On the Optimization of SBST Test Program Compaction

R. Cantoro, E. Sanchez, M. Sonza Reorda, G. Squillero, E. Valea
Politecnico di Torino, Dip. Automatica e Informatica
Torino, Italy
{riccardo.cantoro, ernesto.sanchez, matteo.sonzareorda, giovanni.squillero, emanuele.valea}@polito.it

***Abstract*[1]—Due to the increasing adoption of SBST solutions for both the end-of-manufacturing and the in-field test of SoC devices, the need for effective techniques able to reduce the duration of existing test programs became more pressing. Previous works demonstrated that this task is highly computational intensive and it is beneficial to partition it, e.g., by addressing the test program for one hardware module at a time. However, existing compaction techniques may become completely ineffective when dealing with faults which relate to memory addresses. This paper clarifies this issue and proposes possible solutions. Their effectiveness is experimentally demonstrated on a OR1200 pipelined processor.**

Keywords—test program; SBST; execution time; NOP instructions;

I. INTRODUCTION

End-of-manufacturing test of System-on-Chip (SoC) devices requires a continuous effort to face all the defects possibly affecting the new semiconductor technologies. Moreover, the adoption of electronic systems in safety-critical applications mandates for qualifying in-field test, whose aim is mainly to identify errors provoked by aging phenomena. In this scenario, a major role may be played by functional approaches, which exercise the target circuitry exactly in the same configuration as during the operational phase, and allow running the test at the operational speed. Since SoC devices always include at least one CPU core, one possible solution is forcing the CPU core to execute a test program, and then checking the produced results. This approach, denoted as Software-based Self-test (SBST) [2] was originally introduced to test processor devices [1], but then experienced a good success also in the area of SoC testing, and was extended to memories [6] and peripheral components [7], too. Since it belongs to the general category of self-test solutions, this approach is also suitable to be used for the in-field test of SoC devices used in safety-critical applications.

The quality of an SBST test program can be evaluated by the achieved coverage with respect to the target faults, by the memory occupation, and by the test duration. While the first parameter expresses the quality of the test, the latter two directly measure its cost. Moreover, when applied in the field, SBST is often run during the times left idle by the application, and it is therefore crucial to minimize the duration. While in some cases it is possible to produce SBST test programs having by construction the maximum fault coverage and minimum duration (e.g., when using formal techniques, such as in [3]), in most scenarios they are built incrementally, often starting from design-validation stimuli, possibly incremented with some carefully written pieces of code, each one targeting a different module in the CPU core.

Hence, a new wave of research activities started recently, aimed at developing automatic techniques able to compact existing test programs, while keeping unchanged the fault coverage they can achieve. A preliminary work targeted an 8051 processor core, by partitioning an existing test program in small independent fragments (called *spores*), and then identifying the minimum subset of them able to still guarantee the same fault coverage [4]. Being based on extensive fault simulation, the approach can only work when the fault simulation computational cost is affordable. Another technique assumed that the test program is already partitioned in different sub-programs [8]. Once again, clever techniques can be adopted to identify the minimum subset of these sub-programs. In a recent paper, Gaudesi et al. described a set of techniques, able to trade-off between test execution time minimization and required computational effort [9]. The basic idea behind all the proposed techniques is that compaction can be achieved by removing single instructions, if they proved to be useless in terms of fault coverage. However, when considering some modules interacting with the memory, such as the Fetch Unit, one can easily identify some faults (e.g., those related to the Program Counter) which can become untestable, if the size of the test program is modified. As the number of these length-dependent faults (LDFs) is not negligible, we propose some techniques able to successfully use the compaction algorithms proposed in [9], taking the LDFs into consideration.

The rest of the paper is organized as follows: Section II better highlights the characteristics of the LDFs, and then describes the proposed method. Section III gives details about the experimental environment. Section IV presents some

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

978-1-5386-0362-8/17/$31.00 ©2017 IEEE.

experimental results assessing its effectiveness. Section V finally draws some conclusions.

II. PROPOSED APPROACH

A. Background

We define a test program (TP) as a piece of code, composed of N instructions, that can be run on a target processor. The execution of the TP must start from a well-defined state of the processor, where all the internal memory elements of the CPU store a known value. The TP can achieve a fault coverage (FC) with respect to a given set of faults F. In our experiments, we consider single stuck-at faults, but the proposed solutions are still valid if extended to other fault models as well. A fault is marked as detected when the execution of the TP generates an observable difference with respect to the execution on a fault-free system. We define two test programs "test equivalent" with respect to a set of faults F, if both are able to cover the same subset of faults in F. Compaction algorithms aim at finding a new test program TP', test-equivalent to TP, that minimizes a specific cost function. In our experiments, we focused on the reduction of the execution time, hence reducing the number of instructions of the TP and shortening its code-length. A test program is "valid" if it can be safely executed, terminates correctly and it properly handles all exceptions. Compaction algorithms must reduce the number of instructions of a given TP while guaranteeing its validity.

A strong limitation of existing compaction algorithms arises when in the set F some "special" faults are present that we denoted as "Length-Dependent Faults" (LDFs). We define a fault *f* in F as LDF if its detection requires that the TP includes at least *n* instructions. When the set of faults F covered by a test program TP contains some LDFs, traditional compaction algorithms show very poor performance, or completely fail in reducing its code-length and consequently its execution time. The resulting TP' cannot be shorter than a specific length *n* that depends on the kind of LDFs present in F. The worst case is that n = N. In this case the compaction algorithm is not able to remove any instruction from TP. In common cases, the difference between N and *n* is small, thus the compaction capability is limited to a small number of instructions. After these instructions are removed, no further compaction is possible, resulting in very poor performance. The Program Counter is one of the processor structures where a high number of LDFs are present. Other structures, such as the adder used to compute the target address for jump instructions, may also produce LDFs. In general, testing all the faults associated to these structures would require a test program distributed over the whole memory addressable by the processor. In some cases, any reduction in the size of the test program may turn some LDFs into undetectable. Similar phenomena can be observed when dealing with all registers/modules in a processor related to memory access. Hence, the presence of LDFs has been observed in many modules of the processor.

The goal of this paper is to propose an algorithm able to effectively compact an existing test program even when the target fault list includes LDFs.

B. NOP injection method

The proposed solution stems from one of the compaction algorithms proposed in [9], called *A0* compaction algorithm. Under *A0*, the TP is fault simulated against F and the subset of detected faults φ is obtained. Then each instruction I_i is selected and removed from TP. The resulting program TP' = TP\{I_i} is first checked for validity. In the negative case I_i is restored inside the test program, otherwise TP' is fault simulated against φ. If the fault coverage of TP' on φ is less than 100%, I_i is restored inside the test program. This process is iterated until all the instructions of TP have been evaluated. In the proposed experiments, the instructions to be possibly removed have been selected starting from the bottom of TP since we experimentally verified that this choice is the most effective one. In Section IV, we show that *A0* is not effective when applied to a test program targeting a module with LDFs. Thus, instead of removing instructions, the proposed solution substitutes them with *NOP* instructions.

```
1   Fault simulate TP; let ϕ be the set of faults detected by TP
2   For every instruction Ii, selected from the bottom {
3       Let TP'= TP\{Ii}
        (i.e., let TP' be the test program obtained by removing Ii
        from TP)
4       If TP' is a valid test program AND TP' has a shorter or
        equal execution time than TP then
            Substitute Ii with a NOP instruction
            Fault simulate TP'
5           If all the faults in ϕ are detected by TP' then
                TP = TP'
    } // end for
```

Fig. 1. Pseudo-code for the *NOP* insertion method

The pseudo-code of the *NOP* insertion method is sketched in Fig. 1. At first, the test program TP is fault simulated to obtain the set of covered faults φ. Each instruction I_i is selected (starting from the bottom) and removed from TP. At each step, a logic simulation of TP' = TP\{I_i} is performed. If TP' is valid and its execution time is not increased, a *NOP* instruction is inserted in the position where I_i was before. This new version of TP' is fault simulated against φ; if the whole set φ is covered, TP' becomes the new TP, otherwise I_i is restored inside TP' and the modification is discarded. The insertion of *NOP* instructions allows to keep the code-length of the test program constant, therefore the coverage of LDFs is preserved.

The resulting program is called a NOP-injected test program.

C. Merging NOP-injected test programs

We propose a solution that allows to obtain much higher compaction figures, merging many *NOP*-injected test programs together. Let us have a set of test programs TP_i, each of these designed to cover a set φ_i of faults on a sub-module H_i of the processor. First, test programs are processed with the insertion of *NOP* instructions (see Section II.B), thus a set of *NOP*-injected test programs TP_i' is obtained. At this point a unique test program TP' can be built concatenating all the TP_i' one after the other. The goal is that TP' covers all faults that were covered by the original test programs TP_i. When test programs are

concatenated we need to insert a specific sequence of instructions between TP_i' and TP'_{i+1} to reset the state of the processor. For this reason, the number of instructions in TP' will be equal to the sum of instructions present in all TP_i', plus an overhead. The resulting test program TP' covers the set of faults $\varphi = (\varphi_0 \cup \varphi_1 \cup \varphi_2 \ldots)$ each lying in the correspondent sub-modules of the processor. Using this technique, it is possible to merge test programs designed separately for all the sub-modules of a processor to obtain a test program that tests the whole processor. At this point, TP' can be compacted using another compaction algorithm, since the code-length of TP' is much longer than any single test program TP_i. For this reason, special faults present in fault sets φ_i are not a bottleneck anymore for the compaction process. In our experiments, we performed the final compaction using the *A0* algorithm, selecting the instructions starting from the bottom. At the end of the whole process, a test equivalent test program TP" is obtained, with shorter execution time, that covers all the faults present in the sets of faults φ_i.
The reader should note that the alternative solution based on first concatenating all the test programs, and then directly compacting them is far less efficient, since the effectiveness of the adopted compaction algorithm exponentially decreases with the length of the targeted test program, while its computational cost increases.

III. CASE OF STUDY

To experimentally validate the proposed method, we focused on the test of four different modules present in a standard pipelined microprocessor. The target processor for our experiments was an OpenRISC 1200 [10]. It is based on a 5-stage pipelined architecture, with 32-bit registers and addresses.
We focused on a set of handcrafted test programs targeting four different modules of the processor: Control Unit (CU), Load and Store Unit (LSU), Operand Multiplexer (OPMUX) and Writeback Multiplexer (WBMUX).
Figure 2 shows how the chosen modules are related to the other main blocks of the CPU of the OpenRISC1200 microprocessor.

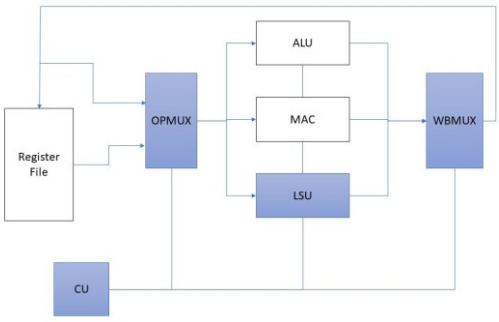

Fig. 2. High-level schematics of the OpenRISC 1200 CPU. The modules under test are highlighted (coloured blocks).

The compaction of the test programs addressing the described modules shows the presence of LDFs in three of them. Table I summarizes the total number of stuck-at faults that can be present inside the modules and it specifies how many of them correspond to LDFs. It can be observed that the CU module does not show the presence of any LDF. In real scenarios, SBST test programs designers do not always have knowledge of the internal structure of the processor under test. For this reason, the presence of LDFs is hardly predictable. Hence, it is important to propose compaction techniques that provide good performance in all cases.

TABLE I. CPU MODULES CHARACTERISTICS

	Total faults	**LDFs**
CU	4,346	0
LSU	1,976	112
OPMUX	2,828	4
WBMUX	1,826	95

IV. EXPERIMENTAL RESULTS

A. Experimental setup

To experimentally validate the proposed method, we implemented it in a Bash script that calls some custom commands, written in C, that perform the necessary operations on the test programs. The fault simulations, targeting single stuck-at faults, are performed with *Synopsys TetraMAX* version J-2014.09-SP2. The RTL model of the OpenRISC 1200 microprocessor has been synthesized using the *FreePDK45 Generic Open Cell Library*, a 45nm technology library from *NanGate* [5]. The experiments were run on a server machine based on an AMD Opteron processor at 2GHz with 64GB of RAM memory. The software environment runs on OpenSUSE 13.2 operating system.

TABLE II. ORIGINAL TEST PROGRAM CHARACTERISTICS

	Size [bytes]	**Execution time [cc]**	**FC [%]**
CU	524	359	72.17
LSU	15,220	15,472	72.15
OPMUX	284	149	76.64
WBMUX	11,708	8,532	70.04

Table II reports details about the test programs developed for these modules. The limited fault coverage that can be achieved by the considered test programs can be explained by the fact that a relevant number of stuck-at faults in the target modules are untestable in the current configuration of the environment in which the CPU is simulated [11].

B. Results

In the first stage of the experiment, the *NOP* injection procedure described in Section II.B has been applied to the test programs from Table II. Table III shows how many instructions from the original test programs have been replaced with *NOP* instructions. Moreover, the required CPU time (expressed in minutes) is reported; most of these computational times are spent for fault simulation, which is the most complex operation involved in this algorithm. The CPU time has been evaluated using traditional system functions; it is referred to the time of execution of the algorithm using a single core of the CPU.

TABLE III. RESULTS FOR THE *NOP* INJECTION PROCEDURE

	***NOP* insertion [%]**	**CPU time [min]**
CU	53.79	38
LSU	43.22	2,294
OPMUX	83.33	10
WBMUX	31.69	114

Compaction of the same test programs has also been performed using the traditional version of the *A0* algorithm, to show that this method does not work properly in the presence of LDFs. Table IV shows the result of the compaction performed with the traditional *A0* algorithm on the same test programs.

TABLE IV. COMPACTION RESULTS WITH *A0* ALGORITHM

	Execution time [cc]		**Compaction [%]**	**CPU time [min]**
	Original	***Compacted***		
CU	359	227	36.77	128
LSU	15,472	14,236	7.99	7,456
OPMUX	149	149	0.00	49
WBMUX	8,532	8,532	0.00	3,039

These results show that the traditional compaction technique gives low compaction ratios and, in two cases (lines OPMUX and WBMUX in Table IV), it does not compact at all, because of the presence of LDFs. In the second stage of the experiment, *NOP*-injected test programs (detailed in Table III) are concatenated one after another to create a unique test program (MERGED-TP). This has been compacted using the traditional version of the *A0* algorithm. Table V shows the result of this experiment.

TABLE V. COMPACTION ON MERGED *NOP*-INJECTED TEST PROGRAMS USING *A0* ALGORITHM

	Size [byte]	**Execution time [c.c.]**		**Compaction [%]**	**CPU time [min]**
		Original	***Compacted***		
MERGED-TP	28,216	14,858	7,420	50.06	14,014

The results show that this technique allows to reduce the number of instructions from the concatenation of *NOP*-injected test programs, reducing the execution time by one half. If we compare the final execution time with the sum of the initial ones (see Table II), we observe a nearly 70% compaction ratio.

The compaction of concatenated *NOP*-injected test programs is a much faster process than compacting directly the concatenation of the original test programs. This is due to the fact that more than half of the instructions of MERGED-TP are *NOP* instructions. This leads to far less complex logic simulations of the execution of the test program on the processor model. Since these logic simulations underlie the fault simulation procedure, all the fault simulations that are performed during the compaction are much faster.

We can thus state that this method allows to achieve a compression figure that is impossible to achieve when we try to compact single test procedures independently using traditional methods.

V. CONCLUSIONS

This paper deals with the automatic minimization of the execution time of SBST test programs. It builds over the techniques proposed in [9], whose effectiveness is impaired by the presence of a category of faults we called *Length-Dependent Faults* (LDFs), mainly related to the addresses used to fetch instructions and more in general to access the memory. As soon as the test program size is reduced, these faults may become untested, and thus prevent any further compaction. This paper faces this issue and proposes an elegant solution, based first of avoiding the elimination of any instruction, which is rather replaced with a NOP instruction. Then, the test programs targeting different modules are merged, and NOP instructions are cleverly removed, achieving the final goal of reducing the test program execution time, without affecting the attained fault coverage. Activities are currently being performed to further validate the approach on other processors.

ACKNOWLEDGEMENTS

This work has been partly supported by the European Commission through the Horizon 2020 Project No. 637616 (MaMMoTH-UP).

REFERENCES

[1] S. Thatte and J. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429–441, June 1980.

[2] M. Psarakis et al., “Microprocessor Software-Based Self-Testing”, IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19.

[3] A. Riefert et al., “An effective approach to automatic functional processor test generation for small-delay faults”, Proceedings of the Conf. on Design, Automation and Test in Europe (DATE), 2014.

[4] E. Sánchez, M. Schillaci, G. Squillero, “Enhanced Test Program Compaction Using Genetic Programming”, 2006 IEEE Congress on Evolutionary Computation, pp. 865-870, 2006

[5] "NanGate FreePDK45 Generic Open Cell Library", [Online]. Available at https://www.si2.org/openeda.si2.org/projects/nangatelib.

[6] A. J. van de Goor et al., “Memory testing with a RISC microcontroller”, IEEE/ACM Design, Automation and Test in Europe, 2010

[7] A. Apostolakis et al., “Test Program Generation for Communication Peripherals in Processor-Based Systems-on-Chip”, IEEE Design & Test of Computers, vol. 26 n. 2, pp. 52-63, 2009.

[8] A. Touati; A. Bosio; P. Girard; A. Virazel; P. Bernardi; M. Sonza Reorda, “An effective approach for functional test programs compaction”, IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2016

[9] M. Gaudesi, I. Pomeranz, M. Sonza Reorda, G. Squillero, “New Techniques to Reduce the Execution Time of Functional Test Programs”, IEEE Transactions on Computers, 2016

[10] OpenRISC 1200 IP Core Specification, http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2For1200%2Fdoc%2Fopenrisc1200_spec_0.7_jp.pdf

[11] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, O. Ballan, “On-line functionally untestable fault identification in embedded processor cores”, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013