# An Effective Approach for Functional Test Programs Compaction

A. Touati, A. Bosio, P. Girard, A. Virazel
LIRMM-UM/CNRS, France
<lastname>@lirmm.fr

P. Bernardi, M. Sonza Reorda
Politecnico di Torino, Italy
<paolo.bernardi, matteo.sonzareorda>@polito.it

***Abstract*—Functional test guarantees that the circuit is tested under normal conditions, thus avoiding any over- as well as under-test. This work is based on the use of Software-Based-Self-Test that allows a special application of functional test to the processor-based systems. This strategy applies the so-called functional test programs that are executed by the processor to guarantee a given fault coverage. The main goal of this paper is to investigate the static test compaction of a given set of functional test programs. The investigation aims at understanding and determining how to select the best functional test program candidates to obtain the smallest set having the best fault coverage. Results carried out on two different microprocessors show that a 49% reduction in test length and a 28.7% reduction in test application time can be achieved.**

**Keywords—Test Compaction, Microprocessor Test, Functional Test, SBST.**

## I. INTRODUCTION

In functional test, only the functional input signals of a circuit (or module) are stimulated and only functional output signals are observed. Therefore, functional test guarantees that the circuit is tested under normal conditions thus avoiding any over- as well as under-test [1].

For processor-based systems, the most efficient strategy for applying a functional test is called *Software-Based Self-Test* (SBST) [2]. This strategy is based on the execution of a set of functional test programs on the target processor. SBST solutions are particularly suited for on-line test of processor-based systems. For example, a dedicated functional test program (or a set of test programs) can be periodically executed during the mission time. In this context, it is clear that the test length (i.e., how many programs) as well as the test time is a very important issue for this type of test. Indeed functional test programs have to be as small as possible while achieving the highest fault coverage.

Some approaches have been proposed so far in the literature to deal with both the memory occupation and the test program length. Although all these approaches share the same title, test compaction process, two main classes of techniques can be identified: dynamic compaction and static compaction. The first one acts during test generation. It can usually achieve good results but at the cost of a high generation time. On the other hand, static compaction is applied after the test generation process. Even if it does not impact the generation time, it can lead to less compaction compared to dynamic compaction since it does not allow any pattern regeneration.

In [3] a dynamic compaction method is proposed. It extracts from a functional test program an independent fragment (called a spore) whose iteration allows achieving a given fault coverage. The authors describe a method to identify the best sequence of spore activations that can achieve the same initial fault coverage, by resorting to an evolutionary approach. The method is effective, but could only be applied under strict constraints as, for example, the test of an arithmetic unit.

In [4] authors propose an algorithm able to automatically compact an existing functional test program. Based on instruction removal and restoration solutions, they showed great compaction capabilities keeping in mind the computational costs.

In [5] a new static compaction procedure was proposed to better fit in-field constraints for cores in embedded systems. By using an evolutionary-based strategy, authors were able to discriminate which are the instructions in a given test program are not contributing on the testing goals. This procedure was applied to a set of test programs (i.e., generated using different approaches) targeting a couple of modules inside the minimips pipelined processor core.

The main goal of this paper is to investigate the static test compaction of a given set of functional test programs. The input is a functional test set composed of many functional test programs. The investigation aims at understanding and determining how to select the best functional test program candidates to obtain the smallest set having the best fault coverage. Conversely to [4] and [5], this approach is applied at test program level and not instruction level.

Moreover, we assume in this work that test programs are totally independent from each other in term of fault coverage. Hence, for each test program, we could compute the set of detected faults with a single fault simulation. Results carried out on two different microprocessors show that a 49% reduction in test length and a 28.7% reduction in test application time can be achieved.

The remaining of the paper is organized as follows. In Section II, we introduce the background of this work. Section III presents the fault coverage analysis by giving the main idea of the proposed compaction technique that is further detailed

in Section IV. Section V analyzes the experimental results. Finally, Section VI concludes the paper.

## II. Context and Background

In this work, to evaluate the effectiveness of the functional test programs compaction, we used two different processors having different architectures.

The first processor is the Intel MC8051 non-pipelined CISC processor, with 8-bit ALU and 8-bit registers [6]. The second processor is the minimips, 5-stages-pipelined RISC processor, with 32-bit ALU and 32-bit registers [7]. Both processors have been synthesized with the same industrial 65nm technology. Table I gives the main characteristics of the synthesized processors in terms of number of Gates, Flip-Flops, Primary Inputs and Primary Outputs.

TABLE I. MC8051 & Minimips Gate-Level Characteristics

| Processor | #Gates | #FFs | #PIs | #POs |
|---|---|---|---|---|
| MC8051 | 4307 | 576 | 65 | 94 |
| minimips | 8612 | 1785 | 38 | 67 |

We started from two different functional test sets targeting two different microprocessors. Both tests have been generated using the µGP tool [8]. This tool, based on a genetic approach, is able to generate a population of functional test programs optimizing one or more constraints. The first set of functional test programs has been generated in order to maximize the power consumption of the MC8051. In [9] we already proved that functional programs maximizing power consumption can achieve important fault coverage. On the other hand, the second set has been generated to maximize the stuck-at fault coverage on minimips.

## III. Fault Coverage Analysis

As already described, the two sets of functional test programs have been generated by considering different targets (i.e., maximizing the power consumption and maximizing the stuck-at fault coverage). Therefore, the first part of our analysis is about the fault coverage achieved by those test sets. The considered fault models are the stuck-at and the transition fault models. This kind of analysis has been already presented in detail in [10] and here we summarize the main concepts for the sake of readability.

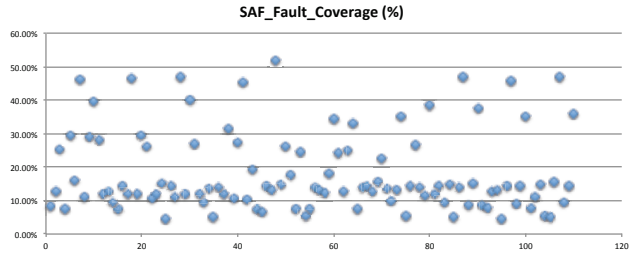


Fig. 1. Stuck-at fault coverage achieved by functional test programs generated for the minimips.

Fig.1 plots the stuck-at fault coverage distribution achieved by the functional test programs generated for the minimips. All fault simulations have been carried out by using a commercial tool [11]. Those functional test programs correspond to the last generation carried out by the µGP tool. As can be seen, the fault coverage varies significantly depending on the programs. Actually, the minimum fault coverage is 4.70% while the maximum is 52%. Table II reports the same analysis for both processors. For each one, we give the maximum and minimum stuck-at as well as transition fault coverage achieved by the functional test programs. Also, we do not forget to give more details about the minimum and maximum memory requirements and test time in columns 5 and 6. The last column reports the number of functional test programs generated for each processor.

TABLE II. Functional Test Programs Analysis

| | Metrics | SAF (%) | TF (%) | Size (Bytes) | Duration (Clock cycles) | #Prg |
|---|---|---|---|---|---|---|
| MC8051 | Max | 43.45 | 25.07 | 618 | $1{,}158*10^3$ | 117 |
| | Min | 35.67 | 19.61 | 346 | $1{,}070*10^3$ | |
| minimips | Max | 52.00 | 43.49 | 1540 | $19{,}61*10^3$ | 110 |
| | Min | 4.70 | 1.02 | 32 | 21.5 | |

As we can see for the minimips, and conversely to the MC8051, from one test program to another we can have a huge difference in terms of memory requirement (i.e., from 32 bytes to 1540 bytes) and test application time (i.e., from 21.5 to 19615 clock cycles). This could be explained by the fact that these test programs were generated to enhance stuck at fault coverage which does not require obligatory a high number of clock cycles. In fact there is no linear relationship between the stuck at fault coverage and the required clock cycles among the generated test programs. We will investigate later the impact of this in the proposed test compaction methodology.

The next analysis we made was about the redundant tested faults (i.e., faults detected by more than one functional test program).

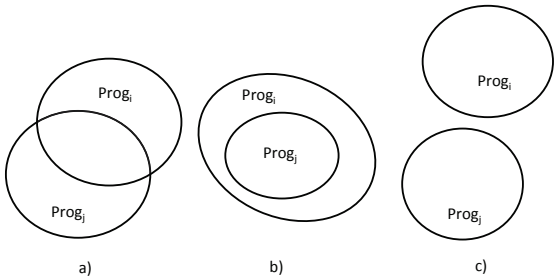


Fig. 2. Test Redundancy.

Fig.2 shows the three possible cases of equivalent faults when considering two functional test programs *i* and *j*. For each program, we plot the set of detected faults defined as $Prog_i$ and $Prog_j$. In the first case shown in Fig.2.a), some faults

are covered by both $Prog_i$ and $Prog_j$ programs. In this case, the same faults are determined by the intersection between detected fault sets. In the second case shown in Fig.2.b), all faults covered by $Prog_j$ are also covered by $Prog_i$. In this case, $Prog_i$ is the superset of $Prog_j$. Finally, in the last case shown in Fig.2.c), there is no relation between the two sets. The two programs detect different faults.

The described cases can be formalized as follows:

a) $Prog_i \cap Prog_j \neq \emptyset$

b) $Prog_i \cap Prog_j = Prog_j$

c) $Prog_i \cap Prog_j = \emptyset$

In the next subsection, we describe how the three cases have been considered in order to deeply analyze the achieved fault coverage.

### A. *Fault Coverage Merging*

The main idea behind Fault Coverage Merging is to understand whether or not it is possible to combine together functional test programs to achieve the highest fault coverage. The final goal is clearly to remove as many test programs as possible without impacting the fault coverage.

Clearly, each program has to be fault simulated in order to compute the set of detected faults.

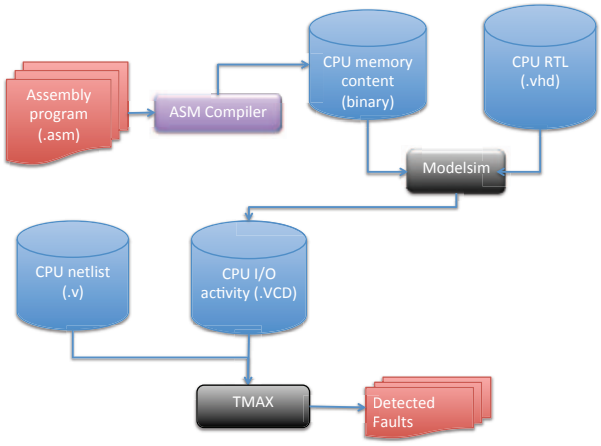


Fig. 3. Flow for determining the set of detected faults.

Fig.3 sketches the flow for determining the set of detected faults. Basically, each functional test program is simulated using *Modelsim* [12] in order to store the Input/Output activity into a VCD file. For this evaluation the RTL description of the microprocessor is used. The VCD file is then used as input for the fault simulator, which is *TetraMAX* [11]. The fault simulator requires the processor described at gate-level (i.e., the netlist). The result of the fault simulation of a given test program *i,* is a set of detected faults which is simply defined as follows:

$$DT_i = \{f_1, f_2, \cdots f_n\} \quad (1)$$

In the charts shown in Fig.4 and Fig.5, we plot the merged stuck-at and transition fault coverage obtained for the minimips. As can be seen in both curves, the fault coverage increases every time a functional test program is fault simulated. From a formal point of view, we see that the relations between the set of detected faults mostly have the same behavior as the cases shown in Fig.2.a) and Fig.2.c). In other words, many functional test programs detect different faults. The limit reached by the merged fault coverage is about 90% for the stuck-at faults and 80% for the transition faults. This is actually very good fault coverage for a functional test.

Another interesting property of the two curves is the fact that they are composed of steps. For example, for the stuck-at faults, we can notice that once the fault coverage reaches 70%, it remains constant for a while and then it directly ramps up to 79%. It means that after reaching 70% of faults, the next ten fault simulated test programs do not contribute to further improve it. Thus, their set of detected faults is completely included in the previous one. This is the case shown in Fig.2.c). In other words, those programs are redundant.

Results for the MC8051 have the same characteristics. The limit reached by the merged fault coverage is about 60% for stuck-at faults and 39% for transition faults. The difference in terms of fault coverage between the two processors is easily explained by the fact that the test set generated for the MC8051 maximizes the power consumption and not the fault coverage.

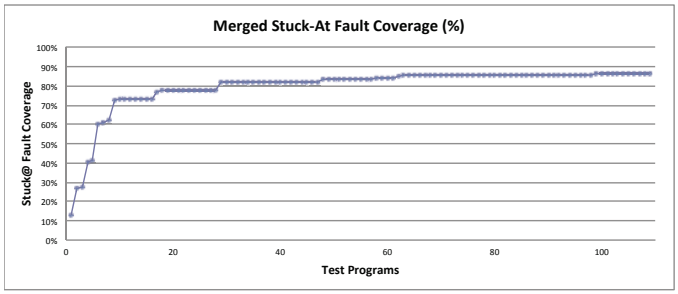


Fig. 4. Merged Stuck-At Fault coverage for the minimips.

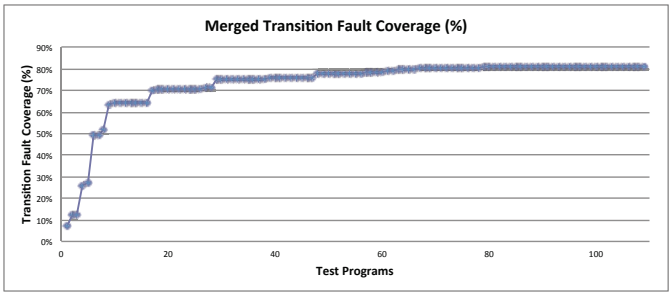


Fig. 5. Merged Transition Fault coverage for the minimips.

The most important result obtained by this analysis is the demonstration that test programs may easily be redundant. Those programs can be removed since they do not contribute to the overall fault coverage.

An interesting property of merging the fault coverage is that the order of test programs execution impacts the number of redundant programs. Let us consider the following

example. We have three test programs P1, P2 and P3. Those programs are characterized by the following DTs:

- $DT_1 = \{f_1, f_2, f_3, f_5\}$
- $DT_2 = \{f_2, f_4\}$
- $DT_3 = \{f_4, f_5\}$

Let us now consider two different program execution orders: P1-P2-P3 and P3-P2-P1. For the first execution order, we start from the execution of P1 that detects the faults in $DT_1$. Then, we execute P2. The overall detected faults is computed by the union between $DT_2$ and $DT_1$:

- $DT_{2\text{-}1} = DT_2 \cup DT_1 = \{f_1, f_2, f_3, f_4, f_5\}$

P2 is not redundant w.r.t to P1 because $DT_1$ is not a superset of $DT_2$. We finally execute P3 and we still compute the final set of detected faults.

- $DT_{3\text{-}2\text{-}1} = DT_3 \cup DT_{2\text{-}1} = \{f_1, f_2, f_3, f_4, f_5\}$

In this case the number of detected faults does not increase because the set $DT_{2\text{-}1}$ is a superset of $DT_3$. Thus, for this program execution order, P3 becomes redundant and thus, it can be removed.

We come back now to the second execution order P3-P2-P1. We apply the same steps as we did for the first one:

- $DT_{3\text{-}2} = DT_3 \cup DT_2 = \{f_2, f_4, f_5\}$
- $DT_{3\text{-}2\text{-}1} = DT_1 \cup DT_{3\text{-}2} = \{f_1, f_2, f_3, f_4, f_5\}$

For the second execution order, we do not identify any redundant program because at each step we increase the set of detected faults. Obviously, the final fault coverage remains the same, irrespective of the order. To conclude, for the first execution order, we can apply only **two** test programs while for the second order we must apply all the **three** programs.

In the next section we will describe the proposed compaction technique based on the program execution order. The goal is to identify the execution order which is able to reduce as much as possible the functional test length.

## IV. COMPACTION TECHNIQUE

In this section, we aim at compacting the given set of functional test programs. Our target is so to keep the achieved fault coverage while removing as much redundant functional test programs as possible. As already stated in the introduction, the proposed approach is a static compaction technique that involves the program execution order to maximize the number of redundant functional test programs.

The proposed approach is similar to the restoration-based technique described in [4], but with the difference that our approach is processed at higher abstraction level (i.e., test program level instead of removing subsets of instructions in one test program). Moreover, since it is applied after the test generation, it is completely independent of the used test generation tool.

The preliminary step required by the compaction technique is the fault simulation of each test program in order to determine the DTs (1).

```
1. Compaction (T)
2.    while (iteration < K)
3.       exe_order = random;
4.       T_tmp = T
5.       cost = remove_redundant  (T_tmp, exe_order)
6.       actual_cost = |T|
7.       while (actual_cost > cost)
8.             exe_order = random;
9.             T_tmp = T
10.            actual_cost = remove_redundant  (T_tmp, exe_order)
11.      iteration = iteration + 1
```

Fig. 6. Compaction pseudo-code.

Fig.6 depicts the pseudo-code of the proposed compaction technique. The input is a given test set T. Actually the main core of the compaction algorithm is composed of two loops. We loop for a given number of iterations K. For each iteration, we randomly specify an execution order for the test programs and we compute the cost by removing the redundant programs (i.e., procedure `remove_redundant`). Then we enter in the second loop. Here we continue to generate random execution orders until the cost is not lower or equal to the old one. The first loop is executed in order to reduce the risk of ending the compaction at a local minimum.

```
1. remove_redundant (T, exe_order)
2.     D = ∅
3.     for each test program i using exe_order
4.           if (DT[i] ∩ D =  DT[i])
5.                remove T[i]
6.           else
7.                D = D U DT[i]
```

Fig. 7. Redundant pseudo-code.

Fig.7 depicts the pseudo-code of the `remove_reduntant` procedure. Here inputs are the test set T and a given execution order. We apply the given execution order in the for-loop and during each iteration we check if the current program *i* is redundant w.r.t. the test programs already applied. Please note once again that the order is specified by the input exe_order. The program *i* is redundant if the intersection between the detected fault set of *i* (DT[i]) and the global detected fault set D is equal to the detected fault set of *i* as described in Section II (i.e., D is a superset of DT[i]). If *i* is redundant we remove it from T, otherwise we compute the new global set of detected fault D as the union between D and DT[i].

The computational cost of the compaction procedure is, in the worst case, equal to K * |T|, where |T| is the number of test programs included in the test set T and K is the maximum number of iterations. Please note that the detected fault sets DTs have been computed before the compaction execution.

## V. EXPERIMENTAL RESULTS

This section presents experimental results carried out on the two case studies: MC8051 and minimips. For all experiments, we fix the maximum number of iterations K equal to 300. For all reported experiments, the compaction CPU time does not exceed one minute.

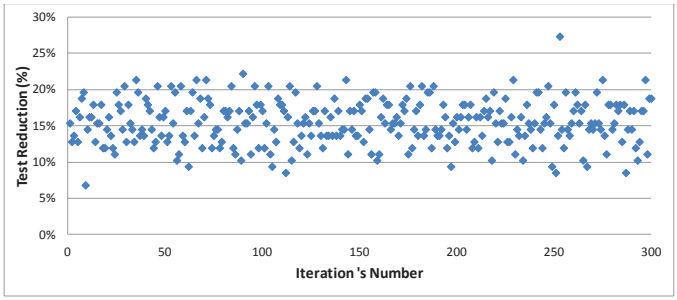


Fig. 8. % of test reduction (S@F MC8051).

Fig.8 plots the chart giving the percentage of test reduction at each iteration of the compaction procedure. On the horizontal axis we plot the iterations (i.e., 300 random execution orders); while on the vertical axis we plot the percentage of test reduction. The test reduction is computed as the number of redundant functional test programs over the total number of programs (i.e., for the MC8051, the number is 117 test programs as reported in Table II). The data are related to the MC8051 and the stuck-at fault coverage. It is interesting to note that the maximum percentage of reduction is 27% while the minimum is only 7%. Please note that irrespective of the execution order, the stuck-at fault coverage is always the same: 60% (see Section III.A).

We performed the same type of experiments for both processors and fault models. Table III summarizes the results obtained. For each processor and fault model we report the maximum and minimum test reduction. For all the cases, the maximum reduction is important since we can remove from 27% up to 49% of test programs without affecting the fault coverage. For all experiments, we fix the number of iterations K equal to 300.

TABLE III. Test Reduction %

| Processor | Test Reduction % | SAF | TF | K |
|---|---|---|---|---|
| MC8051 | Max | 27 | 29 | 300 |
| | Min | 7 | 19 | |
| minimips | Max | 49 | 38 | 300 |
| | Min | 27 | 22 | |

The above results show that by changing the program execution order, it is possible to remove many redundant test programs. Despite this important result, it could be difficult to quantify the real gain. Indeed, we cannot state anything about the test length except regarding the number of test programs, but what about the test time? To answer this question we have to consider the test time required by each test program.

Let us resort to another example to explain this point. We have four test programs P1, P2, P3 and P4. These programs are characterized by the following DTs:

- $DT_1 = \{f_1, f_2, f_3, f_5\}$
- $DT_2 = \{f_2, f_4\}$
- $DT_3 = \{f_4, f_5\}$
- $DT_4 = \{f_1, f_4\}$

For our example, we also consider the test execution time expressed in terms of clock cycles (i.e., how many clock cycles are required by each program):

- TestTime (P1) = 50
- TestTime (P2) = 5
- TestTime (P3) = 15
- TestTime (P4) = 10

TABLE IV. Test time example

| Order | Redundant Programs | CC |
|---|---|---|
| P1-P2-P3-P4 | P3, P4 | 55 |
| P1-P3-P4-P2 | P4, P2 | 65 |
| P1-P4-P3-P2 | P3, P2 | 60 |

Table IV summarizes our example. We consider three execution orders reported in the first column. For each execution order, the second column reports the redundant programs. Those programs are determined by following the compaction procedure described in Section IV. The last column reports the clock cycles required by the merged test sequences. In other words, we compute the CC value as the sum of the clock cycles of each program composing the final test sequence. Thus, for the first execution order, we have 55 clock cycles including the 50 required to execute P1 plus the 5 clock cycles required to execute P2. Once again we do not consider P3 and P4 since they are identified as redundant programs for this order. In the same way we computed the CC for the other execution orders. As it can be seen, the CC varies depending on the programs composing the final test sequence. However, all the execution orders require the execution of two programs over the initial four.

The above example clearly illustrates that by only looking at maximizing the number of redundant programs we could not reach the shortest test time.

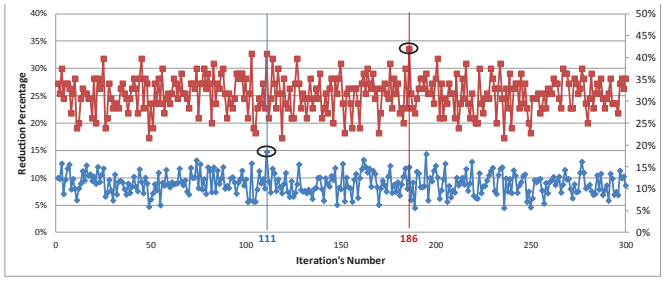


Fig. 9. Reduction and Test Time (Transition Faults minimips).

Fig.9 reports two curves. The red one reports the percentage of test reduction achieved by the compaction procedure during each iteration (the values are reported on the left side of the chart). The blue one reports the percentage of test time reduction (the values are reported on the right side of

the chart). The test time reduction is computed as the sum of clock cycles of the final test sequence over the total number of clock cycles. The latter is the sum of all the 110 programs clock cycles. The data are related to the minimips and the transition fault coverage.

This plot is interesting because it shows that the maximum test reduction (equal to 34%) is obtained at the iteration 186. However, that iteration does not correspond to the lowest number of clock cycles. Indeed, at the iteration 186 we have 15% of test time reduction. The maximum of test time reduction (18%) is obtained at the iteration number 111.

TABLE V. Test Time Reduction %

| Processor | Test Reduction % | SAF | TF | Clock Cycles |
|---|---|---|---|---|
| MC8051 | Max | 28 | 28.7 | $129{,}084*10^3$ |
| | Min | 8 | 19 | |
| minimips | Max | 20 | 18 | 72,290 |
| | Min | 7 | 6 | |

We performed the same type of experiments for both the processors and the fault models. Table V summarizes the results obtained. For each processor and fault model, we report the maximum and minimum test time reduction. For all the cases, the maximum reduction is important since we can reduce up to 28.7% the test time without affecting the fault coverage. Please note that 300 iterations are considered for all the experiments.

To summarize, the proposed compaction technique is able to determine the shortest test set in terms of number of test programs and test time. This kind of technique can be very useful when the test is performed during the mission time where a short test time window is available. In this case the main goal is to reach the maximum of the fault coverage by running the shortest number of test programs as possible or by running the shortest test programs. The user can tune the proposed technique to determine the best test suite.

## VI. CONCLUSION

In this paper, we investigated the static test compaction level that can be achieved starting from a given set of functional test programs. The input of the proposed compaction technique is a functional test set composed of many functional test programs. The compaction technique is able to determine the best functional test program candidates to obtain the smallest set while preserving the original fault coverage. Results carried out on two different microprocessors show that a 49% reduction in test length and a 28.7% reduction in test time can be achieved.


## ACKNOWLEDGEMENT

This work has been partially funded by the International Associate Laboratory (LIA-CNRS) French-Italian research Laboratory on hardware software Integrated Systems LAFISI.


## REFERENCES


[1] J Valka M., Bosio A., Dilillo L., Girard P., Pravossoudovitch S., Virazel A., Sanchez E., Sonza Reorda M., "A Functional Power Evaluation Flow for Defining Test Power Limits During At-Speed Delay Testing", 2011 16th IEEE European Test Symposium (ETS), DOI: 10.1109/ETS.2011.21. (2011)

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

[4] M. Gaudesi, M. Sonza Reorda, I. Pomeranz, "On Test Program Compaction", 2015 20th IEEE European Test Symposium (ETS), DOI: 10.1109/ETS.2015.7138771

[5] R. Cantoro, M. Gaudesi, E. Sanchez, P. Schiavone, G. Squillero, "An evolutionary approach for test program compaction" Test Symposium (LATS), 2015, DOI 10.1109/LATW.2015.7102406

[6] D. Appello, V. Tancorre, P. Bernardi, M. Grosso, M. Rebaudengo, M.S. Reorda, "On the Automation of the Test Flow of Complex SoCs", IEEE VLSI Test Symposium, pp. 166-171, 2006, DOI: 10.1109/VTS.2006.51

[7] "miniMIPS Overview," opencores.org, 2009. [Online]. Available at http://opencores.org/project,minimips.

[8] E. Sanchez, M. Schillaci, G. Squillero, "Evolutionary Optimization: the µGP toolkit: The UGP Toolkit", 2011, DOI 10.1007/978-0-387-09426-7_5

[9] A. Touati, A. Bosio, L. Dilillo, P. Girard, A. Virazel, P. Bernardi, M.S. Reorda, "Exploring the impact of functional test programs re-used for power-aware testing", Design, Automation & Test in Europe Conference & Exhibition (DATE), pp.1277-1280, 2015

[10] A. Touati, A. Bosio, L. Dilillo, P. Girard, A. Virazel, A. Todri-Sanial, P. Bernardi, "A Comprehensive Evaluation of Functional Programs for Power-Aware Test," *IEEE North Atlantic Test Workshop*, pp. 69-72, 2014

[11] Synopsys Inc., TetraMAX®, User Guide 2013

[12] MentorGraphics, ModelSim®, http://www.model.com