On the Automatic Generation of SBST Test Programs for In-Field Test

Andreas Riefert * Riccardo Cantoro † Matthias Sauer * Matteo Sonza Reorda † Bernd Becker *

* Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
{ riefert | sauerm | becker }@informatik.uni-freiburg.de

† Politecnico di Torino
Corso Duca degli Abruzzi 24
10129 Torino, Italy
{ riccardo.cantoro | matteo.sonzareorda }@polito.it

***Abstract*— Software-based self-test (SBST) techniques are used to test processors against permanent faults introduced by the manufacturing process (often as a complementary approach with respect to DfT) or to perform in-field test in safety-critical applications. A major obstacle to their adoption is the high cost for developing effective test programs, since there is still a lack of suitable EDA algorithms and tools able to automatically generate SBST test programs. An efficient ATPG algorithm can serve as the foundation for the automatic generation of SBST test programs.**

In this work we first highlight the additional constraints characterizing SBST test programs wrt functional ones, with special emphasis on their usage for in-field test; then, we describe an ATPG framework targeting stuck-at faults based on Bounded Model Checking. The framework allows the user to flexibly specify the requirements of SBST test programs in the considered scenario. Finally, we demonstrate how a set of properly chosen requirements can be used to generate test programs matching these constraints.

In our experiments we evaluate the framework with the miniMIPS microprocessor. The results show that the proposed method is the first able to automatically generate SBST test programs whose fault efficiency is superior to those produced with state-of-the-art manual approaches.

I. Introduction

Testing a processor against permanent faults arising during the manufacturing process or during the operational phase is a complex task, no matter whether the processor is a stand-alone device or a core within a System on a Chip (SoC). In several cases, the task can be solved by resorting to Design for Testability (DfT) techniques, such as scan. However, functional approaches based on forcing the processor to execute a suitably written test program and checking the produced results are gaining momentum, and increasingly complement existing DfT-based approaches for specific purposes. The new popularity of the functional approach, that was first proposed already three decades ago [1], has several motivations, including the following:

- The higher defect coverage that can be achieved by a functional test, since it typically runs at the processor's operational frequency.
- The lower overtesting it produces, since it excites the processor circuitry exactly as the operational phase does, without the reconfiguration involved by DfT.
- The reduced external ATE requirements: Functional approaches are often implemented in the form of Software-Based Self-Test (or SBST [2]), which involves first uploading the test program at low speed in an internal memory, then running it at full speed, and finally downloading the results produced by the processor at low speed. The external ATE is only involved in the first and last phase, that can be performed at low speed.
- The possibility of exploiting functional test sequences for detection of permanent faults during the operational phase.

Finally, it must be mentioned that sometimes the DfT structures, even if present, cannot be used by the test engineer, e.g., because they are not documented by the processor producer: this is for example the typical scenario in which system companies operate when they develop the in-field test for a processor.

As a consequence, functional test approaches are increasingly adopted by industry for manufacturing ([3][4]) and in-field test [5], and also became a hot research topic.

In this paper we consider a common scenario for companies manufacturing SoC devices to be used in safety-critical applications: The netlist of the processor core is available, DfT structures possibly exist, but some SBST test program is required, either to complement scan test for manufacturing test, or to provide a test usable during the operational phase (*in-field test*) [6]. In this scenario a major issue is the cost for generating such a test program. In fact, commercial tools still lack comprehensive support for functional test program generation, and hence this task has to be performed manually, requiring a significant effort by skilled programmers. It is crucial to note that the complexity of the task not only stems from the difficulty of writing a test program able to achieve a sufficient fault coverage, but also derives from the fact that the test program must be activated in the operational phase. This means that several constraints imposed by the specific test infrastructure of a SoC need to be considered (e.g., related to the size and location of the memory area available for storing the test program code and data, as well as the bus protocol). Moreover, the test program must match a number of additional constraints connected to the specific characteristics of the SBST paradigm: As an example, each time the processor fetches an instruction from the same memory location, the same instruction must be retrieved and executed. Similarly, if the test program reads the content of a memory location, the value returned by the memory must be the same of the last write operation on the same memory location. All these constraints do not exist in other scenarios,

978-3-9815370-4-8/DATE15/©2015 EDAA

e.g., when an ATE tests a processor following the functional approach. As a result, we can claim that writing an SBST test program is more constrained than writing a generic functional test program. To the best of our knowledge the method we propose in this paper is the first able to flexibly specify the constraints characterizing the addressed scenario and to automatically generate SBST test programs matching them.

The approach we propose is based on the usage of Bounded Model Checking (BMC). We first show how to manage stuck-at faults when working with the BMC paradigm; we then describe a method to specify constraints, and we summarize how to express the typical constraints of an SBST scenario.

When compared with other methods (e.g., [7]) our approach is easier to adopt, since it does not require modeling the processor architecture, but only specifying constraints on a high abstraction level. We also show that the way in which constraints are imposed heavily impacts the efficiency of the method, and provide some solutions to effectively trade-off the quality of the achieved results with the required computational effort. Moreover, our algorithm is able to identify untestable faults (in particular those caused by the additional constraints related to the SBST in-field scenario), which is crucial when the processor is used in safety-critical applications, where a given fault coverage is required by standards and regulations (e.g., ISO 26262 [8]). Experimental results gathered on the miniMIPS processor [9] demonstrate that the approach is able to automatically generate an effective SBST program, which is superior to test programs generated by sophisticated manual methods (e.g., [7]).

The organization of the paper is the following. Sections II and III discuss related work and the preliminaries, respectively. Section IV describes the framework used to perform the test generation procedure. In Section V the interface for the specification of constraints is explained and the developed set of constraints for SBST is detailed. Finally, Section VI presents the experimental results and Section VII draws the conclusions.

II. Related work

A significant amount of research has been carried out in the field of functional test pattern generation. Several ATPG tools for sequential or functional test pattern generation are simulation-based. [10] constitutes an example for this approach. In this work an initially random test set is modified incrementally using wavelet and inverse wavelet transformation in order to improve its fault coverage. In [11] a combination of simulation-based sequential ATPG with SAT-based Bounded Model Checking (BMC) is proposed. The ATPG procedure is intended to target testable faults, whereas BMC shall identify untestable faults. [12] provides a detailed comparison of SAT-based BMC with sequential ATPG based on structural methods. In [13] a new approach based on BMC was explored. It proved to be able to effectively and automatically generate test sequences for small-delay faults for a pipelined processor. A major contribution of that paper was the description of a method allowing to specify constraints for the generated test patterns.

An extensive overview of processor and software-based self-test is given in [2]. Several approaches first generate a test sequence for a fault on a low level by only considering the faulty module. Then they try to extend this test to a sequence of processor instructions. The method in [14] computes low level test sequences with an ATPG tool and maps these sequences to instructions with the Cadence SMV BMC. In their experiments the authors investigate the OpenRisc 1200. A similar approach is proposed in [15], where a SAT solver is used for justification and propagation of the precomputed test. An approach for the automatic generation of an SBST test program is presented in [16], where an evolutionary algorithm is used to improve the quality of a given test suite. An additional FPGA-based hardware acceleration is utilized in order to speed up the fault simulation. [17] proposes the use of an evolutionary algorithm in combination with Binary Decision Diagrams (BDDs). Both together are used to generate an SBST test program for path-delay faults. [7] focuses on testing the faults related to the pipeline of a microprocessor. The authors develop an SBST methodology in order to tackle these faults and provide experimental results for the miniMIPS and OpenRISC 1200. Lastly, [18] proposes an instruction-based test generation for a pipelined processor by manually creating a graph model representing the behavior of the processor.

However, all the previously mentioned papers mainly address the generation of SBST test programs for end-of-manufacturing test, and no one specifically addresses the constraints which are typical of in-field test (as this paper does). This means that in practice the generated test programs require a manual step to check their compliance with the SBST constraints, triggering a complex and expensive transformation in the likely case that they don't match them. Moreover, our approach is able to provide test sequences of optimal length and to prove untestability of faults. Furthermore, it provides a flexible interface allowing to specify constraints for the test sequences. This enables the automatic generation of an effective SBST test program for in-field test.

III. Preliminaries

We provide essentials on SAT-based Bounded Model Checking (BMC) together with Craig Interpolation as far as they are necessary for the understanding of the following. More details can be found in [19], where functional ATPG for small-delay faults is presented. The underlying engines of [19] are used in our approach too.

The core of our ATPG framework is a solver which applies BMC together with Craig Interpolation. A classical BMC solver tries to solve a formula, which is defined by an initial state I_0, a transition relation $T_{i,i+1}$ and a target property P_k

$$BMC_k = I_0 \wedge T_{0,1} \wedge ... \wedge T_{k-1,k} \wedge P_k \quad (1)$$

$T_{i,i+1}$ defines the progress of the system from timeframe i to $i+1$, whereas P_k specifies the property to be verified. Starting with $k = 0$ the solver searches for a solution which satisfies the target property or proves that the target property cannot be satisfied within k steps. k is increased stepwise until a solution is found or no new system states can be reached. In

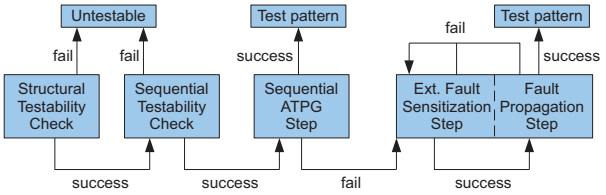

Figure 1. ATPG flow

general the latter case requires very large values for k, which are not feasible in practice.

However, several approaches for a more efficient unreachability proof exist. The solver used for this work [20] uses Craig Interpolants to over-approximate the reachable system states within each step. This in many cases allows to prove effectively that a target property will not be satisfied for arbitrary values of k.

IV. ATPG framework

This section describes the proposed algorithm for functional ATPG. The overall flow of the test generation process is as follows: A stuck-at fault list is used, where each fault is processed one after the other. The test generation for the first considered fault starts from the state after the application of the initialization sequence as we cannot assume the existence of a scan chain. If a test sequence is found, it is appended to the initialization sequence. Then a fault simulation is executed and all faults, which are covered additionally to the target fault, are dropped. The final state of this test sequence is then used as the starting state for the test generation of the next fault.

In the first step of the actual ATPG process (see Figure 1) a check is executed, which determines whether a fault is structurally testable. This check is done with a SAT solver and identifies all faults which are untestable within a scan design. These faults do not have to be considered anymore, as they also will not be testable in the more restrictive functional scenario. This step is convenient as it is significantly faster than the following sequential testability check.

The second step determines whether the considered fault is sensitizable within the sequential test scenario. For the transition relation a '01'-encoding of the circuit is created. The target property requires that the considered faulty signal line is sensitized with the proper logic value. If the solver returns unsatisfiable, it has been proven that the fault is sequentially untestable, i.e., there exists no sequence which can test the fault.

If both the first and the second step were satisfiable, we know that for the currently considered fault a sequence exists which is able to sensitize this fault. The third step then tries to find a test sequence which sensitizes the fault and propagates the fault effects to a primary output. For the transition relation of the formula a 'Good-Bad'-encoding of the circuit is created, where 'Good' models the fault-free and 'Bad' the faulty circuit. The target property requires that a fault effect, i.e., a difference between 'Good' and 'Bad', is visible at a primary output. If the solver returns a satisfiable solution, a valid test sequence can be extracted. Please note that the solver is optimal with regard to the unrolling depth. Thus it always returns the shortest possible test sequence from the given starting state. If the solver returns unsatisfiable, it has been proven that no test sequence of arbitrary length exists which propagates the fault effect to a primary output.

If testing the considered fault requires a too high unrolling depth of the circuit, the solver aborts after a user-defined time bound. In this case the test sequence generation is divided into two steps, namely the extended fault sensitization and the fault propagation step. The idea is to approach the problem with two solver calls, in order to find solutions with higher unrolling depths. In order to compute the extended fault sensitization sequence, again a 'Good-Bad'-encoding of the circuit is utilized. The target property requires the sensitization of the faulty signal line and the propagation of the fault effect to a directly (i.e., in one time frame) reachable flip-flop. If the fault effect could be latched in a flip-flop, a sequence is required which propagates it to a primary output. The fault propagation step utilizes the final state of the extended fault sensitization as its initial state and the same 'Good-Bad'-encoding for the transition relation. In its target property we require the fault effect to be visible at a primary output. If the fault propagation step succeeds the solutions from the extended fault sensitization and the fault propagation together form the test sequence for the considered fault.

If the fault propagation fails, it is most probably due to the fact, that the fault effect was latched in a circuit state which prevents any further propagation. For example, assume a register in a microprocessor which temporarily stores the result of the ALU. This result is only used if the processor is currently executing an ALU instruction. If not, the register value is overwritten in the next clock cycle and a possibly stored fault effect immediately vanishes. This means that the extended fault sensitization step has to latch the fault effect into a system state which allows the further propagation. For this reason we created a heuristic to guide this ATPG step and latch the fault into suitable system states. We have randomly chosen several functional circuit states, where we consecutively inserted a fault effect in each flip-flop and tried to propagate it to a primary output by utilizing our solver. Depending on how often the fault propagation succeeded for a given circuit flip-flop and how long the corresponding sequence was, we heuristically compute a score for each flip-flop which expresses how suitable it is for fault propagation. In the extended fault sensitization step we now modify the target property by requiring the fault propagation to flip-flops with high scores. The extended fault sensitization step is repeated for a user-defined number of iterations. In the first iteration only flip-flops are targeted which are directly reachable from the fault location. If the extended fault sensitization into one or more of these flip-flops succeeds but the consecutive fault propagation to a primary output fails, the next iteration also targets suitable flip-flops which are reachable directly from the flip-flops which could be sensitized in the first iteration.

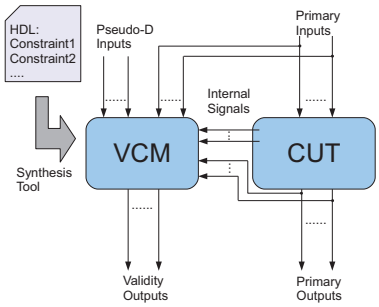

Figure 2. Application of the Validity Checker Module

V. Constraining the input space

A. Validity Checker Module

The functional ATPG algorithm described in the previous section computes pattern sequences, which test a fault by applying values to primary inputs and observing primary outputs. For the applied primary input values each possible combination is allowed and primary outputs can be ovserved at each clock cycle. However, this is unrealistic for the SBST test of a processor. For example, a microprocessor which is embedded in an environment will only load (valid) instructions and data from the available memory. Furthermore, the communication with the memory has to respect a certain memory protocol. If the generated test sequences also have to respect such additional constraints, which are required for functional or software-based self test, the input space has to be constrained in a suitable way.

In [13] a *Validity Checker Module* (VCM) was proposed for the functional test of small-delay faults. The idea of the VCM is to provide an easy-to-use interface for a test engineer for specifying functional constraints (see Figure 2). The VCM is a circuit which can be specified in a hardware description language like VHDL or Verilog. The inputs of the VCM comprise the primary inputs, primary outputs and internal signals of the Circuit Under Test (CUT), which are required to describe the desired constraints. For each constraint the VCM contains one validity output, which is active if the constraint is fulfilled and inactive if not. After the VCM has been designed, it is synthesized to a gate netlist. This netlist is properly combined with the netlist of the CUT. Both together represent the input netlist for the ATPG algorithm.

During the execution of the algorithm the validity outputs are required to be active, thus generating sequences which respect the specified constraints. Please note that the final CUT is not changed in any way. The described process is only required to integrate the specified constraints into the ATPG algorithm.

In [13] a basic usage of the VCM concept was explored, in which only *invariant constraints* have been supported, i.e., constraints which have to be fulfilled at each clock cycle. In this work we extend this concept by considering constraints whose fulfillment is only checked at the last cycle of a sequence and denote them as *target constraints*. Furthermore, we introduced the so called *pseudo-D* inputs for the VCM. A Pseudo-D input is a primary input in the VCM and corresponds to a fault effect at a certain signal line or register in the CUT. This allows the user to specify constraints with regard to fault propagation. By using a unique suffix, a Pseudo-D input is identified by our ATPG tool and connected to the corresponding internal variable.

In general, a constraint can invalidate the test generation for a certain fault, i.e., a fault which was testable before becomes untestable because of the constraint. For this reason it is possible to turn each constraint on or off during the ATPG process, for example when we realize that the constraint overrestricts the search space.

Our framework can also be used as guidance for the development of new constraints. By first executing an unconstrained ATPG run a user can get an impression of how a certain fault is tested. This can be used as a starting point for adding simple and less restrictive constraints. These constraints are then stepwise improved until the generated sequences satisfy all requirements of the user. The execution of the ATPG at each step gives feedback about the quality of the constraint with respect to the given test environment and allows to identify ineffective or overly restrictive constraints. We applied this flow to devise the constraints which are described in the following.

B. Constraints for in-field SBST

Developing a program suitable for in-field SBST is a complex task, as it not only requires a fault to be sensitized and propagated to a primary output, but additionally to respect several further constraints (e.g., concerning the coherence of the memory content). Consequently, not only a proper test sequence for a fault has to be found, but also other constraints have to be considered too, depending on the specific addressed scenario. This requires accurate and flexible modeling of these constraints.

In order to better clarify the kinds of constraints that should be considered when creating an SBST test program for in-field test, we are going to list them in the following.

A first set of constraints (denoted as *functional constraints*) was already introduced in [13], as listed as follows:

1) *Reset*: The system reset signal is only allowed to be active once at initialization and then has to remain inactive, as activating it at certain clock cycles would be hard to control in an in-field SBST scenario.

2) *Interrupt*: For a similar reason, external interrupts are not allowed.

3) *Memory protocol*: The memory protocol adopted by the processor should be enforced, e.g., imposing a given memory response time.

4) *Valid instructions*: If the processor is loading an instruction, only valid instructions are allowed to be applied.

Using the functional constraints during ATPG results in test sequences, which can only be applied to the microprocessor by resorting to an external ATE. It means, they cannot be directly mapped to program and data memory. An extended set of constraints (denoted as *SBST constraints*) has been developed in order to meet this requirement. In the following the additional constraints are described:

5) *Code memory coherence*: During the execution of an SBST test program, successive fetch operations from the same memory cell should return the same instruction.

6) *Data memory coherence*: During the execution of an SBST test program, any couple of read accesses to the same data memory cell should return the same value, unless a write operation is performed on the same cell in the middle. Moreover, any read operations should be preceded by a write operation initializing the accessed memory cell with the required logic value.

7) *Fault detection*: As in the SBST scenario only the memory content after the program execution can be observed, a fault must lead to a memory content which differs from the fault free one.

C. Constraint enforcement

The above constraints can be enforced resorting to different solutions, all mapped on a specific VCM implementation. Different solutions may result in different impact on the ATPG.

All functional constraints (1 to 4) and constraint 7) are implemented as purely combinational blocks. They produce a reduction of the search space in the ATPG process. On the contrary, constraints 5) and 6) are implemented as sequential blocks, which may increase the search space.

An effective solution to force the fulfillment of the constraint 5) is to prevent instructions which could decrease the program counter. This avoids several fetch operations from the same memory cell. In order to implement this constraint we propose to only allow branch instructions executing forward jumps with a fixed value. The value is chosen by taking into account the number of fetch operations which take place after loading the branch instruction until its execution. This number is determined by the number of pipeline stages which have to be passed before the instruction is executed. The constraint is fulfilled also due to the fact that any kind of interrupts are not permitted.

The aforementioned implementation of 5) presents some critical aspects. The first issue concerns the branch prediction unit, which requires jumping to the same memory address more than once. We used a specialized constraint to test this module. It consists of a state machine realizing a small loop, which includes a store instruction. A fault will prevent the proper execution of the store instruction. Furthermore, jumps with a fixed offset result in high instruction addresses being hard to reach. For this reason we propose a specialized constraint, which enables a sequence of two jump and link instructions and one store instruction. A fault in the address logic will cause a jump to a wrong target address and storing the wrong program counter value.

An effective strategy for enforcing the constraint 6) is to define a starting address and then to force each load and store instruction to use a distinct memory address as their target address. In our implementation a general purpose register is used to store the starting address. All load and store instructions use this register value for their target address. After the execution of each load or store instruction the register value is incremented. Thus all memory instructions will always address distinct memory cells. Memory cells which are addressed by load instructions are pre-charged with the computed value.

The constraint 7) can be addressed by exploiting some Pseudo-D inputs (see V-A). We require that either the memory write signal is faulty or the memory write signal is active and a fault effect is visible at either the address or the data bus. The first requirement will cause a store instruction which is only executed in either the fault free or the faulty case. The latter requirement leads to an executed store instruction which either writes to an erroneous memory address or writes an erroneous data value into the memory.

Taken together, the proposed solutions allow the designer to flexibly trade-off between the complexity of the constraint and their effectiveness. In some cases (e.g., forbidding interrupts), the proposed solution slightly reduces the achieved fault coverage, but proved to be computationally effective.

VI. Experimental Results

The miniMIPS processor was used to prove the viability and effectiveness of the proposed approach. The available RT-level description [9] was synthesized with Synopsys Design Vision using an in-house developed library. The resulting gate netlist contained 18,279 gates and 1,966 flip-flops which were all considered for stuck-at fault test generation. The VCM containing all described constraints (Section V-B) was specified in VHDL and comprises about 400 lines of code which resulted in a synthesized netlist consisting of 395 gates and 21 flip-flops. All ATPG experiments were run on one core of an Intel Xeon processor running at 3.3 GHz.

In the first experiment we activated only the functional constraints 1 to 4 (Section V-B). Another experiment applied all previously listed constraints. The runtime needed to generate the test sequences was 89 hours using the set of functional constraints and 295 hours for the complete set of SBST constraints.

While the runtime is relevant, the reader should note that the complexity of our approach is substantially affected by the enforced constraints. Hence, we believe that our approach could be easily adopted on any single pipeline processor / controller, thus covering a good percentage of the current safety-critical embedded systems.

Table I lists the detailed evaluation of all modules of the miniMIPS. The subdivision corresponds to the modules in the available VHDL code. The column *#faults* contains the number of faults for this module, columns *testable* and *untestable* state the number of detected and provably untestable faults. *abort* contains the faults which could not be detected and were not proven as being untestable. The values of *testable*, *untestable* and *abort* are based on the experiment using the functional constraints. The following columns give the fault efficiency of each experiment in percent.

In the last column we compare to the results of [7], which also reported detailed stuck-at fault coverages for the miniMIPS. As their approach is not able to identify untestable faults, their fault coverage is equivalent to the fault efficiency. For the instruction decode unit (*di*), the execution unit (*ex*) and the data forwarding unit (*renvoi*) a range is given, as the modules were further subdivided in [7]. For the branch prediction unit (*predict*) no results are given, as their evaluated miniMIPS version did not contain this module.

Table I
EXPERIMENTAL RESULTS

	faults	testable	untestable	abort	%FE [Functional Const]	%FE [SBST Const]	%FE [7]
pf	2,118	2,109	9	0	100.00	91.97	86.32
ei	1,478	1,470	7	1	99.93	96.82	90.86
di	7,090	6,738	238	114	98.39	92.45	83.53 – 90.24
ex	23,042	22,173	710	159	99.30	96.20	84.12 – 97.85
mem	2,658	2,014	396	248	90.67	71.29	81.87
renvoi	3,746	3,548	198	0	100.00	99.68	86.60 – 93.64
banc	41,600	41,536	64	0	100.00	100.00	99.98
syscop	6,696	5,148	1,548	0	100.00	98.04	87.90
bus_ctrl	1,988	1,877	65	46	97.69	92.20	93.95
predict	20,982	20,923	56	3	99.99	99.34	—
total	111,398	107,536	3,291	571	99.49	97.46	95.08

As it can be seen by the experimental results, our automatic SBST approach yields generally a very high fault efficiency and fault coverage which outperforms the previous results, which were manually generated. Clearly, the number of considered constraints significantly affects the complexity and effectiveness of the ATPG process. Depending on the number of constraints used, the fault efficiency of the generated test sequence ranges from 97.46% when applying all constraints to 99.49% using the reduced set of constraints. When all constraints are applied, our generated test program requires 10,139 clock cycles for execution. [7] has a total length of 7,162 clock cycles. Please note that the additional clock cycles of our approach mainly stem from the subsequences testing the branch prediction unit, which does not exist in the miniMIPS version considered in [7].

VII. Conclusions

We presented an efficient functional ATPG algorithm able to generate effective SBST test programs for in-field testing of stuck-at faults.

In particular, we demonstrated how our framework can be used to flexibly constrain the generated test programs in order to match the specific requirements of in-field SBST testing, which is becoming important for safety-critical applications. To the best of our knowledge, this is the first approach able to automatically generate high quality in-field SBST test programs without the need for manually generated (or adapted) test routines and in-depth knowledge about the evaluated processor.

As a conclusion, we believe that our results demonstrate that the proposed approach, although expensive from a computation point of view, represents a significant step forward, thus paving the way to its adoption with real-sized microcontrollers.

References

[1] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Transactions on Computers*, pp. 429–441, 1980.

[2] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design and Test of Computers*, pp. 4–19, 2010.

[3] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITSA microprocessor functional BIST method," in *IEEE International Test Conference*, pp. 590–598, 2002.

[4] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache resident functional microprocessor testing: Avoiding high speed IO issue," in *IEEE International Test Conference*, pp. 1–7, 2006.

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

[6] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line software-based self-test of the address calculation unit in RISC processors," in *17th IEEE European Test Symposium (ETS)*, pp. 1–6, 2012.

[7] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration Systems*, pp. 1441–1453, 2008.

[8] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *Design, Automation & Test in Europe*, pp. 1462–1467, 2013.

[9] *miniMIPS*. http://opencores.org/project,minimips.

[10] S. K. Devanathan and M. L. Bushnell, "Sequential spectral ATPG using the wavelet transform and compaction," in *IEEE International Conference on VLSI Design*, 2006.

[11] M. R. Prasad, M. S. Hsiao, and J. Jain, "Can SAT be used to improve sequential ATPG methods ?," in *IEEE International Conference on VLSI Design*, pp. 585–590, 2004.

[12] D. G. Saab, J. A. Abraham, and V. M. Vedula, "Formal verification using bounded model checking: SAT versus sequential ATPG engines," in *IEEE International Conference on VLSI Design*, pp. 243–248, 2003.

[13] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. Sonza Reorda, and B. Becker, "An effective approach to automatic functional processor test generation for small-delay faults," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1–6, 2014.

[14] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *IEEE International Test Conference (ITC)*, pp. 1–9, 2006.

[15] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 518–530, 2007.

[16] E. Sanchez, M. Sonza Reorda, G. Squillero, and M. Violante, "Automatic generation of test sets for SBST of microprocessor IP cores," in *18th Symposium on Integrated Circuits and Systems Design*, pp. 74–79, 2005.

[17] K. Christou, M. K. Michael, P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda, "A novel SBST generation technique for path-delay faults in microprocessors exploiting gate- and RT-level descriptions," in *IEEE VLSI Test Symposium (VTS)*, pp. 389–394, 2008.

[18] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1203–1215, 2006.

[19] M. Sauer, S. Kupferschmid, A. Czutro, I. Polian, S. Reddy, and B. Becker, "Functional test of small-delay faults using SAT and craig interpolation," in *IEEE International Test Conference (ITC)*, pp. 1–8, 2012.

[20] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, "Incremental preprocessing methods for use in BMC," *Formal Methods in System Design*, pp. 1–20, 2011.