Efficent Machine-Code
Test-Program Induction

F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

Abstract- *Technology advances allow integrating on a single chip entire system, including memories and peripherals. The test of these devices is becoming a major issue for manufacturing industries. This paper presents a methodology for inducing test-programs similar to genetic programming. However, it includes the ability to explicitly specify registers and resorts to directed acyclic graphs instead of trees. Moreover, it exploits a database containing the assembly-level semantic associated to each graph node. This approach is extremely efficient and versatile: candidate solutions are translated into source-code programs allowing millions of evaluations per second. The proposed approach is extremely versatile: the macro library allows easily changing target processor and environment. The approach was verified on three processors with different instruction sets, different formalisms and different conventions. A complete set of experiments on a test function are also reported for the SPARC processor.*

1 INTRODUCTION

Technology advances allow integrating on a single chip entire system, including memories and peripherals. This new kind of devices is called *Systems-On-a-Chip* (SOCs) and often includes complex cores, like microprocessors, MPEG encoders/decoders and other dedicated blocks (Figure 1). Such cores are usually provided by third parties and designers may include them into SOCs rather easily and quickly.

Before selling any electronic device, integrated-circuit producers need to check the correctness of the manufacturing process, and this test process accounts for a relevant percentage of the total production cost. Noticeably, checking SOC manufacturing is becoming a major issue, mainly because of the complexity of the embedded blocks and their limited accessibility.

This paper takes into account a specific type of cores, namely pipelined microprocessor (μP) cores. Popular solutions for core testing exhibit several drawbacks when adopted for μP cores: both test length and test application may unfeasibly extend, and test architectures based on modifications of the core are likely to degrade performance. Forcing the core to execute a given test program appears a suitable solution since it does not suffer of the above limitations [13]. However, the difficulty to generate such a test program for pipelined designs complicates its adoption.

Recently, Dey et al. proposed a deterministic method named DEFUSE [2] to generate test programs able to detect most manufacturing errors in the arithmetic and logic unit of a μP. The approach is very effective with combinationally testable parts, but shows limitations when hard-to-test sequential modules are addressed. Another approach, proposed by Batcher and Papachristou [1], is based on generating random sequences of instructions. However, this approach requires the insertion of additional hardware in the μP core under test. Recently, Sheen et al. proposed a technique where the processor itself generates test at run-time by self-modifying code [15]. Interestingly, Utamaphethai et al. showed a method for generating test programs for validating the branch prediction mechanism of the PowerPC604 [19]. The methodology generated very effective programs, but also exploits a deep knowledge of the processor and cannot be generally applied.

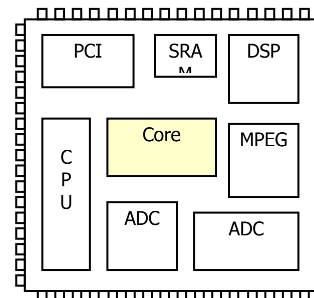

Figure 1: *System-On-a-Chip*

In [3] is proposed a partially automated method for generating test programs from gate-level descriptions. In [4] this method is extended to the case, in which only a high-level description of the processor is available. Both approaches were tuned for non-pipelined μP and relied on a library of chunks of instructions able to activate specific parts in the design. This library is carefully designed by programmers and exploited in a successive automatic phase, aiming at selecting the best macros and identifying their optimal parameters values.

Relying on the ability to evaluate the efficacy of a given test program both at RT- and gate-level, capacity shown in previous contributions, this paper presents a methodology for inducing test-programs for pipelined μP cores. The methodology starts from a genetic programming (GP) scheme. However, problem-specific requirements and characteristics entail an ad-hoc approach.

Next Section better details µP-core test programs, while assembly-level program induction is tackled in Section 3. Section 4 contains some experimental details and Section 5 concludes the paper.

2 MICRO-PROCESSOR CORE TEST PROGRAMS

Production test may consist in forcing the µP core to execute a test program while observing its outputs. This test program should allow discriminating fully working µP cores from malfunctioning ones. The effectiveness of a test program is its ability to expose internal failures, making them observable. For instance, an arithmetic operation is not observable; however, if the test program put the result on an output, it becomes possible to check its correctness from the outside.

Executing all possible operations with all possible operand values and exposing the results is not only inefficient, but also insufficient. Such a test program would be inapplicable for the required size, and may be unable to detect several defects. Indeed, the assembly-language programmer must be aware of several un-intuitive peculiarities.

In modern µP instructions are *pipelined*, this means that consecutively-executing instructions can have their execution overlapped in time. The details of instruction execution are arranged so that the CPU doesn't have to wait for one operation to finish before starting the next.

```
        RegA = 100;
        GOTO LABEL;
        RegA = 0;
        RegA = 10;
LABEL:
        RegA = RegA + 1;
```

Figure 2: *Pipeline Effects*

A striking peculiarity concerns program branches. When the CPU is ready to execute an instruction, it must first *fetch* that instruction (asking memory to retrieve the instruction at the appropriate address) and then *execute* that instruction (figuring out which operation is specified by that instruction and actually carrying it out). In modern pipelined architectures, at any given time, the CPU will be executing some instructions and, *at the same time*, it will be fetching the next instructions in the program. When a jump instruction is executed (for example, a call to a subroutine), the instruction appearing immediately after the call or jump in the code is already fetched in the pipeline. Thus, depending on µP architecture and implementation, the instruction following a branch may execute regardless of which way the branch goes.

For instance, after the pseudo-code in Figure 2, variable A holds 1 and not 101. Several hazards also arise from data dependencies, when consecutive instructions operate on the same data (in Figure 2, for instance, consecutive instructions read and modify the value of register *RegA*, resulting in a hazard).

Pipelined-processor assembly-language programmer must be constantly aware of all these problems while coding. Fortunately, modern compilers handle these peculiarities automatically and most of high-level programmers may ignore them, however, a test program able to test the pipeline must be written directly in machine code.

3 ASSEMBLY-LEVEL TEST-PROGRAM INDUCTION

Genetic Programming (GP) was defined as a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems [9]. GP addresses one of the more desired goals of computer science: creating, in an automated way, computer programs able to solve problems.

In GP context programs are usually represented as *tree*. A tree is a special kind of directed acyclic graph where there is only one path between any two nodes. Tree representations have been traditionally implemented in the *LISP* language as *S-expressions*. However, in recent year, several researchers proposed to modify this conventional representation.

Remarkably, in [6] the whole population was stored as a single directed acyclic graph, rather than as a forest of trees, leading to considerable savings of memory (structurally identical sub-trees are not duplicated) and computation (the value computed by each sub-tree for each fitness case can be cached). In [14] a significant speed-up was achieved extending the representation from trees to generic graphs and parallelizing the evolution process.

For the purpose of this works, however, it is more interesting to examine techniques based on the idea of *compiling* GP programs either into some lower level, more efficient, virtual-machine code or even into machine code.

Pioneering ideas date back to [7]. However, more recently in [11] the author suggested to directly evolving programs in machine-code form for completely removing the inefficiency in interpreting trees. More recently, a genome compiler has been proposed in [5], which transforms standard GP trees into machine code before evaluation. The possibilities offered by the Java virtual machine are also currently being explored [8], [10].

3.1 Directed-Acyclic-Graph Representation

Since the goal is to generate an assembly program, the canonical *S-expression* representation cannot be used. The tree representation was relaxed and the *flow* of the program is represented as a directed acyclic graph (DAG). The semantic of each node in the DAG consists in a pointer to a *macro* inside an *instruction library* and in parameter values. The macro represents a fragment of machine code, usually a single instruction, and parameters represent operand values and registers. It should be remarked that in any assembly language programmers may use several different registers, thus node semantic must include register specification.

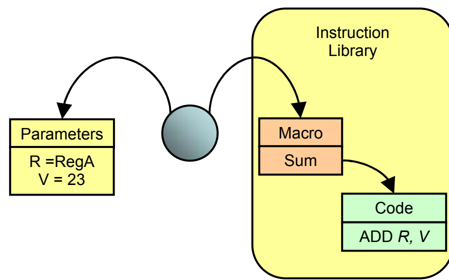

Figure 3: *Generic Node*

A DAG is always translated to a syntactically-correct assembly program. However, it not possible guaranteeing *a-priori* any semantic meaning. An induced program may perform operations on any register, and this exceptional freedom is essential to generate test programs.

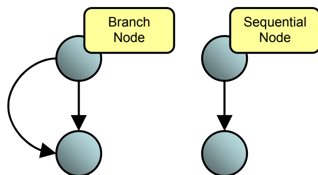

Figure 4: *Sequential and Branch Nodes*

Moreover, the library approach has been developed to enable the genetic core and the DAG structure to work with different assembly languages. Different processors not only implement different instruction sets, but also use different formalisms and conventions. Indeed, the method has been successfully tested with three different processors: an i8051, a CISC (complex instruction set computer) micro-controller developed by Intel; a DLX, an academic processor implementing a 5-stage pipeline [12] and a SPARC, the well known strongly-pipelined RISC (reduced instruction set computer) processor [18]. Additionally, exploiting abstract macros the program is able to infer data dependencies this ability may be used during assembly-level program generation to avoid inconsistencies.

The DAG is built with four kinds of node: a *prologue*, an *epilogue*, *sequential instructions*, and *conditional branches*.

The prologue and epilogue nodes are always present and represent required operations. They depend on the processor and on the operating environment. The prologue has no parent node and it is followed by a child node. Typically, it contains initialization code. The epilogue has no child nodes. These nodes may not be removed from the program, nor changed.

Sequential-instruction nodes (Figure 4, right) represent common operations, such as arithmetic or logic ones. They are characterized of having a single child. Their number of parameters may change, for instance, on certain processor some instructions use an implicit operand, thus the register specification s missing.

Conditional-branch nodes (Figure 4, left) are translated to assembly-level conditional-branch instructions. All common assembly languages implement some *jump-if-condition* mechanisms. Programmers must use two instructions to check a condition: a test and then a conditional branch.

```
    COMPARE A, B
    JUMP-IF-GREATER g_label
    ;
    ; These operations are executed if A <= B
    ;
g_label:
```

Figure 5: *Conditional Branch*

Figure 5 reports the assembly pseudo code for a simple *if-then* construct. It's remarkable that, since all these details are masked by compilers, they do not exist in high-level languages.

All conditional branches implemented in the target assembly languages are included in the macro library.

Unconditional branches are treated as sequential operations, since execution flow does not split.

DAG representation prohibits backward branches, either conditional or unconditional. This characteristic guarantees program termination, since no endless loop may be implemented. However, the effects of this small reduction in semantic power still need to be evaluated with regards to µP test-program diagnostic effectiveness. As a solution, library will probably include a macro containing a *safe* backward jump, but it will be represented as a sequential node at the DAG level.

3.2 Evolution

Test-program induction proceeds on two distinct levels: the syntactic level, where the DAG topology is modified; and the semantic level, where it is the macro referred by a DAG node that is modified. Both kinds of modifications are embedded in an evolutionary algorithm, implementing a $(\mu+\lambda)$ strategy.

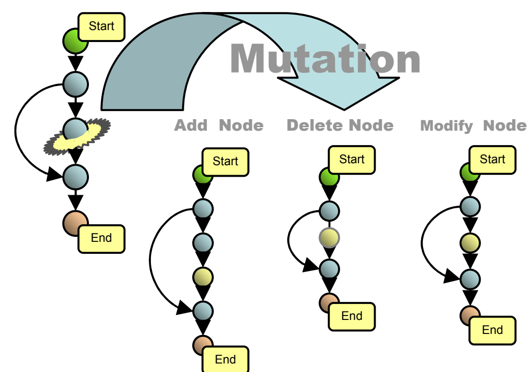

Figure 6: Evolutionary operators

In more details, evolution begins with a population of μ empty programs, i.e., programs composed only of prologue and epilogue. In each generation, λ new individuals are generated by mutating an existing individual, recombination is not exploited. Parents are selected using tournament selection with tournament size τ. Once the parent is selected, mutation operator is applied. Three mutation operators were implemented and are chosen with equal probability:

- **Add node (syntactic):** a new node is inserted after an existing node into the DAG. The new node can be either a sequential instruction or a conditional branch. In both cases, the macro refereed by the node is randomly chosen. If the inserted node is a conditional branch, a node implementing a random sequential instruction is also inserted after the node.
- **Remove node (syntactic):** an existing internal node (not the prologue or the epilogue) is removed from the DAG. If the removed node was the target of one or more branch, parents' edges are updated. On the other hand, removing a conditional branch requires no special attention.
- **Modify node (semantic):** macro parameters of an existing internal node are randomly changed. Parameters include both immediate values and register specifications. However, the macro implemented by the node may not be changed.

After creating new λ individuals, the best μ programs in the population of (μ+λ) are selected for surviving.

The evolution process iterates until population reaches a *steady state* condition, i.e., no improvements are recorded for a S_t generations.

3.3 Evaluation

While the final target of test-program generation is to detect production errors, for the purpose of this paper a more practical goal was adopted.

Macro library was composed to implement the SPARC V8/V9 assembly language. Prologue and epilogue was set to match the GNU C Compiler [17] convention. In this framework generated programs were simply assembled and linked to a stub interface.

Translating a DAG to the SPARC V8/V9 assembly language is rather a simple operation, since violating pipeline timing for data dependencies is legal, i.e., the μP does not hang, although the produced result may not be intuitive. On the other hand, sequences of consecutives branches, conditional or unconditional, are considered illegal. Thus, during translation DAG structure is analyzed, and opportune NOP (useless instructions) are added.

This mechanism allows to generate an executable program extremely fast and to execute it without any overhead.

4 EXPERIMENTAL EVALUATION

A prototype of the proposed approach was implemented in ANSI C and evaluated on a simple test problem. The goal of the experiment was to induce a program able to discriminate between even and odd values, and to set *all bits* in a processor register accordingly (see Figure 7).

```
goal_function(x)
{
  RegA : 32-bit internal register

  if(x is even) {
    RegA = 11111111111111111111111111111111
  } else {
    RegA = 00000000000000000000000000000000
  }
}
```

Figure 7: *Test Goal Function*

The *fitness* value of a solution measures its ability to solve the problem. In this context, it measures the percentage of correct bits in the result, i.e., the number of 1's in *RegA* when x is even, or the number of 0's when x is odd. To better evaluate each candidate solution, the corresponding programs were run 10,000 times with random parameter values, and fitness was averaged on the number of tries.

Noticeably, since the candidate-solution programs are compiled and linked, running 10,000 evaluations require no special effort. On a SPARC Ultra, translating a solution to an assembly program, compiling and linking it requires, on the average, 0.15 seconds of CPU time. Running the program for 10,000 trials requires, on the average, 0.01 seconds of CPU time. The program is therefore able to evaluate each single solution on 1,000,000 different runs in about a single second.

To assess the feasibility of the approach, the test program was tackled with different values of μ, λ, τ. Since the goal of the evaluation was not to solve the test problem, but to evaluate the influence of parameters, steady-state threshold S_t was set to a very low value (100).

Table 1 reports attained results. For each set of μ, λ and τ it is reported the fitness attained by the best individual [Fitness] and the characteristics of the corresponding program, in term of sequential nodes [Sequential] and branch nodes [Branch].

It can be noted that, whenever steady-state threshold was extremely low, several times the optimal program was induced, i.e., a program able to *always* set correctly *all* bits of output register. Fitness around 50% means random drifting, while a higher value indicates that the correct program was actually being induced.

μ	λ	τ	Sequential	Branch	Fitness
1	1	1	7	4	53.44%
5	1	1	6	6	53.46%
10	1	1	3	2	52.40%
50	1	1	1	0	52.35%
1	5	1	4	1	53.54%
5	5	1	7	3	53.74%
10	5	1	10	2	65.69%
50	5	1	6	3	53.53%
1	10	1	16	8	53.85%
5	10	1	9	4	53.52%
10	10	1	33	19	75.58%
50	10	1	18	6	96.94%
1	50	1	16	8	100.00%
5	50	1	10	3	53.97%
10	50	1	14	8	74.43%
50	50	1	37	14	98.50%
5	1	2	5	1	52.53%
10	1	2	4	1	53.38%
5	5	2	7	1	53.47%
10	5	2	13	6	96.94%
50	5	2	5	0	53.52%
5	10	2	7	3	53.55%
10	10	2	13	4	53.82%
50	10	2	10	6	53.94%
5	50	2	16	6	100.00%
10	50	2	15	4	95.32%
50	50	2	19	7	100.00%
5	1	5	3	1	52.29%
10	1	5	2	1	52.25%
50	1	5	1	1	52.34%
5	5	5	8	3	53.28%
10	5	5	20	8	60.89%
50	5	5	6	5	53.44%
5	10	5	22	6	84.95%
10	10	5	22	8	57.62%
50	10	5	14	5	53.54%
5	50	5	15	7	100.00%
10	50	5	17	6	100.00%
50	50	5	10	4	53.94%

Table 1: *Test Program Evaluation*

5 CONCLUSIONS

An evolutionary algorithm for inducing programs was presented. First, the method includes the ability to explicitly specify registers either as operands or targets. Furthermore, it relaxes the usual tree-based representation, resorting to DAG. Finally, it couples a standard GP approach with a database containing the assembly-level semantic associated to DAG nodes.

Exploiting DAG and library the proposed approach is extremely efficient and versatile. Candidate solutions are translated into source-code programs that can be assembled and linked using standard compilers. Such executable programs allow a fast evaluation: millions of runs may be performed in just a second.

Moreover, the approach is versatile. The macro library allows changing the target μP and environment easily. The approach was verified on three processors with different instruction sets, different formalisms and different conventions.

This method can be seen as a general enhancement of standard GP, however it was specifically devised for inducing test-program for μP cores, a critical area in modern industry. Experimental results show its efficacy and feasibility, thus authors are now actively applying it in the electronic area.

References

[1] K. Batcher, C. Papachristou, "Instruction Randomization Self Test For Processor Cores", *Proceedings IEEE VLSI Test Symposium*, 1999, pp. 34-40

[2] L. Chen, S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", *IEEE VLSI Test Symposium*, 2000, pp. 255-262

[3] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores," *IEEE Conference on Design & Test in Europe*, March 14-16, 2001

[4] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Effective Techniques for High-Level ATPG," *IEEE Asian Test Symposium*, November 20-21, 2001

[5] A. Fukunaga, A. Stechert, D. Mutz, "A genome compiler for high performance genetic programming", Genetic Programming 1998: Proceedings of the 3rd Annual Conference, 1998, pp. 86-94

[6] S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs", *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, 1994, pp 154-159

[7] R. M. Friedberg, "A Learning Machine: Part {I)", *IBM Journal of Research and Development*, 1958, vol. 2, n. 1, pp 2-13

[8] S. Klahold, S. Frank, R. E. Keller, W. Banzhaf, "Exploring the possibilites and restrictions of genetic programming in Java bytecode", *Late Breaking Papers at the Genetic Programming 1998 Conference*, 1998

[9] J. R. Koza, "Genetic programming", *Encyclopedia of Computer Science and Technology*, vol. 39, Marcel-Dekker, 1998, pp. 29-43

[10] E. Lukschandl, M. Holmlund, E. Moden, "Automatic evolution of Java bytecode: First experience with the Java virtual machine," Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming, 1998, pp 14-16

[11] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," *Advances in Genetic Programming*, 1994, pp. 311-331

[12] J.L. Hennessy, D.A. Patterson, “DLX architecture”, in *Computer Architecture, a Quantitative Approach*, Morgan Kaufmann Publishers.

[13] C.A. Papachristou, F. Martin, M. Nourani, “Microprocessor Based Testing for Core-Based System on Chip”, *ACM/IEEE Design Automation Conf*erence, 1999, pp. 586-591

[14] R. Poli, “Evolution of graph-like programs with parallel distributed genetic programming”, *Genetic Algorithms: Proceedings of the 7th International Conference*, 1997, pp 346-353

[15] J. Shen, J. Abraham, D. Baker, T. Hurson, M. Kinkade, “Functional verification of the Equator MAP1000 microprocessor,” *Proceedings 36th Design Automation Conference*, 1999, pp. 169 -174

[16] A. Samuel, “Some studies in machine learning using the game of checkers”, *IBM Journal of Research and Development*, 3(3), 1959, pp. 210–229

[17] R. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Cambridge MA, 02139, 1989

[18] SPARC International, *The SPARC Architecture Manual (Version 8)*. Prentice Hall, NJ, 1992

[19] N. Utamaphethai, R.D. Blanton and J.P. Shen, “Superscalar Processor Validation at the Microarchitecture Level,” *12th IEEE International Conference on VLSI Design*, 1999, pp. 300-305