Automated Test Program Reordering for Efficient SBST

R. Cantoro, E. Cetrulo, E. Sanchez, M. Sonza Reorda, A. Voza
Politecnico di Torino, Dip. Automatica e Informatica
Torino, Italy

***Abstract*[1]—Software-based Self-test (SBST) is one of the techniques adopted to detect latent faults in safety-critical applications, thus aiming at preventing them from producing failures. When adopted for in-field test, not only the achieved fault coverage, but also the test duration of SBST test programs become critical parameters. Sometimes, these test programs are created following guidelines allowing to guarantee a given Fault Coverage with reduced test duration. In other cases, existing test programs are re-used. Hence, it is important to devise automatic techniques able to modify them in such a way that the fault coverage is kept unchanged (or increased) while the test duration is reduced. This paper presents a possible approach in this direction. Its effectiveness is evaluated on some test programs targeting the openMSP430 processor. Experimental results show that the proposed method is able not only to significantly reduce the test duration (up to 26%), but also to further increase the achieved Fault Coverage, while keeping the required computational time acceptable.**

I. INTRODUCTION

There is a growing number of domains (e.g., biomedical, automotive, aerospace, telecommunications) where electronic systems are used for implementing safety-critical applications. In all these scenarios, it is crucial to identify techniques able to guarantee that a given level of dependability can be achieved. When dealing with permanent faults, this requires not only being able to perform an effective end of production test, good enough to identify faulty components with extremely high accuracy, but also to run test procedures during the operational phase, which may identify faults which arose in this phase, due for example to aging.

In this context, different solutions are often combined, ranging from Design for Testability (DfT) to functional test. Different products and different semiconductor technologies may require a different mix of solutions to achieve the best results.

This paper focuses on functional test, which is now extensively used to complement other test solutions, especially because it turns to be able to detect some defects (e.g., those related to timing behavior) which can more difficult to detect with other techniques. Moreover, functional test works on the system in the same configuration of the operation scenario, and hence it is less likely to produce over testing.

When the target system includes at least one CPU, the functional test is typically based on a test program, which is run by the processor. By looking at the produced results we can detect possible faults affecting the system (*Software-based Self-test*, or *SBST*) [2]. In the recent past, several research efforts revamped the effectiveness of SBST (whose first examples date back to more than 30 years ago [1]) by introducing new techniques belonging to this paradigm, and providing algorithms able to support the test engineer in the generation of SBST test programs addressing the different modules in a CPU, in the memory, in the peripherals, and in the interconnection structures. SBST also demonstrated to be particularly effective when used for in-field testing, since it can more easily match the several constraints coming from the application environment (e.g., in terms of intrusiveness) than other solutions, such as those based on DfT [8].

Clearly, the major parameter characterizing any test solution (including SBST) lies in the achieved Fault Coverage. At the same time, especially when used for *true* in-field test (e.g., when the test program is activated exploiting the time periods left idle by the application) [7], a major parameter characterizing a given SBST test program corresponds to its duration. In some cases, the test program is developed by following an algorithm able to optimize the test duration [3]. In other cases, the test program already exists, and we would like to optimize it. For this reason, some efforts have been recently started [4][5][6], to identify automatic or semi-automatic ways to reduce the duration of an SBST program, while maintaining its Fault Coverage.

In this paper, we propose a new and original method belonging to the same category, which has been named *ARES* (*Automated Reordering for Efficient SBST*). The key idea behind the method is that we can significantly reduce the duration of a test program by first dividing it in pieces (called *groups*), and then reordering them in a clever manner, such that some of the pieces can be removed. Remarkably, experimental results show that in some case, the approach can also improve the achieved Fault Coverage. In the paper, we first provide a description of how an existing test program can be partitioned in suitable groups, and then we propose a method to identify the most suitable ordering of the groups, so that duration can be reduced while at least keeping the same Fault Coverage. Special heuristic techniques are proposed to reduce the computational cost for selecting the most promising ordering solutions.

The rest of the paper is organized as follows. In Section II we describe the proposed method. In Section III we report some experimental results proving its effectiveness on a sample processor. In Section IV we draw some conclusions.

II. PROPOSED METHOD

A. Background

The problem presented in this paper is described as follows. It is assumed that a test program TP for a given processor is available. The test program consists of *n* instructions, its execution lasts for *cc* clock cycles, and achieves a fault coverage FC. In this paper, single stuck-at faults are

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

considered. We assume that a fault is detected when it generates a difference compared with the fault-free system on any bus signal and at any clock cycle. The adoption of different fault models or detection mechanisms does not affect the effectiveness of the technique.

The problem is to find a new test program TP' that achieves a fault coverage FC' equal or higher than FC after a number of clock cycles *cc'* lower than *cc*. For this purpose, we partition the instructions in the original TP test program into NG non-overlapping groups. Additional constraints may exist on the partitioning of all the instructions in groups.

We propose an algorithm, called *ARES* (*Automated Reordering for Efficient SBST*), whose purpose is to automatically modify a given test program by decreasing its duration (i.e., the number of clock cycles for its execution) without decreasing (in some case increasing) its fault coverage. The basic idea behind the ARES algorithm stems from a simple observation: the faults detected by a given piece of code depend on the input data for the code, and by the state of the system (processor and memory) when the code starts. If the system starts from a different initial state, it is possible that different faults are detected, even if the same set of input data are applied. In principle, for each piece of code we could identify the initial state which maximizes its fault coverage: however, this would result in an unacceptably high computational effort. Hence, the main idea behind the ARES algorithm is to first partition an existing test program into several pieces (hereinafter called *groups*), and then to reorder them in such a way, that the initial states for the different pieces are globally the most suitable ones. The duration of the test program corresponds to the number of executed instructions. We refer to the duration of a test program as test length. If we assign the final ending instruction to a particular group, by reordering we can change the position of this group in the final test program and, as a consequence, the duration of the test program will correspondingly change. Clearly, since the number of possible ordering of the test programs grows exponentially with the number of groups, we need some a smart technique to select a small number of them, which will then be fault simulated in order to assess their effectiveness.

To summarize, the proposed algorithm works in two stages:

1. Instruction group partitioning and generation of the groups
2. Group reordering and generation of the list of possible best results.

B. Stage 1 – Instruction group partitioning and group generation

The inputs of the Stage 1 of the ARES algorithm are:

- The original test program *OTP*
- The number of instruction groups *NG* to be divided
- The number of test programs *NTP* to be generated
- The number of attempts of logic simulations *NALS* to be performed.

A sequence of instructions can represent a group if this sequence does not contain macro definitions, procedure definitions, or jump instructions to a target instruction outside the sequence. For this reason, we define the term *admissible region* as the region of a test program where the code can be split. A *non-admissible region* can be the macro definition region, the procedure definition region or the region between a target and its related jump instruction.

There are two methods to perform the Stage 1: manual and automated. The two approaches are independent with respect to the Stage 2 because this last step needs only of a test program partitioned in a certain number of groups. For partitioning, the manual approach can use functional information as to divide the test program in groups related to the different parts of each module.

In the following, we present an automatic method whose pseudo-code is reported in Fig. 1. The Stage 1 is in charge to create non-overlapping groups, delimited by *divisor characters*. Given a certain target number of groups, different grouping solutions can be identified and, according to a *quality* parameter, only the best one is returned.

```
Input: originalTestProgram (OTP), NumberTestPrograms (NTP),
NumberOfGroups (NG), NumberAttemptLogicSimulations (NALS)
Variables: GroupedPrograms{} (GPs), PartitionedTestProgram
(PTP), ElectedProgram (EP)
begin
Z := NumLines(OTP)/NG
for i = 0 up to NTP do
 |  while groupsPartitioning(OTP, Z) is success do
 |   |  PTP := groupsPartitioning(OTP, Z))
 |   |  GPs{i} := verificationChecking(PTP, NALS)
 |  loop
loop
EP := qualityAnalysis(GPs)
Return (EP)
```

Fig. 1. *Pseudo-code for Stage 1*

For each *for* loop, the considered OTP, given as input to *groupsPartitioning()*, initially has not divisor characters; inside this function, the test program is divided into groups, leading to a Partitioned Test Program (PTP). Fig. 2 reports the pseudo-code of the *groupsPartitioning()* procedure, whose goal is to automatically partition the OTP into groups.

```
Input : originalTestProgram (OTP), AVGDistanceBetweenLabels (J)
Variables : Line (L) = Second-Part, indexGroups (IG) = 0,
PartitionedTestProgram (PTP)
begin
PTP := OTP
while Third-Part do
 | L := L + abs(J + Rand(-X, +X))
 | if (L is in an admissable region) then
 |  | IDGroup(InsertGroup(IG++), L, PTP)
 | end if
loop
if RunLogicSimulation(PTP) is failed then
 |     return (FAIL, PTP)
else
 |     return (SUCCESS, PTP)
end if
```

Fig. 2. *Pseudo-code for groupsPartitioning()*

The *IDGroup()* function writes at row L the ID of the IG-th group in the PTP test program. *IDGroup()* generates the partitioned test program. The OTP is divided in three different parts:

1. Macro and constants initialization
2. Where the test starts

3. Where interrupt service routines and procedures are implemented.

The L variable corresponds to a pointer to the lines of the OTP starting from the second part.
After group partitioning, the *verificationChecking()* function is activated, whose pseudo-code is reported in Fig. 3. This function performs the logic simulation on each sorted grouped program.
The *grouped program* corresponds to the PTP variable.
In the *verificationChecking()* function, the input argument is the PTP that was generated by the previous function *groupsPartitioning().*

```
Input : PartitionedTestProgram (PTP) ,
NumberAttemptLogicSimulation (NALS)
Variables : Stats (STATS), MaxTestLength (MAXTL),
AvgBestTestLength (AVGBTL), NumFails (NUMF), %sel
begin
for i = 0 up to NALS do
 |  STATS{i} := LogicSimAndComputingTestLength(sort(PTP))
loop
MAXTL = findmax(STATS, NALS)
NUMF = findfails(STATS, NALS)
AVGBTL = findAvgBest(%sel, STATS, NALS)
return (STATS, MAXTL, AVGBTL, NUMF)
```

Fig. 3. *Pseudo-code for verificationChecking()*

PTP is sorted by the *sort()* function and the logic simulation is executed on this sorted test program and the information results are stored in an array of statistic figures, called STATS. The *sort()* function generates a seed (ID) string that is used to generate the sorted test program that is given as input to the logic simulation. The logic simulation is performed NALS times for NALS different sorted test programs, starting from the PTP test program. The value for NALS should be chosen on the based on statistical analysis. At the end of these NALS logic simulations, the following values are computed and returned:

- the maximum test length between the sorted test programs (MAXTL)
- the number of failures of logic simulation on the sorted test programs related to the original PTP partitioned test program
- the average best test length between the sorted test programs.

STATS contains different information:

- The duration of each sorted test program
- Whether the i-th sorted test program fails the logic simulation or not
- The seed (ID) string of each sorted test program
- The PTP test program to reconstruct a sorted test program by its seed string.

The values returned by the *verificationChecking()* function are elaborated by the *qualityAnalysis()* procedure, whose pseudo-code is shown in Fig. 4.

```
Inputs : GTPs → {MaxTestLength(MAXTL), AvgBestTestLength
(AVGBTL), NumFails(NUMF), ElectedProgram(EP)}
Variables : bestGTP = worstGTP
begin
for each currGTP in GTPs  do
 |  if fbestGTP(currGTP, bestGTP) then
 |   |  bestGTP := currGTP;
 |  end if
loop
return (bestGTP→EP)
```

Fig. 4. *Pseudo-code for qualityAnalysis()*

In the *qualityAnalysis()* function, for each grouped test program, the *quality* is computed and it is compared with the maximum current quality related to another grouped test program. At the end, the grouped test program related to the maximum quality is returned.
fbestGTP() is an heuristic function that evaluates the quality of a grouped test program according to the quality parameter. The parameters used to compute the test program quality are:

1. Number of Failures NF, used in the formula: -NF/NALS = XXNF
2. AvgBestTestLength, used in the formula: AvgBestTestLength/MAXABS= XXABTL
3. MaxTestLength, used in the formula: MaxTestLength/MAXABS = XXMTL

MaxTestLength is the maximum test length for a single grouped test program. MAXABS is the maximum between the test lengths among all the grouped test programs. We discard all the test lengths that overcome the test length of the original test program. The quality can be computed as:

```
quality = ((a)·XXNF + (b)·XXABTL + (c)·XXMTL)
```

where a, b, c ∈ □ such that a > b > c.
These coefficients refer to the weight of each one of these parameters and they have to be calibrated with experimental analyses.
In practice, the Stage 1 is used to check what is the best grouped test program to be sent to Stage 2 for effective reordering.
The Stage 1 can be implemented also by a manual approach by following the constraints explained above.

C. Stage 2 - Group reordering and generation of the list of possible best results

From the original test program, we generate several possible seeds. A *seed* is a unique identifier that associates the original test program with the generated sorted one. Each seed is used to reconstruct the related test program with a certain reordering. These seeds are randomly generated, by picking a random instruction group and assigning it a random position in the final test program. Among SN possible seeds, we perform the fault simulation of those programs that overcome a given threshold in terms of duration (ThTL), only.

```
Input : originalTestProgram (OTP), SeedNumber (SN),
ThresholdTestLength (ThTL)
Variables :  SeedsList (SL) {} , CurrentSortedTestProgram (CSTP),
CSTPs {SortedTestProgram, NumTestLength}, Coverage (CV)
begin
for i=0 up to SN do
 |  SL{i} := seedGeneration(NG)
```

```
  | CSTP := reorderingAlgorithm(SL{i}, OTP)
  | CSTPs{i} := LogicSimAndComputingTestLength (CSTP)
loop
{CSTPs, SL} := cutThreshold(SL, CSTPs, ThTL)
for each CSTP in CSTPs do
  | CV{i} := RunFaultSimulation(CSTPs)
loop
return (CV, SL)
```

Fig. 5. *Pseudo-code for Stage 2*

In the *seedGeneration()* function, a seed string is generated, that defines the order of instruction groups in the sorted test program. In this function, each instruction group belonging to the original test program is placed in a random position (between 0 to NG-1) in the final test program.

```
Inputs : Ngroups (NG)
Variables : SeedList (SL), GroupSets init{NG} (GS) ,
TemporalRandomIndex (TRI), TemporalRandomValue (TRV)
begin
for i=0 up to len(NG) to do
  | while (SL{TRI := rand(0, NG-1)} not initialized ) do
  |   | while (GS{TRV := rand(0, NG-1) not used} do
  |   |   | SL{TRI} := TRV
  |   |   | GS{TRV} := used
  |   | loop
  | loop
loop
return (SL)
```

Fig. 6. *Pseudo-code for seedGeneration()*

The generated seed string will be used in the *reorderingAlgorithm()* function to append, inside a new file that corresponds to a new sorted test program, the several instruction groups in the position dictated by this seed string.
The *reorderingAlgorithm()* function uses the seed string to determine the order of instructions in a new (sorted) test program. The original test program is divided in three parts: the first part, the central part, and the last one. The first part is the segment of constants and macros initialization. The last one is the portion after which the last instruction will be executed that can contain the implementations of the procedures. Finally, the central segment contains the groups to be sorted.

```
Inputs : originalTestProgram(OTP) , Ngroups (NG), SeedList(SL)
Variables : ReorderedTestProgram(RTP), seedElement (SE),
GroupsTestProgram {} (GTP), InitTestProgram (ITP),
ProcTestProgram (ProcTP)
begin
ITP = pickfirstpart(OTP)
GTP = pickgroupspart(OTP)
ProcTP = pickprocpart(OTP)
RTP := Print(ITP)
for each SE in SL do
  | RTP := Print (GTP{SE})
loop
RTP := Print(PTP)
return (RTP)
```

Fig. 7. *Pseudo-code for reorderingAlgorithm()*

After the *reorderingAlgorithm()* function, a logic simulation and the computation of the test length are performed on the current (sorted) test program by means of the *LogicSimAndComputingTestLength()* function.

The purpose of the *cutThreshold()* function (see the pseudo-code of the Stage 2 in Fig. 5) is to reduce the number of CSTPs that is an array of structures where each one contains all the information of the logic simulation. In particular, each item of the list is removed from it, according to the comparison between the test length threshold and the test length resulted from the logic simulation of the current seed. To summarize, *cutThreshold()* deletes all the seeds whose duration is lower than the given threshold.
At the end, the fault simulation is executed on the sorted test programs related to the seeds contained in the CSTPs returned by *cutThreshold()*. The Stage 2 returns the resulting fault coverage of each sorted test program and its related seed.
The drop in the test duration with respect to the original test program is obtained because, by moving the group (in the last part of the intermediate region of the test program) containing the end-of-test instruction, the test may terminate before the original one.

III. EXPERIMENTAL RESULTS

A. *Experimental setup*

We wrote a prototypical tool implementing the ARES algorithm using Python and Bash (the number of lines is around 300). The tool interacts with Mentor ModelSim for the logic simulation and to Synopsys TetraMAX for the stuck-at fault simulation and ATPG. The tool was run on an Intel Xeon 2 GHz with 8 GB of RAM, using at most one thread.

Resorting to the tool, we experimentally validated the proposed algorithm on a target core corresponding to *openMSP430*, a 16-bit microcontroller described in Verilog and available through OpenCores [9]. The core has some embedded peripherals like a 16x16 HW Multiplier, Watchdogs, Timers. When synthesized with a 65-nm CMOS technology library, the openMSP430 resulted in approximately 8k gates [9]. The reader should note that the size and complexity of this CPU module is comparable with many similar CPU modules used in safety-critical embedded applications, e.g., in the automotive domain. Table I reports the main modules of the core under test and the corresponding number of stuck-at faults. The total number of faults labeled as undetectable by TetraMax is 726.

TABLE I NUMBER OF FAULTS FOR EACH MODULE OF THE OPENMSP430 CORE

Whole CPU	Frontend	Register file	ALU	Memory backbone	Multiplier
29,424	5,328	9,472	2,568	2,054	6,362

The ARES algorithm was experimentally verified in three different test programs, whose main characteristics are reported in Table II.

TP1 has been generated by analyzing the architecture of the core and focusing on the generation of the instructions related to the main blocks, like Multiplier, Register File and ALU. After having analyzed each block and manually generated the corresponding test programs, these last ones have been merged in a single test program.

For TP2, test vectors for the ALU were generated by running the ATPG on some sub-modules. The generated test vectors were then turned into instructions acting on the corresponding input values. For the test of other sub-modules, some extra code was added, using other instructions and addressing modes. The test of the multiplier module has been generated by using Pseudo Random numbers based on a Fibonacci 16-bit LFSR model. All test chunks have been first generated in C and then translated into assembly language by *msp430-gcc*. At the end, everything was merged in the final test program.

TP3 was generated starting from the test program examples provided by the developers of the processor. Starting from the best test programs, some new code was added, targeting rarest faults not yet detected. Afterwards, ATPG was used to generate test vectors for the ALU module, as done for TP2. For the register file module, a March algorithm was implemented. Then, instructions implementing the different addressing modes were added to test the frontend module, as well as instructions to test the hardware multiplier.

TABLE II THE ORIGINAL TEST PROGRAMS

	Size [KB]	Test Length [#clock cycles]	FC %
TP1	507.1	118,752	94.50%
TP2	109.1	33,496	93.48%
TP3	355.2	119,402	94.45%

B. Results

To prove the effectiveness of the ARES algorithm, the three test programs have been analyzed using different group configurations. We started from a maximum number of groups for each test program. This value has been chosen by evaluating the *quality* parameter in order to not affect itto avoid affecting it. Starting from this maximum value, we then progressively reduced the number of groups of the treated test program. For each grouped test program, we selected from a pool of test programs that were (logically) simulated a random number corresponding to the number of fault simulations we need to be performed. In Table IV, the original values and the values obtained by the algorithm in terms of test length (TL) and fault coverage (FC) are reported. The required total computation time in hours, that involves the automated *Stage 1* and *Stage 2,* is also shown.

In Table V each original test program has been compared with its best sorted version, obtained by executing the fault simulation several times (40 for TP1 and TP3, and 80 for TP2). After the execution of all fault simulations, the one with the highest fault coverage for each test program using the ARES algorithm has been collected. The ARES algorithm seems to optimize well the test programs targeting the register file and the memory backbone. This is mainly due to the high number of memory accesses (read and write). Differently than for the ALU and multiplier modules, the memory backbone and the frontend modules are not controllable directly through a set of instructions. In the case of the ALU, we have a relatively minor increase because the original test program was already well optimized.

In Fig. 8, the values trend of the fault coverage when the number of groups is changed (assuming the values 100, 150, 200, 253) in TP1 are displayed. There are 3 classes of lines: the red ones correspond to the maximum fault coverage achieved among 10 test programs that have been randomly chosen. Instead, the blue lines are the average over 10 fault simulations. Each circle is a group range in terms of test length. In most of test length ranges, the maximum value (represented by the red line) is always above the original coverage of the original test program. In TP1, the optimization seems to work well even considering that the average does not have a falling trend. This is important because the closer the average is to the original FC, the more probable is that a good FC can be achieved with less *test attempts* (#TA). It means that the algorithm has the capability to be inexpensive in term of test duration with a good result, because it is based only on the logic simulation. Hence, the information provided by the logic simulation are enough to understand (without executing the fault simulation) the effectiveness of a sorted test program.

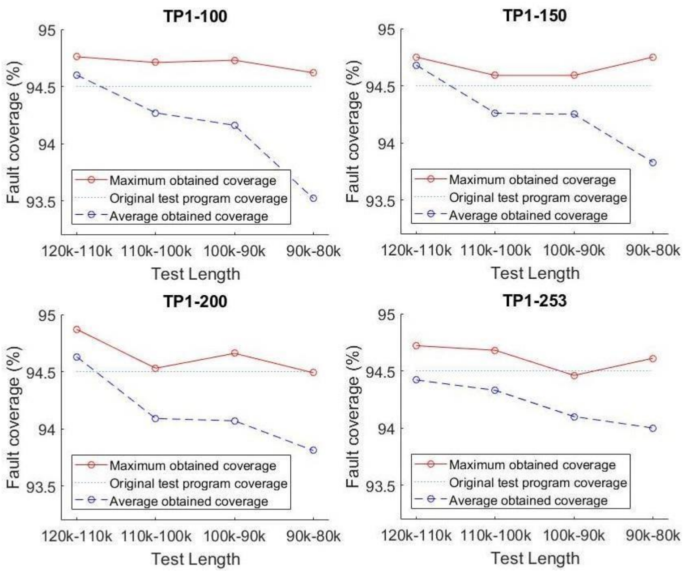

Fig. 8. *FC vs. Test Length for various group sizes for TP1*

Figure 9 shows the values of the fault coverage for different numbers of groups (100, 165) and with decreasing test duration. It is clear that the FC drops faster compared to the other two test programs. This happens due to the shorter length of this test program. In fact, it is more probable that, in this kind of test, the presence of *dead codes*, defined as pieces of code that do not increase the fault coverage and are redundant from the fault simulation point of view, are rarer.

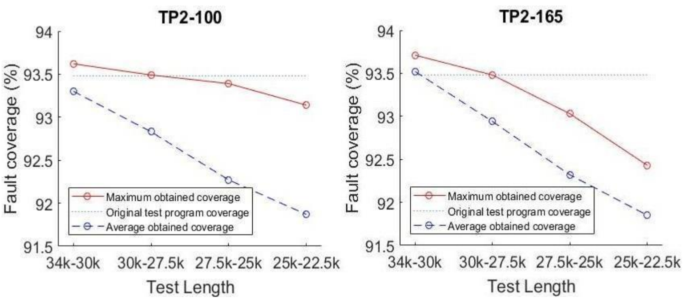

Fig. 9. *FC vs. Test Length for various group sizes for TP2*

If the number of fault simulation experiments increases, the

average fault coverage can be more accurate and the maximum fault coverage can be higher. But thanks to the ARES algorithm and its compaction efficiency, a good result can be achieved.
Finally, Table III reports a comparison between the best trade-off FC-TL on TP3 with different number of groups.

As a conclusion, the following two observations can be drawn:

- The performance of the ARES algorithm depends on the structure of the original test program. In general, the longer a test program is, the higher the amount of *dead-code* it contains, and hence the possibilities for compaction.
- The partitioning into groups plays a fundamental role for the optimization. The optimal configuration can be achieved using different configurations through which the test program has been divided, because each test program can have different intrinsic characteristics in terms of length and complexity. The optimal number of instruction groups is a bit more than half of the maximum number of groups of a test program.

IV. CONCLUSIONS

This paper deals with the compaction of an existing SBST test program. It proposes a method that is based first on automatically partitioning the program in chunks, named *groups*. Secondly, several possible reordering of the groups are considered; each of these reordering is preliminarily evaluated via inexpensive logic simulation. The most promising ones are then fault simulated, and the best is selected. Interestingly, experimental results gathered on a target CPU core show that the method is able not only to significantly reduce the duration of the existing test programs, but also to slightly increase the achieved fault coverage. The whole computation effort remains acceptable.
Work is currently being done to further improve the method and to assess its effectiveness on other CPU cores.

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", 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] M. Gaudesi et al., "New Techniques to Reduce the Execution Time of Functional Test Programs", IEEE Transactions on Computers, 2017.

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

[7] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 24, Issue: 1, 2005, pp. 88 – 99.

[8] Jacob A. Abraham et al., "Special session 8B — Panel: In-field testing of SoC devices: Which solutions by which players?", 2014 IEEE 32nd VLSI Test Symposium (VTS).

[9] "OpenMSP430 project", [Online]. Available at "https://opencores.org/project,openmsp430".

TABLE III COMPARISON AMONG BEST SORTED TEST PROGRAMS AND ORIGINAL TP3

Test Program-# of groups	TP3-100	TP3-250	TP3-373
Fault Coverage ARES	94.58%	94.84%	94.69%
Test Length ARES	94,842	88,240	99,942
Original Fault Coverage	94.45%		
Original Test Length	119,402		
Coverage: ARES vs Original	+0.13%	+0.39%	0.24%
Test Length: ARES vs Original	-20%	-26%	-16.3%

TABLE IV SUMMARY COMPACTION/MAXCOVERAGE/TOTAL COMPUTATION TIME AMONG ALL TEST PROGRAMS

Test Program	**Original TL**	**New TL**	**Compaction**	**Original FC**	**New FC**	**Computation Time [h]**
TP1	118,752	86,234	27%	94.50%	94.84%	54.29
TP2	33,496	27,980	16%	93.48%	93.74%	25.8
TP3	119,402	88,240	26%	94.45%	94.94%	81.44

TABLE V COMPARISON BEST TEST-PROGRAM GENERATED WITH THE ORIGINAL ONE FOR EACH MODULE

Test Program-# of groups	**Top Module**	**Frontend**	**Register file**	**ALU**	**Memory backbone**	**Multiplier**
TP1 Original	94.50	88.83	95.27	98.83	94.94	95.98
TP1-200 ARES	94.87	89.04	95.67	98.91	95.43	96.17
	+0.37	+0.11	+0.40	+0.08	+0.49	+0.19
TP2 Original	93.48	86.89	93.26	98.05	92.76	96.74
TP2-50 ARES	93.74	87.10	93.94	98.17	93.31	96.44
	+0.26	+0.21	+0.68	+0.12	+0.55	-0.30
TP3 Original	94.45	88.94	95.22	97.82	93.69	96.72
TP3-250 ARES	94.94	89.04	95.77	97.78	95.27	97.04
	+0.49	+0.10	+0.55	-0.04	+1.58	+0.32