Exploiting Evolutionary Computation in an Industrial Flow for the Development of Code-Optimized Microprocessor Test Programs

Riccardo Cantoro, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero
Politecnico di Torino
Corso Duca degli Abruzzi 24
10129 Torino, Italy
{riccardo.cantoro, marco.gaudesi, ernesto.sanchez, giovanni.squillero}@polito.it

ABSTRACT
It is well-known that faults affecting an electronic device may compromise its correct functionality, and industries have to check that their devices are fault-free before selling them. In case of a processor core, this task may be accomplished by running specially written "test" programs. In industrial embedded applications, however, shrinking such programs is strictly required. The hard problems of generating and code-optimizing test programs are tackled in this paper by exploiting an evolutionary approach.

Categories and Subject Descriptors
B.7.1 [**Integrated Circuits**]: Types and Design Styles – *microprocessors and microcomputers*; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.3.4 [**Programming Languages**]: Processors – *code generation.*

General Terms
Algorithms, Reliability.

Keywords
Software-Based Self-Test; Testing, Evolutionary Computation.

1. INTRODUCTION
Reliability of embedded systems is becoming a fundamental requirement, due to the fact that these devices are more and more often included in safety-critical applications. System integrity is checked by extra hardware, such as parity errors in memories, or instruments reporting hardware errors to the operating systems; alternatively, periodical self-test phases are performed in software by means of special test programs. Processor testing through the test programs execution is not a new strategy, in fact, its introduction dates back to 1980 [1], and it is usually referred to as Software-Based Self-Test (SBST). SBST techniques are based on the execution of test programs allocated in memory, collecting results obtained at the end of the run and comparing them with the expected signature. Since the periodic testing procedures need to coexist with the user applications, new standards, (i.e., ISO 26262 for automotive, and DO-254 for avionics) specify constraints regarding execution time, memory occupation, test frequency, and test coverage. A survey of the most important SBST techniques can be found in [2]. Evolutionary algorithms were already used in several works, aiming to the generation of high-quality test programs. Since 2000, the possibility to evolve assembly programs was exploited for microprocessor design validation [3] and microprocessor post-silicon validation [4]. Genetic Programming was also used for the automatic compaction of test programs [5][6].

This paper presents evolutionary approaches for generating microprocessor test programs, and for code-optimizing them. Moreover, the code-optimization process is applicable to existing test programs.

GECCO'15 Companion, July 11-15, 2015, Madrid, Spain
ACM 978-1-4503-3488-4/15/07.
http://dx.doi.org/10.1145/2739482.2764673

2. PROPOSED APPROACH
A test program is composed of assembly instructions compliant with the microprocessor's Instruction Set Architecture (ISA). The proposed approach for generating test programs exploits the evolutionary optimizer called μGP (MicroGP) [7]. The user is in charge of creating the *constraint library* for μGP, which defines the characteristics of compliant individuals.

The definition of the microprocessor's ISA is defined and bounded in the μGP constraint library; then, the framework using this is able to create syntactically correct test programs. These programs represent the population *individuals*, that are evaluated to get its *fitness* value. The evaluation process involves different steps: program compilation, simulation, and fault simulation. The fault simulation process consists on running the test program while injecting the processor faults, in order to observe differences in the processor behavior compared to the golden run. Every time the two behaviors are different, the fault is considered as detected. At the end of the fault simulation, the total amount of detected faults is used as fitness value.

During the first step of the evolution, a random population of individuals (i.e., test programs) is created; afterwards, applying the typical operators of mutation and recombination, new individuals are generated and evaluated; according to the fitness value, the best individuals are maintained in the population, while the others are discarded. At the end of the evolution, the best individual in the population is not yet optimized in terms of execution time or memory occupation, even if its fault coverage is as high as expected. A further process is then required to compact the program. Since it may be very hard to identify redundant instructions by means of code inspection, evolutionary computation may be suitable for this purpose.

The key-point of the compaction strategy is to identify the building blocks of the original test program, i.e., only one or a set of assembly instructions, and try to eliminate some parts inside the building blocks that are considered as redundant. Each building block is provided with a *weight* that indicates the

probability to be removed from the test program; this parameter allows the test engineer to identify critical instructions and to assign them low weights. According to the weights, the evolutionary framework (based on µGP) generates the individuals represented by a string of bits, where each bit indicates whether the building block of the original test program must be kept or not in the final programs. During the second optimization run, the fitness function is composed of two parameters: since the optimized test programs should maintain the same goodness in detecting faults, the first parameter is the *Saturated Fault Coverage* (FCs%), which saturates to the fault coverage value of the original test program, with a given tolerance (e.g., 99% of the original value). The second parameter instead, depends on the optimization goal: for example, the number of removed building blocks that may positively impact the memory occupation. During the evolution, an individual composed of all the building blocks is inserted within the initial random population. Thus, it was possible to start the process with one candidate solution already yielding the desired primary goal.

3. CASE STUDY

The proposed approaches were applied to the academic processor miniMIPS [8], which includes a 5-stage pipeline; it was synthesized with Synopsys Design Compiler targeting an in-house developed library; the gate-level net-list contained 16,303 gates and 1,967 flip-flops, corresponding to 115,508 stuck-at faults. Two modules of the microprocessor were used as case studies: the *decode unit* (7,502 stuck-at faults), which interprets the instruction opcodes, and the *forwarding and interlock unit* (3,738 stuck-at faults), a hardware solution to deal with data hazards. The fitness values required by the evolutionary optimizer µGP were calculated resorting to the logic simulator Mentor Modelsim and the fault simulator Synopsys TetraMAX. Experiments were executed on a workstation based on 2 Intel Xeon E5450 CPUs; on this machine, the evaluation of an individual required about 6 - 12 seconds; the experiments were parallelized on 8 cores. The same number of individuals (10 thousands) was generated and evaluated for all the experiments, each one lasting between 5 to 10 hours depending on the test programs length.

Table 1. Test programs code-optimization results.

		Decode Unit		Forwarding Unit	
		FC%	#Instr.	FC%	#Instr.
New generated programs	Before optim.	79.6	447	86.6	456
	After optim.	78.8	325	86.6	383
Existing programs	Before optim.	77.2	258	86.7	243
	After optim.	76.5	75	86.7	131

Two experiments were performed aiming at maximizing the fault coverage. All the assembly instructions of the miniMIPS were defined in the µGP's constraint library. As reported in Figure 1, the initial fault coverage on the decode unit was enhanced from a very low value (less than 40% of detected faults) up to the final value of 79.6%, while on the forwarding unit the optimization process brings up to the final value of 86.6%.

Later, two code-optimization experiments were performed on the test programs generated in the first phase, so that a subset of instructions was eliminated. The programs were fault-simulated by targeting the faults of the modules under consideration. It was possible to obtain a memory compaction of about 15% for the forwarding unit, without fault loss; better results (almost 30% of compaction) were obtained on the decode unit, but in this case 1% of fault coverage loss was tolerated. The effectiveness of the code-optimization was also tested on an existing suite of test programs, developed by the authors in previous works. The fault coverage values of the selected test programs were 77.2% on the decode unit and 86.7% on the forwarding unit. The code-optimized test program for the forwarding unit was half the size of the original one, without fault loss; a tolerance of 1% of fault coverage loss brought to a better compaction ratio (70%) on the decode unit (Table 1).

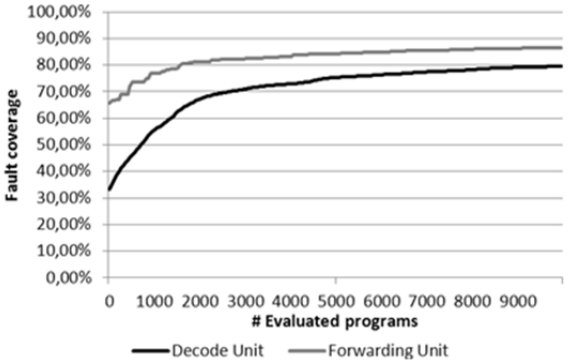

Figure 1. Fault coverage trends during the experiments.

4. CONCLUSIONS

The paper has tacked the complex problem of testing a microprocessor by means of a specialized suite of test programs. Since memory and timing issues are relevant in embedded systems, code-optimization of such programs represents a harder problem; it has been shown how evolutionary computation is suitable in this case. The proposed approach resorts on a framework based on the evolutionary optimizer µGP, for generating, maximizing the fault coverage, and compacting test programs. The same framework has been used also for optimize existing test programs, showing that a certain evidence of redundant instructions existed; the test programs have been compacted up to 70% with a negligible loss of fault coverage.

REFERENCES

[1] S.M. Thatte and J. A. Abraham, Test Generation for Microprocessors, *IEEE Trans. on Computers,* vol. 29, n. 6, pp. 429-441, 1980.

[2] M. Psarakis, D. Gizopoulos, E. Sanchez and M. S. Reorda, Microprocessor Software-Based Self-Testing, *IEEE DESIGN & TEST OF COMPUTERS*, vol. 27, n. 3, pp. 4-19, 2010.

[3] E. Sanchez and G. Squillero, Evolutionary Techniques Applied to Hardware Optimization Problems: Test and Verification of Advanced Processors, in *Studies on Computational Intelligence, Vol 66, Advances in Evolutionary Computing for System Design*, V. P. a. D. S. Lakhmi C. Jain, Ed., Springer, 2007, pp. 83-106.

[4] E. Sanchez, M. S. Reorda, G. Squillero and W. Lindsay, Automatic Test Programs Generation Driven by Internal Performance Counters, in *Microprocessor Test and Verification*, 2004.

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

[6] R. Cantoro, M. Gaudesi, E. Sanchez, P. Schiavone and G. Squillero, An Evolutionary Approach for Test Programs Compaction, *Latin American Test Symposium*, 2015.

[7] E. Sanchez, M. Schillaci, G. Squillero, Evolutionary Optimization: the µGP toolkit, Springer, 2011.

[8] http://opencores.org/project,minimips