New Techniques to Reduce the Execution Time of Functional Test Programs

New Techniques to Reduce the Execution Time of Functional Test Programs

M. Gaudesi, I. Pomeranz, *Fellow, IEEE*,
M. Sonza Reorda, *Fellow, IEEE*, G. Squillero, *Senior, IEEE*

Abstract— The compaction of test programs for processor-based systems is of utmost practical importance: Software-Based Self-Test (SBST) is nowadays increasingly adopted, especially for in-field test of safety-critical applications, and both the size and the execution time of the test are critical parameters. However, while compacting the size of binary test sequences has been thoroughly studied over the years, the reduction of the execution time of test programs is still a rather unexplored area of research. This paper describes a family of algorithms able to automatically enhance an existing test program, reducing the time required to run it and, as a side effect, its size. The proposed solutions are based on instruction removal and restoration, which is shown to be computationally more efficient than instruction removal alone. Experimental results demonstrate the compaction capabilities, and allow analyzing computational costs and effectiveness of the different algorithms.

Index Terms— software-based self-test, test compaction, test generation, test program

1 Introduction

IN functional test, only the functional input signals of the unit under test are stimulated, and only its functional output signals are observed. A common approach to functional test for processor-based systems is *Software-Based Self-Test* (SBST) [3], which consists of forcing the CPU to execute a test program and checking the results. SBST has some important properties:

- it allows testing the system at-speed, since the test program is executed at the same frequency of application programs;
- it does not require costly, high-speed testers, since low frequency interfaces can be used to upload the test program into a memory accessible by the processor and to retrieve the results at the end of the test;
- it implicitly tests both the modules composing a system (such as processor, bus, memories, peripherals) and their interconnections;
- finally, such tests can be easily tweaked to tackle different types of defects, to match new constraints (e.g., related to power [35]), and to provide diagnostic information [6].

Functional test based on SBST is widely adopted for the test of single devices such as Systems on a Chip (SoCs), boards, and complete systems [4]; its adoption spans from end-of-manufacturing to qualification test and in-field testing. In the last case, the role of functional test is particularly relevant in safety-critical applications, where standards and regulations specify the target fault coverage to be achieved, and both minimum cost and minimal invasiveness on the application environment are highly desirable [32]. The constraints mandated by the ISO26262 for automotive applications are a paradigmatic example.

Although SBST was originally intended to test processors and processor cores, some recent works show that it can be effectively adopted also to target memories [39], caches [38][41] and peripherals [40], as well as special kinds of processors, such as VLIWs [37]. Several efforts are currently being done to develop effective techniques for writing test programs addressing common modules in a CPU, such as Branch Prediction Units [42], or for extracting test programs from existing application code [43].

A major limitation of SBST lies in the cost for the development of suitable test programs. Although the first solutions for creating functional test programs were proposed more than three decades ago [1] [2], only recently research led to techniques that allow test engineers to reliably devise test programs achieving high fault coverage for processors and controllers [32].

Lately, automatic approaches for generating a test program have been proposed [5][31], at least for small- and medium-size microprocessors. However, most approaches still rely on manual effort and pseudo random generators [28], and in both cases test program size and duration are usually far from being optimal.

This paper tackles the reduction of the execution time of functional tests, an important activity as the duration of the test is a critical parameter in many different contexts. For instance, when a test program is run in the field, it often exploits the time slots left idle by the main application [33], and a long duration is likely to impair its applicability. Its duration also affects the required amount of power and energy, which may represent critical parameters in some cases, for example for nodes in Wireless Sensor Networks [36]. Simi-

- *M. Gaudesi is with Ominee, S.r.l.. E-mail: marco.gaudesi@ominee.com.*
- *I. Pomeranz is with Purdue University. E-mail: pomeranz@purdue.edu.*
- *M. Sonza Reorda and G. Squillero are with Politecnico di Torino. E-mail: matteo.sonzareorda@polito.it, giovanni.squillero@polito.it.*

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

larly, when a test program is part of the end-of-manufacturing test process, its duration directly influences the cost of the test process, and any reduction immediately turns into a money saving.

Reducing the size or the execution time of a test is called "compaction". Generally speaking, compaction techniques can be classified either as "dynamic" or "static". Dynamic techniques work directly during test generation, while the static ones act on an existing test set in a separate phase after generation. The two types of approaches are not necessarily exclusive, and some authors proposed to mix them [18].

Dynamic test compaction is known to be able to reduce both the size and the duration of a test program. Researchers in [35] and [44] proposed some techniques to consider test program size during test program generation. However, the dynamic approach may significantly increase the complexity of the test generation process, and, more importantly, it cannot be used when the test set is already available. The possibility to reuse, with possible minor modification, existing sets of test programs (e.g., coming from the validation process, or including application code) is a key advantage, and a very popular strategy in practice. All such test programs need to be optimized after their creation. Furthermore, it has been shown that even after applying dynamic compaction there is still room for further improvement by a static test compaction procedure [9].

One approach to static test compaction is to pinpoint and remove the parts of a test that are not strictly necessary for achieving the target fault coverage. Dealing with test program compaction, this means finding and omitting all the instructions that do not (directly or indirectly) contribute to the target fault coverage.

Any test program can be translated into a sequence of binary stimuli, and such a sequence can theoretically be compacted resorting to well-known methods [8]-[21]. However, the resulting binary sequence may not correspond to a valid sequence of instructions any more. Consequently, any test program compaction methodology must operate at the level of assembly program, facing therefore specific problems that are not present in binary test sequence compaction:

- Test programs are executed on systems that include memories; hence, compaction should take care of a more complex scenario when dealing with a test program.
- Test programs may include control-flow instructions implementing loops and branches. Thus, differently from binary test sequences, the duration is not necessarily proportional to the size of the test. Although removing an instruction is likely to decrease the time required to apply the test, a reduction in size does not necessarily turn into a reduction in test duration. For instance, the removal of key instructions, such as a conditional branch allowing to skip the execution of some instructions, can increase the overall execution time.
- The removal of an instruction may trigger special events (e.g., exceptions, infinite loops) which in some cases hang the whole system, or force it into hard-to-manage situations.

At present, there are few papers on test program compaction in the literature. In [7] a method was proposed, based on extracting from a test program a set of independent fragments, called *spores*. An evolutionary algorithm is then used to select the minimum sequence of spores whose execution allows achieving a given fault coverage. The method is shown to be effective on blocks like the arithmetic unit, but can only be applied under strict constraints and requires an ad-hoc simulator to perform the analysis. The method proposed in [34] deals with the special case in which multiple test programs are available, and selects the subset that minimizes the duration, while keeping the same global fault coverage.

This work is among the first proposing a generic and fully automatic procedure for removing instructions from an existing test program to reduce its duration without reducing its fault coverage. The scenario we consider is very common in practice: we do not assume the knowledge of any specific information about the test program itself, and we simply aim at compacting it while preserving the initial fault coverage with respect to a given fault model.

We consider different solutions. As a baseline, we remove one instruction at a time from the test program, checking via fault simulation whether the resulting fault coverage remains the same. This approach, although sometimes very effective, requires high computational effort. We also propose more complex solutions based on instruction restoration: a group of instructions is initially removed from the test program; omitted instructions are then restored one by one until the initial fault coverage is achieved. In this way, we can reduce the required computational effort, while still achieving significant compaction.

Experimental results are reported using a MIPS-like processor as a test case, and considering test programs addressing faults in specific modules within the processor itself. The proposed approach is applicable when fault simulation is computationally feasible, and this is typically the case in embedded system applications resorting to small- and medium-sized microcontrollers and CPU cores. In the paper we also show that our technique can effectively work on one processor module at a time, thus reducing the required computational effort even more. Although our experiments targeted single stuck-at faults, the proposed approach is independent of the adopted fault model, if a fault simulation tool is available.

The rest of the paper is organized as follows: Section II reviews earlier works on test compaction. Section III describes the proposed method. Section IV presents experimental results. Section V draws some conclusions.

2 Background

Extensive research has been performed on the compaction of binary test sequences. The first work to show that it is possible to omit a test vector from a given sequence without losing fault coverage is reported in [9], showing the usefulness of omitting test vectors even when test generation exploited dynamic test compaction. It also demonstrates that test generation procedures cannot avoid the inclusion of unnecessary test vectors in the sequence. Various implementations of test vector omission were described in [9]-[17].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2643663, IEEE Transactions on Computers

In general, if the test vector at clock cycle u is omitted from a test sequence T, every fault that is detected by T at clock cycle u or higher may lose its detection. The procedure described in [9] simulates these faults in order to determine whether or not the test vector at clock cycle u can be omitted. To reduce the computational effort, if the test vector at clock cycle u can be omitted, the procedure applies binary search to compute the longest subsequence that can be omitted starting at clock cycle u.

The restoration-based procedure described in [11] is a more efficient variation of test-vector omission. This procedure initially removes a subset of test vectors from the sequence, then selectively restores them into the sequence to restore the original fault coverage. Test vectors are restored by considering one fault at a time. This contributes to the efficiency of the procedure. In the procedure described in [14], parallel pattern simulation is used to accomplish the restoration process with a computational effort that is equivalent to that of fault simulation of the sequence.

The procedure described in [15] modifies the test vectors in a sequence. The procedure described in [16] allows omitted test vectors to be reintroduced into the sequence. The goal of both approaches is to achieve test compaction beyond that achievable with the basic test vector omission or restoration procedures.

A higher level, assertion-based dynamic test compaction procedure was described in [19]. The compaction of a set of binary test sequences was considered in [8], [20], [21].

The first results of an approach similar to the one presented here were reported in [27]. However, in [27] we targeted the simpler problem of reducing the size of the test program, while here the focus is on the execution time. The two goals are clearly related, since a reduction in the test program size often translates into a reduction in its execution time. However, as seen before, this relationship is not always given, and there are cases in which a reduction in the size produces an increase in the execution time.

Reducing the execution time is often more important in real applications than reducing the test program size, since it directly affects the test cost and feasibility (for in-field test). Unfortunately, the addressed problem is also much harder to solve. Moreover, this paper considers test programs addressing a whole processor, while in [27] we focused on test programs aimed at testing faults belonging to two modules, only.

3 Proposed method

This work describes two approaches (denoted as *A0* and *A1*) to compact SBST test programs. To evaluate the two approaches, we consider both their computational requirements and their compaction performance. Although the target of the test programs considered in this paper is the set of faults within a processor, the approaches can be easily generalized to test programs targeting other modules (e.g., peripheral components) within a processor-based system.

We assume that the test program *TP* is already available. *TP* is a sequence of N instructions $TP = (I_0, I_1, \dots, I_{N-1})$ executed from a completely specified state, i.e., a state where all memory elements, such as RAM cells and flip-flops, have known values. *TP* achieves a fault coverage *FC* with respect to a given set of faults F. In the experiments reported in this paper we consider single stuck-at faults, but the proposed approaches can be applied to different fault models.

In the experiments, we mark a fault as detected when a difference with respect to the fault-free system is observed on any output signal. This assumption does not match the real scenario if other observation mechanisms are adopted, such as looking at the values produced in user registers, cache lines, or memory. Nevertheless, the adopted mechanism for marking a fault as detected is able to provide results which are meaningful in many scenarios. For some low-frequency small microcontrollers SBST is used resorting to an ATE monitoring the output signals, and in SoC design it is becoming popular to introduce a programmable MISR to observe the processor bus during SBST execution. Moreover, if the SBST test program is suitably written to store results in memory, the fault coverage figures that can be achieved observing the output signals or the final memory content are comparable.

Our goal is to find a new test program $TP' = (I'_0, I'_1, \dots, I'_{R-1})$ composed of a subset of the instructions of *TP*, in the same relative order, that achieves at least the same fault coverage when started from the same initial state, and minimizes the execution time. Being composed of a subset of the original instructions, as a side effect, the methodology will also reduce the size of the test program.

We call "valid" a test program that can be safely executed, terminates correctly, accesses only legitimate memory locations, properly handles all exceptions, and does not violate the constraints imposed by the current scenario. We assume that *TP* is a valid SBST program. Even if we neglect additional constraints that could possibly exist on the behavior of the processor during the execution of the test, the removal of instructions may yield to invalid test programs. The most common problems are new exceptions (such as division by zero), and infinite loops. As the effect of the removal of a single instruction cannot be foreseen with a static analysis, we identify these situations through simulation.

3.1 Compaction by random instruction removal: A0

The most straightforward solution to test program compaction is based on single instruction removal. The first algorithm we consider here, called *A0*, is a greedy local search, and follows the idea proposed in [9] for binary test sequences.

The *A0* algorithm is sketched in Fig. 1. In each step, one random instruction $I_i \in TP$ is selected and removed, and the resulting test program $TP' = TP \backslash \{I_i\}$ fault simulated. If TP' is no longer a valid test program, or the attained fault coverage FC' is less than the original one, or the execution time does not decrease, then I'_i is pushed back into TP' and marked so it will never be chosen again. Otherwise, I'_i is permanently removed from TP'. The process is then repeated until all instructions have been considered.

The number of steps required by *A0* to compact *TP* is O(N).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2643663, IEEE Transactions on Computers

In the generic i-th step ($i \in [0, N\text{-}1]$), the evaluation of the fault coverage involves the simulation of a test program composed of a sequence of N-i to N instructions. Fault simulation itself has at least a polynomial complexity in the size P of the unit under test, and represents the sole computationally demanding step of the algorithm. Hence, the complexity of *A0* is $O(N^2 \times P^\beta)$, where $O(P^\beta)$ represents the computational complexity of fault simulation.

```
1   Fault simulate TP; let F be the set of faults detected by TP
2   For every instruction I_i, selected in a random order {
3       Let TP'= TP\{I_i}
        (i.e., let TP' be the test program obtained by removing I_i from
        TP)
4       Fault simulate TP'
5       If TP' is a valid test program AND all the faults in F are de-
        tected AND TP' has a shorter execution time than TP then
            TP = TP'
    } // end for
```

Fig. 1. Pseudo-code for the *A0* algorithm

It is possible to identify cases, in which the removal of an instruction may increase the achieved fault coverage, for example because the instruction was blocking the propagation of the fault effects to the output signals. Obviously, the chances to increase the fault coverage by removing instructions are generally low, but strongly depend on how the original test program was generated. This phenomenon was already observed in [9].

In principle, *A0* can be iterated by running a new optimization step against the test program obtained. This approach requires a significant computational effort, but according to our experience rarely produces a better result than *A0*.

3.2 Restoration-based algorithms: A1xx

An advantage in terms of CPU effort can be achieved by a family of algorithms (denoted as *A1xx*), which are based on first removing a given block of instructions, and then restoring them one at a time until the original fault coverage is recovered.

A1xx algorithms were considered in [27]; however, the target of the optimization was the reduction in the test program size, without considering its duration. The underlying idea is similar to the *restoration-based* procedure from [11], where it was applied to binary sequence compaction. However, in [11], all or most of the test vectors of a binary sequence are initially omitted. In the case of a test program, such an action is likely to create invalid programs, requiring the restoration of a large number of instructions before being able to evaluate the fault coverage again. This issue, that does not exist with binary sequences, is avoided with test programs by removing small blocks of instructions and restoring instructions from every block immediately after it is removed in order to restore the fault coverage. Moreover, at every step the algorithm must check whether the current test program is still valid and still guarantees the same fault coverage and has the same or lower execution time.

The pseudo-code of the generic *A1xx* algorithm is shown in Fig. 2. In the proposed algorithm the original test program is split into m segments (step 2) composed of adjacent instructions. For every segment S_i (starting from the last), we first remove all the instructions it is composed of, thus typically reducing the fault coverage. Let Φ_i be the set of faults that are detected by the segments S_i to S_{m-1}, i.e., faults that may become undetected due to the removal. Then, we start restoring one instruction from S_i at a time until all the faults in Φ_i are detected again, and the execution time is not increased.

As *A0*, *A1xx* algorithms are based on a preliminary fault simulation (step 1). Then, the algorithm performs as many iterations as the number of segments. For every iteration, the algorithm triggers one fault simulation step (step 6) for each instruction restoration. Hence, in the worst case (no instruction removed) the number of fault simulation steps (and thus the required computational effort) for *A1xx* is the same as for *A0*. However, the advantage of *A1xx* algorithms over *A0* lies in the fact that the number of fault simulations may become lower than the number of instructions (like in *A0*), because iterations corresponding to instructions that do not need to be restored (because the fault coverage has already been restored with the same or lower execution time) are not performed. As a consequence, the computational advantage of *A1xx* algorithms with respect to *A0* is greater when the achieved compaction is higher.

Step 4 does not require any computational effort, since the computation of Φ_i (i.e., the set of faults to be considered at each iteration) can be performed once during step 1. Similarly, the cost for the test in step 7 to check whether the new test program is valid or not has a very limited cost. In fact, every fault simulation in step 6 requires a preliminary fault-free simulation, which can easily determine whether the considered program is valid or not.

```
1   Fault simulate TP; let F be the set of faults detected by TP. Mark
    each fault with the instruction that first detects it
2   Split TP into m segments S_0 … S_m-1
3   For every segment S_i, starting from the last one {
4       Let Φ_i be the set of faults which are first detected by the in-
        structions in the segments from S_i to S_m-1 (included)
5       Let TP' be the test program initially obtained by removing
        from TP all instructions belonging to S_i
6       Fault simulate TP'
7       If TP' is a valid test program AND it detects all the faults in Φ_i
        AND it has the same or lower execution time then
                set TP = TP'
                goto 3 (next iteration)
8       Select one instruction I from S_i
9       TP' = TP' ∪ I (i.e., restore I)
10      goto 6
    } // end for
```

Fig. 2. Pseudo-code for the generic *A1xx* algorithm

Segments are selected starting from the last one in the program. The advantage of this approach is that the fault simulation cost of the first steps is rather low, since they require the fault simulation of faults belonging to the selected segment and the following ones, only. In the following steps, the computational cost typically decreases due to the performed compaction. Thus, the algorithm delivers useful results with lower computational effort, enabling the user to trade off between cost and quality.

Several versions of *A1xx* are possible, depending on how the segments are defined (step 2), and how the instructions of the generic segment are selected for restoration (step 8).

Concerning the former point, here we only consider the algorithms for which the original test program T is partitioned into m segments, each composed of a fixed number n of consecutive instructions (apart from the last). Hence, the first segment S_0 will be composed of the first n instructions in T (I_0 to I_{n-1}), the second segment of the instructions I_n to I_{2n-1}, and so on. Clearly, the choice of the optimum value for n may be relevant. The *A1xx* variants with segments of variable size are not analyzed here, as they were shown to be unlikely to yield satisfactory results [27].

Concerning the order according to which instructions belonging to the current segment are restored, the following policies are considered:

- Forward (*A1Fn*): instructions are restored one by one starting from the first in the segment.
- Back (*A1Bn*): instructions are restored starting from the last in the segment.
- Random (*A1Rn*): instructions are restored following a random order.

These three variants, with n ranging from 2 to 10, have been thoroughly evaluated and the results are summarized in Section 4.2. Thanks to the analysis performed in [27], in this paper we do not analyze the effect of applying different restoration techniques in sequence.

As mentioned before, when removing the instructions belonging to a given segment S_i, the algorithm only needs to simulate faults originally detected by instructions in segments S_i to S_{m-1}. The identification of the instruction that first detects each fault is done as a byproduct of step 1.

4 Experimental Evaluation

4.1 Experimental setup

To experimentally validate the proposed algorithms, we implemented them in a prototypical tool written in Java (about 2,000 lines of code). To evaluate the fault coverage, *Synopsys TetraMAX* is used.

In the experiments we tackle a MIPS-like processor [22]: its architecture is based on 32-bit registers and addresses, and includes a five-stage pipeline, accounting for about 250k equivalent gates when synthesized using the *FreePDK45 Generic Open Cell Library* from *NanGate* [23]. The size and complexity of the considered processor is comparable with the one of several microcontrollers (e.g., Intel 8051 or ARM M0) which are popular for many safety-critical applications in areas like automotive, space and avionics.

4.2 Algorithm tuning

We first performed a preliminary analysis aimed at comparing various *A1xx* alternatives. In particular, we first compared the forward (*A1Fx*), backward (*A1Bx*), and random (*A1Rx*) restoration schemes with different sizes of the segments.

The results of this preliminary analysis, which are not reported here in detail due to space limitations, suggest that the choice of the parameters of *A1xx* is not so critical. Generally speaking, looking at the compaction, the *A1Fx* version appears able to deliver slightly higher compaction than *A1Bx* and *A1Rx*, and smaller block sizes are more effective. On the other hand, looking at the computational effort, smaller block sizes yield longer runs, as the resulting algorithms are more similar to *A0*.

The result can be intuitively explained as follows: each segment requires an initial fault simulation (step 6 in Fig. 2) when all its instructions have been removed; hence, the choice of smaller segments will involve a higher number of these fault simulation runs; at the same time, operating at finer granularity increases the possibilities to detect useless instructions. In fact, our algorithm is able to remove a single instruction from a segment only if this instruction is the last to be restored. The smaller the segment size is, the higher this probability is.

Based on the above results, for the following experiments we selected *A1F3*. To provide the reader with a better understanding of the behavior of *A1F3* with respect to *A0*, we plotted in Figure 3 a paradigmatic example of the achieved compaction (CR) with respect to the CPU time (COST) during a compaction run of a difficult module. Both indicators are relative: CR is the ratio between the execution time of the optimized test program and the original one; COST is the ratio between the computational effort required to optimize the test program and one full fault simulation of the original test program on the target module. In this way the reported results are less dependent on the effectiveness of the adopted fault simulator, which in its current version is not optimized for dealing with functional test programs. Finally, the interaction between our tool and the fault simulator is far from being optimized in this version of our environment, thus requiring an amount of CPU time which could be easily reduced in a more engineered version.

In Figure 3, the performance of *A0* (in red) is compared against the performance of *A1F3* (in blue) in terms of achieved compaction after every fault simulation: it is apparent that *A13F* is superior when the CPU budget is limited, while in the long period *A0* is able to achieve a better compaction result.

4.3 General results on single processor modules

We report results obtained with the proposed methodology on the different modules of the target microprocessor. In the experiments, test programs have been generated manually by a test engineer, focusing on each module independently, which corresponds to a quite common industrial practice [32].

By addressing separately the compaction of the test program for each single module, we reduce the computational cost compared to the situation where we compact the test program for the whole processor. We show that combining the compacted programs we can achieve nearly equivalent results with higher efficiency and scalability.

Table I lists the considered modules, together with the corresponding number of faults; the table also shows the size of the relevant test program (in instructions), its length (in clock cycles), and the stuck-at fault coverage (*FC*) it achieves.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2643663, IEEE Transactions on Computers

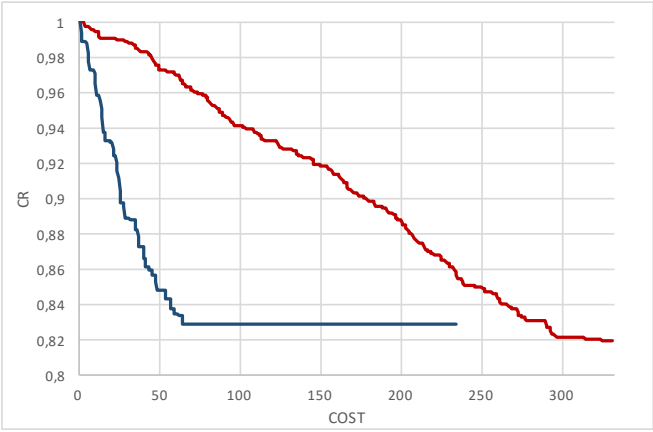

Fig. 3. Performance comparison between A0 (in red) and A1F3 (in blue) on the bypass unit.

We are not considering the compaction of the test programs for the register bank and the execution stage, as the test programs for these two modules are very specific and can hardly be optimized. In particular, the test of the former unit can be written in a minimal form by following the deterministic algorithm described in [29], while the test of the latter unit can effectively be performed resorting to the algorithm proposed in [30], which directly leads to an optimized test program.

The low coverage for the memory access stage is due to constraints imposed for the test: only a small portion of the memory was available during the test and therefore addressable.

Globally, test programs attain a fault coverage of 79.10% on the listed modules, and the coverage increases to 92.39% when the Execution Unit and the Register Bank are also considered, as these two modules contain a high number of easy-to-test faults.

We ran the tools implementing the *A0* and *A1F3* algorithms, computing the same figures used in the previous sub-section: compaction ratio (CR) and computational cost (COST).

Table II shows the results obtained by applying the considered algorithms to the test programs addressing each of the considered modules in the processor. For each module the statistics are reported about the attained test program execution time reduction (CR) and computational cost (COST), as defined above.

Results in Table II show that the considered algorithms can reduce the duration of each test program down to about 70% of its original duration. *A1F3* produces on the average a compacted test program that is about 5% longer than *A0*, but is 21% faster in getting the result. Given the significant computational effort required by test program compaction, this represents a remarkable result.

In our experiments, we worked on a set of test programs, each developed to maximize the stuck-at fault coverage in a different module. By construction, the compaction process does not decrease this fault coverage figure. However, it is possible that the test program for a module also detects faults in another module. By compacting each test program separately, it may happen that the global test coverage on the whole processor is slightly decreased. We experimentally evaluated this loss and found it is very limited, corresponding to fault coverage reduction from 92.39% to 92.28%.

During the optimization process, faults on lines connected with the address bus are not considered, because to detect them it is necessary to execute instructions located in specific memory locations and, consequently, a general approach for compaction would not be able to reduce the size of the test. Moreover, the procedure for detecting faults on lines connected with the address bus is strongly dependent on the circuitry and the compaction algorithm does not use any knowledge about the architecture of the modules. Indeed, the global fault coverage figure achieved by all test programs before and after compaction has been computed by considering these faults, too.

Based on the results of Table II we can also observe that the computational cost of the compaction algorithm (as given by the COST parameter) basically depends on the number of instructions composing the target test program: the COST parameter is maximum for the Bypass unit and the Address calculation stage, whose test programs are significantly longer than the others. Clearly, the computational time for compaction directly depends on the size of the target module and on the execution time of the corresponding test program (because they directly affect the time for fault simulating the original test program). However, the number of fault simulation steps required by the compaction algorithm (and reflected by the value of COST) mainly grows with the size of the original test program. Finally, further experimental results not reported here for lack of space confirm that the computational effort decreases when the achieved compaction is higher.

4.4 General results on the whole processor

In this sub-section we focus on the case, in which a single test program targeting the faults in the whole microprocessor is considered. Such a monolithic test program can be compacted by the proposed approach as well.

In Table III the row labeled “Monolithic” shows data for a test program tackled as a whole. This test program has been designed by an expert test engineer and its execution time is particularly reduced, being heavily based on loops. Its original fault coverage is about 92%.

On the other side, the row labeled “Module by module” reports the results on the whole microprocessor that have been obtained by optimizing one test program from the test set at a time, and eventually concatenating them, following the approach reported in the previous sub-section.

In both lines, columns 2 and 3 report the size and duration of the original test program. The following columns report the behavior of the two compaction algorithms we consider in this paper and show that they are able to effectively compact test programs both working module by module and in a monolithic way.

It should be noted that the “COST” column reports the relative costs compared to a full fault simulation of the original

test program, and not the absolute values. Hence, figures in different rows cannot be directly compared.

5 Conclusions

Test program compaction is crucial, especially when SBST is used for in-field test of safety-critical applications. In this scenario, small- and medium-sized microcontrollers are often adopted, possibly within a SoC. For such devices, fault simulation is feasible, although computationally expensive. Compaction can be tackled by resorting to different strategies, characterized by different figures in terms of achieved compaction and required computational effort. This paper showed approaches aiming at compacting existing test programs: the first one is based on instruction removal alone; the others are based on instruction removal and restoration, and require a significantly lower computational effort.

Experimental results show first that test programs can be effectively compacted using an algorithm exclusively based on the results of fault simulation. Secondly, we experimentally demonstrated that the removal-restoration approaches represent a reasonable trade-off between the achieved compaction and the required computational effort. Other solutions (e.g., [7]) may achieve higher compaction with a much higher computational effort, preventing their usage on a complete processor. Similarly, one could even imagine to adopt completely different solutions, e.g., based on modifying instructions in the test program, thus probably achieving an even higher compaction. However, this would increase even further the complexity of the compaction algorithm.

Test program compaction requires significant computational effort, and most of its cost is due to fault simulation. The test set is commonly composed of many different test programs targeting different functional units, developed at different times. We showed that such programs can be optimized independently, considering only faults in the unit they were intended to tackle. In this way we can tame the computational requirements and improve scalability. We also demonstrated that the proposed approach is effective even when the test is composed of a single monolithic program targeting the whole microprocessor.

The method can be easily extended to deal with other fault models, provided that a fault simulation tool is available, able to provide the fault coverage figure corresponding to each considered subset of instructions.

We are currently working on the development of further optimization techniques, which may increase the achieved compaction while limiting the computational cost.

References

[1] S. Thatte, J. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429–441, 1980
[2] D. Brahme, J. Abraham , “Functional testing of microprocessors”, IEEE Transactions on Computers, vol. C-33, no. 6, pp. 475 - 485, 1984
[3] M. Psarakis et al., “Microprocessor Software-Based Self-Testing”, IEEE Design & Test of Computers, vol. 27, no. 3, 2010, pp. 4-19
[4] A. Jutman et al., “High Quality System Level Test and Diagnosis”, IEEE Asian Test Symposium (ATS), 2014
[5] A. Riefert et al., “An effective approach to automatic functional processor test generation for small-delay faults”, Design, Automation and Test in Europe Conference (DATE), 2014
[6] P. Bernardi et al., “An Effective technique for the Automatic Generation of Diagnosis-oriented Programs for Processor Cores”, IEEE Trans. on Computer-Aided Design , vol. 27, pp. 570-574, 2008
[7] E. Sánchez et al., “Enhanced Test Program Compaction Using Genetic Programming”, IEEE Congress on Evolutionary Computation (CEC), pp. 865-870, 2006
[8] T. M. Niermann et al., “Test compaction for sequential circuits”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 260 - 267, 1992
[9] I. Pomeranz, S. M. Reddy, “On Static Compaction of Test Sequences for Synchronous Sequential Circuits”, Design Automation Conf. (DAC), 1996, pp. 215-220
[10] M. S. Hsiao et al., “Fast Algorithms for Static Compaction of Sequential Circuit Test Vectors”, VLSI Test Symp., 1997, pp. 188-195
[11] I. Pomeranz, S. M. Reddy, “Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits”, IEEE Int’l Conference on Computer Design (ICCD), 1997, pp. 360-365
[12] M. S. Hsiao, S. T. Chakradhar, “State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits”, Design Automation and Test in Europe (DATE), 1998, pp. 577-582
[13] S. K. Bommu et al., “Static Compaction using Overlapped Restoration and Segment Pruning”, IEEE International Conference on Computer-Aided Design (ICCAD), 1998, pp. 140-146
[14] X. Lin et al., “SIFAR: Static Test Compaction for Synchronous Sequential Circuits Based on Single Fault Restoration”, IEEE VLSI Test Symposium (VTS), 2000, pp. 205-212
[15] I. Pomeranz, S. M. Reddy, “Vector Replacement to Improve Static Test Compaction for Synchronous Sequential Circuits”, IEEE Trans. on Computer-Aided Design, Feb. 2001, pp. 336-342
[16] I. Pomeranz, S. M. Reddy, “Enumeration of Test Sequences in Increasing Chronological Order to Improve the Levels of Compaction Achieved by Vector Omission”, IEEE Trans. on Computers, July 2002, pp. 866-872
[17] I. Pomeranz, S. M. Reddy, “Vector Restoration Based Static Compaction using Random Initial Omission”, IEEE Trans. on Computer-Aided Design, Nov. 2004, pp. 1587-1592
[18] S.N. Neophytou, M.K. Michael, “Test Set Generation with a Large Number of Unspecified Bits Using Static and Dynamic Techniques”, IEEE Transactions on Computers (vol. 59, no. 3), 2010, pp. 301-316
[19] J.G. Tong et al., “Test compaction techniques for assertion-based test generation”, ACM Trans. on Design Automation of Electronic Systems (TODAES), December 2013, pp. 1-29
[20] I. Pomeranz, “Concatenation of Functional Test Subsequences for Improved Fault Coverage and Reduced Test Length”, IEEE Transactions on Computers, vol. 61, no. 6, 2012, pp. 899-904
[21] I. Pomeranz, “Two-Dimensional Static Test Compaction for Functional Test Sequences”, IEEE Transactions on Computers, vol. 64, no. 10, 2015, pp. 3009-3015
[22] “miniMIPS Overview,” opencores.org, [Online]. Available at http://opencores.org/project,minimips
[23] “NanGate FreePDK45 Generic Open Cell Library”, [Online]. Available at https://www.si2.org/openeda.si2.org/projects/nangatelib
[24] P. Bernardi et al., “On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors”, 14th Int.l Workshop on Microprocessor Test and Verification, 2013, pp. 52-57
[25] P. Bernardi et al., “On the in-Field Functional Testing of Decode Units in Pipelined RISC Processors”, IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2014, pp. 298-303
[26] E. Sanchez et al., Evolutionary Optimization: the µGP toolkit, Springer, 2011
[27] M. Gaudesi et al., “On Test Program Compaction”, IEEE European Test Symposium (ETS), 2015
[28] P. Parvathala et al., “FRITS - a microprocessor functional BIST method”, IEEE International Test Conference (ITC), 2002, pp. 590-598
[29] D. Sabena et al., “A new SBST algorithm for testing the register file of VLIW processors”, Design, Automation and Test in Europe Conference (DATE), 2012, pp. 412 – 41
[30] D. Gizopoulos et al., “An effective BIST scheme for arithmetic logic units”, IEEE International Test Conference (ITC), 1997, pp. 868 – 877
[31] S. Gurumurthy et al., “Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor”, IEEE International Test Conference (ITC), 2006
[32] P. Bernardi et al., “Development Flow for On-Line Core Self-Test of Automotive Microcontrollers”, IEEE Transactions on Computers, 2016, Vol. 65, Issue 3, pp. 744 – 754

[33] A. Merentitis et al., "Directed Random SBST Generation for On-Line Testing of Pipelined Processors", 14th IEEE International On-Line Testing Symposium, pp. 273 – 279, 2008

[34] A. Touati et al., "An effective approach for functional test programs compaction", IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2016

[35] Jun Zhou, H. -J. Wunderlich, "Software-Based Self-Test of Processors under Power Constraints", Design Automation & Test in Europe Conference (DATE), 2006

[36] A. Merentitis et al., "Low Energy Online Self-Test of Embedded Processors in Dependable WSN Nodes," IEEE Trans. on Dependable and Secure Computing, vol. 9, no. 1, pp. 86-100, Jan.-Feb. 2012

[37] D. Sabena et al., "On the Automatic Generation of Optimized Software-Based Self-Test Programs for VLIW Processors," IEEE Trans. on Very Large Scale Integration Systems, vol. 22, no. 4, pp. 813-823, April 2014

[38] G. Theodorou et al., "Software-Based Self-Test for Small Caches in Microprocessors," IEEE Trans. on Computer-Aided Design, vol. 33, no. 12, pp. 1991-2004, Dec. 2014

[39] A. J. van de Goor et al., "Memory testing with a RISC microcontroller", Design Automation and Test in Europe Conference (DATE), 2010

[40] A. Apostolakis et al., "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices", IEEE Design & Test of Computers, 2009, Vol. 26, Issue 2, pp. 52 – 63

[41] G. Theodorou et al., "Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2013, Vol. 21, Issue 4, pp. 786 – 790

[42] E. Sanchez, M. Sonza Reorda, "On the Functional Test of Branch Prediction Units", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2015, Vol. 23, Issue 9, pp. 1675 – 168

[43] S.V. Kodakara et al., "Extracting effective functional tests from commercial programs", 33rd IEEE VLSI Test Symposium (VTS), 2015

[44] N. Kranitis et al., "Low-cost software-based self-testing of RISC processor cores", Design, Automation and Test in Europe Conference (DATE), 2003, pp. 714 - 719

Giovanni Squillero received his M.S. and Ph.D. in computer science in 1996 and 2001, and he is an Associate Professor in Politecnico di Torino, Italy. His research mixes bio-inspired metaheuristics with electronic CAD and computational intelligence, machine learning, games, and multi-agent systems. Squillero is a member of the *IEEE Computational Intelligence Society Games Technical Committee*.

Marco Gaudesi received the B.S. and M.S. degree in Computer Engineering at Università Palermo in 2007 and Politecnico di Torino in 2010 respectively. He received his Ph.D. in 2015 from Politecnico di Torino. He is now a Data Scientist at Ominee S.r.l. in Torino.

Matteo Sonza Reorda received his M.S. degree in Electronics in 1986 and Ph.D. degree in Computer Engineering in 1990, respectively, both from Politecnico di Torino. He currently is a Full Professor at the Dept. of Control and Computer Engineering of the same University. He is a Fellow of IEEE. His research interests include test of SoCs and fault tolerant electronic system design.

Irith Pomeranz received the B.Sc degree (Summa cum Laude) in Computer Engineering and the D.Sc degree from the Department of Electrical Engineering at the Technion - Israel Institute of Technology in 1985 and 1989, respectively. From 1989 to 1990 she was a Lecturer in the Department of Computer Science at the Technion. From 1990 to 2000 she was a faculty member in the Department of Electrical and Computer Engineering at the University of Iowa. In 2000 she joined the School of Electrical and Computer Engineering at Purdue University. Her research interests include testing of VLSI circuits, design for testability, synthesis and design verification. She is a Fellow of IEEE and a Golden Core Member of the IEEE Computer Society.

TABLE I. MODULES AND RELATED TEST PROGRAMS

Module	Faults (#)	Size (instr)	Time (CC)	FC (%)
Address calculation stage	1,025	174	334	77.89
Instruction extraction stage	1,096	47	139	90.28
Instruction decoding stage	5,365	65	1,895	73.92
Memory access stage	1,398	50	836	53.06
Bypass unit	3,295	341	493	92.63
Coprocessor system	4,652	55	194	80.43
Bus controller	1,616	92	1,378	80.61
Total	18,447	824	5,266	79.10

TABLE II. COMPACTION RESULTS (MODULE BY MODULE)

	A0		A1F3	
	CR	COST	CR	COST
Address calculation stage	0.700	150.92	0.736	91.31
Instruction extraction stage	0.706	40.37	0.711	28.04
Instruction decoding stage	0.981	62.73	0.983	46.98
Memory access stage	0.822	43.64	0.871	36.09
Bypass unit	0.819	332.30	0.829	233.89
Coprocessor system	0.736	44.73	0.860	40.84
Bus controller	0.798	74.08	0.821	76.12

TABLE III. COMPACTION RESULTS (WHOLE PROCESSOR)

	Instructions	Clock Cycles	A0		A1F3	
			CR	COST	CR	COST
Module by module	824	5,266	0.859	748.77	0.881	553.27
Monolithic	373	16,738	0.893	306.25	0.896	288.84