2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)

RSBST: A Rapid Software-based Self-test Methodology for Processor Testing

Vasudevan M S[1], Santosh Biswas[2], and Aryabartta sahu[1]
[1]Department of CSE, IIT Guwahati, Assam, India
[2]Department of EECS, IIT Bhilai, India
Email: santoshbiswas402@yahoo.com

***Abstract*— Light-weight software-based test (SBST) techniques are increasingly being used for testing of modern processors because of the ease of synthesis using evolutionary approaches, coverage for difficult to test faults, non-intrusive nature, low hardware overhead etc. However, the test synthesis time required by SBST is high. In this paper, an advancement SBST technique, termed as Rapid SBST (RSBST) is proposed that reduces the overall test synthesis time by reusing the simulation responses of existing test programs of identical observability. The test codes, developed using the evolutionary process, that produce similar fault simulation results are reused for the fault evaluation. We exploit this reusability to enhance the speed of the test synthesis. The efficacy of the proposed scheme is demonstrated on a 32-bit MIPS processor and on a minimal configuration of 7-stage SPARC V8 Leon3 soft processor. The test code generated achieves a fault coverage of 97.3% for the MIPS processor and 96.2% for the Leon3 soft processor. The test pattern generation time is 90 hours and 98 hours for these two processors, respectively. For the similar processors, traditional SBST requires 122 hours and 142 hours, respectively while providing a coverage of 93.9% and 92.9%. So it may be concluded that the proposed RSBST technique speeds up by a factor of 1.35 while maintaining the fault coverage above 96.2%.**

I. INTRODUCTION

As the processor technology is complex and expanding, the reliability of embedded processors is highly critical during the phases of chip manufacturing, and operational stages. Identification of all physical faults has been ever-challenging because the test patterns must be applied at the operational frequencies of ICs, which are extremely high. This at-speed testing feature is very difficult to achieve with external tester technologies as the tester frequencies could not reach up to the processor frequencies [1]. To solve this, Software-based self-test (SBST) methodologies [2], [3], [4] have cultivated software-based test codes to be applied on the processors as test routines. These test codes are a sequence of instructions with selected operands that could validate the processor functionality. The SBST approaches are non-intrusive because the chip design does not necessitate any modification for testing. These light-weight test codes are uploaded into the memory locations and the responses are downloaded and compared for the fault identification. Furthermore, SBST does not require any extra hardware which leads to a reduced test cost and zero chip area penalty [5]. For these reasons, SBST is exceedingly used for embedded processor testing.

The advancements in manual test pattern generation [6], [7], [8] have substantially contributed in developing effective test programs. In these works, complex functional test patterns are developed for testing pipelined processors with multithreading, dynamic instruction execution, and multicores. Nonetheless, the cost of test pattern development is a tradeoff because the assembly programmer has to devise complicated, cost prohibitive test programs for processors manually.

In [9], Corno et.al. describes μGP, an evolutionary approach to automatically synthesize assembly code test programs [9]. The self-adaptive architecture of μGP searches for enhanced test programs. The earlier μGP approaches employ statement coverage as the code coverage metric [9], [10]. Later, the same approach was extended with toggle, branch, expression, and condition coverages [11], [12]. However, the testability of corner case faults, termed as hard-to-test faults, would be significantly low since they are not easily traceable by the synthesized test programs.

These previous μGP techniques, proposed in [9], [10], [11], [12], could not guarantee the test quality because these hard-to-detect faults were left undetected and the code coverage-based fault evaluation metrics do not hold strict correlation with gate-level fault models. These methods could not realize a gate-level fault coverage of more than 90% for the synthesized test programs. These μGP experiments converge in tens of hours [10], which is reasonably fast but lacks adequate fault coverage. On the other hand, the recent SBST automation approaches [13], [14] have improved in terms of coverage but rely on time-consuming test generation techniques.

Suriasarman et al. [14] have proposed a greedy framework for the conventional μGP approach of test synthesis to discover the hard-to-detect faults of the processor. Here, they have integrated a greedy framework into the μGP-based SBST synthesis of processor cores along with a testability analysis feature. To evaluate the quality of the synthesized test programs, they employed a behavioral-level fault model and finally, reported that the synthesized test solutions could trace 95.8% of the testable faults of a Leon3 processor and 96.32% of the testable faults of a MIPS processor. Although the greedy component improves the test quality, this test generation procedure [14] would have to bear approximately 170 hours of fault evaluation and comparison. So, the test synthesis has a slower convergence because the evolutionary module focuses on the comprehensive search for the hard-to-detect faults. Eventually, 40% of the hard-to-detect faults are traced and identified by Suriasarman et al. [14], but the test synthesis time is longer.

This paper discusses a rapid software self-test technique termed as, rapid SBST (RSBST), where the test synthesis is faster compared greedy based evolutionary approaches, at the same time does not drop the fault coverage. In RSBST, redundant test solutions are identified and their simulation results are reused for a faster fault coverage evaluation. This reusability scheme is developed to expedite the traditional evolutionary test synthesis. Many of the test solutions developed using the evolutionary process would

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00038

produce similar fault simulation results. Thus, in RSBST, these fault simulation results are reused to evaluate the fault coverage of test programs with similar characteristics of fault identification. This substitution could remarkably reduce the test synthesis time without compromising the fault coverage.

To summarize, we propose RSBST, which is an effective test synthesis, where *hard-to-detect faults are identified and the reusability of test programs helps in the reduction of test synthesis time. These hard-to-detect faults are detected with the help of a greedy-cover based evolutionary test synthesis. The test synthesis time is reduced by reusing the simulation responses of equally-observable, existing test programs.*

This paper is organized as follows: The next section discusses the proposed rapid test synthesis methodology for SBST programs. Section III discusses and analyzes the experimental results to assess the performance improvement of RSBST. Section IV concludes the paper and give further research directions.

II. RAPID TEST SYNTHESIS METHODOLOGY FOR SBST

As suggested in traditional SBST synthesis [9], [10], [11], [12], the automation of test programs for processors is performed with the help of an evolutionary approach that develops various test solutions using genetic operators. Each of these test solutions is evaluated in terms of fault coverage for the selection of fittest programs. In the proposed approach, an enhanced greedy-cover based test synthesis, termed as RSBST, is used for a faster cultivation of test programs with a sequence of instructions that could also trace and detect the hard-to-test faults.

In the traditional SBST, to trace the manufacturing and online faults, the faulty and non-faulty processor models are simulated. The simulation responses would comprise the contents of the observable locations (registers, memory locations, primary output etc.), the fault coverage, and the fault list. Further, the contents of the observable destinations are picked from the simulation responses and compared for fault detection. Intuitively, the fault coverage of a test program is completely associated with the observability of the processor modules. But the simulation of faulty processor models and the successive response collection are exceedingly time-consuming. In the proposed RSBST technique, we reuse these simulation responses for equally-observable test programs, to reduce the overall test generation time.

The overall approach of RSBST automation scheme for a processor is shown in Fig. 1. In this scheme, an evolutionary core develops a test solution of optimum fault coverage exploiting an initial population of test programs, represented as directed acyclic graphs (DAGs), and an instruction library. Formerly, all the test programs were evaluated using an external evaluator. To reduce this cost of fault evaluation, we have introduced an observability comparator, which compares and identifies the test programs with similar observability. This would effectively reduce the number of simulations as the simulation responses of the parent programs, stored in a database of simulation responses, could be reused for the offspring programs with equal observability. This database is updated with the simulation responses after each external simulation.

While evaluating each test solution, the observability comparator calculates the contents of the observable destinations of a test program. This content is contrasted with the observability values of the parent chromosome to check for the

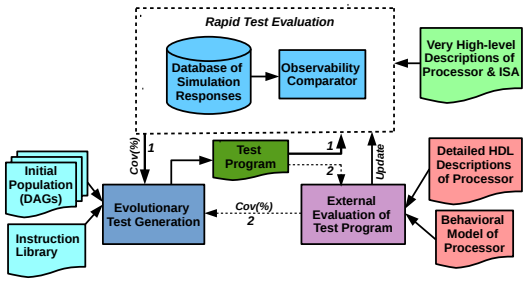

Fig. 1. **RSBST Automation Scheme**

sw $1, $2($3)
- Parameter Constant R1 R2
- Parameter Integer -128 127
- Parameter Constant R1 R2

Fig. 2. **A Sample Macro**

```
lui  Rs,Imm
lui  Rt,Imm
add  Rdest,Rs,Rt
sw   Rdest,offset(Rs)
```

Fig. 3. **A Sample Test Program**

scope of reusability of the test program. The observability values, fault coverage, and the fault list of the parent chromosomes are stored in a database of simulation responses.

The fault coverage of a test program, synthesized by the evolutionary core, is evaluated using either of the path 1 with bold lines or the path 2 with dashed lines, as shown in Fig. 1. Path 1 denotes the reuse of simulation responses using a rapid test evaluation method and path 2 denotes the regular, external evaluation of the test program. High-level processor descriptions and high-level simulations are used for rapid test evaluation whereas detailed HDL descriptions of the processor and time-consuming HDL simulations are used for the external evaluation of test programs.

A. *Basics of Test Program Synthesis*

The test program generation process uses the instruction library of the processor to constitute the test program. The instruction library comprises all the instructions of the ISA of the processor. Each entry of the instruction library is called a macro, which is an instruction with randomly selected operands. The instruction library may also contain multiple valid macros of the same instruction. A sample macro for store instruction is shown in Fig. 2. Here, the store instruction is encoded between two registers and a 16-bit constant. The first and third parameters ($1 and $3) denote the two registers, each of them chosen from R1 and R2. The second parameter ($2) is the 16-bit constant, which may have a value between -128 and 127. Individual test solutions, which are self-test programs, such as the assembly code in Fig. 3, are valid sequence of macros.

In most of the IC designs, the gate-level structural descriptions are not available to generate the conventional fault models. So, various high-level fault models are generated using behavioral level fault modeling, where different faults such as the *stuck-at-0* and *stuck-at-1* faults are injected into the Register Transfer Level (RTL) descriptions. For example, in the input stuck-at fault model, the input is stuck to 0 or 1 for a *bit* or *bit_vector* type signal and stuck to false or true

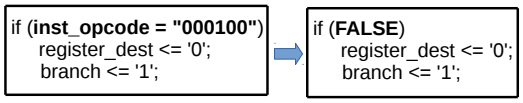

Fig. 4. **If-stuck-else Fault in Instruction Decoding in RTL**

```
            lui   R1,imm1
            lui   R2,imm2
Target :    addi  R1,R1,1
            bne   R1,R2,Target
            sw    R1,offset(R3)
```

Fig. 5. **Test Program with Branch Instructions**

for a Boolean type signal in the RTL statements. Presumably, most of the hardware faults are covered if the behavioral fault coverage is good enough because of the robust correlation (above 95%) between the behavioral faults and the physical faults as demonstrated in [15], [16], [17].

This approach has a higher level of abstraction compared to the gate-level fault modeling because the fault models are associated with the behavioral level descriptions [18], [19]. For example, an $if\ stuck\ else\ fault$ shown in Fig. 4 is the failure to execute an if condition, i.e., the if condition statement is replaced by $if(FALSE)then$.

The evolutionary core synthesizes the fragments of assembly code (macros) taken from the instruction library to generate self-test program solutions as shown in Fig. 5. This test program has loop-based branch instructions. The operands R1 and R2 are loaded with immediate values, where the value loaded in R1 is less than that of R2, and R1 is incremented until R1 and R2 have equal values. Finally, the content of R1 is stored in memory.

Test programs with branch instructions, as shown in Fig. 5, could be used to detect some of the instruction fetch and decode faults of the control components. For example, a behavioral $if\ stuck\ else\ fault$ in the description shown in Fig. 4 could be detected by the test program shown in Fig. 5. So, during the execution of a branch instruction, the if block will not be executed and the control signal $branch$ is not activated. This test program branches only if the $branch$ signal is activated. This indicates that R1 is not incremented up to R2 and finally, the content of R1 stored in the memory is observed to identify the fault.

B. Evolutionary Approach for Automation of Test Synthesis

The evolutionary core develops a genetic algorithm (GA) based automated test code synthesis procedure. In this method, a parent population of test program solutions is modified in each generation using mutation and selection operators. Mutation operator explores the search space for diverse test solutions and selection operator selects the fittest of them in order to generate the offsprings which eventually become the parent population for the next generation.

Evolutionary strategies (ES) are employed to automate the test synthesis using a directed acyclic graph (DAG) method. A DAG represents a test solution and the nodes of the DAG are the pointers of different macro elements of the instruction library as shown in Fig. 6. Epilogue and prologue nodes (I_0 and I_F) are the initial and final empty nodes. A $\mu + \lambda$ strategy of ES with an initial generation of μ DAG test solutions is carried out to develop efficient assembly programs that could validate the processor components. In every generation, new λ offsprings are created using the following mutation operators:

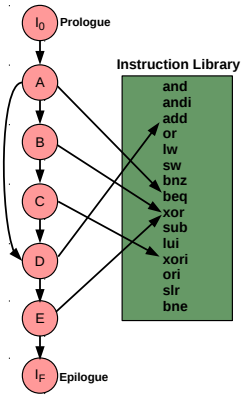

Fig. 6. **A DAG with Pointers to Instruction Library**

- Add node: A node is added to the DAG.
- Remove node: A node is removed from the DAG.
- Modify node: A node is modified in the DAG.

Among the $\mu + \lambda$ individuals of a generation, μ fittest offsprings are selected by the tournament selection operator of tournament size τ. In each generation, these individual test programs are evaluated using behavioral fault model to select the fittest population. An efficient selection of the objective function would help in carrying out the progressive development of the genetic population in consecutive generations and the test solutions evolve through the generations until an optimal solution is achieved.

From a testing point of view, the test program instructions which tests the functionality of the processor components gain high coverage as they could propagate nearly all faults of the functional components. So, the intermediate test programs that detect many of the arithmetic and logic functional faults in ALU are likely to survive to the later generations. But the test programs that detect certain non-functional hard-to-detect faults are less likely to survive if they do not validate any such functional modules. For example, some faults in the control unit are excited only if a beq (i.e., branch) instruction immediately comes after a $load$ instruction. Although some test programs detect such faults which are extreme corner cases, they may not survive to the subsequent generations. It is observed that these solutions are preserved only if the objective function of the evolutionary approach deals with the coverage of the freshly and exceptionally identified faults. So, A greedy-cover based objective function is used to synthesize test solutions with instruction sequences that could detect the uncovered faults.

C. Scheme for Faster Evaluation of Test Programs in RSBST

The test generation time is high for the greedy-cover based test synthesis because the test solution with highest fault coverage is not selected always. In fact, the test solutions with considerably lesser fault coverage could even be selected for the future generations if few hard-to-detect faults are detected. So, the convergence of the traditional SBST will require more generations of chromosomes and thus, is slower.

While the evolutionary process progresses, it is highly likely that the individual solutions have similarities in fault simulation results. If the test individuals have identical instruction sequences, the fault simulation results could be reused to reduce the test synthesis time. The initial μ

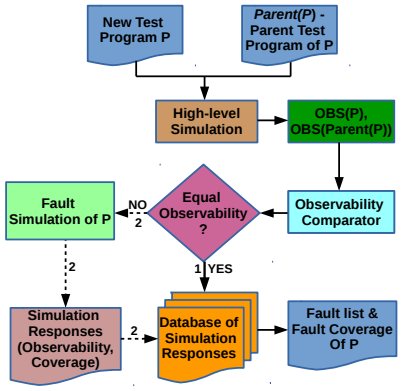

Fig. 7. **Test Program Evaluation in RSBST**

chromosomes of a generation could be reused from the previous generation, thus avoiding re-simulation. But the new λ chromosomes, which are cultivated using the μ individuals of the current generation, has to be dealt with a faster and high-level comparison of the states of the observable destinations.

After the behavioral fault simulation, faults are identified by comparing the contents of the observable locations of the processor. If the values stored in these observable locations following the execution of an offspring solution are same as that of its parent solution, the set of faults that they could identify will also be same; i.e., equally-observable test solutions will have equal fault coverage. In that case, a re-simulation of the offspring solution could be avoided by reusing the identified fault list and the fault coverage of the parent solution.

In the Fig. 7, the framework for the rapid test evaluation of Fig. 1, is elaborated. Here, a high-level test program simulation, which has low timing requirement, is used to get the contents of all observable locations. Further, the observability of a newly generated test program P and its parent are compared to identify the scope of reusability. If they are equal, the simulation responses of the parent solution could be taken and reused from the simulation library, where the simulation responses of the test solutions of the previous generations are stored. If they are not equal, P is to be simulated for the responses. Here, the path 1 with bold lines denotes the rapid test evaluation method and path 2 with dashed lines denotes the external evaluation of the test program. After the simulation, the simulation library is updated with the achieved responses. Finally, the fault list and the fault coverage are achieved from the responses, stored in the simulation library.

For example, the test program shown in Fig. 8(a) has equal observability as that of the test program in Fig. 8(b). These two programs have identical functionalities executed on the same registers and the memory locations; i.e., the registers $R1$ and $R2$ are assigned with X and $X+1$ respectively, and the value of $R2$ is subsequently stored in the memory location $R3{+}offset$. When these two programs are executed independently, the memory, registers, and primary output values are observed to be identical. As equally-observable test programs would have equal fault coverage, the simulation responses of the test program shown in Fig. 8(a) could be reused for the test program shown in Fig. 8(b).

The fault simulation results of the candidate test solutions of a specific generation could be reused for the selected

```
addi  R1,R0,X
addi  R2,R0,X+1
sw    R2,offset(R3)
```

(a) **Test Program 1**

```
addi  R1,R0,X
addi  R2,R1,1
sw    R2,offset(R3)
```

(b) **Test Program 2**

Fig. 8. **Equally-Observable Test Programs**

Algorithm 1: Reusability of Greedy Coverage Method in RSBST Approach

Input: i^{th} generation of solutions $S_i^1, S_i^2, \ldots, S_i^{\mu+\lambda}$.
CF_i be the list of all covered faults until the generation i.
OBS_i^j be the values of observable destinations when S_i^j is executed.

Output: Modified CF_{i+1} of the $(i+1)^{th}$ generation.

1 If $OBS_i^j = OBS_i^{parent(j)}$, then the faut coverage list of S_i^j = the fault coverage list of $S_i^{parent(j)}$;
2 Else, evaluate the fault list of S_i^j using fault simulation.
3 The fitness function of S_i^j is $|F_j|$, which is the cardinality of the set of its newly covered faults, where F_j = fault coverage list of $S_i^j - CF_i$;
4 If the individual S_i^j is selected, update CF_{i+1} with CF_i and the newly detected faults: $CF_{i+1} = CF_i \cup F_j$;

μ offsprings of the next generation since an offspring is selected and replicated directly from one of its parents. While cultivating the remaining λ individuals using the genetic operators, an observability based reusability method of RSBST as described in Algorithm 1 could be used to further reduce the test synthesis time.

Let OBS_i^j be the values of the observable locations after the execution of the test solution S_i^j, which is the j^{th} individual of the population in the i^{th} generation, on the processor. Let $S_i^{parent(j)}$ be the parent solution of S_i^j. The proposed method of reusability of the fault simulation observations are described in the Step 1 and Step 2 of Algorithm 1. If the values of the observable locations of S_i^j are equivalent to that of $S_i^{parent(j)}$, the fault coverage list of $S_i^{parent(j)}$ could be reused for S_i^j. Otherwise, the fault coverage of S_i^j has to be evaluated using behavioral fault simulation.

In this approach, a set CF_i refers to the list of all covered faults until the generation i, and F_j is the set of newly detected faults by S_i^j. In step 3, the objective function is defined as $|F_j|$, which is the cardinality of the set of the newly covered faults. This greedy approach tends to protect the chromosomes that detect the hard-to-detect faults through the generations. Finally, CF_{i+1} is created by merging CF_i and the list of fresh faults that are detected by the selected chromosomes in the i^{th} generation, as described in Step 4.

III. Experimental Results

For the experimental evaluation of our RSBST test synthesis, we have used the similar kind of setup, benchmark processors, and fault models considered in [14]. We have also used a 32-bit MIPS processor and a Leon3 processor model of SPARC V8 architecture with a 7-stage pipeline to be tested with the help of 10 behavioral fault representations shown in Table I. The MIPS processor is synthesized using 810 lines of VHDL code and the Leon3 processor has 5017 lines of

TABLE I

10 CATEGORICAL FORMS OF BEHAVIORAL FAULTS [18]

Fault Representation	Failure Type
Input stuck-at fault	Any primary input signal
Output stuck-at fault	Any primary output signal
If stuck then fault	If block is executed always
If stuck else fault	Else block is executed always
Elsif stuck then fault	Elsif block is executed always
Elsif stuck else fault	Elsif block is failed to get executed always
Assignment statement fault	Assignment of new values to a signal is failed always
Dead clause fault	A selected When clause in a case statement
Micro-operation fault	Micro-operations are failed always
Local stuck data fault	A signal object in a local expression

TABLE II

SPECIFICATIONS FOR THE PROPOSED AUTOMATED TEST SYNTHESIS

Specification	Value
Size of population (μ)	10
Number of offsprings (λ)	5
Number of generations	400
Selection methodology	Tournament selection
Size (τ) of the tournament	2
Steady-state threshold	40
Evolutionary methodology	ES(μ + λ) approach
Fault coverage evaluation method	Behavioral fault evaluation

VHDL code. The command-line options of ModelSim 10.5b simulator are used to execute the synthesized test programs on the faulty and non-faulty models of the processor. The fault simulation responses are extracted to evaluate the test programs and thereafter, the fittest solutions are selected. Generally, the synthesized test programs are of 40-60 lines of assembly code.

The functional and control components of the MIPS processor are tested in a software simulation environment of 270 fault models. To evaluate the test program, memory locations, 256 general purpose registers, and the primary outputs are extracted from the simulation responses using Python scripts and compared with the golden responses. The (μ+λ) evolutionary test synthesizer is developed using ANSI C with 3 mutation operators. A set of macros corresponding to the 9 instructions of the MIPS processor are used for developing the constituents of the instruction library.

For the MIPS processor, the enhancement in the development of the test synthesis using the proposed RSBST scheme is shown in Fig. 9(a). The average fault coverage of the conventional μGP scheme [10] with behavioral fault model achieves an adequate coverage (80-85%) after 50 generations, whereas the existing greedy-based GA [14] could only cover 85% of the faults after 75 generations. Eventually, 93.9% of the behavioral faults are detected by the μGP approach [10] and 96.3% of faults are detected by the greedy-based GA [14]. However, our RSBST test synthesis yields more than 85% of fault coverage before 50 generations and conclusively, carries out an improved coverage of 97.3%.

For the Leon3 processor, the progress in the achieved fault coverage using the RSBST scheme is shown in the Fig. 9(b). The conventional μGP approach [10] with the behavioral fault model, yields a fault coverage of 80-85% before 150 generations whereas the greedy-based GA [14] accomplishes above 80% coverage only after 200 generations. Finally, μGP approach [10] could detect 92.9% of the faults and the greedy-based GA [14] comes up with a fault coverage of 95.8%. However, the proposed RSBST test synthesis covers more than 85% of the possible faults before 100 generations and ends with a final fault coverage of 95.9%.

To identify the equally-observable test programs, the observability comparator is developed using ANSI C. This module stores the contents of the observable destinations to identify the redundant test programs which could be internally evaluated. In Table III, the fault coverage, time, and the amount of chromosome reuse are shown. The chromosome reuse refers to the percentage of chromosomes reused throughout the test synthesis.

The parameter values used for the proposed automated synthesis are shown in Table II. The size of the initial population (μ) is 10 and the number of offsprings to be generated in each generation (λ) is 5. The chromosomes of each generation are selected using a tournament selection operator where the tournament size (τ) is 2. The evolutionary core executes the test synthesis for 400 generations and terminates if there is no improvement for 40 generations, which is the steady-state threshold.

For the MIPS processor, the μGP approach [10] consumes 122 hours for the test synthesis and the greedy-based GA [14] takes 168 hours. The RSBST approach consumes only 90 hours, which is 46.4% faster than the greedy-based GA [14], and with a better coverage of 97.3% as shown in Table III. For the Leon3 processor, the μGP approach [10] takes 142 hours for the convergence of the test synthesis and the greedy-based GA [14] consumes 172 hours. The proposed RSBST approach consumes only 98 hours, which is 43% faster than the greedy-based GA [14], with an improved coverage of 96.2% as shown in Table IV.

The chromosome reusability is exploited to accelerate the convergence of the greedy-based GA [14]. For the μGP and the greedy-based GA [14], the simulation responses of the selected chromosomes ($\mu = 10$) of each generation are adopted and reused directly from the parent chromosomes, which saves 66.6% of test synthesis time. For our proposed RSBST approach, the offspring chromosomes ($\lambda = 5$) are substituted by the equally-observable parent chromosomes along with the reuse of the selected chromosomes ($\mu = 10$). Eventually, the overall reusable chromosomes would mount up to 82.1% for the MIPS processor and 80.8% for the Leon3 processor.

IV. CONCLUSIONS

In this work, a faster SBST synthesis of processor cores is employed using an accelerated greedy-based evolutionary method (RSBST), where the test programs that could detect the hard-to-detect faults are developed. From the results, we could conclude that using a more comprehensive fault model, our strategy develops test solutions that could detect 97.3% of the testable behavioral faults of the MIPS processor in 90 hours and 96.2% that of the Leon3 processor in 98 hours. This affirms a chromosome (test program) reuse of 82.1% for the MIPS processor and 80.8% for the Leon3 processor. A faster and profound test synthesis could be developed using the fragment-wise reusability of test programs. Even if the observability of 2 test programs are different, the identical, and data independent code fragments (chunks) could be extracted from these test programs and reused. Also, the fault equivalence techniques could be used for reducing the volume of simulations and the test generation time.

REFERENCES

[1] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded processor-based self-test*. Springer Science & Business Media, 2013, vol. 28, ISBN: 978-1-4020-2801-4.

[2] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001, pp. 92–96.

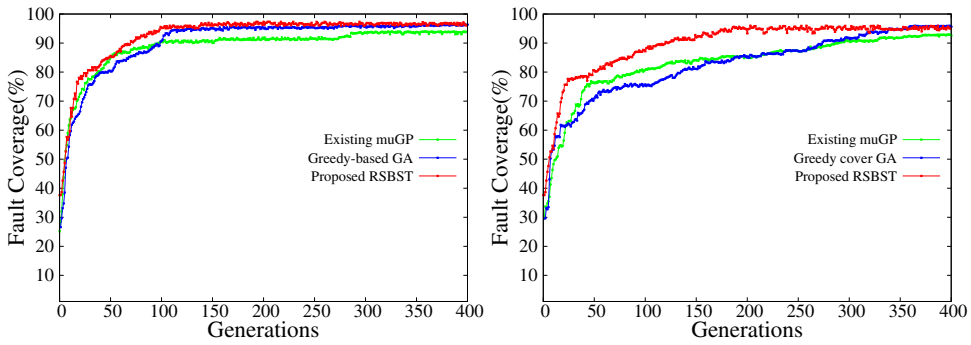

Fig. 9. **Average Fault Coverage of a MIPS Processor (Left Graph) and a Leon3 Processor (Right Graph) over 400 Generations using 1) μGP [10] with Behavioral Fault Model 2) Greedy-based GA [14] 3) Proposed RSBST**

TABLE III

MIPS PROCESSOR - ACHIEVED COVERAGE AND TIME OF THE 1) μGP [10] WITH BEHAVIORAL FAULT MODEL 2) GREEDY-BASED GA [14] 3) PROPOSED RSBST METHOD

Framework	Simulation Environment	Behavioral Fault Coverage	Test Synthesis Time	Chromosome Reuse	Remarks
μGP by G.Squillero [10]	Modelsim Version 5.7a	93.9%	122 Hours	66.6%	Lesser fault coverage but test synthesis is faster.
Greedy GA by V.M.Suriasrman et al. [14]	GHDL	96.3%	168 Hours	66.6%	Improved fault coverage but test synthesis consumes huge time
Proposed RSBST	Modelsim Version 10.5b	97.3%	90 Hours	82.1%	Improved fault coverage and faster test synthesis

TABLE IV

LEON3 PROCESSOR - ACHIEVED COVERAGE AND TIME OF THE 1) μGP [10] WITH BEHAVIORAL FAULT MODEL 2) GREEDY-BASED GA [14] 3) PROPOSED RSBST METHOD

Framework	Simulation Environment	Behavioral Fault Coverage	Test Synthesis Time	Chromosome Reuse	Remarks
μGP by G.Squillero [10]	Modelsim Version 5.7a	92.9%	142 Hours	66.6%	Lesser fault coverage but reasonable test synthesis time.
Greedy GA by V.M.Suriasrman et al. [14]	GHDL	95.8%	172 Hours	66.6%	Adequate fault coverage but longer test synthesis
Proposed RSBST	Modelsim Version 10.5b	96.2%	98 Hours	80.8%	Improved fault coverage and faster test synthesis

[3] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 548–553.

[4] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 369–380, 2001.

[5] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, April 2005.

[6] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[7] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 16, no. 11, pp. 1441–1453, 2008.

[8] M. De Carvalho, P. Bernardi, E. Sánchez, M. S. Reorda, and O. Ballan, "Increasing the fault coverage of processor devices during the operational phase functional test," *Journal of Electronic Testing*, vol. 30, no. 3, pp. 317–328, 2014.

[9] F. Corno, E. Sánchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 102–109, 2004.

[10] G. Squillero, "Microgpan evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 247–263, 2005.

[11] E. Sánchez, M. S. Reorda, and G. Squillero, "Efficient techniques for automatic verification-oriented test set optimization," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 93–109, 2006.

[12] F. Corno, E. Sánchez, and G. Squillero, "Evolving assembly programs: how games help microprocessor validation," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 695–706, 2005.

[13] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of sbst programs."

[14] V. M. Suryasarman, S. Biswas, and A. Sahu, "Automation of test program synthesis for processor post-silicon validation," *Journal of Electronic Testing*, vol. 34, no. 1, pp. 83–103, Feb 2018.

[15] A. Karputkin and J. Raik, "A synthesis-agnostic behavioral fault model for high gate-level fault coverage," in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1124–1127.

[16] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "An rt-level fault model with high gate level correlation," in *Proc. High-Level Design Validation and Test Workshop, IEEE International.*, 2000, pp. 3–8.

[17] M. Karunaratne, A. Sagahayroon, and S. Prodhuturi, "RTL fault modeling," in *Proc. 48th Midwest Symposium on Circuits and Systems.*, 2005, pp. 1717–1720.

[18] C.-I. H. Chen, "Behavioral test generation/fault simulation," *IEEE Potentials*, vol. 22, no. 1, pp. 27–32, 2003.

[19] R. Leveugle and K. Hadjiat, "Multi-level fault injections in vhdl descriptions: Alternative approaches and experiments," *Journal of Electronic Testing*, vol. 19, no. 5, pp. 559–575, Oct 2003.