On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors

P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso,
J. Lagos-Benites, E. Sanchez, M. Sonza Reorda
Dipartimento di Automatica e Informatica
Politecnico di Torino – Torino, Italy

O. Ballan
STMicroelectronics
Agrate Brianza – Milano, Italy
oscar.ballan@st.com

***Abstract*—Software-based Self-Test (SBST) can be used during the mission phase of microprocessor-based systems to periodically assess the hardware integrity. However, several constraints are imposed to this approach, due to the coexistence of test programs with the mission application. This paper proposes a method for the generation of SBST programs to test on-line the Address Calculation Unit of embedded RISC processors, which is one of the most heavily impacted by the on-line constraints. The proposed strategy achieves high stuck-at fault coverage on both a MIPS-like processor and an industrial 32-bit pipelined processor; these two case studies show the effectiveness of the technique and the low effort.**

Keywords-SoC, pipelined processors, on-line testing, SBST

I. INTRODUCTION

Many manufacturers all over the world (including those dealing with safety-critical applications such as space, vehicle and medical ones) are demanding smaller, higher performance, less expensive, and less power consuming electronic components. However, these constraints risk compromising the quality, since they make a challenge testing the device. For example, car manufacturers were recently forced to recall newly designed vehicles, because key electronic components responsible for critical parts, such as the air-bag and braking systems, were presenting failures, thus reducing product quality and putting people's lives at risk.

The use of electronic components responsible for safety- and mission-critical parts raises the necessity for high-dependability systems. Such requirement demands a number of auditing processes during the product's lifecycle, as well as some in-mission periodic testing for error detection. In fact, car manufacturers are adopting the ISO/DIS 26262 [1] standard which demands the adoption of the on-line self-test technique as an essential test process in critical electronic vehicle parts to insure high quality and mission safety throughout the product useful life.

Electronic components controlling critical parts include microprocessors as the key piece for the proper functioning of an application. The microprocessor usually monitors inputs coming from system sensor devices and re-acts accordingly, processing information and elaborating correct outputs for actuator devices. This is a common characteristic of real-time applications, which need to be constantly operating; therefore, the execution of an on-line test must be extremely fast and strictly non-invasive, in such a way that the application can still work correctly.

A viable solution that matches the above constraints is a carefully adapted *Software-Based Self-Test* (SBST) approach, forcing the processor to interrupt at regular intervals the mission software and run constrained test routines. Several works have been presented in the literature covering the topic of on-line SBST and showing why software-based techniques are often preferable to hardware approaches. Very effective SBST generation approaches are described in [2][3][6][8].

In this paper we concentrate on the on-line SBST of a very sensitive microprocessor component, the address calculation unit. This part of a processor is used to calculate the addresses for load and store memory accesses, usually summing a base address to an offset. The address calculation module is intensively used along the mission time; therefore, its components are more likely to early degrade than others (e.g., ALU ones). A failure in this module can provoke sneaky device behaviors, such as accessing wrong data in a still legal but misplaced address range; the effect is a deviated but not disruptive application execution that may not be revealed by hardware approaches, such as watchdog timers and software-implemented redundancy.

In the on-line testing context, the generation of SBST programs targeting the address calculation module is really critical because of the constraints imposed by the coexistence of the test routines with the mission application. Ideally, the mission application should freely access data and code memory; therefore, the test routines are constrained to use a limited memory address range for writing signatures and reading known data.

In this work we propose a strategy for developing SBST programs valid for on-line testing of address calculation units in pipelined microprocessor. The proposed test generation method provides guidelines for the selection of the on-line compliant address range reserved for test purposes and includes a test program synthesis flow based on atomic blocks, which is very effective in terms of stuck-at fault coverage and takes into consideration the on-line constraints.

The paper is structured as follows: we first introduce background concepts (Section II). Then (Section III), we discuss the methodology for generating SBST patterns for the address calculation unit. We gathered some comparative data (Section IV) on two test cases corresponding to a couple of pipelined microprocessors: an industrial System-on-Chip by STMicroelectronics and a MIPS-like processor. Finally, we draw some conclusions.

978-1-4673-0697-3/12/$31.00 ©2012 IEEE

II. SBST FOR ON-LINE TESTING

The principle of software-based self-test (SBST) is to run functional test patterns, based on the processor instruction set, exploiting processor resources to test the processor itself and the components around it [2]. SBST is one of the strategies firmly included in the manufacturing flow of microprocessors. Industrial experiences, such as [10] and [11], have confirmed the suitability of the methodology. In [4] an interesting case targeting a multicore processor is presented. In that experience functional patterns are loaded in cache and applied to each of the 8 Processing Units belonging to a 4 GHz multicore server in order to perform partial-good device binning.

SBST is also an emerging alternative for identifying faults during normal operation of the product, by performing on-line testing [5]. Several reasons push this choice: SBST does not require external test equipment, it has very little intrusiveness into system design and it minimizes power consumption and hardware cost with respect to other on-line techniques based on circuit redundancy. It also allows at-speed testing, a feature required to deal with some defects prompted by deep submicron technology advent.

Additional aspects have to be taken into account when dealing with test program generation for on-line purposes. The test program must first be able to properly excite the considered processor modules, and then, once the results have been produced, it must turn them observable from the outside in a transparent way that does not affect the normal operation of the mission application. The most important constraints for on-line testing include:

- *Preserving the processor status*: the status of the interrupted mission (i.e., the processor status register content) has to be saved and restored at the end of the test.
- *Execution time*: duration must be as short as possible complying with the requirements stated in the adopted safety standard [1].
- *Memory content*: it is crucial to prevent mission software and test programs from overriding information belonging to other processes. Code and data memory belonging to the test procedures must be clearly defined and limited considering the system memory map of the device.
  - *Code Memory footprint*: the code memory space required by the test programs should not be excessive, and it must conform to the established memory limits.
  - *Data Memory footprint*: the data memory space must also be as short as possible. However, the data memory placement can play a significant role with respect to the effectiveness of the generation process.

Different approaches have been proposed to test, by means of SBST techniques, processor modules such as the Address Calculation unit. However, in most cases, the previously mentioned constraints for on-line testing were not explicitly considered during the test program development. In some cases, for instance, the test application time is not suitable for on-line testing [3]; in others, the code size becomes also an unconsidered constraint [8]. Finally, in some approaches (for example [7]) the address calculation module is not explicitly targeted, and no special considerations have been taken into account regarding the placement of the data and code memory blocks devoted to allocate the on-line testing program. The resulting programs are, in any case, a good starting point and can be transformed into on-line test programs, as detailed in [6], where enforced constraints are considered.

In this work we detail the constraints and a generation method to test the Address Calculation unit during the mission phase of the device.

III. PROPOSED SBST GENERATION APPROACH FOR ON-LINE TESTING OF ADDRESS CALCULATION UNIT

The aim of this paper is to introduce an effective strategy to generate SBST programs, or routines, suitable to be run periodically during the device mission. The illustrated strategy falls into the non-concurrent on-line testing domain because the mission application is interrupted at regular intervals to let the self-test routines run.

The processor component specifically targeted by the proposed approach is the address calculation module; this module is in charge of calculating the addresses for memory access operations performed when *load* and *store* instructions are executed. This computation is performed by a dedicated adder, whose purpose is to sum an offset to a base value to address the RAM memory for reading or writing a value.

Usually, this adder is not part of the processor ALU, and thus it does not perform any arithmetic computations required by instructions like *add* and *sub*. Testing an adder is often deemed as a trivial task, but in the case of the address generation module, controllability and observability are limited, and on-line requirements pose additional constraints.

The criticalities in testing this module by using a software-based approach are mainly due to the type of instructions (load and store and all their variants) that activate the address calculation. In fact, a test program including many of such instructions may potentially induce some undesirable effects:

- Store instructions may corrupt the mission data and compromise the correct resuming of the system.
- Load instructions may retrieve data from memory zones (i.e., the parts containing the mission application variables) whose content can hardly be forecasted a priori, therefore compromising the signature generation, no matter how it is calculated.

Taking these factors into consideration during on-line test generation is a must; careful planning of a SBST generation campaign should early consider memory access constraints.

Figure 1 shows the conceptual view of the generation approach we propose. The main block in the scheme is called *Generation flow*. Other than usual inputs, such as the netlist of the circuit and the fault list of the tackled module, the Generation flow considers a set of constraints imposed by the coexistence of the test routines and the mission application, and leverages on a SBST atomic program template whose structure is tailored to take the on-line constraints into account.

The Generation flow is composed of a set of steps that are detailed in the next paragraphs; along the generation process, the fault coverage is a feedback measure that allows the program generation to proceed towards a set of routines guaranteeing high fault coverage. The final result of the generation process is a test suite mainly performing a sequence of load and store instructions, along with some

arithmetic instructions devised to setup base and offset values for memory access.

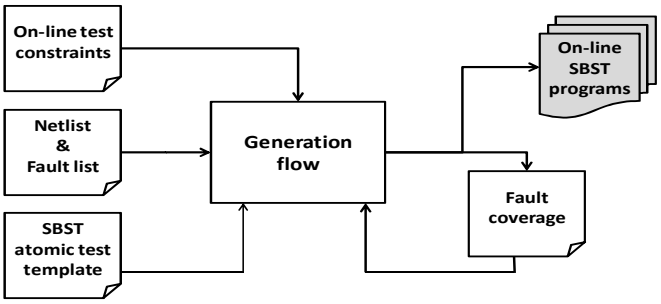

Figure 1. Conceptual view of the proposed generation approach.

A. On-line test constraints

Section II outlined the general constraints that a self-test procedure has to respect. In the specific case of the address calculation module, limitations are even stricter:

- Store instructions have to write only in reserved memory locations, possibly contiguous in the addressable space
- Load instructions have to read from memory sections never varying in content, possibly reserved for this purpose
- Memory zones reserved for writing and reading should be separated, to avoid the aforementioned problems
- Both read and write zones need to be programmed with suitable initial values
  - Arbitrary values are suitable for reading zones, provided that they are diverse within the zone; in this way the method minimizes the aliasing potentially stemming from a compromised memory access (e.g., when the reading zone is all 0s, accessing a wrong address in the range may cause non-detection)
  - The all 0s value is fine for the writing zone, since values transferred to the memory are sourced from the reading zone, which is properly filled.

The setup of the initial memory content is a critical operation, since it may be affected by faults in the same logic for address calculation to be tested. To avoid undesired fault effects, direct memory access (DMA) driven initialization is suggested; for example, the reading zone may be filled by copying part of the test routine code which is invariant, while this is not guaranteed for the mission program code.

The first issue arising from these limitations concerns the selection of memory zones. This selection directly reflects in the effectiveness of the generation process. Ideally, reserving a single small memory portion is desirable; however, a too small address range may prevent the achievement of a high coverage. For making an effective selection, it is useful to define first the acceptable amount of memory `M` that can be reserved for testing purposes (called *Test Memory*). This portion of memory can be eventually shared with test programs to store signatures; therefore, we suggest that the test of the address calculation module should be the first module considered in a SBST suite development plan.

To select the range `R` defining the starting and ending addresses of the Test Memory given the dimension `M` of the available memory, the following simple formula can be used, where `N` is equal to the number of addressing bits; the formula assumes that the Test Memory is located in the middle of the address space of the whole memory:

$$R = [2^{N-1}-M/2,\ 2^{N-1}+M/2-1] \quad (1)$$

The effectiveness of this choice is due to the fact that in this way the Hamming distance between the extreme address values is maximum, i.e., it is equal to `N`, allowing the address calculation adder outputs to exhibit a large value variance.

For instance, let us suppose that the addressing space is 32 bits, but only with 128KB available within the address range R_{data} = [0x40000000, 0x40001FFFF] corresponding to an addressable subspace of 17 bits. Supposing to have 1KB to be allocated to test purposes, the selected address range is $R = [2^{16}-512,\ 2^{16}+512-1]$ relative to the R_{data} segment start address. Hexadecimal values better show the effect of this choice considering the Hamming distance in the range R = [0x4000FE00, 0x400101FF]; all 17 less significant bits are changing values. Figure 2 visualizes the effect of such address range over the address calculation adder. As a counter-example, let us locate the 1KB test data on top of the memory addressable space, in the range R = [0x40000000, 0x400003FF]; in this case, only 10 bits in the address range can be varied.

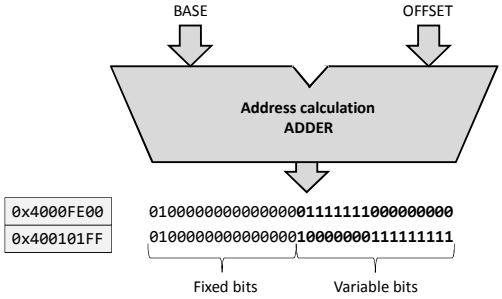

Figure 2. Effect of address range selection for address calculation adder test.

The identified memory space can be used either for writing or reading by defining 2 sub-zones.

As it can be noticed, the test effectiveness of the address calculation adder is strongly limited by the available address range, since several adder output bits may be fixed to a constant value. To overcome this problem, a viable solution can be based (whenever possible) on accessing other chip resources located out of the Test Memory (i.e., `0xBFFF000` may be a suitable complement to the previously selected address range). As an example, in most real cases the addresses reserved for peripheral core registers can be easily and safely accessed. Moreover, the selected complementary locations may be used for reading only, thus guaranteeing that their content is never changed.

The test program structure described in the next paragraph is suitably studied to cope with the identified requirements.

B. Atomic block structure

Roughly speaking, test programs targeting processing modules need to apply suitable values to the module inputs by means of controlling instructions; some instructions are used to setup the data to be elaborated by the tested module when a target instruction is executed, and then results are propagated to the processor outputs.

Testing the Address Calculation unit by using the SBST principles means reading and writing data from the memory at suitable locations. Let us consider the following generic memory access instructions (store and load):

```
Sd Rx, base (offset)
Ld Rx, base (offset)
```

When using such instructions, the actual address is calculated by adding the values provided as *offset*, and *base*, which usually correspond to two registers or to a register and an immediate value encoded in the instruction, respectively. The execution of this instruction excites the Address Calculation module by applying the *base* and *offset* values at the module inputs.

For testing sakes, base and offset have to hold suitable values to achieve fault coverage; in the on-line scenario, additional constraint must be considered and the resulting addresses must belong to a small memory range corresponding to the Test Memory.

To overcome the issue of performing effective sum operations while matching these strong constraints, we propose the usage of an *atomic block* devised to support the test generation process. The atomic block illustrated in the following can be used as a building element for the test program targeting the address calculation unit. The usage of similar blocks, called *building blocks*, was introduced in [12]; however, the authors did not consider the particularities of the address calculation modules of pipelined processors, neither the constraints regarding on-line testing.

In short, the proposed structure first loads a data value from a test memory location in the read reserved space; then this value is modified, and finally, it is saved again in a writable test memory zone. The pseudo-code of the atomic block is shown in Figure 3.

The first two instructions in Figure 3 (lines 1 and 2) load random values in the registers *rA* and *rB*. The value in *rA* is a constrained random address value (*rd constrained*) selected within the readable Test Memory addressing range: a known value must be stored previously in memory at this address. On the other hand, the value placed in *rB* is purely random without any constraint. The addition and subtraction instructions of lines 3 and 4 prepare registers *rC* and *rD*, which activate the arithmetic adder; in addition, these instructions manipulates the registers involved in the load instruction placed at line 5, that accesses the memory location at the address in *rA*, during which the address calculation module performs the addition between *rB* and *rC*. The memory value is read in register *rE*. The instruction at line 6 manipulates *rE* by applying the function *f(rE, rD)*, which is aimed at merging the results of the memory read (line 5) and the arithmetic addition (line 4). During our experiments, the function *f(rE, rD)* was replaced by the logic XOR instruction on the registers *rE* and *rD*. The advantage introduced by this function is that both address calculation and arithmetic adder are tackled obtaining high levels of fault coverage without additional effort. The value in *rE* is later stored in memory completing the observability task of the atomic block for the test of both adders.

In the second part of the pseudo-code, (lines 7-9) new random values are loaded in *rA* and *rB*; both of them are constrained values (*wr constrained*), since they are used to calculate the destination address to store *rE* in line 9. Clearly, the addition of *rA* and *rB* must produce a value placed in the writable Test Memory. Finally, it can be noted that line 9 collects the information elaborated by the atomic block and uses a store instruction to send it to the appropriate location.

```
1.    rA ← RNDM (rd constrained) value
2.    rB ← RNDM (unconstrained) value
3.    rC ← rA – rB
4.    rD ← rB + rC
5.    ld_inst rE, M[rB,rC]
6.    rE ← f(rE, rD)
7.    rA ← RNDM (wr constrained) value
8.    rB ← RNDM (wr constrained) value
9.    sd_instr rE, M[rA, rB]
```

Figure 3. Atomic block pseudo-code.

The structure proposed for the atomic block is as general as possible, so as to cope with different processor realizations. In the example above we supposed that the two elements involved in the memory access instructions correspond to a couple of registers. Clearly, depending on the targeted case, the adopted solution may include a couple of registers or an immediate value and a register.

C. *Test program building flow*

A suitable test program can be built by exploiting the atomic block introduced in the previous section. Its final form is a sequence of atomic blocks including selected values for exciting and propagating the address calculation adder faults. Possibly, no loops should be included in the test program code since those constructs usually lead to a long execution time, even if they permit saving program code lines.

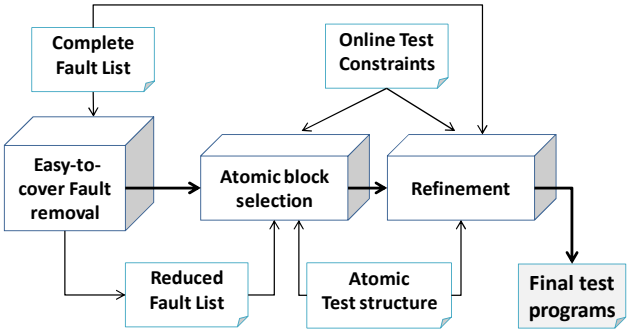

Figure 4. Proposed framework for on-line testing.

To perform accurate and fast selection of the required values, we propose a three-step generation process, whose graphical view is reported in figure 4. The result of this flow is a program composed of a sequence of carefully selected atomic blocks to be sequentially executed. The three steps differ due to the set of faults they work on, and on the test generation method they adopt.

The proposed flow derives from this concept: in any circuit there are faults that are easy-to-cover, i.e., they are detected by a large set of patterns, while there are other faults that require very specific test sequences. Therefore, we propose:

1. to initially remove from the fault list a possibly large set of easy-to-cover faults, even using unconstrained test programs;
2. to produce focused atomic block values considering the remaining faults, in this phase the on-line constraints are taken into account; more in details
   a. when the required level of coverage is reached, the obtained test program is graded with respect to the whole fault list (including easy-to-cover faults previously removed)
   b. most of easy-to-cover faults will result covered also in this case; therefore
   c. the coverage figure will only slightly decrease;
3. to integrate some more atomic blocks with proper values to refine the test program.

The usual fault coverage trend observed along the generation process is shown in figure 5.

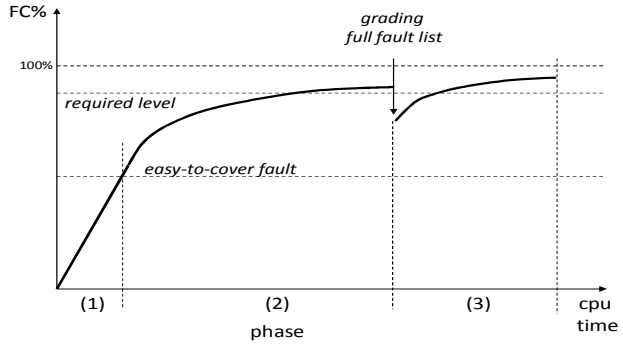

Figure 5. Fault coverage general trend along the generation process.

1) Easy-to-cover Faults Removal

In this phase a preliminary test program is written and evaluated without considering any on-line test constraint. This program may be composed of few manually selected instructions exciting the address calculation unit; particular cases, such as 0 values for base and offset, and random values may be included. As a result, many faults will be covered, which are said to be easy-to-cover. These faults are usually related to the logic parts of the address calculation adder located close to its inputs and outputs.

The coverage achieved by this process is usually quite high and up to 50% in some cases. The faults not covered are used as the input fault list for the next step.

2) Atomic block selection

Starting from the reduced fault list inherited from the first phase, the generation process strongly relies on the atomic test structure; such a generation process must comply with the constraints imposed by the on-line test environment. Therefore, in this phase the faults not belonging to the easy-to-cover category are considered, only; several test generation strategies can be adopted to identify the values to complete the atomic blocks and create a suitable program, as described in the next subsections. The goal is to reach the highest possible fault coverage, considering the cumulative effect of the test programs resulting from phases 1 and 2. As shown in the experimental results, the coverage on the complete fault list may reach up to more than 90%.

3) Refinement

At this point of the generation flow, the program obtained during phase 1 is removed. The coverage of the single program developed during phase 2 is evaluated; what is normally observed is a slight decrease in the fault coverage level, but this is usually not substantial, since most of the easy-to-cover faults are detected by both programs.

Starting from this new fault list, the refinement process is another generation step that tackles the few faults not yet detected by the on-line oriented program produced in step 2.

We underline that the major novelty of the proposed approach lies in the introduction of the atomic block and in the adoption of a 3-steps test generation method, while the techniques used in each step are not crucial.

IV. Case Studies

In order to verify the effectiveness of the proposed approach we performed our experiments on two 32-bit pipelined microprocessor case studies:

1. an automotive microcontroller by STMicroelectronics
2. a demonstrative case based on the miniMIPS core [13].

Following the flow described in Section III, initial constraints were defined in both the presented cases. Firstly, it was assumed that the size of the memory devoted to allocate the on-line testing programs for the address calculation unit is 4KB (2KB for code and 2KB for data). In addition, realistic spaces for code and data memories were also defined for the specific test purpose of the considered unit.

Special considerations have to be done regarding the atomic block selection methodology, since it is possible to adopt different strategies to identify the number of atomic blocks composing the on-line test program, as well as the most suitable values for the operands involved in every atomic block. In the presented cases we adopted two strategies briefly described in the following:

- *Evolutionary optimization tool:* it is possible to use an evolutionary optimization tool such as the one described in [9]. The evolutionary tool can include the atomic blocks as a building element to assemble the test programs. In this case, the evolutionary optimizer can fine-tune the constrained and unconstrained random values in order to maximize every atomic block testing capacities.
- *Random:* a random strategy can utilize the proposed atomic block as an assembling structure where to put the constrained and unconstrained values. A test program can be generated out of a predefined number of atomic blocks.

Special care needs to be taken for fault coverage assessment. In [14] the authors describe the method used in this work to evaluate the effectiveness of a SBST test set: this step is very critical, since it has to provide a fair coverage evaluation while taking into account the observability constraints in the on-line test application context, and fault simulation can be a very intensive computational task.

The first case study is a SoC including a 32-bit pipelined microprocessor based on the Power Architecture™. It is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers. This device contains over 2 million logic gates, including a 576 KB code Flash memory (2 KB of this memory are devoted to store the

address calculation test program) and a 128 KB data RAM (a contiguous 2 KB space is reserved within it for the Test Memory). It also includes other modules, such as an interrupt controller, different buses and I/O interfaces, and a debug controller. Address calculation is performed within the ALU, where two parallel adders are included in the same unit; using different ports, they are in charge of performing both arithmetic operations and address calculations. This module counts 689 gates and 4,188 stuck-at faults; an additional difficulty is that faults in the module affecting arithmetic or address calculations cannot be distinguished.

We applied the generation flow described in the previous section including 1) an easy-to-cover fault removal consisting in a loop-based SBST strategy [3] mainly devoted to cover arithmetic adder faults, 2) an evolutionary approach exploiting the μGP[3] tool [9] that only includes a macro implementing the atomic block described in section III.B, and 3) a refinement phase, again resorting to the same tool and considering corner cases and operating on the full fault list. The progression in the fault coverage values for stuck-at faults is shown in Table I. Interestingly, the final test program counts only 31 atomic blocks, each composed of 13 instructions.

TABLE I. STUCK-AT FAULT COVERAGE [%] OBTAINED ALONG THE FLOW

Case study	**Easy-to-cover fault removal**	**Test program generation**		**Refinement**
		Cumulative (phases 1+2)	*Phase 2 only*	
1	56.67	91.79	86.66	95.11
2	58.02 (59,51)	88.09 (90.72)	81.76 (82.85)	94.27 (96.38)

The second case study is based on *miniMIPS* [13], a processor core based on the MIPS I ISA. It features 32-bit buses for data and addresses, and includes a 5-stage pipeline. It was synthesized using Synopsys Design Compiler and an in-house developed library, resulting in 33,117 logic gates. In this case, the adder performing the address calculation is a clearly separated unit within the execution stage, counting 342 logic gates and 1,988 stuck-at faults; both address calculation and arithmetic adders count 757 gates and 4,408 stuck-at faults.

The same 3-step strategy was used in this case. However, we employed a random approach for the atomic block selection operation. The results are also shown in Table I, where fault coverage values are reported for both adders as well as for the address calculator adder alone (in parenthesis). For the miniMIPS case, the atomic block was reduced to only 6 instructions, and the final program counts 65 atomic blocks.

The proposed methodology is exploited resorting to two different generation strategies, an EA- and a random-based one. Remarkably, in both cases good coverage results were obtained, and the final test programs take about 1.6 KB, and require about 800 clock cycles to execute.

In order to corroborate the importance of the selection of the test memory placement, we performed a new experiment using a variable number of atomic blocks, and different code memory allocation constraints: the entire unconstrained processor addressing space, the 2KB space starting at address 0x00000000, and a configuration especially selected for on-line test, i.e., the address range 0x20007C00-0x200083FF. Table II reports the obtained coverage results.

It can be observed that a careful selection of the memory space attains better coverage results, and that a small (2 KB) Test Memory allows achieving a fault coverage comparable to that achievable having the whole memory accessible (which is hardly the case for on-line test).

TABLE II. STUCK-AT FAULT COVERAGE FOR DIFFERENT ATOMIC BLOCK/MEMORY CONFIGURATIONS FOR CASE STUDY 2 (BEFORE REFINEMENT)

Atomic blocks [#]	**Memory allocation constraints**		
	entire memory	***2K beginning***	***2K selected for on-line***
8	80.58 %	69.57 %	77.11 %
16	82.24 %	72.03 %	80.68 %
24	82.95 %	72.59 %	81.54 %
32	83.15 %	73.26 %	82.85 %

V. CONCLUSIONS

On-line test application poses a number of constraints to the SBST approach for microprocessor-based systems. The most critical aspects related to the problem were reviewed, and a new methodology for the generation of test programs for address calculation circuitry was proposed. The method is mainly based on the adoption of an atomic block, which can be replicated several times in the test programs; the choice of the parameters for each atomic block can be performed using different techniques. Experimental results obtained on both an industrial and a representative case study demonstrates the efficacy of the approach under on-line constraints.

REFERENCES

[1] ISO/DIS26262 "Road vehicles – functional safety", 2009 (proposed std.).

[2] M. Psarakis, et al., "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, n. 3, pp.4 – 19, May-June 2010

[3] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 24, n. 1, Jan. 2005, pp. 88 – 99

[4] J. Crafts, et al., "Testing the IBM Power 7 TM 4 GHz Eight Core Microprocessor," IEEE International Test Conference, 2010, pp.1-10

[5] H. Al-Asaad, et al., "Online BIST for embedded systems," IEEE Design & Test of Computers, vol. 15, issue 4, pp. 17–24, October 1988

[6] A. Merentitis, et al., "Directed Random SBST Generation for On-Line Testing of Pipelined Processors", IEEE On-Line Testing Symposium, 2008, pp. 273–279

[7] E. Sanchez, et al., "On the transformation of manufacturing test sets into on-line test sets for microprocessors", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2005, pp. 494- 502

[8] J. Shen, J. and A. Abraham, "Synthesis of Native Mode Self-Test Programs", in Journal of Electronic Testing: Theory and Applications, vol. 13, n. 2, October 1998, pp. 137-148

[9] E. Sanchez, et al., "Evolutionary Optimization: the μGP toolkit", Springer, 2011

[10] P. K. Parvathala, et al., "Functional random instruction testing (FRITS) method for complex devices such as microprocessors", *United States Patent 6948096*, 2005

[11] I. Bayraktaroglu, et al., "Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues", IEEE International Test Conference, 2006, paper 27.2

[12] F. Corno, et al., "Fully automatic test program generation for microprocessor cores", Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 1006- 1011

[13] miniMIPS processor, available at http://opencores.org/project,minimips

[14] P. Bernardi, et al., "Fault grading of Software-Based Self-Test procedures for dependable automotive applications", IEEE Design, Automation and Test in Europe Conference and Exhibition, 2011