IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 10, 2019, accepted May 8, 2019, date of publication May 15, 2019, date of current version May 28, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2917036

Fault Grading Techniques of Software Test Libraries for Safety-Critical Applications

ANDREA FLORIDIA, (Student Member, IEEE), ERNESTO SANCHEZ, (Senior Member, IEEE), AND MATTEO SONZA REORDA, (Fellow, IEEE)
Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Andrea Floridia (andrea.floridia@polito.it)

ABSTRACT The adoption of complex and technologically advanced integrated circuits (ICs) in safety-critical applications (e.g., in automotive) forced the introduction of new solutions to guarantee the achievement of the required reliability targets. One of these solutions lies in performing in-field test (i.e., the test performed when the device is already deployed in the mission environment) to detect faults that may arise in this phase of electronic circuit life. In this scenario, one increasingly adopted approach is based on the software test libraries (STLs), i.e., suitable code which is run by the CPU included in the system and is able to detect the existence of possible permanent faults both in the CPU itself and in the rest of the system. In order to assess the effectiveness of the STLs, fault simulation is performed, so that the achieved fault coverage (e.g., in terms of stuck-at faults) can be computed. This paper explains why the 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), shows why it can be highly computationally expensive, and overviews some solutions to reduce the computational cost and possibly trade-off between results accuracy and cost.

INDEX TERMS Functional safety, fault simulation, software test libraries, ISO 26262.

I. INTRODUCTION

Dependability of electronic systems has been a major concern since decades, leading to technical solutions which have been extensively used, for example in domains such as space, aircrafts, military, nuclear plant control. More recently, new application domains such as automotive, biomedical or telecommunications drastically changed the scenario to be considered (e.g., by emphasizing parameters such as cost and Time To Market), and forced the adoption of new solutions for the development of dependable electronic systems. However, the continuous evolution in terms of target applications and adopted semiconductor technologies accelerated even further the request for new techniques able to support the designer of these kinds of electronic systems. In particular, the last period is experiencing further changes in this domain, due to the massive adoption of electronic systems for applications where the requirements in terms of safety are very strict, while the high performance required asks for advanced (and thus less reliable) semiconductor technologies and architectures (e.g., multi-core ones). As a result, the so-called functional safety standards (as the ISO 26262 for the automotive domain, or in general the IEC 61508 for safety-critical applications) appeared in the last decade to regulate the usage of electronics devices in these domains. Such standards impose reliability figures which are normally achieved by leveraging different safety mechanisms. According to the ISO 26262 (in the following it is used as reference, but similar concepts can be found in any functional safety standard), a *safety mechanism* is a portion of the system oriented to the detection of faults, controlling system failures in order to achieve and/or maintain a safe state. The most commonly adopted safety mechanisms are:

- Logic and Memory BIST (LBIST and MBIST respectively): they are mainly intended for in-field testing, but also used for end-of-manufacturing test;
- End-to-End Error Correction Codes (ECCs) for protecting memories against bit flips due to radiations;
- Dual-Core LockStep (DCLS): two processor cores (the main and the checker respectively) are paired together and always fed with the same identical inputs. Their outputs are continuously monitored by a set of comparators, so that any failure in one of the two processors can be immediately detected;
- Software Test Libraries (STLs), which are a set of Software-Based Self-Test (SBST) routines commonly used for in-field testing.

The associate editor coordinating the review of this manuscript and approving it for publication was Luca Cassano.

ECC and Lockstep are the de-facto standard solutions against the *Single-Point Faults* for the highest reliability applications (ASIL C and D according to the ISO 26262). Such faults would directly cause critical failures without a suitable safety mechanism guarding for them. However, *Latent Faults* (i.e., faults not causing directly a failure, but that can become dangerous in conjunction with a second fault) could invalidate the functionalities of these safety mechanisms. BIST-based solutions are adopted for the *Power-On Self-Test* (POST) that is the in-field test performed when the device is powered up. Although being particularly effective solutions, the applicability of these solutions is limited to the POST since they are normally invasive solutions that damage the system status, and the test application time could be quite long. This could be problematic if the time interval between two consecutive power-on is long. For these reasons, STLs represent a better solution for the on-line testing, which is the in-field test performed concurrently with the user application (e.g., an Operating System). When dealing with on-line testing, the test phase must fit in relatively short periods (in the order of tens of milliseconds, or less), while intrusiveness should be minimal.

The idea behind procedures composing STLs is relatively simple and known since decades [13], [14], [27], [28]: if 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. By looking at the produced output data, one can detect a number of faults possibly existing within the processor module (and around it). By adopting suitable techniques, the obtained fault coverage can achieve sufficiently high values.

The advantages offered by this solution for in-field test have pushed producers of devices to be used in safety-critical applications (e.g., in the automotive sector) to deliver libraries of such Self-test procedures (sometimes known as Software Test Libraries, or STLs). These procedures are developed by the semiconductor company, which owns all the structural information about the device and can guarantee that their execution allows achieving a given figure in terms of Fault Coverage. Once available, Self-test procedures are then integrated by the system company (often corresponding to a Tier1 company in the automotive domain) into the application software and invoked when required (e.g., periodically, or during the application idle times). In this way, the required safety level can be obtained without transferring any sensible information from the semiconductor to the system company. Thus, practical industrial applications of STLs range from the on-line test of processor cores for ASIL A or B applications (the lowest level of safety level according to the ISO 26262) to the test of latent faults in safety mechanism as the DCLS and the ECC [24] for ASIL C or D applications. For the DCLS, STLs are particularly suitable for avoiding latent faults accumulation in both the checker and the main core, that could cause failures being masked. This approach is today widely supported by many semiconductor and IP companies, such as Infineon [1], STMicroelectronics [2], Cypress [3], Renesas [4], Microchip [5], ARM [6].

When working at the development of a Software Test Library for a given device, the test engineer must face two main issues:

- How to effectively write their code, minimizing the related effort (e.g., by following some guidelines, while clearly re-using the code developed for previous versions of the same device, or for similar ones, if possible)
- How to compute in a time effective and precise manner the fault coverage achieved by the procedures.

While a number of papers focused on the first point, proposing techniques to write Self-test procedures for different modules in a system (e.g., [7]–[10]) and to optimize them [11], this paper focuses on the second point. Computing the Fault Coverage achieved by a given test sequence is typically done resorting to Fault Simulation. In the '90s, a wide research effort led to the development of effective and highly optimized algorithms for fault simulation (e.g., [12]). Such algorithms have been widely adopted in commercial tools (e.g., fault simulators and ATPG tools) which are nowadays used in different commercial environments for generating and grading test patterns for a given digital circuit. However, those tools and methodologies targeted a rather different problem than the one considered in this paper. In fact, traditional fault simulation approaches are intended to compute the fault coverage (typically, with respect to stuck-at faults) provided by a sequence of input vectors applied to a generic sequential circuit, assuming 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-production testing but is very different to the one related to in-field testing.

When considering the effects stemming from the execution of a Self-test procedure, we are focusing on a processor executing a program stored in a memory and producing some output results, also written in memory and observed at the end of the procedure execution. This scenario is quite different than the previous. For example, the sequence of inputs for the processor (e.g., the sequence of executed instructions) often changes due to fault effects, as it 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 if it produced a wrong result in memory at the end of the Self-test procedure execution. This analysis is normally part of the Functional Safety Verification flow [22], [23]. Specifically, the fault grading of STLs becomes relevant during the Failure Modes Effects and Diagnostic Analysis (FMEDA) [23]. FMEDA is a structured approach that aims at defying failure modes, computing the failure rate, devising proper safety mechanisms for the identified failures and measuring their effectiveness. For better fitting this scenario, the so-called *functional fault simulators* recently appeared on the market (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.

The purpose of this paper is many-fold. First and foremost, we want to emphasize the inadequacy of traditional fault simulation methodologies to effectively compute the fault coverage produced by a library of Self-test procedures and clarify the differences between sequential circuit fault simulation (SC-FSIM) and Self-test procedures fault simulation (STP-FSIM). Secondly, we explain the reasons for (and provide experimental evidences of) the huge computational cost required to perform STP-FSIM. Finally, we overview some solutions able to significantly speed-up the STP-FSIM, sometimes at the expenses of a limited loss in accuracy. All the proposed solutions are based on the usage of commercially available EDA tools, thus being easily adoptable by professionals in the field. To the best of our knowledge, with respect to similar works [20], [21], this paper is the first attempt to provide a comprehensive and commented overview about the techniques that can be adopted to effectively perform the fault grading of STLs.

The rest of the paper is organized as follows: in Section 2 the required terminology and background information are provided; in Section 3 and 4 detailed descriptions of possible fault simulation techniques are illustrated; Section 5 presents the gathered experimental results; finally, in Section 6 we discuss the conclusions of the present work.

II. BACKGROUND

In this Section, we aim at pinpointing the differences between the problem faced by traditional Fault Simulation techniques, developed in the past to support sequential circuit fault simulation (typically used for assessing the effectiveness of test solutions for the end-of-manufacturing testing) and the one of computing the Fault Coverage achieved by a Self-test procedure belonging to a Software Test Library (STL) for in-field test. To make the explanation easier, in the following we will assume that the common stuck-at fault model is adopted, although the following concepts can be extended to several other fault models (e.g., the transition delay one).

A. SEQUENTIAL CIRCUIT FAULT SIMULATION (SC-FSIM)

In this case, the goal is to compute the Fault Coverage achieved when a given test sequence is applied to the Circuit Under Test (CUT), which correspond to a combinational or sequential digital circuit (Figure 1). Typically, the test we refer to when performing such a computation is the final test executed at the end of the device manufacturing. In this scenario, the CUT is mounted on an Automatic Test Equipment (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 values, or test patterns, 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.

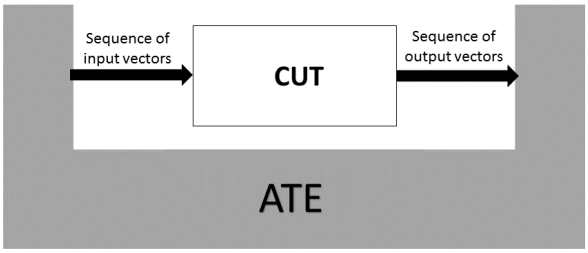

FIGURE 1. Test vectors-based end-of-manufacturing test scenario for a generic sequential CUT.

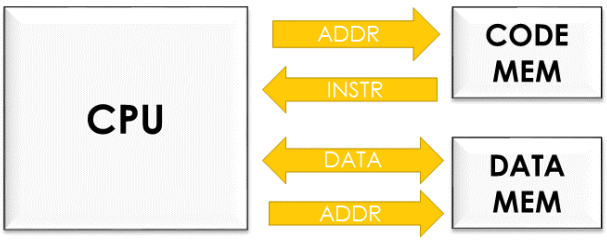

FIGURE 2. STL-based in-field test scenario for a CPU-based system.

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 can be dropped, and it is not necessary anymore to simulate its effects in the following. This mechanism (known as *Fault Dropping* [15]) significantly reduces the computational cost of Fault Simulation, since most of the faults are only simulated for a limited period of time.

B. 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 quite different. This process is also known as *Fault Grading* [21] of Self-test procedures. In particular, in this case the scenario to be considered is more complex, since the typical application of STLs is for in-field test, which is performed when the device is already in the operational phase, without the support of any ATE. In fact, 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 2). For the sake of simplicity, and since I/O operations are conceptually similar to memory operation, we will neglect I/O operations in the

following. In practice, the self-test routines often accumulate the produced results in a single register in order to create a *test signature*. Such signature is then checked by the Self-test code itself against a golden one: a fault is considered as detected if the execution of the test code produced some wrong signature value of the execution. At the end of the test, the Self-test code itself performs the check on the signature and returns by storing a flag in the memory, stating whether a fault has been detected and/or the computed signature. Hence, the following statements 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 should execute;
   - values coming from the data memory, corresponding to the values of the accessed memory cells.

   In both cases, the input sequence corresponds to the content of the memory cells 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 if some specific condition is true 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). Once again, SC-FSIM techniques (and tools) are hardly able to manage such a scenario;
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 is a quite different task than SC-FSIM, requiring 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 may be 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: faults triggering an exception or forcing the processor to enter an endless loop may be easily found. 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.

III. BASIC STP-FSIM TECHNIQUES

One of the goals of this work is to overview different fault simulation techniques aiming at effectively performing the fault grading of a set of Self-test procedures. In the following, possible approaches for STP-FSIM are described. The different approaches differ, first of all in terms of a) fault detection mechanism, and b) input stimuli. The former point 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 latter instead means how the instructions and data are fed to the CPU. In the following, 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).

First, the Fault Grading of a single Self-test procedure is considered. Then, the analysis is extended to the Fault Grading of an entire Software Test Library. For each approach, advantages and drawbacks are presented, which are then validated in the experimental part of this work.

A. FAULT GRADING OF A SINGLE SELF-TEST PROCEDURE

For performing the fault grading of a single Self-test procedure, it is possible to adopt different approaches, that we categorize in the following three main alternatives:

1. STP-FSIM0: this approach is based on the traditional SC-FSIM. Although being originally conceived for a rather different purpose than the fault grading of Self-test procedures, traditional fault simulation techniques and tools can be adopted even in this case. 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, and 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, which are fed sequentially to the CPU each time it performs a fetch operation, and from the other side to the data values the CPU reads from the memory each time it performs a read memory access. Since we are considering a traditional SC-FSIM method, the input sequence is fixed. Hence, no matter if any fault changes or delays 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, and as soon as a difference is detected, the corresponding fault is labeled as detected. In this way, this approach could lead to wrong (i.e., larger than real) figures in terms of fault coverage, since faults might provoke a difference in one or more output signals, but such a difference could later be masked, and hence 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

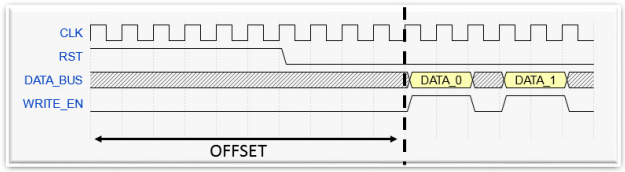

FIGURE 3. Graphical representation of the fault detection mechanism for STP-FSIM1.

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, at the time the result is going to be written, only. This situation is represented in Figure 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, and 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, and eventually 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), while the other components 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 possible to model also the scenario in which different data are retrieved or stored to or from the data memory. 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. Simulation of memories is computationally expensive and since fault dropping is not performed, the fault simulation tends to be slow and memory intensive.

B. FAULT GRADING OF A SOFTWARE TEST LIBRARY

Normally, Self-test procedures are developed according to a divide and conquer strategy. If the target module is a CPU, this is partitioned into submodules, and for each submodule, a specific Self-test procedure is developed [9]. The methodologies described in the previous sub-section represent a good 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 belonging to it 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 and (during the STL development) to better guide the development and 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 submodule can also detect faults present in different submodules. 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.

IV. OPTIMIZED FAULT SIMULATION TECHNIQUES

The aim of this section 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, Fault Dropping is not exploited at all. Hence, in the following it is discussed a suitable way to still adopt such a mechanism.

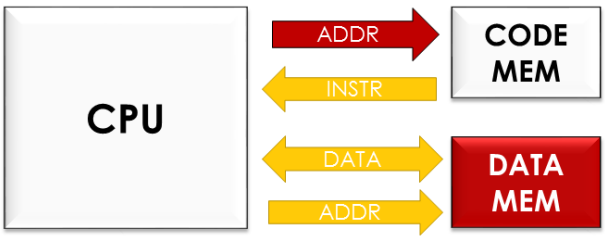

FIGURE 4. STP-FSIM3 scenario: The observability locations are marked in red.

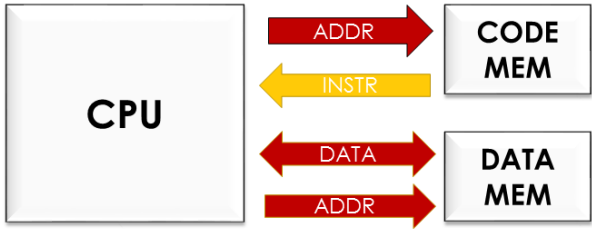

FIGURE 5. STP-FSIM4 scenario: The observability locations are marked in red.

A. STP-FSIM3

The main difference of this method (Figure 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 motivation for this choice originates from the fact that some faults can cause a different sequence of instructions to be fetched from the code memory. This normally 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 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 saving in computational cost may be relevant.

B. STP-FSIM4

A further optimization (Figure 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.

Table 1 summarizes the fault simulation methodologies presented in this paper. For each method, all the relevant characteristics are reported. Interestingly, 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.

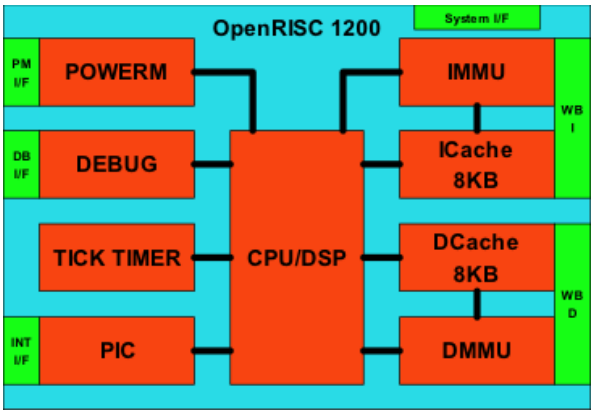

FIGURE 6. The OR1200 architecture.

V. EXPERIMENTAL RESULTS

In this section, a brief overview of the flow used for assessing the fault coverage of a Software Test Library is first presented. Then, we will present the relevant characteristics of the Self-test procedures and the fault simulation environment we used to quantitatively evaluate the effectiveness of the different techniques. Finally, we will present and discuss the gathered experimental results.

Experiments were conducted on the open source OpenRisc1200 (OR1200) soft-core processor [16], [17]. It consists of a 32-bit scalar RISC with Harvard architecture, MMU and basic DSP functionalities. The OR1200 includes a CPU and basic peripherals (e.g. timer, interrupt controller) as shown in Figure 6. The OR1200 was inserted in a SoC, which comprises a WishBone Interface, a Flash memory and a RAM. The Flash and RAM memories are 2MB each. The OR1200 was synthesized and mapped to a 65nm CMOS technology library, using Synopsys Design Compiler as logic synthesis tool.

TABLE 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+Memories	None	Final Memory Content	End of Test Program	No	Complete
STP-FSIM3	CPU+Memories	None	Final Memory Content Instruction Address Signals	End of Test Program All Clock Cycles	Yes	High
STP-FSIM4	CPU+Memories	None	Data Memory Signals Instruction Address Signals	Memory Write Operations All Clock Cycles	Yes	High

TABLE 2. STL characteristics.

Self-test procedure	Duration [Clock Cycles]	Size [Bytes]
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

TABLE 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%

We emphasize the fact that the OR1200 is representative of a class of CPUs which is widely used in practice in safety-critical applications. Despite its relatively small size, its adoption as a test case in this paper does not limit the generality of the adopted results. In fact, moving to larger processors simply increases the computational effort and memory required for fault grading, while the comparative behavior of the different fault grading techniques outlined in the previous sections remains the same.

The Self-test procedures we used for our experiments were developed resorting to different development strategies (random, deterministic, ATPG-based) [9]. The Self-test procedures compute internally the result of the test and then this is written to a known memory location in the system RAM. Each test program addresses stuck-at faults of a specific part of the CPU: Register File (RF), control unit, Operand multiplexers, ALU, Multiply and accumulator (MAC) unit, Load Store Unit (LSU), Fetch and Decode, Writeback multiplexers. The final STL includes eight Self-test procedures, whose main features are listed in Table 2.

Concerning the fault simulation campaigns, the Z01X tool by Synopsys has been used. Z01X is a functional fault simulator, which is widely used for functional safety analyses. Its fault simulation algorithm is based on a compiled event-driven concurrent engine and it supports both the 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 6). For STP-FSIM1, the observation points were limited to the Data WishBone Interface (namely WB D in Figure 6). 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 MB each. During STP-FSIM3 the RAM memory content and the Instruction WishBone Interface (WB I in Figure 6) were exclusively observed, while for STP-FSIM4 Data and Instruction WishBone Interface were observed. Among the other functionalities offered by the tool, there is also hyperfaults [18] 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 GB 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 Safe Faults [19], i.e., faults that in the application environment cannot produce any failure. When removing Safe Faults, the achieved Fault Coverage can be significantly higher. Finally, the fault simulations were performed with a zero-delay mode (i.e., all combinational and sequential delays were ignored).

In Table 3, we reported the gathered experimental results for the methodologies presented in Section 3 and 4. 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

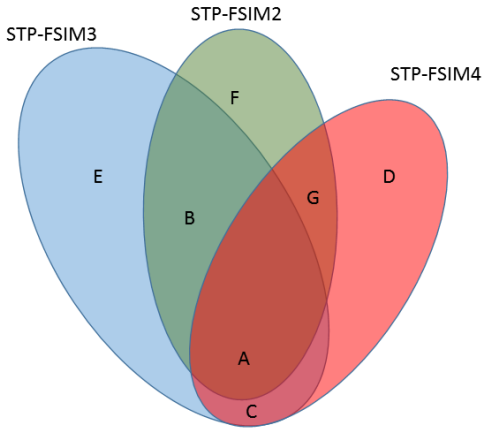

FIGURE 7. The detected faults by STP-FSIM2, STP-FSIM3, STP-FSIM4 and their possible intersections. In this case, E = F = G = 0.

STP-FSIM2 (in bold, 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 non-negligible 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, [26]), it emerged that the three approaches detect slightly different sets of faults. Let us denote with A, B, C, D, E, F and G the sets of faults to be considered when comparing the sets of faults detected by the three fault simulation approaches (Figure 7). In Table 4 instead, it is detailed the number of faults within each set. As shown in Figure 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 4 that the set A is the largest one, in fact, 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

TABLE 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

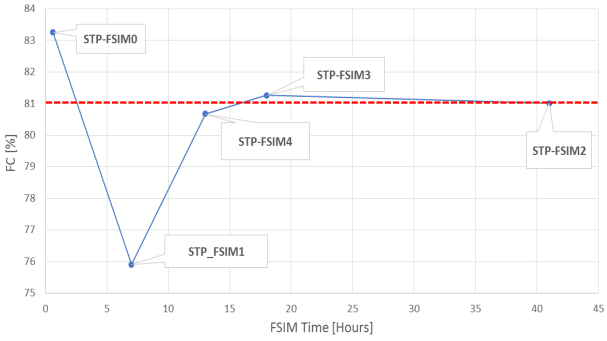

FIGURE 8. Fault simulation methodologies: Fault coverage accuracy versus Fault simulation time. The red line represents the exact FC figure.

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 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, and *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 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.

Figure 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).

The reader should also note that by carefully developing 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.

TABLE 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

TABLE 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

As an example, consider Table 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 [25]) and ALU some bogus data transit on that signal which are not reflected in the 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 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.

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 reliability 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, let us consider how the lockstep principle works. The key idea 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.

TABLE 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

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 comparison of the signature by the self-test routines, thanks 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 an in-house DCLS version of the OR1200 confirmed. The lockstep was applied to the lowest possible level, that is at the CPU level. Table 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. This is because for the results in Table 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 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).

VI. CONCLUSIONS

The main objective of this paper is to overview the different approaches for performing the fault simulation of a STL. Through the experiments, it has been shown the inadequacy of the traditional fault simulation approaches when dealing with this kind of task, since the nature of the problem evolved

from end-of-manufacturing to in-field test methodologies and so the fault simulation methodologies should evolve, too. When dealing with the development of an STL, fault simulation is a crucial step, often requiring a huge computational effort, and a combination of all these techniques should be exploited for maximum efficiency. During the early phases, fast fault simulations (e.g.,STP-FSIM0 and STP-FSIM1) may be preferred for refining the programs. Then, once the Self-test procedures are mature enough, the test engineer should move to more accurate fault simulations (e.g., STP-FSIM3 or STP-FSIM4). Finally, when computing the final fault coverage of the entire STL, they should resort to STP-FSIM2. However, it was also shown that when developing an STL for a DCLS-based SoC, STP-FSIM0 can be used as it is for the whole development, avoiding long and expensive fault simulations.

ACKNOWLEDGMENT

The authors would sincerely thank Federico Salieri, Jean-Marc Forey and Mel Gilmore from Synopsys Inc. for the valuable talks about fault simulation.

REFERENCES

[1] *Software Test Library Infineon*. Accessed: 2019. [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/
[2] *Software Test Library STMicroelectronics*. Accessed: 2018. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf
[3] *Software Test Library Cypress*. Accessed: 2018. [Online]. Available: http://www.cypress.com/file/249196/download
[4] *Software Test Library Renesas*. Accessed: 2018. [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read
[5] *Software Test Library Microchip*. Accessed: 2018. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf
[6] *Software Test Library ARM*. Accessed: 2018. [Online]. Available: https://developer.arm.com/technologies/functional-safety
[7] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design Test Comput.*, vol. 27, no. 3, pp. 4–19, May/Jun. 2010.
[8] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 88–99, Jan. 2005.
[9] P. Bernardi, R. Cantoro S. De Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65. no. 3, pp. 744–754, Mar. 2016.
[10] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of SBST programs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3055–3066, Oct. 2016.
[11] M. Gaudesi, I. Pomeranz, M. S. Reorda, and G. Squillero, "New techniques to reduce the execution time of functional test programs," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1268–1273, Jul. 2017.
[12] T. Niermann, M. Wu-Tung, J. Cheng, and H. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 11, no. 2, pp. 198–207, Jul. 1990.
[13] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 429–441, Jun. 1980.
[14] L .Chen and S. Dey, "Software-based Self-testing methodology for processor cores," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, Mar. 2001.
[15] S. Gai and P. L. Montessoro, "The fault dropping problem in concurrent event-driven simulation," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 10, no. 8, pp. 968–971, Aug. 1990.
[16] *OR1200 Instruction Set Architecture*. Accessed: 2018. [Online]. Available: https://openrisc.io/
[17] *OR1200 Design*. Accessed: 2018. [Online]. Available: https://github.com/openrisc/or1200
[18] S. Gai, P. L. Montessoro, and F. Somenzi, "MOZART: A concurrent multilevel simulator," *IEEE Trans. Comput. Aided Design Integr.*, vol. CAD-7, no. 9, pp. 1005–1016, Sep. 1988.
[19] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, and M. S. Reorda, "About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications," in *Proc. IEEE 19th Latin-Amer. Test Symp. (LATS)*, Mar. 2018, pp. 1–6.
[20] F. Pratas, T. Dedes, A. Webber, M. Bemanian, and I. Yarom, "Measuring the effectiveness of ISO26262 compliant self test library," in *Proc. 19th Int. Symp. Qual. Electron. Design (ISQED)*, Mar. 2018, pp. 156–161.
[21] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *Proc. Design, Automat. Test Eur.*, Mar. 2011, pp. 1–2.
[22] G. Bahig and A. El-Kadi, "Formal verification of automotive design in compliance with ISO 26262 design verification guidelines," *IEEE Access*, vol. 5, pp. 4505–4516, 2017.
[23] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2017, pp. 970–975.
[24] M. Restifo, P. Bernardi, S. De Luca, and A. Sansonetti, "On-line software-based self-test for ECC of embedded RAM memories," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2017, pp. 1–6.
[25] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti, and G. Squillero, "Software-based self-test techniques for dual-issue embedded processors," *IEEE Trans. Emerg. Topics Comput.*, to be published.
[26] P. Bernardi, D. Piumatti, and E. Sanchez, "Facilitating fault-simulation comprehension through a fault-lists analysis tool," in *Proc. IEEE 10th Latin Amer. Symp. Circuits Syst. (LASCAS)*, Feb. 2019, pp. 77–80.
[27] D. Gizopoulos, A. Paschalis, and Y. Zorian, "An effective built-in self-test scheme for parallel multipliers," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 936–950, Sep. 1999.
[28] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, Apr. 2005.

ANDREA FLORIDIA received the M.Sc. degree in computer engineering from the Politecnico di Torino, Torino, Italy, in 2017, where he is currently pursuing the Ph.D. degree with the Department of Control and Computer Engineering. His research interests include test of multi-core systems-on-chip and fault simulation techniques.

ERNESTO SANCHEZ received the degree in electronic engineering from Universidad Javeriana, Bogota, Colombia, in 2000, and the Ph.D. degree in computer engineering from the Politecnico di Torino, Italy, in 2006, where he is currently an Assistant Professor with the Department of Control and Computer Engineering. His main research interests include microprocessor testing and evolutionary computation.

MATTEO SONZA REORDA received the M.S. degree in electronics and the Ph.D. degree in computer engineering from Politecnico di Torino, Italy, in 1986 and 1990, respectively, where he is currently a Full Professor with the Department of Control and Computer Engineering. His research interests include test of SoCs and fault tolerant electronic system design.