Emulation is performed. It aims at recording all the partial golden signatures to be used successively. The external interface performs the steps GE1-7. It is worth mentioning that during Step GE3, the JTAG register of the Fault Injection Manger is not altered since no injections are required. The emulation and partial signatures recording terminate when the STOP bit is set. After having recorded all the partial signatures, the Fault Emulation begins. The steps FE1-6 compose the Fault Emulation. Differently than in the Golden Emulation, the JTAG infrastructure does not require a reset. Similarly, the JTAG register of the Observation Domain is not altered since already configured during the Golden Emulation. As it can be seen in Step FE6, the fault emulation proceeds until a mismatch in the partial signature occurs (implementing in practice the fault dropping).

```
        /* ******* GOLDEN EMULATION ******** */
/* GE1 */
ActivateReset();
ActivateResetJTAG();
/* GE2 */
ReleaseResetJTAG();
/* GE3 - Program Repetitive Timer and STOP Generation */
ProgramObservationDomain(REG_VAL);
/* GE4 - ASIC and Observation Domain are active */
ReleaseReset();
/* GE5 - Start recording partial signatures */
CurrentPointer = 0;
do
{
    /* GE6 - Read JTAG Register of Observation Domain */
    ValueRegJTAG = ReadObservationDomainJTAG();
    if (ValueRegJTAG[LOAD] == 1)
    {
        /* GE7 - Retrive the new partial signature */
        Signatures[CurrentPointer]=ValueRegJTAG[Signature];
        CurrentPointer++;
    }
} while (ValueRegJTAG[STOP] != 1);

        /* ******** FAULT EMULATION ******** */
Detected = 0;
for (i = 0; i < TotFaults; i++)
{
    /* FE1 */
    ActivateReset();
    /* FE2 - Inject Fault */
    ProgramFaultInjectionManager(REG_VAL);
    /* FE3 - ASIC and Observation Domain are active */
    ReleaseReset();
    /* FE4 - Start recording partial signatures */
    CurrentPointer = 0;
    do
    {
        /* FE5 - Read JTAG Register of Observation Domain */
        ValueRegJTAG = ReadObservationDomainJTAG();
        if (ValueRegJTAG[LOAD] == 1)
        {
            /* FE6 - Compare the partial signature */
            Diff=Signatures[CurrentPointer]-ValueRegJTAG[Signature];
            if (Diff != 0)
            {
                Detected++;
            }
            CurrentPointer++;
        }
    } while (Diff == 0 && ValueRegJTAG[STOP] == 0);
}
```

Listing 6.1: C-like pseudo-code of the fault injection flow.

6.3 Experimental results

The proposed emulation platform was validated with a Xilinx FPGA model xc7z020clg4000-1. As case study, the 8051 microcontroller was selected and implemented with a 65nm ASIC technology. For the experiments, stuck-at faults within the processor were exclusively targeted (about 50k faults). The Observation Domain was configured as follows: as strobe points, the system bus signals. These signals produce a 32-bit signature, which is also the MISR width. Consequently, the FIFO memory had a depth of 32 words, each word 34-bit long (32 bits for the MISR value, 2 bits for STOP and LOAD flags). As signals to be compared for generating the STOP signal of Figure 6.3, the program counter was selected (12 bits). Finally, a 32-bit configuration was selected for the Repetitive Timer. In Table 6.1 the post-layout results in terms of FPGA resources utilization is reported. Most of the resources required are for the ASIC Domain (i.e., the fault-injectable netlist). Instead, the overhead of the proposed platform is relatively modest. It requires 1,445 LUTs and 1,459 flip-flops. The FIFO within the Observation Domain is most demanding one. As workload for the analysis, an application program that generates the first 15 elements of the Fibonacci Series was selected (which lasts slightly more than 2k clock cycles).

The first set of experiments compares the results produced by the proposed fault emulator and a commercial fault simulator (Z01X) in terms of fault detected. The only discrepancies (less than the 1%) that were found concern how unknown (i.e., X) logic values are treated. In fault simulation, the Xs at strobe points cause the current fault to be marked as potentially detected. In real hardware, the X value does not exist. However, it is worth noting that this is a known limitation [85] of fault emulation in FPGAs. For the remaining faults, there was a complete

Table 6.1: FPGA implementation details

Module	**LUT**	**FF**
TAP Controller	184	75
Fault Inj. Manager	40	22
Observation Domain	1,221	1,362
ASIC Domain	27,192	3,352
Total	28,637	4,811

match between the two. The second set of experiments focused (Figure 6.4) on the use and configuration of the fault dropping mechanism. Four different runs were performed, each differing for the number of partial signatures produced. In the first one, partial signatures were not used (one final signature). This can be achieved by using an expiration time greater than the workload duration (i.e., 2k clock cycles). The second one, only one partial signature was used (the timer expires after 1.5k clock cycles). In the third one, two partial signatures were used (expiration set to 1k clock cycles). In the last one, the Repetitive Timer was configured to expire every 500 clock cycles, producing 4 partial signatures.

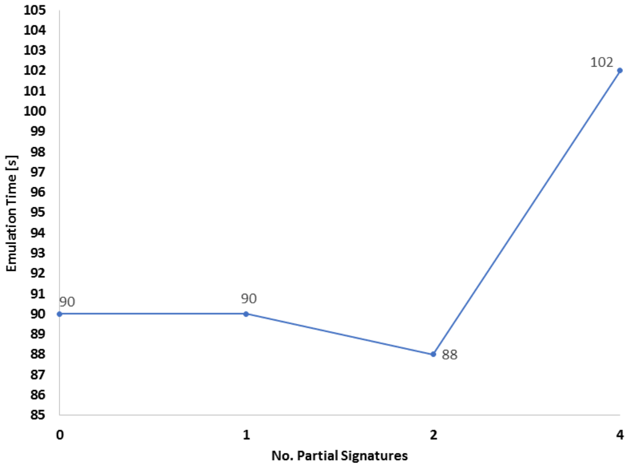


Figure 6.4: Emulation Time vs Number of partial signatures.

As shown, it is possible to reduce the execution time (by 2 seconds) even for short workload as the one selected for the experiments. Selecting one partial signature did not yield appreciable advantages, since the timer expiration time is too close to the end of the emulation. In case of four partial signatures, the experiment produced an outcome similar to the one that can be expected with fault simulators (i.e., when trying to drop faults too often). Indeed, the duration of the workload is too short to amortize the timing overhead required for shifting out and comparing 4 partial signatures plus the final one. Thus, this last case clearly shows that the fault dropping configuration is a critical aspect that should not be overlooked.

The reasons for a better execution time with two partial signatures can be found in the plot of Figure 6.5 derived through Z01X (which allows for a clock cycle resolution, unlike the proposed emulator). Figure 6.5 plots the histogram of the fault detection instants and its cumulative distribution function. Nearly the 80% of the detected faults already produces faulty effects at the strobe points after 1k clock cycles. It is also noteworthy that only 12k faults are detected. Thus, the 76% of the faults do not produce a failure. Therefore, for the 76% of the cases comparing the partial signatures is totally useless.

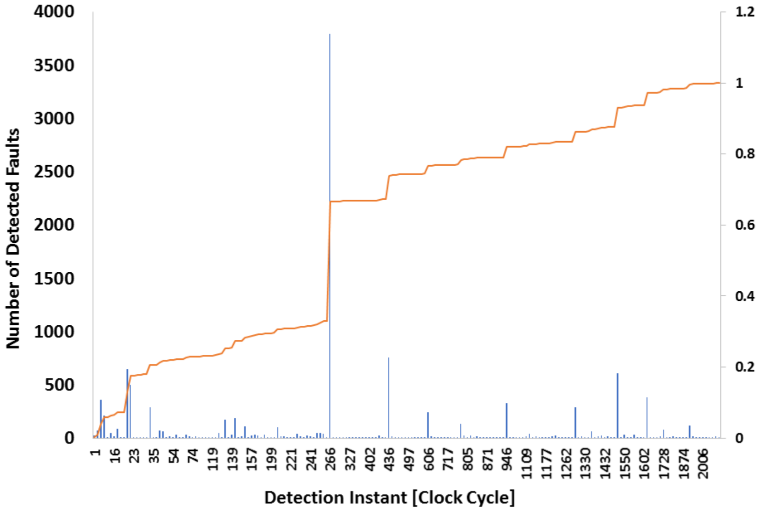


Figure 6.5: Histogram and cumulative distribution function of the detection instants (expressed in clock cycles).

Chapter 7

Conclusions and future directions

In this manuscript different aspects of the self-testing for automotive MPSoCs were addressed. Mainly, the software-based ones were considered since in the state of the art, most of the recent research efforts were directed to the hardware-based ones. Beside, also new possible approaches such as the hybrid one were explored. The second part of the thesis dealt with the improvement of the methodologies for the functional fault grading, being one of the major bottlenecks in the development of any self-test mechanisms (not only for the automotive domain) when the device complexity increases (as it happens with MPSoCs).

Concerning the first part, it was described a cache-based approach for achieving a deterministic execution of self-test procedures when executed in field in a MPSoC (Chapter 2). The proposed methodology is able to deliver stable signature and deterministic fault coverage, most notably without requiring additional on-chip resources. Through the experiments, it was demonstrated the applicability to any self-test procedure without altering its overall memory footprint. On the other hand, as the experiments confirmed it requires slightly more clock cycles to be executed compared to other strategies (e.g., the TCM-based ones).

The problematic discussed in that chapter opens a new interesting research area. While considering stuck-at faults, few specific test programs exhibit these issues in a multi-core execution. Instead, it might be further emphasized with delay faults which require test patterns applied in a timed sequence. Further research efforts should be directed toward this end, which might be of particular interest in emerging research areas such as the one of the System-Level Test (SLT) [78].

Different multi-core decentralized schedulers were introduced (Chapter 3). These are intended to be applied in different scenarios (i.e, single/multi-resource and homogeneous/heterogeneous) and compatible with the modern industrial development flow for STLs [13]. The decentralized approach has been shown to be a valid alternative in multi-core systems, which are highly non-deterministic themselves. Instead of controlling which test programs is executed by each core at any given time, each core independently executes its test programs. Besides reducing the overall test application time when in field, the selfish decentralized scheduler also yields a better predictability in terms of execution time when compared to other solutions. Concurrently, the memory overhead is minimized, as most of the data structures resides in the stack memory without replication of code and/or data. The proposed decentralized architecture allows to embed different schedulers in the same system, as they are completely independent. That is, it promotes also code re-usability, which is an attractive feature to have when dealing with modern automotive systems.

Through the experiments, it was shown that any multi-core scheduler is inevitably affected by the underlying system architecture (i.e., architecture of the interconnect, memory hierarchy, parallelism of the memories). In this manuscript, the focus was on system interconnect based on crossbar switch. Although they are barely adopted in modern embedded multi-core systems, future research efforts should be directed to analyze shared bus interconnects. Furthermore, more advanced interconnection schemes such as those proposed by the Network-on-Chip philosophy should be investigated as well (even though they have not yet reached the mainstream industrial safety-critical applications).

Several self-test mechanisms for DCLS processors were analyzed in Chapter 4. These self-test mechanisms can be used for the on-line testing of the comparators required for implementing a DCLS configuration. It was proven through the experiments the inadequacy of an STL to address all the possible permanent faults within the comparators. At the same time, it was shown the overhead in terms of area of a pure hardware self-test module. An alternative self-test approach is then introduced: it is based on the insertion into the system of a hardware module (LSMU) that assists test programs for the generation the required test stimuli. Therefore, the proposed strategy is hybrid, since it is based on both software and hardware.

The proposed strategy provides several advantages:

- Low hardware overhead compared to a pure hardware approach: part of the test patterns generator is implemented in software;
- Flexibility: as it is partially based on software, it inherits its flexibility: the test can be split in different test sessions to fit the test time budget when online. Moreover, the test patterns are not fixed anymore and can be updated as required;
- Promotes IP re-usability: the hardware module is the least intrusive as possible. Indeed, it is not required any modification nor a detailed knowledge of the processor core. The system designer oversees the integration of the module in the final SoC, as it happens with a standard IP;
- Scalability: the hardware overhead minimally depends on the complexity of

the considered processor. The ISM depends only on the considered architecture (e.g. 32 or 64-bit), while the CSSM depends on the number of control signals of the processor core.

Additionally, it was described a set of countermeasures to be used against single-point faults that could arise within the LSMU, in order to improve the safety of the module. Latent faults within the LSMU can be addressed following the same approach of a pure hardware module (i.e., LBIST-based approaches).

Concerning the second part of the thesis, to alleviate the effort required for functional fault simulation, in Chapter 5 the focus was on fault dropping strategies. These are fully compatible with modern fault simulators and intended to be integrated in the standard STL development flow.

As possible solution, fault emulation was considered in Chapter 6. This represents the most interesting future direction concerning functional fault grading for dependability dependability analyses. Indeed, emulation might be the answer to the growing complexity of the MPSoCs, complementing fault simulation. In particular, the integration of these methodology in the development flow of software-based safety mechanisms (for implementing both self-testing and general hardening mechanisms) should be the next step. Finally, considering this last topic, since the average operational life of electronic devices increases, hardware fault injection mechanisms enabling the injection of multiple faults should considered as well.

Bibliography

[1] Rangachari, S., Jalan, S.: Self test for safety logic. United States Patent US9964597B2, 8 May 2018.

[2] https://openrisc.io/.

[3] https://github.com/openrisc/or1200.

[4] J. Abraham and S. Thatte. "Test Generation for Microprocessors". In: *IEEE Transactions on Computers* 29.06 (June 1980), pp. 429–441. ISSN: 1557-9956. DOI: 10.1109/TC.1980.1675602.

[5] A. Apostolakis et al. "Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors". In: *2009 14th IEEE European Test Symposium*. 2009, pp. 33–38. DOI: 10.1109/ETS.2009.31.

[6] A. Apostolakis et al. "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors". In: *IEEE Transactions on Computers* 58.12 (2009), pp. 1682–1694. DOI: 10.1109/TC.2009.118.

[7] J. Ax et al. "CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories". In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2018), pp. 1030–1043. DOI: 10.1109/TPDS.2017.2785799.

[8] O. Bailan et al. "Verification of soft error detection mechanism through fault injection on hardware emulation platform". In: *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2010, pp. 113–118. DOI: 10.1109/DSNW.2010.5542611.

[9] R. Banakar et al. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*. 2002, pp. 73–78. DOI: 10.1145/774789.774805.

[10] P. Bernardi, D. Piumatti, and E. Sanchez. "Facilitating Fault-Simulation Comprehension through a Fault-Lists Analysis Tool". In: *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*. 2019, pp. 77–80. DOI: 10.1109/LASCAS.2019.8667573.

[11] P. Bernardi et al. “A DMA and CACHE-based stress schema for burn-in of automotive microcontroller”. In: *2017 18th IEEE Latin American Test Symposium (LATS)*. 2017, pp. 1–6. DOI: 10.1109/LATW.2017.7906767.

[12] P. Bernardi et al. “A new hybrid fault detection technique for systems-on-a-chip”. In: *IEEE Transactions on Computers* 55.2 (2006), pp. 185–198. DOI: 10.1109/TC.2006.15.

[13] P. Bernardi et al. “Development Flow for On-Line Core Self-Test of Automotive Microcontrollers”. In: *IEEE Transactions on Computers* 65.3 (2016), pp. 744–754. DOI: 10.1109/TC.2015.2498546.

[14] P. Bernardi et al. “Fault grading of software-based self-test procedures for dependable automotive applications”. In: *2011 Design, Automation & Test in Europe* (2011), pp. 1–2.

[15] P. Bernardi et al. “MIHST: A Hardware Technique for Embedded Microprocessor Functional On-Line Self-Test”. In: *IEEE Transactions on Computers* 63.11 (2014), pp. 2760–2771. DOI: 10.1109/TC.2013.165.

[16] P. Bernardi et al. “On-line functionally untestable fault identification in embedded processor cores”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 1462–1467. DOI: 10.7873/DATE.2013.298.

[17] P. Bernardi et al. “On-line functionally untestable fault identification in embedded processor cores”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 1462–1467. DOI: 10.7873/DATE.2013.298.

[18] P. Bernardi et al. “On-line software-based self-test of the Address Calculation Unit in RISC processors”. In: *2012 17th IEEE European Test Symposium (ETS)*. May 2012, pp. 1–6. DOI: 10.1109/ETS.2012.6233004.

[19] P. Bernardi et al. “Software-Based Self-Test Techniques for Dual-Issue Embedded Processors”. In: *IEEE Transactions on Emerging Topics in Computing* 8.2 (2020), pp. 464–477. DOI: 10.1109/TETC.2017.2758641.

[20] A. Biondi and M. Di Natale. “Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2018, pp. 240–250. DOI: 10.1109/RTAS.2018.00032.

[21] L. Burgun et al. “Serial fault emulation”. In: *33rd Design Automation Conference Proceedings, 1996*. 1996, pp. 801–806. DOI: 10.1109/DAC.1996.545681.

[22] S. Campagna and M. Violante. “An hybrid architecture to detect transient faults in microprocessors: An experimental validation”. In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 1433–1438. DOI: 10.1109/DATE.2012.6176590.

[23] R. Cantoro et al. “About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications”. In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. 2018, pp. 1–6. DOI: 10.1109/LATW.2018.8349679.

[24] J. Celaya and U. Arronategui. “A Highly Scalable Decentralized Scheduler of Tasks with Deadlines”. In: *2011 IEEE/ACM 12th International Conference on Grid Computing*. 2011, pp. 58–65.

[25] C. L. Chen and M. Y. Hsiao. “Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review”. In: *IBM Journal of Research and Development* 28.2 (1984), pp. 124–134. DOI: 10.1147/rd.282.0124.

[26] P. Civera et al. “Exploiting circuit emulation for fast hardness evaluation”. In: *IEEE Transactions on Nuclear Science* 48.6 (2001), pp. 2210–2216. DOI: 10.1109/23.983197.

[27] J. Daveau et al. “An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip”. In: *2009 IEEE International Reliability Physics Symposium*. 2009, pp. 212–220. DOI: 10.1109/IRPS.2009.5173253.

[28] J. Daveau et al. “An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip”. In: *2009 IEEE International Reliability Physics Symposium*. 2009, pp. 212–220. DOI: 10.1109/IRPS.2009.5173253.

[29] L. Entrena et al. “SET Emulation Considering Electrical Masking Effects”. In: *IEEE Transactions on Nuclear Science* 56.4 (2009), pp. 2021–2025. DOI: 10.1109/TNS.2009.2013346.

[30] A. Floridia and E. Sanchez. “A JTAG-based fault emulation platform for dependability analyses of processor-based ASICs”. In: *2021 12th IEEE Latin America Symposium on Circuits and System (LASCAS)*. In Press, pp. 1–6.

[31] A. Floridia and E. Sanchez. “Hybrid On-Line Self-Test Strategy for Dual-Core Lockstep Processors”. In: *2018 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2018, pp. 1–6. DOI: 10.1109/DFT.2018.8602982.

[32] A. Floridia, E. Sanchez, and M. Sonza Reorda. “Fault Grading Techniques of Software Test Libraries for Safety-Critical Applications”. In: *IEEE Access* 7 (2019), pp. 63578–63587. DOI: 10.1109/ACCESS.2019.2917036.

[33] A. Floridia et al. “A Decentralized Scheduler for On-line Self-test Routines in Multi-core Automotive System-on-Chips”. In: *2019 IEEE International Test Conference (ITC)*. 2019, pp. 1–10. DOI: 10.1109/ITC44170.2019.9000129.

[34] A. Floridia et al. "Deterministic Cache-based Execution of On-line Self-Test Routines in Multi-core Automotive System-on-Chips". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 1235–1240. DOI: 10.23919/DATE48585.2020.9116239.

[35] A. Floridia et al. "Parallel software-based self-test suite for multi-core system-on-chip: Migration from single-core to multi-core automotive microcontrollers". In: *2018 13th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*. Apr. 2018, pp. 1–6. DOI: 10.1109/DTIS.2018.8368558.

[36] Andrea Floridia and Ernesto Sanchez. "On-line self-test mechanism for Dual-Core Lockstep System-on-Chips". In: *Microelectronics Reliability* 112 (2020), p. 113770. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2020.113770. URL: http://www.sciencedirect.com/science/article/pii/S0026271420300317.

[37] N. Foutris et al. "MT-SBST: Self-test optimization in multithreaded multicore architectures". In: *2010 IEEE International Test Conference*. 2010, pp. 1–10. DOI: 10.1109/TEST.2010.5699277.

[38] Eiji Fujiwara. *Code Design for Dependable Systems: Theory and Practical Application*. USA: Wiley-Interscience, 2006. ISBN: 0471756180.

[39] S. Gai and P. L. Montessoro. "The fault dropping problem in concurrent event-driven simulation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.8 (1991), pp. 968–971. DOI: 10.1109/43.85734.

[40] S. Gai, P. L. Montessoro, and F. Somenzi. "MOZART: a concurrent multilevel simulator". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7.9 (1988), pp. 1005–1016. DOI: 10.1109/43.7799.

[41] M. Gaudesi et al. "New Techniques to Reduce the Execution Time of Functional Test Programs". In: *IEEE Transactions on Computers* 66.7 (2017), pp. 1268–1273. DOI: 10.1109/TC.2016.2643663.

[42] D. Gizopoulos, A. Paschalis, and Y. Zorian. "An effective built-in self-test scheme for parallel multipliers". In: *IEEE Transactions on Computers* 48.9 (1999), pp. 936–950. DOI: 10.1109/12.795222.

[43] D. Gizopoulos et al. "Architectures for online error detection and recovery in multicore processors". In: *2011 Design, Automation Test in Europe*. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763096.

[44] H. Grigoryan et al. "Generic BIST architecture for testing of content addressable memories". In: *2011 IEEE 17th International On-Line Testing Symposium*. July 2011, pp. 86–91. DOI: 10.1109/IOLTS.2011.5993816.

[45] M. Grosso, M. Sonza Reorda, and S. Rinaudo. "Software-Based Self-Test for Delay Faults". In: 586 (2020). DOI: 10.1007/978-3-030-53273-4_1.

[46] F. Hapke et al. "Cell-Aware Test". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.9 (2014), pp. 1396–1409. DOI: 10.1109/TCAD.2014.2323216.

[47] T. Hsieh et al. "Software-hardware-cooperated built-in self-test scheme for channel-based DRAMs". In: *2017 International Test Conference in Asia (ITC-Asia)*. 2017, pp. 107–111. DOI: 10.1109/ITC-ASIA.2017.8097122.

[48] T. Hsieh et al. "Tolerance of performance degrading faults for effective yield improvement". In: *2009 International Test Conference*. 2009, pp. 1–10. DOI: 10.1109/TEST.2009.5355594.

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

[50] *Road vehicles – Functional safety*. Standard. International Organization for Standardization, 2011.

[51] Xabier Iturbe et al. "The Arm Triple Core Lock-Step (TCLS) Processor". In: *ACM Trans. Comput. Syst.* 36.3 (June 2019). ISSN: 0734-2071. DOI: 10.1145/3323917. URL: https://doi.org/10.1145/3323917.

[52] M. Kaliorakis et al. "Online error detection in multiprocessor chips: A test scheduling study". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. July 2013, pp. 169–172. DOI: 10.1109/IOLTS.2013.6604071.

[53] O. Khan and S. Kundu. "Hardware/Software Codesign Architecture for Online Testing in Chip Multiprocessors". In: *IEEE Transactions on Dependable and Secure Computing* 8.5 (Sept. 2011), pp. 714–727. ISSN: 2160-9209. DOI: 10.1109/TDSC.2011.19.

[54] O. Khan and S. Kundu. "Thread Relocation: A Runtime Architecture for Tolerating Hard Errors in Chip Multiprocessors". In: *IEEE Transactions on Computers* 59.5 (May 2010), pp. 651–665. ISSN: 2326-3814. DOI: 10.1109/TC.2009.76.

[55] J. Kim et al. "All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer Memory". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 622–633. DOI: 10.1109/ISCA.2016.60.

[56] N. Kranitis et al. "Software-based self-testing of embedded processors". In: *IEEE Transactions on Computers* 54.4 (2005), pp. 461–475. DOI: 10.1109/TC.2005.68.

[57] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. “Fault emulation: a new approach to fault grading”. In: *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. 1995, pp. 681–686. DOI: 10.1109/ICCAD.1995.480203.

[58] Li Chen and S. Dey. “Software-based self-testing methodology for processor cores”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.3 (2001), pp. 369–380. DOI: 10.1109/43.913755.

[59] Y. Li, O. Mutlu, and S. Mitra. “Operating system scheduling for efficient online self-test in robust systems”. In: *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. Nov. 2009, pp. 201–208.

[60] Y. Liu et al. “Deterministic Stellar BIST for In-System Automotive Test”. In: *2018 IEEE International Test Conference (ITC)*. 2018, pp. 1–9. DOI: 10.1109/TEST.2018.8624872.

[61] M. Lv et al. “Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software”. In: *2010 31st IEEE Real-Time Systems Symposium*. 2010, pp. 339–349. DOI: 10.1109/RTSS.2010.30.

[62] R. Mariani, G. Boschi, and F. Colucci. “Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508”. In: *2007 Design, Automation Test in Europe Conference Exhibition*. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364641.

[63] Riccardo Mariani, Federico Colucci, and Peter Fuhrmann. “Safety Integrity of Memory Sub-Systems in Automotive Microcontrollers”. In: *SAE Transactions* 116 (2007), pp. 486–496. ISSN: 0096736X, 25771531. URL: http://www.jstor.org/stable/44719916.

[64] Riccardo Mariani, Thomas Kuschel, and Hiroshi Shigehara. “A flexible microcontroller architecture for fail-safe and fail-operational systems”. In: (Nov. 2020).

[65] P. C. Maxwell et al. “THE EFFECT OF DIFFERENT TEST SETS ON QUALITY LEVEL PREDICTION: WHEN IS 80% BETTER THAN 90%?” In: *1991, Proceedings. International Test Conference*. 1991, pp. 358–. DOI: 10.1109/TEST.1991.519695.

[66] T. McLaurin. “Periodic Online LBIST Considerations for a Multicore Processor”. In: *2018 IEEE International Test Conference in Asia (ITC-Asia)*. 2018, pp. 37–42. DOI: 10.1109/ITC-Asia.2018.00017.

[67] S. Milewski et al. “Full-scan LBIST with capture-per-cycle hybrid test points”. In: *2017 IEEE International Test Conference (ITC)*. 2017, pp. 1–9. DOI: 10.1109/TEST.2017.8242036.

[68] E. Moghaddam et al. "Test point insertion in hybrid test compression/LBIST architectures". In: *2016 IEEE International Test Conference (ITC)*. 2016, pp. 1–10. DOI: 10.1109/TEST.2016.7805826.

[69] N. Mukherjee et al. "Ring Generator: An Ultimate Linear Feedback Shift Register". In: *Computer* 44.6 (2011), pp. 64–71. DOI: 10.1109/MC.2010.334.

[70] N. Mukherjee et al. "Time and Area Optimized Testing of Automotive ICs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020), pp. 1–13. DOI: 10.1109/TVLSI.2020.3025138.

[71] O. Mutlu and T. Moscibroda. "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 146–160.

[72] M. Nakazato et al. "Design for Testability of Software-Based Self-Test for Processors". In: *2006 15th Asian Test Symposium*. 2006, pp. 375–380. DOI: 10.1109/ATS.2006.260958.

[73] A. Nardi and A. Armato. "Functional safety methodologies for automotive applications". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 970–975. DOI: 10.1109/ICCAD.2017.8203886.

[74] M. Nicolaidis. "Theory of transparent BIST for RAMs". In: *IEEE Transactions on Computers* 45.10 (1996), pp. 1141–1156. DOI: 10.1109/12.543708.

[75] T. M. Niermann, W. -. Cheng, and J. H. Patel. "PROOFS: a fast, memory efficient sequential circuit fault simulator". In: *27th ACM/IEEE Design Automation Conference*. 1990, pp. 535–540. DOI: 10.1109/DAC.1990.114913.

[76] Antonis Paschalis and Dimitris Gizopoulos. "Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. DATE '04. USA: IEEE Computer Society, 2004, p. 10578. ISBN: 0769520855.

[77] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269.

[78] I. Polian et al. "Exploring the Mysteries of System-Level Test". In: *2020 IEEE 29th Asian Test Symposium (ATS)*. 2020, pp. 1–6. DOI: 10.1109/ATS49688.2020.9301557.

[79] M. Portela-García et al. "Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures". In: *IEEE Transactions on Dependable and Secure Computing* 8.2 (2011), pp. 308–314. DOI: 10.1109/TDSC.2010.50.

[80] F. Pratas et al. “Measuring the effectiveness of ISO26262 compliant self test library”. In: *2018 19th International Symposium on Quality Electronic Design (ISQED)*. 2018, pp. 156–161. DOI: 10.1109/ISQED.2018.8357281.

[81] M. Psarakis et al. “Microprocessor Software-Based Self-Testing”. In: *IEEE Design Test of Computers* 27.3 (2010), pp. 4–19. DOI: 10.1109/MDT.2010.5.

[82] F. Reimann et al. “Advanced diagnosis: SBST and BIST integration in automotive E/E architectures”. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2014, pp. 1–6. DOI: 10.1145/2593069.2602971.

[83] A. Riefert et al. “A Flexible Framework for the Automatic Generation of SBST Programs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (2016), pp. 3055–3066. DOI: 10.1109/TVLSI.2016.2538800.

[84] E. Sanchez, L. Sterpone, and A. Ullah. “Effective emulation of permanent faults in ASICs through dynamically reconfigurable FPGAs”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–6. DOI: 10.1109/FPL.2014.6927478.

[85] Shih-Arn Hwang, Jin-Hua Hong, and Cheng-Wen Wu. “Sequential circuit fault simulation using logic emulation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.8 (1998), pp. 724–736. DOI: 10.1109/43.712103.

[86] P. Singh, D. L. Landis, and V. Narayanan. “Test Generation for Precise Interrupts on Out-of-Order Microprocessors”. In: *2009 10th International Workshop on Microprocessor Test and Verification*. 2009, pp. 79–82. DOI: 10.1109/MTV.2009.14.

[87] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael. “DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors”. In: *IEEE Transactions on Computers* 65.5 (2016), pp. 1453–1466. DOI: 10.1109/TC.2015.2449840.

[88] J. E. Smith and A. R. Pleszkun. “Implementing precise interrupts in pipelined processors”. In: *IEEE Transactions on Computers* 37.5 (1988), pp. 562–573. DOI: 10.1109/12.4607.

[89] G. Tshagharyan, G. Harutyunyan, and Y. Zorian. “An effective functional safety solution for automotive systems-on-chip”. In: *2017 IEEE International Test Conference (ITC)*. 2017, pp. 1–10. DOI: 10.1109/TEST.2017.8242075.

[90] Wei-Cheng Lai and Kwang-Ting Cheng. “Instruction-level DfT for testing processor and IP cores in system-on-a-chip”. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 59–64. DOI: 10.1109/DAC.2001.156108.

[91] B. Wilkinson. “On crossbar switch and multiple bus interconnection networks with overlapping connectivity”. In: *IEEE Transactions on Computers* 41.6 (1992), pp. 738–746.

[92] P. Wohl, J. A. Waicukauski, and T. W. Williams. “Design of compactors for signature-analyzers in built-in self-test”. In: *Proceedings International Test Conference 2001 (Cat. No.01CH37260)*. 2001, pp. 54–63. DOI: 10.1109/TEST.2001.966618.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.