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

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

Panagiotis Georgiou[1,2], Xrysovalantis Kavousianos[1,2], Riccardo Cantoro[1] and Matteo Sonza Reorda[1]
[1]Dept. of Control and Computer Engineering, Politecnico di Torino, Italy
[2]Dept. of Computer Science and Engineering, University of Ioannina, Greece

***Abstract*—Software-based self-test (SBST) is being widely used (often as a complement to other techniques) 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 SBST method. 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.**

I. Introduction

Current ICs 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, sometimes impose the use of non-intrusive test methods [15]. Therefore, design-for-testability solutions, like scan or built-in self-test, can be complemented with functional solutions, such as software-based-self-test for processor-based ICs. SBST executes test-programs that activate potential faults inside the circuit and propagate the errors to observable sites, like the memory [22], [23], [29]. SBST is fully autonomous, it does not alter the circuit structure and it does not affect its performance. Moreover, it is applied exactly at the operating conditions without over-testing the circuit, it facilitates the periodic monitoring in-the-field with limited intrusiveness with respect to the normal-mission operation [11], [20], and it does not compromise the internal state of the circuit [23].

The major challenge of SBST is the generation of small test-programs that offer high fault coverage in short test-time [21], [31], [35]. SBST programs can be generated manually [28], semi-automatically [12] or automatically, targeting different processor architectures and fault models [29]. Various methods target microprocessors with caches [9], shared-memory schemes [2], floating-point units [36] and dual-issue processors [6]. Deterministic techniques exploit the regularity of sub-modules [4], [5], [13], [17], [18], [24], [25], while others use automatic-test-pattern-generation (ATPG) [27] and evolutionary algorithms [3], [27], [30]. In [7] the effectiveness of SBST for a given level of dependability is evaluated.

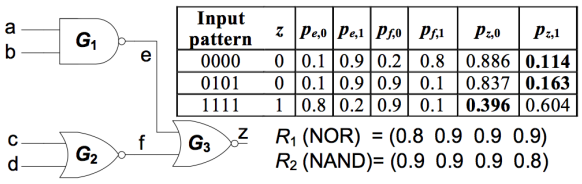
Despite their benefits, SBST methods suffer from drawbacks too. First of all, they often target the traditional stuck-at fault model, which is inadequate for detecting many defects. In addition, most SBST techniques are not systematic, therefore they require extensive human intervention and long development 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 by the most commonly used fault models [32]. 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 processor in 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 processor. 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 and in abort-at-first-fail environments in manufacturing testing.

II. Motivation

SBST methods typically target only the stuck-at fault model [3]–[7], [12], [13], [17]–[20], [22]–[25], [27], [29], [30], [35], [36]. However, 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 [14], [26] for non-scan sequential circuits modeled at the register-transfer level (RTL). However, RTL models limit the effectiveness of these

This work has been supported by the European Union through the H2020 project no. 637616 (MaMMoTH-Up).



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

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 processors have very few (or even none) observable outputs (the main observation site is the embedded memory of the system). 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 on the other hand are very effective in detecting silicon defects in structural testing [16], [33], [34]. They are probability measures that reflect the likelihood of error detection at circuit outputs; they are computed without explicit fault grading, hence the computation (linear in the number of gates) is feasible for large circuits. At first 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. Let y be the output of a NAND gate with inputs a, b. We have $p_y^0 = p_a^0 p_b^0 (1 - r_i^{00}) + p_a^0 p_b^1 (1 - r_i^{01}) + p_a^1 p_b^0 (1 - r_i^{10}) + p_a^1 p_b^1 r_i^{11}$ and $p_y^1 = p_a^0 p_b^0 r_i^{00} + p_a^0 p_b^1 r_i^{01} + p_a^1 p_b^0 r_i^{10} + p_a^1 p_b^1 (1 - r_i^{11})$. Likewise, signal probabilities can be computed for other gate types [33]. 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 t_j. The deviation values at the circuit outputs are indicative of the probability for arbitrary defects to be detected at these outputs.

Fig. 1 shows a circuit with three gates G_1, G_2, G_3, and confidence level vectors R_1, R_2 assigned to the NOR and NAND gates [33]. 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 most promising pattern for detecting defects is $(a, b, c, d) = (1, 1, 1, 1)$ 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 *test macros* [12], [19]. Test macros are very effective for stuck-at faults, but they have never been exploited for non-modeled faults. Every macro is associated with one machine-level instruction executing a specific function wrapped with additional instructions that set the *macro parameters* (i.e., the operand values) and propagate

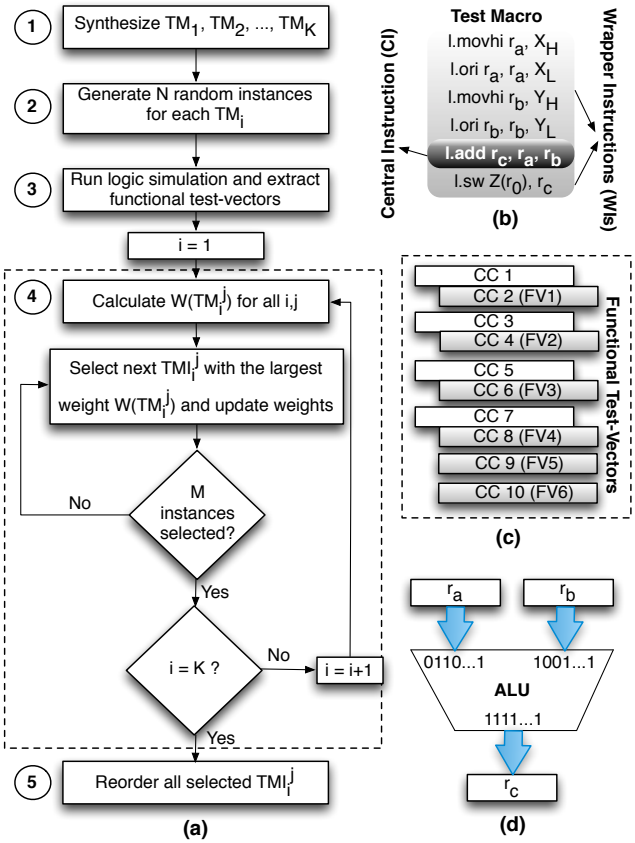


Fig. 2. (a) Test Flow (b) TMI example (c) Functional Vectors (d) ALU Test

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 defects, and by randomly varying their operands. Each test-macro instance (TMI) is evaluated by the means of a novel output-deviation-based metric computed at 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

Fig. 2a presents the proposed test-generation method, which consists of five steps. 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 [1] executes the arithmetic operation $r_c = r_a + r_b$ (r_a, r_b, r_c are general purpose registers), and it exercises 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, r_c, which are set by the means of the *Wrapper-Instructions (WIs)*. For example, the test-macro generated for the instruction *add* r_c, r_a, r_b is shown in Fig. 2b. The first four instructions are WIs 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 CI and the last instruction is a WI that stores the result at the physical address obtained by combining Z, r_0.

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

The defect coverage of every test-macro depends on the operands of the central and wrapper-instructions. To this end, we propose a completely automated process to generate test-macro instances ($TMIs$) with high non-modeled defect 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 the registers r_a, r_b, r_c, and their values, as well as the values X, Y and Z.

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 at the inputs of the ALU (see Fig. 2d). These logic values constitute one functional test-vector/response applied by the TMI. Every TMI generates a number of functional test-vectors/responses that excite various units 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 (they are fetched from the memory), and the functional test-vectors are generated at their second clock cycles. The last two instructions require one clock cycle to be executed (they are fetched from the cache), 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.

In order to evaluate the functional test-vectors of the $TMIs$ 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.

In order to identify the most effective functional test-vectors, we apply first the vectors that excite one unit, and we calculate the maximum deviation value that is generated at every output 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 $FV(TMI_i^j, 1)$, $FV(TMI_i^j, 2)$, ..., $FV(TMI_i^j, NV_i)$ 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 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 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. Therefore, we set $Faults(p)$ 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 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

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

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 defects.

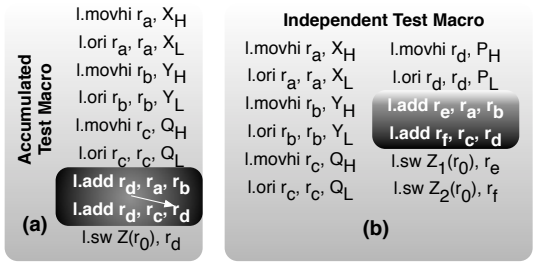


Fig. 3. (a) A-TM example (b) I-TM example

At the next step, for every test-macro TM_i the M most effective out of the N random instances TMI_i^j are selected as follows: at first, the instance TMI_i^j with the highest weight $W(TMI_i^j)$ is selected. Since TM_i^j tends to detect many defects at the outputs $p \in \bigcup_{k=1}^{NV_i} MS(TMI_i^j, k)$ the effectiveness of the rest of the instances with high deviations at these outputs decreases (less defects tend to remain undetected at these outputs). Therefore, we decrease the priority of these outputs by a constant factor F and the weights of the remaining instances are re-computed by applying again eq. (1), (2). Then, the next instance with the highest weight is selected and the same process continues until M instances are selected. The higher is the value of F, the faster we select instances that maximize the deviation values at all the outputs.

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 in such a way to achieve the highest defect-coverage ramp-up. In this case, all the $M \times K$ instances are evaluated together by considering all the targeted units of the processor. We set $Faults(p)$ at their initial values and the weights $W(TMI_i^j)$ are re-computed. The instance with the highest weight is selected every time and the values of $Faults(p)$ are updated as shown before.

The test-macros generated using this method are very effective in detecting non-modeled defects. However, the use of a single CI reduces the delay-defect coverage in units like the ALU, which require the application of pairs of successive functional test-vectors, and thus pairs of successive *CIs*. In Fig. 3 we present: a) the *Accumulated-Test Macro* (*A-TM*) that embeds two similar successive *CIs* (one operand of the second *CI* is the result of the first *CI*), and b) the *Independent-Test Macro* (*I-TM*) that embeds two independent successive *CIs*. 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 of the second CI. However, in the case of *I-TMs* separate evaluation and selection of the first and second *CI* is required (note that both CIs detect defects, but most of the delay-defects are detected by the second *CI*). To this end, different priority values PR and sets MS of outputs with high deviation values are manipulated for the first and the second *CI* of all *I-TMIs*, while the generation of the *I-TMIs* is done as follows: multiple random instances of regular *TMIs* with a single CI are generated, but only the functional test-vector of their CI is evaluated using eq. (1), (2). The two *CIs* with the highest weights are combined to generate an *I-TMI* (the *CI* with the highest weight is used as second in the pair to favor the detection of delay defects). Then, the selected CIs are removed and the same process is repeated for generating the next N *I-TMIs* using the remaining *CIs*.

IV. Experimental Results

In order to evaluate the proposed methodology we run extensive experiments on the 32-bit scalar OR1200 RISC processor [1], which has a Harvard micro-architecture, 5-stage integer pipeline, virtual-memory support (MMU) and basic DSP capabilities. The most important units of the processor are the *ALU*, the *Multiply-Accumulate (MAC)*, the *Load-Store*, the *Instruction-Decoder*, the *Register-File* and the *Exception-Handling* unit. The processor has one embedded instruction cache and one embedded data cache of size 8KB each. The RTL netlist of the processor was synthesized into a gate-level netlist 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 proposed methodology was developed using C++ and Python. All the major units of the processor were targeted except of the exception-handling unit and the embedded caches, which require special test-generation mechanisms. 122 $A-TMs$ and 122 $I-TMs$ were generated consisting of 3-12 and 3-18 instructions, respectively (one $A-TM$ and one $I-TM$ for every instruction used as CI). For each test-macro 150 random instances were generated (overall 18,300 $A-TMIs$ and 18,300 $I-TMIs$) and 12 test-programs were automatically generated as follows:

- **Baseline (A-TM):** 2, 5 and 10 instances were selected randomly out of the 150 instances generated for every $A-TM$ (overall 244, 610 and 1220 $A-TMIs$).
- **Baseline (I-TM):** 2, 5 and 10 instances were selected randomly out of the 150 instances generated for every $I-TM$ (overall 244, 610 and 1220 $I-TMIs$).
- **Prop. (A-TM):** 244, 610 and 1220 $A-TMIs$ were selected using the proposed method.
- **Prop. (I-TM):** 244, 610 and 1220 $I-TMIs$ were selected using the proposed method.

For every baseline approach 3 different test-programs were generated, and the average defect coverage is reported. The running time of the proposed method on a single 64-bit CPU running at 1.2GHz 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, they were applied using commercial tools for evaluating the generated test-programs on a server with 48 CPUs running at 2.5 GHz. The stuck-at fault-simulation time for each test-program

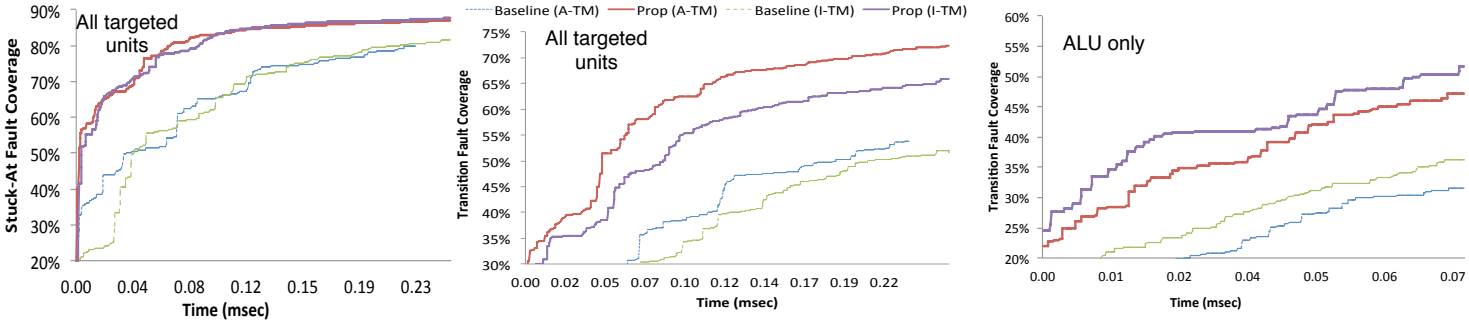


Fig. 4. Stuck-At and Transition-Delay Fault Coverage (244 $TMIs$).

was between 3.5 days (for 244 $TMIs$) up to 12.5 days (for 1,220 $TMIs$), and the transition-delay fault-simulation time was between 4.5 days and 2 weeks.

Fig. 4a and Fig. 4b present the stuck-at and the transition fault-coverage of all the targeted units of the processor for the test-programs with 244 $TMIs$. The x-axis presents the test-time (clock cycles) and the y-axis presents the fault coverage. It is obvious that the proposed methods drastically outperform the baseline approaches in terms of stuck-at and transition fault-coverage, but most importantly they offer very high coverage ramp-up, which can further reduce the test-time in strictly constrained manufacturing and periodic-test applications. 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 coverage very close to the one reported in [17] for the same processor (although in that case test generation was manual). Moreover, the use of general purpose processors for specific applications may restrict the usage of certain parts of the processor [10], thereby further reducing the maximum attainable fault-coverage for these devices.

$I{-}TMs$ are conjectured to offer higher delay fault coverage than $A-TMs$ as they do not suffer from correlation between the operands of the first and the second CI. However, in the case of the OR1200 processor, all possible CIs pairs that exercise the MAC unit are inherently correlated because they use accumulator-type instructions. This feature imposes an additional WI between the first and the second CI in the $I-TMs$ to break this correlation, which adversely affects the non-modeled delay fault coverage of this unit. As shown in Fig. 4, $I-TMs$ offer higher stuck-at coverage for the processor, and higher transition-fault coverage for units like the ALU that do not suffer from this limitation. However, the reduced effectiveness of $I-TMs$ for detecting delay defects at the MAC unit adversely affects the defect-coverage of the processor. Nevertheless, $I{-}TMIs$ are expected to offer higher delay-defect coverage than $A{-}TMIs$, except for units similar to the MAC unit, where $A-TMs$ should be used instead.

Table I presents the test-program size, the test-time, the stuck-at fault-coverage and the transition fault-coverage for the baseline (BSL) and the proposed approaches. We note that the proposed method offers the highest benefits when a

TABLE I
SOFTWARE-BASED-SELF-TESTING RESULTS

		Test Program Size (KBytes)		Test Time (msec)		Stuck-At Faults (%)		Transition Faults (%)	
	TMIs	BSL	Prop	BSL	Prop	BSL	Prop	BSL	Prop
A-TM	2x122	8.9	9.8	0.058	0.064	80.3	87.7	54.1	73.6
	5x122	19.1	20.8	0.104	0.112	86.7	89.8	67.9	78.3
	10x122	36.1	36.1	0.179	0.179	89.1	90.4	74.3	81.0
I-TM	2x122	9.6	13.8	0.061	0.081	81.9	88.9	52.2	70.3
	5x122	21.0	27.7	0.111	0.139	88.5	90.7	65.3	75.3
	10x122	39.9	47.4	0.193	0.224	91.0	91.2	72.7	76.7

small number of $TMIs$ are selected because the proposed output-deviation based metric is very effective in identifying the $TMIs$ with the highest non-modeled fault coverage. 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 (note that when the number of the selected $TMIs$ increases some less effective ones are inevitably selected and the large gap between the proposed and the baseline approach reduces). We also note that the larger size (and test-time) of the test-programs based on $I-TMIs$ as compared to $A-TMIs$ is due to the higher number of WIs in $I-TMIs$ (see Fig. 3). Moreover, the proposed method tends to select $TMIs$ with large numbers of WIs, therefore it generates test-programs slightly larger than the baseline method. Nevertheless, the defect coverage of the proposed method remains higher even in cases that the baseline test-programs contain more $TMIs$.

Finally, in Fig. 5 we compare test-programs with 244 and 610 A-TMIs generated using the proposed method against various SBST programs [8]. The proposed test-programs outperform the rest of the programs in terms of both test-program-size and test-time, while they offer higher (or similar) stuck-at and transition fault-coverage. Similarly, our method achieves a comparable stuck-at fault coverage with respect to [17], with much lower test duration. 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

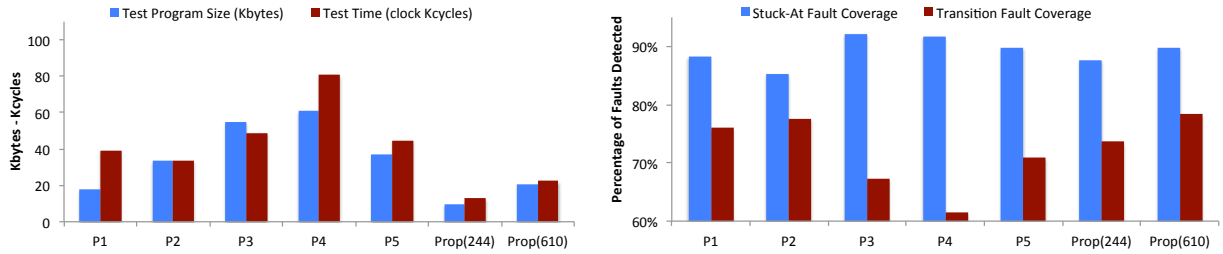


Fig. 5. Comparisons against other SBST programs.

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

[2] 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.

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

[4] ——, "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.

[5] ——, "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.

[6] ——, "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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] F. Corno, E. Sanchez, M. S. 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] 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.

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

[25] E. Sanchez and M. S. 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.