Fault-Independent Test-Generation for Software-Based Self-Testing

Fault-Independent Test-Generation for Software-Based Self-Testing

Panagiotis Georgiou, Xrysovalantis Kavousianos, *Member*, IEEE, Riccardo Cantoro, *Member*, IEEE and Matteo Sonza Reorda, *Fellow*, IEEE

***Abstract*—Software-based self-test (SBST) is being widely used in both manufacturing and in-the-field testing of processor-based devices and Systems-on-Chips. Unfortunately, the stuck-at fault model is increasingly inadequate to match the new and different types of defects in the most recent semiconductor technologies, while the explicit and separate targeting of every fault model in SBST is cumbersome due to the high complexity of the test-generation process, the lack of automation tools, and the high CPU-intensity of the fault-simulation process. Moreover, defects in advanced semiconductor technologies are not always covered by the most commonly used fault-models, and the probability of defect-escapes increases even more. To overcome these shortcomings we propose the first fault-independent method for generating software-based self-test procedures. The proposed method is almost fully automated, it offers high coverage of non-modeled faults by means of a novel SBST-oriented probabilistic metric, and it is very fast as it omits the time-consuming test-generation/fault-simulation processes. Extensive experiments on the OpenRISC OR1200 processor show the advantages of the proposed method.**

***Index Terms*—Fault coverage, Non-modeled Defects, Output Deviations, Software-Based Self-Test (SBST), Testing.**

I. Introduction

The deep sub-micron semiconductor technologies combined with the advanced architectural innovations have significantly improved the performance of modern Systems on Chip (SoCs). Especially when used in safety-critical applications, SoCs require advanced testing techniques for screening defective devices. However, the strict design constraints and the need to test the target devices at the normal mode of operation impose the use of non-intrusive test methods [26]. In addition, any test applied in-the-field should not compromise the internal state of the Device Under Test (DUT) [34]. Therefore, design-for-testability solutions are complemented with functional solutions, such as SBST for processor-based ICs and SoCs.

SBST executes test programs that activate potential faults inside the circuit and propagate the errors to observable sites, like the memory [33], [34], [39]. SBST has several properties, which are very important for in-field test of safety-critical devices. At first it does not require the assistance of any automatic-test-equipment (ATE), therefore it is completely autonomous. In addition, it is not intrusive (it does not alter the circuit structure), therefore it does not affect the performance of the processor. Moreover, it is applied exactly at the operating conditions, while at the same time it does not excite any redundant faults avoiding thus over-testing the core under test. Finally, SBST facilitates the periodic monitoring in-the-field with limited intrusiveness with respect to the normal-mission operation [19], [31], and it does not compromise the internal state of the circuit [34].

Based on the SBST paradigm, several semiconductor and IP companies provide nowadays self-test libraries with their products, which can be easily integrated by their customer into the application code [1]–[6]. The major challenge in the development of these self-test libraries is the generation of small test-programs that offer high fault coverage in short test-time [32], [41], [45]. SBST programs can be generated manually [38], semi-automatically [20] or automatically, targeting different processor architectures and fault models [39]. Various methods target microprocessors with caches [17], shared-memory schemes [8], floating-point units [46] and dual-issue processors [13]. Deterministic techniques exploit the regularity of sub-modules [11], [12], [21], [22], [28], [29], [35], while others use automatic-test-pattern-generation (ATPG) [37] and evolutionary algorithms [10], [37], [40]. Several methods explore the application of SBST to test peripheral modules [9], [25]. In [14] the effectiveness of SBST for a given level of dependability is evaluated.

Despite their benefits, SBST methods suffer from several drawbacks. At first they often target only the stuck-at fault model, which is inadequate for detecting many defects. In addition, most SBST techniques are not systematic, therefore they require extensive human intervention and long development times. Moreover, they involve the CPU-intensive process of fault-simulating multi-million gate designs for multi-million clock cycles using multiple fault models and specialized functional (non-scan) simulators. Besides these deficiencies, the shrinking process technologies, the physical limits of photo-lithographic processes and new materials introduce new defects that are not always accurately modeled even by the most commonly used fault models [42]. Therefore, fast, low-cost and highly effective SBST-based techniques are required to improve the defect screening of processor-based devices.

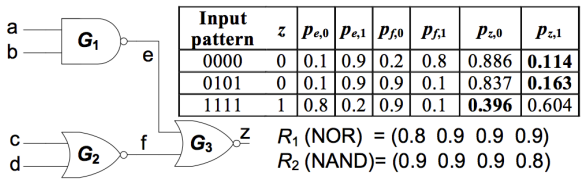
In this paper, we present the first fault-independent SBST method. The proposed method offers short test-program generation time as it is almost fully systematic, and it exploits multiple design models of the device-under-test (DUT) in

This work was submitted for review on 19/09/2018. A preliminary version of this work has been presented at the 24^{th} IEEE International Symposium on On-Line Testing and Robust System Design.

P. Georgiou and X. Kavousianos are with the Department of Computer Science, University of Ioannina, Ioannina 45445, Greece (e-mail: pgeorgio@cs.uoi.gr, kabousia@cs.uoi.gr).

R. Cantoro and M. Sonza Reorda are with the Department of Control and Computer Engineering, Politecnico di Torino, Turin 10129, Italy (e-mail: riccardo.cantoro@polito.it; matteo.sonzareorda@polito.it).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability



Input pattern	z	$p_{e,0}$	$p_{e,1}$	$p_{f,0}$	$p_{f,1}$	$p_{z,0}$	$p_{z,1}$
0000	0	0.1	0.9	0.2	0.8	0.886	**0.114**
0101	0	0.1	0.9	0.9	0.1	0.837	**0.163**
1111	1	0.8	0.2	0.9	0.1	**0.396**	0.604

Fig. 1. Output deviation calculation example

order to maximize the non-modeled fault coverage of the test-programs under strict test-application-time and test-program-size constraints. The test-programs are evaluated by means of a novel and very effective SBST-oriented probabilistic metric, which considers both the architectural model and the synthesized gate-level netlist of the DUT. The proposed metric is very fast as it omits the time-consuming functional fault-simulation, and it can be applied to any SBST-based method. In addition, the very high fault-coverage ramp-up of the generated test-programs offers additional test-time benefits in periodic-testing as well as in abort-at-first-fail environments in manufacturing testing. Extensive experiments on a core corresponding to the OpenRISC OR1200 processor [7] show the advantages of the proposed method.

The organization of this paper is as follows: Section II presents background material and the motivation of this work. Section III presents the proposed test-generation method. Section IV presents an experimental evaluation of the method, and Section V concludes the paper.

II. Motivation

The large computational overhead, the long running times and the high test-generation complexity prevent SBST methods from targeting other fault models than the stuck-at fault model. As it is shown in Section IV, even the simple task of fault simulating 50Kbytes of test code on the Open Risc 1200 processor using commercial tools, requires several days for stuck-at faults and even weeks for transition faults. By taking into account that test-generation is highly complex and even more CPU-intensive than fault simulation, we understand why most SBST methods target only the stuck-at fault model [10]–[14], [20]–[22], [28]–[31], [33]–[35], [37], [39], [40], [45], [46] .

Even though targeting multiple fault models is a rather unrealistic goal for SBST, the defect coverage of the test programs can be enhanced by probabilistically evaluating their potential to detect arbitrary defects without targeting any particular fault model. Such an approach was proposed in [23], [36] for non-scan sequential circuits modeled at the register-transfer level (RTL). However, RTL models limit the effectiveness of these methods for detecting silicon defects. Moreover, these methods require an automatic-test-equipment (ATE) to apply the test sequences and to monitor the outputs at each clock cycle, whereas SBST implies the observation of the content of the data memory at the end of the test. Therefore, they are not suitable for SBST, which is by definition fully autonomous and ready to be applied in-the-field without the need of any ATE.

Gate-level output-deviations were shown to be very effective in detecting silicon defects in structural testing [27], [43], [44]. They are probability measures that reflect the likelihood of error detection at circuit outputs and they are computed without explicit fault grading, hence the computation is feasible for large circuits (the computational cost grows linearly with the number of gates). Initially, a probability map, the confidence-level (CL) vector, is assigned to every circuit gate. For every input pattern, each line i is assigned signal probabilities p_i^0, p_i^1 to be at logic 0 and 1, respectively. The CL vector R_i of gate G_i with m inputs has 2^m components $r_i^{0\ldots00}, r_i^{0\ldots01}, \ldots, r_i^{1\ldots11}$, each denoting the probability that the gate output is correct for the respective input combination. For example, r_i^{11} denotes the probability that the output y of the 2-input gate G_i is correct when the logic values at the inputs a, b of the gate are set equal to $ab = 11$. Every 2-input logic gate G_i is assigned a CL vector $R_i = (r_i^{00}\ r_i^{01}\ r_i^{10}\ r_i^{11})$. Let G_i be a NAND gate. Then the CL vector can be used to define the probability that the output y is correct as follows:

$$p_y^0 = p_a^0 p_b^0 (1 - r_i^{00}) + p_a^0 p_b^1 (1 - r_i^{01}) + p_a^1 p_b^0 (1 - r_i^{10}) + p_a^1 p_b^1 r_i^{11}$$
$$p_y^1 = p_a^0 p_b^0 r_i^{00} + p_a^0 p_b^1 r_i^{01} + p_a^1 p_b^0 r_i^{10} + p_a^1 p_b^1 (1 - r_i^{11})$$

Note that $p_y^0 + p_y^1 = 1$. Signal probabilities can be computed for other gate types as well [43]. The gate-level CL vectors can be generated from simple transistor-level failure probabilities [43], or by using alternative ways, like layout information, inductive-fault analysis [24], and failure-data analysis.

For any gate G_i let its fault-free output value for input pattern t_j be d, with $d \in 0, 1$. The output deviation $\Delta G_{i,j}$ of G_i for t_j is defined as $P_{G_i}^{d'}$ (d' is the complement of d), and it is a measure of the likelihood that the gate output is incorrect for input pattern t_j. The deviation values at the circuit outputs are indicative of the probability for arbitrary defects to be detected at these outputs (the higher is the deviation value at an output, the higher is the likelihood of observing an error at the corresponding output).

Example 1. Fig. 1 shows a circuit consisting of three gates G_1, G_2, and G_3, with two different confidence level vectors (R_1 and R_2) assigned to the NOR and NAND gates [43]. The first column of the Table in Fig. 1 presents three test patterns and their respective fault free value at output z. The next six columns present the signal probabilities computed at the internal circuit nodes using the aforementioned confidence level vectors. For each input pattern, the output-deviation is the probability the output 'z' to be faulty, which is shown in bold in the two last columns of the Table. Note that the deviation at output z is higher when the last test pattern is applied (it is equal to 0.396) as compared to the other two patterns (it is equal to 0.114 and 0.163 respectively). Therefore, the last test pattern $(a, b, c, d) = (1, 1, 1, 1)$ is the most promising for detecting defects as it provides the highest output deviation value. ■

In this paper we propose the first output-deviation-based metric that exploits the architectural and the gate-level models of the processor to evaluate SBST sequences. Even though this metric can enhance the non-modeled fault-coverage of any SBST technique, we apply it on the particular case of *test macros* [20], [30] that have been proven to be very effective for stuck-at faults. A test-macro is a sequence of assembly-level

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability

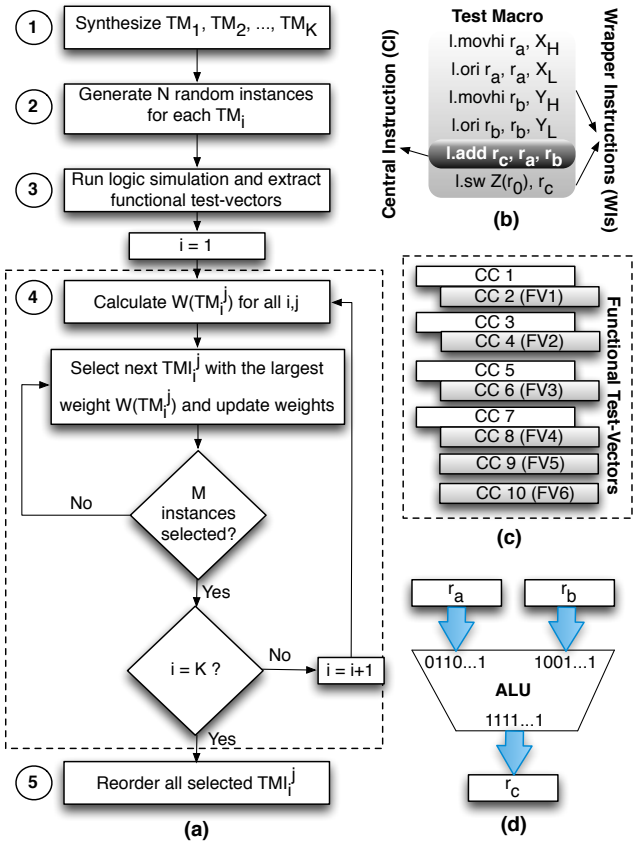


Fig. 2. (a) Test Flow (b) TMI example (c) Functional Vectors (d) ALU Test

instructions, with one instruction executing a specific function and additional instructions that set the *macro parameters* (i.e., the operand values) and propagate the results to observable memory positions. At the architectural level, instruction-based *test-macros* are synthesized that exercise the various modules of the processor and observe the responses. Multiple instances of every test-macro are generated by combining instructions that maximize the probability of detecting non-modeled faults, and by randomly varying their operands. Each test-macro instance (TMI) is evaluated by means of a novel output-deviation-based metric computed on the gate-level netlist of the processor, and the most effective ones are selected to synthesize test programs, according to specific test-time and test-program-size constraints.

III. SBST Using Output-Deviations

In this section we present the proposed test generation method. We first introduce the basic test generation flow for detecting non-modeled faults, and then we elaborate on the particular case of non-modeled delay-faults at computational units.

A. Basic test generation flow

The basic test generation method consists of five steps and it is shown in Fig. 2a. Initially, one test-macro is manually generated for every instruction of the processor. This instruction is called hereafter the *Central-Instruction (CI)* of the macro and it exercises specific units of the processor. For example, the instruction *add* r_c, r_a, r_b of the OR 1200 processor [7] executes the arithmetic operation $r_c = r_a + r_b$ (r_a, r_b, r_c are general purpose registers), and it exercises both the ALU and the control unit. If the result of the addition is observable, then this instruction constitutes a test for these units. The quality of this test depends on the contents of r_a, r_b, which are set by means of additional instructions, the *Wrapper-Instructions (WIs)*. The wrapper-instructions assign specific values to the operands of the central-instruction, and store the result at observable memory positions.

Example 2. The test-macro generated for the instruction *add* r_c, r_a, r_b is shown in Fig. 2b. The first four instructions are wrapper-instructions that load the high/low 16-bit parts of registers r_a, r_b with the 16-bit values X_H, X_L, Y_H, Y_L. The next instruction is the central-instruction and the last instruction is a wrapper-instruction that stores the result at the physical address obtained by combining the immediate offset Z with register r_0 that holds always the value 0. ■

Similar macros are generated for every instruction of the processor, with two exceptions. The first one is related to wrapper-instructions. Such instructions do not necessarily become central-instructions, because the faults activated by them are observable when they are executed as wrapper-instructions. Second, the central-instructions that do not produce directly observable results (e.g. branch instructions) use wrapper-instructions to provide the observable results. For example, upon correct execution of a branch-based central-instruction, a wrapper-instruction stores a pre-determined value at an observable memory position, thereby making the results of the branch instruction observable. Even though the generation of the test-macros requires some knowledge of the instruction set and the architecture of the processor, it is an one-time and rather simple task for every processor architecture.

The fault coverage of every test-macro depends on the operands of the central and wrapper-instructions. Unfortunately, there is no straightforward method to select the most effective operands for detecting non-modeled faults. To this end, we propose an almost fully automated process to generate test-macro instances $(TMIs)$ with high non-modeled fault coverage. Let K be the number of different macros TM_1, TM_2, ..., TM_K generated at the first step (one for every CI). Then, N random instances $TMI_i^1, TMI_i^2, \ldots, TMI_i^N$ are generated for every TM_i at the second step, by randomly varying every operand of TM_i. For the particular macro shown in Fig. 2b the $TMIs$ are generated by varying registers r_a, r_b and the values for X, Y, and as a result register r_c and the value for Z also change (we note that whenever a TMI with invalid values is generated it is discarded).

At the third step, we run logic simulation on the gate-level netlist of the processor using the $N \times K$ $TMIs$ generated at the previous step, and the logic values generated at the inputs/outputs of selected units of the processor are recorded at the clock cycles when these units are excited by each TMI_i^j. For example, when the CI shown in Fig. 2b is executed, the inputs/outputs of the ALU are recorded during the clock cycle when the arithmetic values stored into registers r_a and r_b are applied to the inputs of the ALU (see Fig. 2d). These logic values constitute one functional test-vector/response applied

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability

by the TMI. Every TMI generates a number of functional test-vectors/responses that excite various units of the processor and propagate the results to observable sites of the processor. These vectors are generated during the clock cycles when their responses are observable either immediately (e.g. the output of the ALU stored into register r_C) or later through the wrapper-instructions.

One example is shown in Fig. 2c for the test-macro of Fig. 2b. The first four instructions require two clock cycles to be executed and the functional test-vectors are generated at their second clock cycles. The last two instructions require one clock cycle to be executed and the functional test-vectors are generated at both clock cycles. The more effective are the functional test-vectors of TMI_i^j in detecting defects, the higher is the test-quality of TMI_i^j. The potential of TMI_i^j for detecting non-modeled faults depends on the quality of the functional test-vectors/responses generated by TMI_i^j. When a functional test-vector generated by TMI_i^j activates a defect and propagates the error to the output of the exercised module, then there is a high probability that this error will propagate to an observable site of the processor.

In order to evaluate the functional test-vectors of the $TMIs$ we propose a new output-deviation-based metric. Specifically, the combinational logic of every processor unit exercised by a functional test-vector is used to calculate the deviations of its outputs for this test-vector, as it is shown in Section II. For example, in Fig. 2d we show the functional test-vector $FV5$ generated during the execution of the CI shown in Fig. 2b, where the ALU receives inputs from registers r_a, r_b and it stores the result in register r_c. The computation starts from the inputs to the outputs of the ALU (note that, in the general case, the mapping between inputs and outputs of every combinational block may not be trivial).

According to the theory of output-deviations the most effective test-vectors are the test vectors that produce the highest deviation values at the outputs of the circuits under test [43]. In order to identify those vectors, we apply first all the generated functional vectors that excite one unit, and we calculate the maximum deviation value that is generated at every output bit p of that unit. This process is applied separately for every TM_i, because different test-macros exercise different parts of the units. Let NV_i be the number of functional test-vectors generated by TM_i. In the particular case of TMI_i^j the functional vectors $FV(TMI_i^j,1)$, $FV(TMI_i^j,2)$, $\ldots$, $FV(TMI_i^j,NV_i)$ are generated. The output deviations of $FV(TMI_i^j,k)$ are computed for $j \in [1,N]$, $k \in [1,NV_i]$, and the proximity of each deviation value to the highest deviation value found at every output among all the generated functional vectors is calculated. Let $Max(TM_i,p,v)$ be the highest deviation value found for all functional test-vectors of every instance TM_i^j of TM_i ($j \in [1,N]$) at output p for logic response v ($v = 0,1$). Then, for each $FV(TMI_i^j,k)$ all the pairs (p,v) with deviation values $Dev(FV(TMI_i^j,k),p,v)$ higher than a threshold value constitute the set $MS(TMI_i^j,k)$ (the rest of the pairs are not further considered for this functional test-vector). This threshold value THR is a percentage of the highest deviation value $Max(TM_i,p,v)$ at this output, i.e., $Dev(FV(TMI_i^j,k),p,v) \geq THR \times Max(TM_i,p,v)$, with THR usually in the range $90\% - 100\%$. Note that the higher is the value of THR, the more strict is the selection process towards pairs (p,v) with high deviation values.

Besides the high deviation values, the potential of $FV(TMI_i^j,k)$ to detect defects at the outputs p and logic response v with $(p,v) \in MS(TMI_i^j,k)$ depends on two additional parameters. The first one is the number of faults that are observable at output p, $Faults(p)$, which is proportional to the size of the logic cone driving p. This parameter is modeled by setting $Faults(p)$ equal to the number of gates in the fan-in cone size of p. The second one is the number of functional test-vectors generated by all random instances of TM_i that provide high deviation for each pair (p,v). The higher is this number, the higher is the probability that, eventually, some of the selected $TMIs$ will provide functional test-vectors with high deviation values for this pair. Therefore, this output-logic value pair is considered as an easy pair, and it is given low priority $PR(TM_i,p,v)$ to bias the selection towards $TMIs$ that embed functional test-vectors with high deviation values at more difficult pairs. $PR(TM_i,p,v)$ is set equal to the inverse of the proportion of all functional test-vectors generated by every random instance of TM_i that offer high deviation value for (p,v). $FV(TMI_i^j,k)$ is assigned a weight equal to

$$WFV(TMI_i^j,k) = \sum_{(p,v)\in MS(TMI_i^j,k)} Faults(p) \times PR(TM_i,p,v) \quad (1)$$

Note that the weight increases as the fan-in cone-size of p and the priority of (p,v) increase. Then, a weight is assigned to TMI_i^j equal to the sum of the weights of its functional test-vectors

$$W(TMI_i^j) = \sum_{k=1\ldots NV_i} WFV(TMI_i^j,k) \quad (2)$$

The higher is the value of $W(TMI_i^j)$, the more effective is the instance TMI_i^j for detecting non-modeled faults.

Every instance TMI_i^j with high weight $W(TMI_i^j)$ is expected to detect many defects at the outputs $p \in MS(TMI_i^j,k)$, i.e., the outputs with high deviation values at the functional test-vectors generated by TMI_i^j. When this instance is selected, the potential of the rest of the instances of the same test-macro to detect defects at the same outputs $p \in MS(TMI_i^j,k)$ decreases (less defects are anticipated to remain undetected at the logic cones of these outputs). To reflect this fact the number of faults $Faults(p)$ at the logic cone of every output $p \in MS(TMI_i^j,k)$ is divided by a constant factor F after TMI_i^j is selected. This reduces the weight of all the functional test-vectors that provide a high-deviation value at output p, since their effectiveness for detecting defects drops after the selection of TMI_i^j. The higher is the value of F, the higher is the priority given to instances with high deviation values at other outputs. Therefore, all the weights are re-computed by applying again eq. (1), (2) and the next instance with the highest weight is selected. This process is iterated until M instances are selected for every test-macro.

The M most effective instances selected from every test-macro maximize the defect coverage at the particular processor

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability

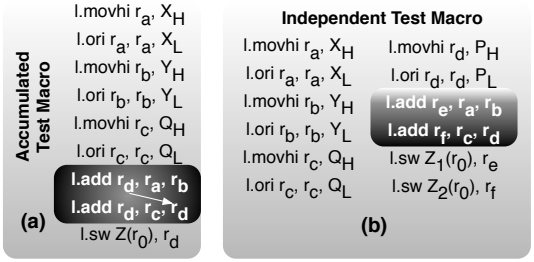


Fig. 3. (a) A-TM example (b) I-TM example

units excited by the respective test-macro under test-program-size constraints. Therefore, in order to achieve high defect-coverage ramp-up for the whole processor, the test-macro instances must be applied in the specific order that maximizes the defect-coverage at all processor units at the same time. To this end, after the M most effective instances of every test-macro are selected, all the $M \times K$ instances are evaluated again and they are re-ordered by considering the whole processor circuit. In particular, all the $M \times K$ instances are evaluated together by considering all the targeted units of the processor. First, we reset the values of $Faults(p)$ at their initial values and the weights $W(TMI_i^j)$ are re-computed. Then, $M \times K$ iterations are applied, and at each iteration the instance with the highest weight is selected as the next in the order, and the values of $Faults(p)$ are updated as shown before. We note that the final order of the test-macro instances depends on both the size of the excited units and the quality of the generated functional vectors, as it is evaluated by the proposed output-deviation based metric.

Even though the test program generation is not fully automatic, it requires only limited designer/test-engineer intervention, mostly during the development of the test-macro templates. However, this intervention is at the architectural level and it does not require the specific gate-level or transistor-level model of the processor, which decouples the proposed method from any implementation details. We note that, such an intervention is an one-time task, while the rest of the test-generation method (which is usually applied multiple times during the development process) is fully automatic.

B. Test generation for high delay-defect coverage.

Even though the test-macros generated using the method proposed in Section III-A are very effective in detecting non-modeled faults, they offer limited detection of delay-faults in execution units like the ALU. Detection of delay-faults requires pairs of test-vectors to be applied to the inputs of the exercised units. Therefore, delay-fault oriented test-macros are developed that embed pairs of central instructions exercising the processor units in successive clock cycles with different test-vectors. Such macros are called *Independent-Test Macros* (*I-TMs*).

Even though *I-TMs* can be generated for most of the processor's units, some units are accumulator-type units (like the Multiply-Accumulate unit of the OR1200 processor), which are designed to execute successive operations of accumulator-type only. Such operations require one operand of the second central instruction to be the result of the execution of the first central instruction. For such units $I-TMs$ are inapplicable, and *Accumulated-Test Macros* (*A-TMs*) are used instead. Each $A-TM$ embeds two similar successive central-instructions, where one operand of the second *CI* is the result of the first *CI*.

Fig. 3 presents an example of an *A-TM* and an *I-TM* test-macro. Since both *CIs* in every *A-TM* instance (*A-TMI*) consist of the same instruction, the test-generation flow is exactly the same with the flow shown in Fig. 2a except of the additional functional test-vector that is generated by the second CI. However, in the case of *I-TMs* separate evaluation and selection of the first and second central-instructions is required (note that both of them detect defects, but most of the delay-defects are detected by the second central-instruction). To this end, different priority values PR and sets MS of outputs with high deviation values are manipulated for the first and the second central-instruction of all *I-TMIs*.

The generation of the *I-TMIs* is done as follows: multiple random instances of regular *TMIs* with a single central-instruction are generated, but only the functional test-vector of their central-instruction is evaluated using eq. (1), (2). The two central-instructions with the highest weights are combined to generate an *I-TMI* (the central-instruction with the highest weight is used as second in the pair to favor the detection of delay defects). Then, the selected central-instructions are removed and the same process is repeated for generating the next N *I-TMIs* using the remaining central-instructions.

Even though $A-TMs$ can be potentially used to test all the units of the processor, $I-TMs$ are conjectured to offer higher delay fault coverage than $A-TMs$ because they do not suffer from the correlation between the operands of the first and the second central-instruction. Moreover, $A-TMs$ can be considered as a specialization of $I-TMs$, because every $A-TMI$ can be substituted by an equivalent $I-TMI$ with one of the operands of the second CI be a value equal to the expected result of the execution of the first CI. Therefore, $A-TMs$ are used only for accumulator-type units, and $I-TMs$ are used for the rest of the units.

The synthesis of $A-TMs$ and $I-TMs$ is straightforward for computational units like the ALU but there are various non-computational units that require the synthesis of special test-macros in order to detect delay faults. One such example is the *Load-Store* unit of the OR1200 processor, which transfers the data between the processor and the memory. Even though some delay faults of this unit are exercised by the wrapper instructions of the test-macros generated for other units, most of them remain undetected unless specific test-macros are synthesized for this unit. One test-macro example targeting delay-faults at this unit is presented in Fig. 4a. The main purpose of this macro is to exercise the *Load-Store* unit by the means of different memory addresses and data transferred between the registers and the memory. Specifically, the wrapper-instructions load the registers r_a, r_b with the 32-bit values $X(X_H, X_L)$ and $Y(Y_H, Y_L)$, while the central instructions transfer these data between the registers and the cache memory of the processor (the value X is moved from register r_a to a memory position defined by the contents of r_b, then it is transferred to register r_c and finally from register r_c to memory

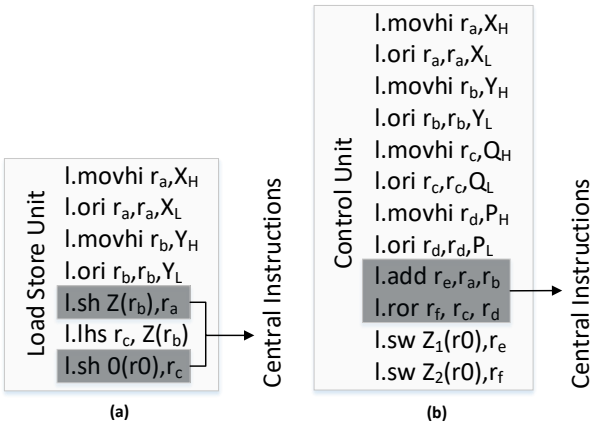


Fig. 4. Load-Store Unit & Control Unit TM Example

position 0).

Another unit that requires the synthesis of special test-macros is the *Control* unit. Similar to the *Load-Store* unit, the *Control* unit is also exercised by the wrapper and central instructions of all test-macros, but it cannot be sufficiently tested for delay defects unless multiple combinations of different instructions targeting different units are applied in consecutive clock-cycles. In Fig. 4b, a test-macro example for the *Control* unit is presented. The first eight instructions are wrapper-instructions that load the registers r_a, r_b, r_c, r_d with the 32-bit values X, Y, Q, P, then two arbitrary central-instructions follow that apply an arithmetic and a logical computation using these registers, and the last two wrapper-instructions store the results of the computations at the physical addresses Z_1, Z_2. In a similar manner test-macros can be synthesized for every unit of the processor.

IV. Experimental Results

The proposed method was developed using C++ and Python, and experiments were performed using the 32-bit scalar OR1200 RISC processor. The OR1200 processor has a Harvard micro-architecture, 5-stage integer pipeline, virtual-memory support (MMU) and basic DSP capabilities. The processor has one embedded instruction cache and one embedded data cache of 8KB each. The gate level netlist of the processor was synthesized using the NanGate 45nm technology and commercial tools. The gate-level netlist (excluding the memories) consists of 17.6K cells. One instruction is fetched from the cache at every clock cycle, while additional clock cycles are required when the data are fetched from the memory and/or conditional branch instructions are executed. The clock frequency for the OR 1200 processor was set equal to 200 MHz.

Almost all units of the processor were targeted, i.e. the *ALU*, the *Multiply-Accumulate (MAC)*, the *Load-Store*, the *Program Counter Generator*, the *Instruction Fetcher*, the *Operands Mux*, the *Write-Back Mux* and the *Instruction-Decoder* (the *Exception-Handling* unit and the caches require special test generation mechanisms). 137 test-macros (including $A-TMIs$, $I-TMIs$, Load-Store and Control unit $TMIs$) were generated, each consisting of 3 to 17 instructions. For each test-macro 120 random instances were generated (overall

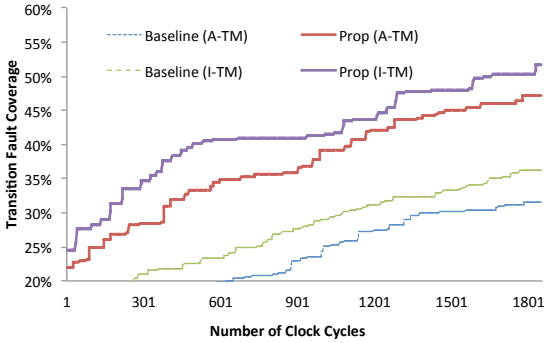


Fig. 5. $I-TMs$ vs $A-TMs$ for unit (*ALU*)

16,440 $TMIs$) and 6 test-programs were semi-automatically generated as follows:

- **Baseline**: 2, 4 and 8 instances were selected randomly out of the 120 instances generated for every test-macro (overall 274, 548 and 1,096 $TMIs$).
- **Proposed**: 274, 548 and 1,096 $TMIs$ were selected using the proposed method.

We note that the values of 2, 4 and 8 instances were intentionally chosen in order to generate small, medium and large test-programs, respectively. The running time of the proposed method on a single 64-bit CPU running at 1.2 GHz was less than one day in the worst case.

All test-programs were evaluated for detecting non-modeled faults using two surrogate fault models: the stuck-at and the transition-delay fault models. None of these models were explicitly targeted by the test-macro generation process. Instead, they were used to evaluate the potential of the proposed method to detect non-modeled faults. Even though the proposed method does not involve any fault simulations, such simulations were used to evaluate the generated test-programs and thus assess the effectiveness of the proposed method. These simulations were applied using commercial tools on a server with 48 CPUs running at 2.5 GHz. The stuck-at fault-simulation time for each test-program was between 5 hours (for 274 $TMIs$ on the *Write-Back Mux* unit) up to 5 days (for 1,096 $TMIs$ on the *Multiply-Accumulate unit*), and the transition-delay fault-simulation time was between 6 hours and 6.5 days on the respective cases. The stuck-at fault simulation and the transition-fault simulation on the whole processor (for 1,096 $TMIs$) require 11.5 days and 2 weeks, respectively.

As it was explained in Section III-B, both $A-TMs$ and $I-TMs$ can be used for most of the processor-units, but the superiority of $I-TMs$ in detecting transition delay faults makes them more favorable as compared to $A-TMs$. As it is shown in Fig. 5, $I-TMs$ are more effective than $A-TMs$ on the ALU unit of the OR1200 processor in both the proposed and the baseline approaches. Therefore, all the test-programs were composed of $I-TMs$ for every unit, except of the $Multiply-Accumulate$ unit, where only $A-TMs$ are applicable.

In order to show that the efficiency of the proposed method only slightly depends on the randomness of the initial set of $TMIs$, we generated (randomly) three different initial sets of 16,440 $TMIs$, and we run the proposed method

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability

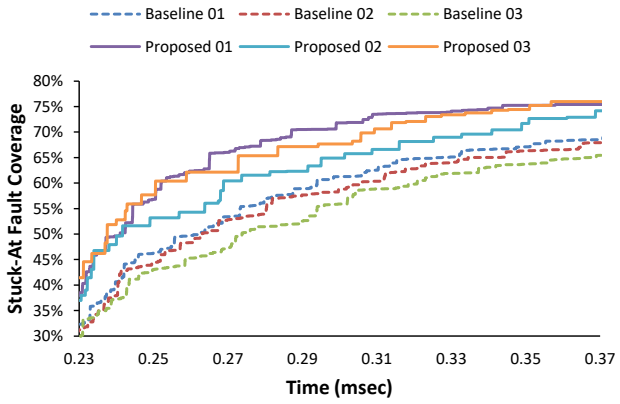


Fig. 6. Stuck-at fault variation results of *ALU*

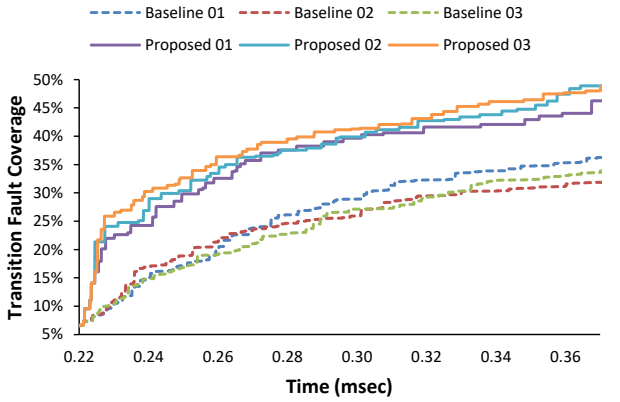


Fig. 7. Transition-fault variation results of *ALU*

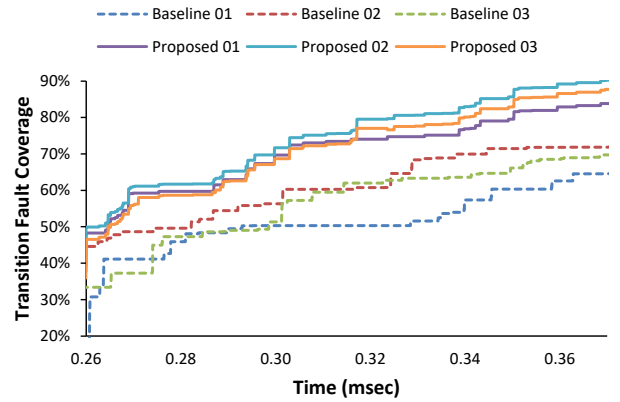


Fig. 8. Stuck-at fault variation results of *MAC* unit

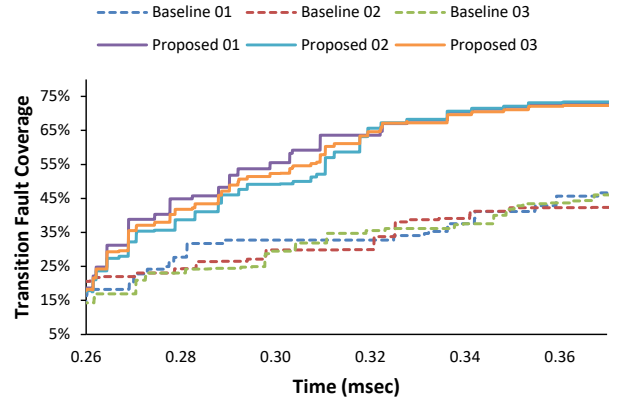


Fig. 9. Transition-fault variation results of *MAC* unit

TABLE I
SOFTWARE-BASED-SELF-TESTING RESULTS

		Test Program Size (KBytes)		Test Time (msec)		Stuck-At FC (%)		Transition FC (%)	
TMIs	**#**	**BSL**	**Prop**	**BSL**	**Prop**	**BSL**	**Prop**	**BSL**	**Prop**
2x137	1	10.9	13.2	0.064	0.079	82.58	87.45	59.00	73.32
	2					83.01	87.84	60.04	73.79
	3					81.96	87.08	58.70	73.32
4x137	1	19.8	23.5	0.109	0.122	86.39	88.92	66.85	75.89
	2					86.53	88.91	66.91	76.06
	3					86.44	88.86	66.84	76.04
8x137	1	37.5	41.8	0.186	0.203	88.14	89.56	73.51	77.62
	2					87.81	89.67	72.62	77.36
	3					88.08	89.78	72.62	77.40

three times selecting each time the 274, 548 and 1,096 most effective $TMIs$ from each one of these sets. We repeated the same approach for the baseline approach, and we generated 3 different baseline test-programs for every different number of 274, 548 and 1,096 $TMIs$. Fig. 6 and Fig. 7 present the stuck-at and the transition fault-coverage for the *ALU* unit for the three test-programs generated using the proposed approach, as well as the three test-programs generated using the baseline approach. Fig. 8 and Fig. 9 present the stuck-at and the transition fault-coverage for the *MAC* unit for the respective cases. In every chart the x-axis presents the test-time (in msec) and the y-axis presents the fault-coverage. It is obvious that there is no significant variation on the fault coverage of the proposed test-programs, while in all cases the proposed method clearly outperforms the baseline approach. The proposed method offers very high coverage ramp-up, which can further reduce the test-time in strictly constrained manufacturing and periodic-test applications.

We note that the proposed method cannot achieve complete stuck-at fault coverage on the OR1200 processor, due to the existence of functionally untestable faults, but it achieves a stuck-at fault coverage that is close to the one achieved by the method in [28], which is based on manual test generation and represents one of the highest reported values in the literature for the OR1200 processor. Moreover, the use of a processor for specific applications may restrict the usage of certain parts of the processor [18], thereby further reducing the maximum attainable fault-coverage for these devices. For example, the stuck-at fault coverage for the MAC unit is 99.0% in [28] and 95.67% in the proposed method. The results for the ALU unit are 91.9% in [28] and 88.57% in the proposed method, and the results for the Instruction Fetch unit are 23.5% and 74.19%, respectively. The number of instructions composing the test according to [28] is equal to 31.728 while in the proposed method it was equal to 8,487. We have to note that these results were obtained using different synthesized netlists of the processors and different fault lists.

Table I presents the test-program size, the test-time, the stuck-at fault-coverage and the transition fault-coverage of the baseline (BSL) and the proposed approaches for each one of the three proposed and baseline test-programs. We note that the proposed method offers the highest benefits when a small number of $TMIs$ are selected because the proposed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDMR.2019.2911022, IEEE Transactions on Device and Materials Reliability

TABLE II
STUCK-AT & TRANSITION FAULT COVERAGE PER UNIT

		Stuck-At Faults (%)		Transition Faults (%)		Number of Faults per Unit
	TMIs	BSL	Prop	BSL	Prop	
MULT	2x137	87.60	94.92	63.79	86.14	32,844
	4x137	92.54	95.41	73.64	86.96	
	8x137	94.00	95.67	80.87	88.23	
ALU	2x137	76.90	80.78	47.64	57.46	12,368
	4x137	82.41	85.86	57.97	67.19	
	8x137	85.34	88.57	67.71	70.05	
GENPC	2x137	57.78	58.84	31.71	33.56	4,072
	4x137	58.93	59.91	33.22	34.26	
	8x137	60.44	60.73	35.56	36.09	
CTRL	2x137	83.67	84.30	73.26	74.25	3,982
	4x137	83.94	84.80	74.14	74.40	
	8x137	84.19	84.87	75.03	75.11	
OPMUX	2x137	97.16	97.16	93.36	93.65	2,574
	4x137	97.16	97.16	93.64	93.71	
	8x137	97.16	97.16	93.73	93.73	
IF	2x137	71.41	73.42	50.84	52.74	2,322
	4x137	72.69	73.90	52.58	54.08	
	8x137	73.57	74.19	53.55	54.51	
LSU	2x137	82.45	84.69	55.95	59.89	2,282
	4x137	84.87	85.91	58.05	60.51	
	8x137	85.37	85.97	58.75	61.09	
WBMUX	2x137	75.28	76.78	53.81	56.30	1,826
	4x137	75.61	77.04	55.94	57.22	
	8x137	76.79	77.97	56.72	57.88	

output-deviation based metric is very effective in identifying the $TMIs$ with the highest non-modeled fault coverage (when the number of the selected $TMIs$ increases some less effective $TMIs$ are inevitably selected and the large gap between the proposed and the baseline approach reduces). Therefore, its effectiveness depends mostly on the potential of the output-deviation based metric to identify the most effective $TMIs$, and less on the amount of randomization (the variation of the proposed method among the three test programs is less than 0.5% in almost all cases). Moreover, the proposed method tends to select $TMIs$ with large numbers of WIs, therefore it generates test-programs slightly larger than the baseline method, even though they both select the same number of $TMIs$. Nevertheless, the defect coverage of the proposed method remains higher even in cases that the baseline test-programs contain more $TMIs$.

Table II compares the proposed method against BSL in terms of the average (among the three test-programs) fault coverage achieved for every targeted unit of the processor. The results on the various units are reported in descending order of the size of the units (the large units are reported first). It is obvious that the proposed method achieves higher stuck-at and transition fault coverage in almost all cases, but the higher benefits are on the two largest units, the *ALU* and the *Multiply-Accumulate (MAC)* unit. We note that the variation of the fault coverage of the proposed method was very low (less than 1%) in all cases, while it was higher for the baseline method. For example, the stuck-at fault coverage for the *Multiply-Accumulate (MAC)* unit for 274 $TMIs$ was between 84.48% and 89.18% for the baseline method and between 94.48% and 95.17% for the proposed method. The corresponding ranges for the *Write-Back Mux* unit were $75.14\% - 75.68\%$ and $76.68\% - 76.86\%$ respectively.

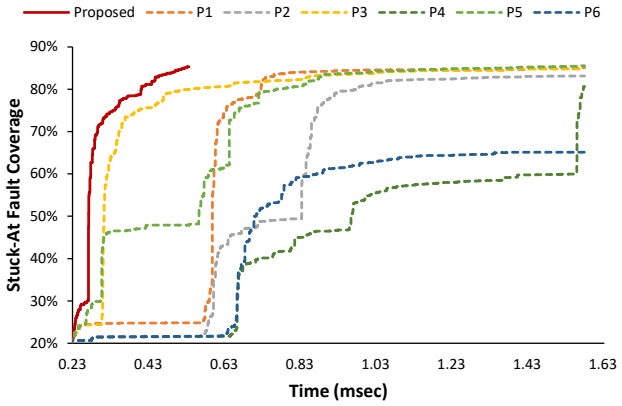


Fig. 10. Comparisons with other SBST programs (stuck-at faults).

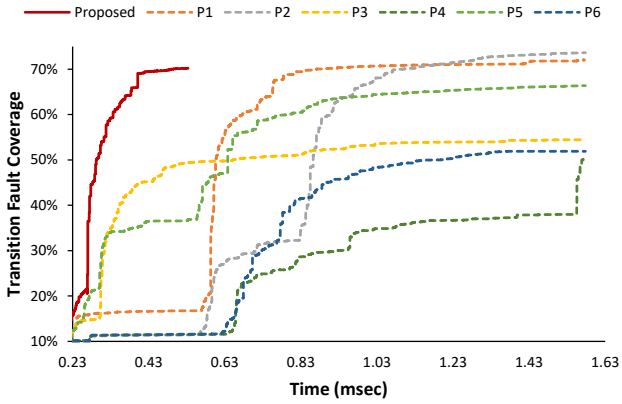


Fig. 11. Comparisons with other SBST programs (transition faults).

In Fig. 10 and Fig. 11 we compare test-programs generated using the proposed method with 274 $TMIs$ against SBST programs P1 to P6 (details about these test programs can be found in [15], [16]) in terms of stuck-at and transition-fault coverage. Test programs P1 to P6 have been generated manually, requiring several weeks of significant test-engineering effort. The proposed test-programs clearly outperform the rest of the programs in terms of test-time and test-program size (the test-program size is proportional to the test-time in most of the cases), while they offer higher or similar (in some cases) stuck-at and transition fault-coverage. Nevertheless, in every case the proposed test-programs offer considerably higher defect-coverage ramp-up, and taking also into account that they were generated very fast and almost fully systematically, the superiority of the proposed method becomes apparent.

V. CONCLUSIONS

In this paper we have presented an SBST test generation method that offers high non-modeled fault coverage in a semi-automatic manner and with short computational time. Instead of applying time-consuming fault-simulations using multiple fault-models, the proposed method uses logic simulation and a novel SBST-oriented probabilistic metric that exploits both

the architectural and the gate-level model of the processor. The proposed method is fast, it is almost fully automated, and it achieves high non-modeled fault-coverage ramp-up. Experiments on the OR1200 processor demonstrate the advantages of the proposed SBST method.

REFERENCES

[1] [Online]. Available: https://developer.arm.com/technologies/functional-safety

[2] [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[3] [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read

[4] [Online]. Available: http://www.cypress.com/file/249196/download

[5] [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

[6] [Online]. Available: https://www.hitex.com/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/

[7] "Openrisc 1200." [Online]. Available: {https://opencores.org/ocsvn/openrisc/openrisc/trunk/or1200/doc/openrisc1200_spec.pdf}

[8] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *IEEE Trans. on Computers*, vol. 58, no. 12, pp. 1682–1694, 2009.

[9] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda, "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices," *IEEE Design Test of Computers*, vol. 26, no. 2, pp. 52–63, March 2009.

[10] P. Bernardi *et al.*, "On-line software-based self-test of the address calculation unit in risc processors," in *17th IEEE ETS*, May 2012.

[11] ——, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *14th Intern. Workshop on Microprocessor Test and Verification*, Dec 2013, pp. 52–57.

[12] ——, "On the in-field functional testing of decode units in pipelined risc processors," in *IEEE Intern. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 299–304.

[13] ——, "Software-based self-test techniques of computational modules in dual issue embedded processors," in *20th IEEE European Test Symposium (ETS)*, May 2015, pp. 1–2.

[14] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *Design, Automation Test in Europe*, March 2011.

[15] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J.-G. Mess, "An analysis of test solutions for COTS-based systems in space applications," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018.

[16] R. Cantoro, E. Sanchez, and M. Sonza Reorda, "On the detection of board delay faults through the execution of functional programs," in *18th IEEE Latin American Test Symposium (LATS)*, 2017.

[17] S. D. Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, July 2011.

[18] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke processors for applications with ultra-low area and power constraints," in *2017 ACM/IEEE 44th ISCA*, 2017, pp. 41–54.

[19] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Trans. on Computers*, vol. 58, no. 8, pp. 1063–1079, Aug 2009.

[20] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante, "On the Test of Microprocessor IP Cores," in *IEEE DATE*, 2001, pp. 209–213.

[21] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 102–109, Mar 2004.

[22] E. Sanchez and M. Sonza Reorda, "On the functional test of branch prediction units," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1675–1688, Sept 2015.

[23] H. Fang, K. Chakrabarty, A. Jas, S. Patil, and C. Tirumurti, "Functional test-sequence grading at register-transfer level," *IEEE Trans. on VLSI Systems*, vol. 20, no. 10, pp. 1890–1894, Oct 2012.

[24] F. J. Ferguson and J. P. Shen, "A cmos fault extractor for inductive fault analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 11, pp. 1181–1194, Nov 1988.

[25] M. Grosso, W. Perez, D. Ravotto, E. Sanchez, M. Sonza Reorda, and J. V. Medina, "A software-based self-test methodology for system peripherals," in *2010 15th IEEE European Test Symposium*, May 2010, pp. 195–200.

[26] ISO/DIS26262, "Road vehicles - functional safety," 2009.

[27] X. Kavousianos, V. Tenentes, K. Chakrabarty, and E. Kalligeros, "Defect-oriented lfsr reseeding to target unmodeled defects using stuck-at test sets," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 12, pp. 2330–2335, Dec 2011.

[28] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-sbst methodology for efficient testing of processor cores," *IEEE Design Test of Computers*, vol. 25, no. 1, pp. 64–75, Jan 2008.

[29] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, April 2005.

[30] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective software self-test methodology for processor cores," in *DATE*, 2002, pp. 592–597.

[31] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 88–99, Jan 2005.

[32] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *DATE 2001*, 2001, pp. 92–96.

[33] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, May 2010.

[34] F. Reimann, M. Glaß, J. Teich, A. Cook, L. R. Gómez, D. Ull, H. J. Wunderlich, U. Abelein, and P. Engelke, "Advanced diagnosis: Sbst and bist integration in automotive e/e architectures," in *2014 51st ACM/EDAC/IEEE DAC*, June 2014, pp. 1–6.

[35] D. Sabena, M. Sonza Reorda, and L. Sterpone, "A new SBST algorithm for testing the register file of VLIW processors," in *DATE 2012*, pp. 412–417.

[36] A. Sanyal, K. Chakrabarty, M. Yilmaz, and H. Fujiwara, "Rt-level design-for-testability and expansion of functional test sequences for enhanced defect coverage," in *IEEE ITC*, Nov 2010, pp. 1–10.

[37] M. Schölzel, T. Koal, and H. T. Vierhaus, "Systematic generation of diagnostic software-based self-test routines for processor components," in *19th IEEE European Test Symposium*, May 2014, pp. 1–6.

[38] P. Singh, D. L. Landis, and V. Narayanan, "Test generation for precise interrupts on out-of-order microprocessors," in *10th International Workshop on Microprocessor Test and Verification*, Dec 2009, pp. 79–82.

[39] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "Daemonguard: O/s-assisted selective software-based self-testing for multi-core systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2013, pp. 45–51.

[40] G. Squillero, "Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs," *Computing*, vol. 93, no. 2-4, pp. 103–120, Oct. 2011.

[41] S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. on Computers*, vol. C-29, no. 6, pp. 429–441, June 1980.

[42] B. Vermeulen, C. Hora, B. Kruseman, E. Marinissen, and R. Rijsinge, "Trends in testing integrated circuits," in *ITC*, 2004, pp. 688–697.

[43] Z. Wang and K. Chakrabarty, "Test-quality/cost optimization using output-deviation-based reordering of test patterns," *IEEE Trans. on CAD*, vol. 27, no. 2, pp. 352 –365, 2008.

[44] Z. Wang, H. Fang, K. Chakrabarty, and M. Bienek, "Deviation-based lfsr reseeding for test-data compression," *IEEE Trans. on CAD*, vol. 28, no. 2, pp. 259 –271, 2009.

[45] C. H. P. Wen, L.-C. Wang, and K.-T. Cheng, "Simulation-based functional test generation for embedded processors," *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1335–1343, Nov 2006.

[46] G. Xenoulis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Instruction-based online periodic self-testing of microprocessors with floating-point units," *IEEE Trans. on Dependable and Secure Computing*, vol. 6, no. 2, pp. 124–134, April 2009.