Genetic Programming and Evolvable Machines, 6:3, 247–263, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s10710-005-2985-x

# MicroGP—An Evolutionary Assembly Program Generator

GIOVANNI SQUILLERO giovanni.squillero@polito.it
*Politecnico di Torino – DAUIN, Corso Duca degli Abruzzi 24, 10129, Torino–Italy*

*Submitted May 30, 2003; Revised October 27, 2004*

**Published online:** 12 August 2005

**Communicated by:** Una-May O'Reilly

**Abstract.** This paper describes $\mu$GP, an evolutionary approach for generating assembly programs tuned for a specific microprocessor. The approach is based on three clearly separated blocks: an evolutionary core, an instruction library and an external evaluator. The evolutionary core conducts adaptive population-based search. The instruction library is used to map individuals to valid assembly language programs. The external evaluator simulates the assembly program, providing the necessary feedback to the evolutionary core. $\mu$GP has some distinctive features that allow its use in specific contexts. This paper focuses on one such context: test program generation for design validation of microprocessors. Reported results show $\mu$GP being used to validate a complex 5-stage pipelined microprocessor. Its induced test programs outperform an exhaustive functional test and an instruction randomizer, showing that engineers are able to automatically obtain high-quality test programs.

**Keywords:** evolutionary algorithms, micro-processors, assembly programs generation

## 1. Background

While Alan Turing was probably the first to suggest that the computing machine could manipulate itself just as well as any other data, the pioneering attempt of generating programs in an automatic way date back to 1958. Friedberg defined an assembly language and a set of random variations, then, he tested the randomly mutated code against a given problem trying to cultivate a solution [1, 2]. Indeed, the approach showed limited success. Evaluating programs to measure their performance was a very slow process, and the required computational effort was probably the most limiting factor.

Nowadays, with the increased computational power brought by modern computers, evolutionary techniques are routinely exploited in many combinatorial optimization problems. Several different paradigms have been proposed, with different representations and operators, or simulating the evolution process at different levels. However, there is quite a significant difference between solving a numerical problem and generating a functioning, and hopefully effective, program. Currently, the best known approach tackling the evolution of programs is Genetic Programming (GP). Koza defines it as "a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems. Genetic programming is based on the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as *crossover* (*sexual recombination*) and *mutation*" [3].

In the GP framework, however, the term "program" usually stands for "expression." Such expressions may be represented as *trees* (directed acyclic graphs where there is only one path between any two nodes) and are traditionally implemented in the LISP language as *S-expressions* (*symbolic expressions*) [4, 5]. Evaluating expressions with an interpreter rather than a compiler is a source of inefficiency in GP, and the community tried to overcome this drawback. Different techniques based on the idea of compiling GP programs either to some lower level, more efficient, virtual-machine code or even into machine code were reported. A genome compiler has been proposed in [6], which transforms standard GP trees into machine code before evaluation. In recent years several researchers proposed modifications to this representation. In [7] the whole population was stored as a single directed acyclic graph, rather than as a forest of trees, considerably saving 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 [8] a significant speed-up was achieved extending the representation from trees to generic graphs and parallelizing the evolution process.

More radically, some authors suggested evolving the programs directly in a machine-code form to completely remove the inefficiency of interpreting trees [9], or directly manipulating machine code storing the program as linear strings [10]. The latter approach, called CGPS (compiling genetic programming system), was first exploited on the SPARC, a RISC (reduced instruction set computer) microprocessor. Then it was renamed to AIM-GP (automatic induction of machine code—genetic programming) and extended to CISC (complex instruction set computer) microprocessors [11]. The possibilities offered by the Java virtual machine have also been explored [12].

Despite the important emphasis on evolving Turing complete programs, most of the work has been performed to merely speed-up the evaluation of individuals. Differently, this paper describes an evolutionary approach for generating Turing-complete programs tweaked for a target microprocessor called MicroGP ($\mu$GP). The main goal of $\mu$GP is not to optimize the evaluation of the individuals, but rather to generate syntactically correct assembly programs of variable size that *fully exploit* the assembly syntax, including the different addressing modes, the instruction set asymmetries, subroutines and interrupt calls. Although the resulting method can be exploited in different context, it was initially designed for test-program generation for microprocessors.

This paper describes the $\mu$GP framework, reporting the new experimental evaluation on a test-program generation problem for design validation of a pipelined microprocessor. Remarkably, the approach enables getting meaningful results on such a complex device in reasonable CPU time. All new results are compared with previous ones.

The next section describes the approach; Section 3 reports an experimental evaluation; Section 4 concludes the paper.

## 2. Automatic test program generation

A test program is an assembly program devised to extract information that reveals the correctness or valid operation of the machine that executes it, rather than calculating a function or performing a task. Test programs may be used to validate the correctness of a microprocessor design or to check the correct functionality of a device after production. In the former case, the goal of test-program generation is to achieve maximal code coverage

and a collection of test programs are used in a process similar to debugging. In the latter, the goal is to generate a program capable of exposing internal malfunctions via observable outputs. The term *test program* is commonly used whether the goal is *validation* or *testing*.

An early attempt to evolve assembly programs for post production test was presented in [13]. The approach relied on a library of fragments of code carefully and skillfully written by hand, called macros. The optimal sequence of macros was heuristically determined, and then a genetic algorithm optimized their parameters. The approach is quite effective, but hardly scalable. An Intel i8051, a very simple microprocessor, required 213 macros; and the macro list was carefully compiled by an experienced engineer in two working days.

The embryonic idea of a general framework for generating assembly language program starting from simple instructions and not macros, was presented in [14]. It represented programs as directed acyclic graphs, and implemented a straightforward $(\mu + \lambda)$ evolution. Remarkably, the overall structure resembled the *linear graph GP*, developed independently, and concurrently, in [15]. Shortly after, the naïve approach was successfully used in [16] for validating a microprocessor design, and provided effective results in an almost completely automatic way.

Subsequently, $\mu$GP was improved, adding recombination and self-adaptation [17] (called *auto-adaptation* in the paper) and the evaluation procedure was expanded. The enhanced methodology was used on different problems [18, 19] and the results of [13] were easily outperformed. Paraphrasing Samuel [20], *$\mu$GP was eventually able to do what was needed to be done, without being told exactly how to do it*. This is true at least in the very specialized context of microprocessor validation.

In 2003, $\mu$GP was rewritten from scratch. It has now the ability to exploit single instructions as in [14], complex macros as [13] or a combination of them. The internal representation has been completely disconnected from assembly specification. The number of different nodes has been reduced, and their expressiveness increased. The directed acyclic graph representation has been transformed to handle generic directed cyclic graphs. A new abstraction level has been added, enabling support for subroutines and software interrupts.

## 3. The $\mu$GP architecture

The $\mu$GP architecture is biased by the goal of being broadly applicable. It is composed of three clearly separated blocks (Figure 1): an *evolutionary core*, an *instruction library* and an *external evaluator*. The *evolutionary core* adapts and generates a population of individuals. It uses self-adaptation mechanisms, dynamic operator probabilities, dynamic operator strength, and variable population size. The *instruction library* is used to map individuals to valid assembly language programs. It contains either a highly concise description of the assembly syntax, or more complex, parametric fragments of code. Finally, the *external evaluator* simulates the assembly program, providing the necessary feedback to the evolutionary core.

The next subsections detail the architecture.

### *3.1. Instruction library*

An assembly program may be seen as composed of different *sections*. Those sections are pieces of code each with a kind of syntactic format. For instance, the main body of the

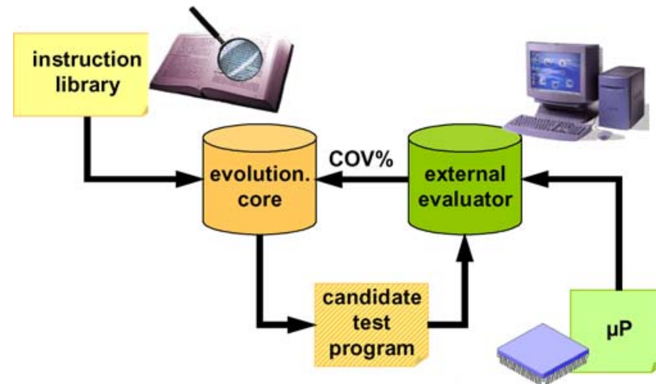


*Figure 1.* $\mu$GP is based on three clearly separated blocks: an evolutionary core which conducts population-based search, an instruction library which maps individuals to valid assembly programs and an external evaluator that simulates the assembly program, providing the necessary feedback to the evolutionary core in the form of a coverage percentage.

```
.macro
      add $1, $2
.parameter constant R1 R2
.parameter integer -128 127
.endmacro
```

*Figure 2.* A simple macro.

program is a section. Usually, a few starting lines are fixed and required by the operating system and by the environment. Conversely, all subroutines may be considered a different section. Some keywords, like "**proc**" always appear at the beginning of definitions, while some others, like "**endp**," always conclude them. We use this concept of sections to group code of the same syntactic format within an assembly source program

The *instruction library* is designed to easily be set up and understood by a human operator. It allows the operator to specify most syntactic details of the assembly language, like the format of labels, subroutines and comments. The operator also enumerates each different section of a program and defines a set of macros, i.e., fragments of code of arbitrary length, with an arbitrary number of parameters represented as "`$n`." A very simple macro is shown in Figure 2. It encodes a single instruction, an "`add`" instruction, between a register and an 8-bit constant. The register is the first parameter (`$1`) and may be either R1 or R2. The 8-bit constant is the second parameter (`$2`) and takes values between –128 and 127.

A sensible use of the instruction library allows enumerating all valid instructions in a very compact way. Parameters may be used to encode operands, instructions and addressing modes, as in Figure 3.

Formally, the instruction library supports seven different types of parameters:

```
.macro
      $1 $2, $3
.parameter constant ADD SUB OR AND XOR
.parameter constant R1 R2
.parameter constant @R1 @R2 #R1 #R2
.endmacro
```

*Figure 3.* A macro describing some arithmetic/logic instructions and different addressing modes.

- *Integer:* represents a numeric value and may be used as immediate value, offset or any other numeric data supported by the assembly language. The valid range must be specified.
- *Constant:* represents a string inside a predefined set. They are usually used for specifying a register together with a specific addressing mode. For instance, the third parameter in Figure 3 specifies either R1 or R2, and two possible addressing modes, denoted with "@" and "#."
- *Inner forward label:* a reference to a subsequent macro in the same section.
- *Inner backward label:* a reference to a preceding node in the same section. Such labels may produce endless loops and non-terminating programs.
- *Inner generic label:* a reference to a generic node in the same section, either preceding or subsequent. Such labels may produce endless loops and non-terminating programs.
- *Outer label:* a reference to the first macro in a different section. Typically the entry point of a subroutine.
- *Unique tag:* a string guaranteed to be unique during program evolution. It can be used to define local labels inside the macro. For example, see the complex macro shown in Figure 4.

The first and the last macro of a section are called respectively *prologue* and *epilogue* and are kept distinct from the other macros.

```
.macro
      mov $1, $2          ; inizialize counter R1 or R4
$3:   call $4             ; call sub in other frame
      $5 $1               ; increment (or decrement) counter
      jz $3               ; loop until zero
      jmp $6              ; then jump to a previous location
.parameter constant R1 R4
.parameter integer -128 127
.parameter unique_tag
.parameter outer_label SubsFrame
.parameter constant INC DEC
.parameter inner_backward_label
.endmacro
```

*Figure 4.* A more complex macro.

Finally, most of the macros are used to specify single instructions. However, more complex structures can be used, and this opportunity may be useful in some context.

The instruction library also allows specifying most syntactic details of the assembly language, like the format of label and subroutines, the format of comments, etc.

### *3.2. Representation*

Individuals are represented as directed graphs, and each graph may be seen as a collection of loosely linked sub-graphs, each one associated with a specific section of the instruction library. Each node has an ancestor and a successor, except two special nodes: the *head* and the *tail*. The head has no ancestor, while the tail has no successor. Nodes that are neither tails nor heads are called *internal*. The presence of head and tail is required in all sub-graphs, even if they may have no internal nodes. The successor of node $n$ is denoted with $succ(n)$.

Each node implements a macro of the instruction library. The first node encodes the prologue of a section, the last the epilogue. All internal nodes implement random macros. Each node contains a link to its successor and encodes all parameters required by the macro, thus a node may be connected to any number of other nodes, according to the number of *label* parameters specified in the macro. An *inner forward label*, an *inner backward label*, or an *inner generic label* specifies an edge connecting a node to an internal node of the same sub-graph. An *outer label* specifies an edge connecting a node to the head of a different sub-graph.

Figure 5 details a node implementing a 3-parameter macro in a program. Each node is linked to its successor by a vertical arrow. The third parameter of the macro is a label and its value is represented by an arrow pointing to a previous node.

### *3.3. Evolutionary core*

The $\mu$GP evolutionary core exploits a generational strategy. A population of $\mu$ individuals is stored and in each generation $\lambda$ genetic operators are applied. Operators are chosen according to their activation probabilities. Since each genetic operator produces a variable offspring, the number of individuals at the end of each generation is not fixed. After offspring generation the population is sorted, the best $\mu$ individuals are selected for survival and transferred to the next generation. The evolutionary core implements both a crossover (recombination) operator and different mutation operators. All activation probabilities are endogenous parameters automatically self-adapted by the algorithm.

The evolution process iterates until population reaches a *steady state* condition, i.e., no improvements are recorded for a number $S_t$ of generations, or a maximum number of generations have been evaluated.

Parents are selected using tournament selection with tournament size $\tau$.

***3.3.1. Crossover.*** The crossover operator recombines the genetic material of two different parents, generating two new individuals. The strongly connected nature of the graphs prevents the straightforward usage of canonical approaches, such as the exchanging sub-trees crossover or even the structure-preserving crossover. On the other hand, a simplistic crossover such as uniform crossover would annihilate the structure of the individuals, resulting in an almost-complete random rebuilding of the whole genetic material.

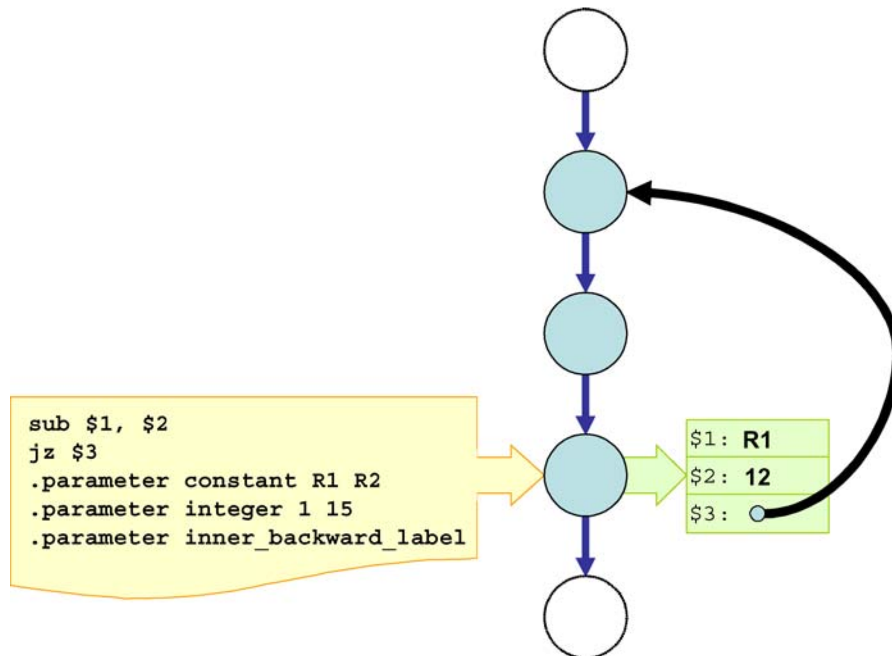


*Figure 5.* A program with one node implementing a 3-parameter macro.

The crossover operator developed in the evolutionary core is able to recombine genetic materials, while preserving the structure and the successful strategies evolved in the parents. Crossover is performed by swapping two compatible *cores*. A core $\kappa$ is defined as a strict subset of nodes where:

- no node contains, as a parameter, an edge to a node outside the core;
- no node outside the core contains, as a parameter, an edge to a node inside the core.
- all nodes are consecutive: $n_i \in \kappa \wedge succ(succ(n_i)) \in \kappa \Rightarrow succ(n_i) \in \kappa$.

A core may span multiple sections. If a node in the core contains an edge to the head of a different sub-graph (i.e., an *outer label*), all that sub-graph is included in the core.

Two cores are compatible if they include the same sections. A core including the head of a section is compatible only with cores that include the head of the same section. Similarly, a core including the tail of a section is compatible only with cores that include the tail of the same section. These rules guarantee that cores may be exchanged maintaining graphs consistency.

To select a core, first a random sequence of nodes is chosen from a sub-graph. Then, the core is iteratively extended to fulfill all requirements. Once two cores are selected in the two parents, if they are compatible they are swapped, generating two new individuals.

***3.3.2. Mutations.*** The $\mu$GP evolutionary core takes advantage of four different mutation operators. In addition, to control the evolution process further, $\mu$GP exploits the concept

of *mutation strength*. The mutation strength $S \in [0, 1[$ measures how deeply a genetic mutation transforms the parent (when $S \to 1$ the diversity increases). Since most mutation operators perform an elementary modification, the number of consecutive modifications may be increased in order to tune the overall mutation strength. The number of consecutive elementary operations performed is probabilistically determined according to the value of $S$. Each time an elementary step is performed, there is a probability $S$ to apply a subsequent mutation on the same parent. Thus, the expected number of consecutive mutations may be calculated as $N = \frac{1}{1-S}$.

Mutation strength is an endogenous parameter self-adapted during evolution. In most of the experiments, $\mu$GP exploits large mutations at the beginning of the evolution process, while it reduces their strength in the end. Thus, it autonomously molds the search process from the *exploration* of the space to the *exploitation* of the results.

Four different mutation operators have been included in the current version of the evolutionary core:

- *Add*: New nodes are added to the target graph, in a randomly chosen sub-graph. New nodes encode random macros and all parameters are randomly chosen. Generating an *outer label* may require creating a new sub-graph as well.
- *Remove*: Nodes are deleted from the target graph, from a randomly chosen sub-graph. If the removal of a node creates a disconnected sub-graph, the disconnected sub-graph is entirely deleted.
- *Add/remove*: Either a new node is added or an old one is deleted, as in the two previous mutations. This mutation quickly increases diversity and is mostly used in the earliest phases of the evolution.
- *Change*: All parameters in some nodes are randomly modified. Removing a node containing *outer label* (a reference to a macro in a different section) may create a disconnected sub-graph, causing its deletion. Conversely, creating a *outer label* may require creating a new sub-graph.

### 3.4. Self adaptation

The $\mu$GP evolutionary core internally tunes the activation probabilities of all genetic operators and the mutation strength. By modifying these parameters, the algorithm is able to shape the search process significantly improving its performance.

Activation probabilities are initially set to a common value: $p_{xover} = p_{mut/A} = p_{mut/R} = p_{mut/AR} = p_{mut/C} = \frac{1}{5}$. During evolution, probability values are updated according to the operator results: let $N^{OP}$ be the number of activations of the operator $OP$ in the last generation; let $S^{OP}$ be the number of successful invocations of the genetic operator $OP$ in the last generation, i.e., the number of activations where the resulting individual attained a fitness value higher than its parents'. At the end of each generation, the new value of the activation probability for the operator $OP$ is calculated as $p_{OP}^{new} = \alpha \cdot p_{OP} + (1 - \alpha) \cdot \frac{S^{OP}}{N^{OP}}$. Activation probabilities are then normalized and forced to avoid values below .01 and over 0.9. If $N^{OP} = 0$, then activation probability is slowly pushed towards its initial value. The coefficient $\alpha$ introduces an inertia to avoid unexpected abrupt changes.

Experimental data shows that in the beginning of the evolution, the evolutionary core takes most advantage from the *Add* and *Add/Remove* mutations, making individuals grow

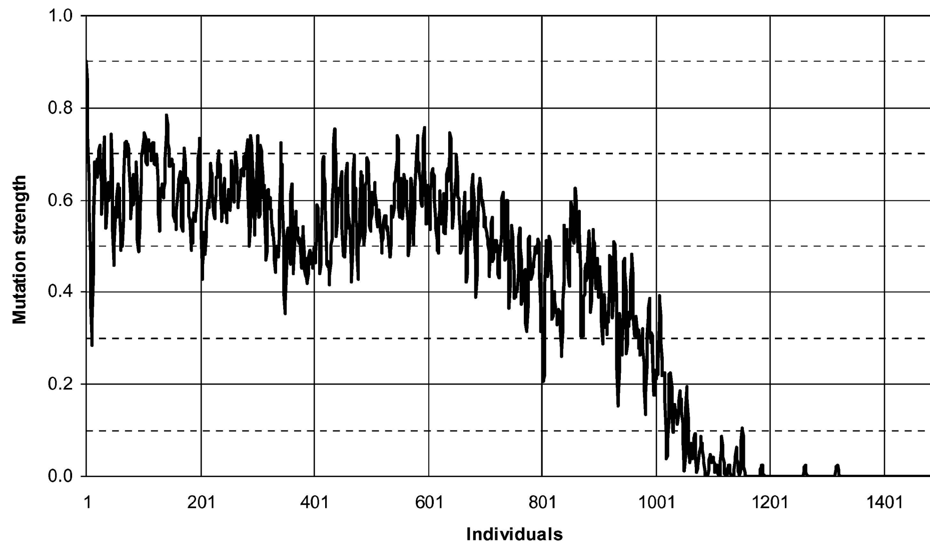


*Figure 6.* Mutation strength self adapting over one typical run.

quickly. Later, in the end of the process, those operands are overridden by the *Sub* and *Change* mutations.

During evolution, mutation strength is also varied according to the evolution results. Intuitively, in the beginning it is better to adopt a high value, allowing the offspring to strongly differ from parents. On the other hand, at the end of the search process, it is preferable to reduce diversity, allowing only for small mutations.

Let $I_H$ be the number of newly created individuals attaining a fitness value higher than their parents over the last $H$ generations. At the end of each generation, the new mutation strength $S$ is calculated as $S^{new} = \alpha \cdot S + (1 - \alpha) \cdot \frac{I_H}{H \cdot \lambda}$. Then $S$ is saturated to 0.9. Initially, the maximum value is adopted ($S = 0.9$), considering all $H(\mu + \lambda)$ individuals as improvements. The coefficient $\alpha$ introduces an inertia. Figure 6 shows the mutation strength during a typical run of $\mu$GP.

### 3.5. External evaluator

The external evaluator is a key element in the $\mu$GP architecture. It performs the phenotypic evaluation of each individual and sends back a fitness score to the evolutionary core. The versatility of $\mu$GP strongly depends on it.

Depending on the evaluation setup, the assembly program may be assembled, linked and simply executed to evaluate it. Or its execution may be *simulated* against a hardware model of a target microprocessor. During this simulated execution, the behavior of the hardware model is observed and the results of the simulation are eventually used as a feedback to the evolutionary core. Thus, the *result* produced by the program is not directly considered, but the program is evaluated with respect to *how it is executed* by the target hardware model.

### 3.6. Implementation

The evolutionary core of $\mu$GP was implemented in ANSI C language with about 6000 lines, and the source is freely available from the *CAD Group* web site.[1] Instruction library and external evaluator are strictly problem dependent, and need to be developed manually for each application.

## 4. Experimental evaluation

To evaluate the effectiveness of the proposed approach, it has been used to generate a test program for validating a microprocessor design. All parameters were set to their default values as shown in Table 1.

Despite the recent advance of formal methods, such as model checking [21], equivalence checking [22], theorem proving [23], etc., simulation is still a key step in the validation of modern microprocessors. Engineers run massive simulations to increase confidence on device correctness and to get insightful information. Results may be compared to an instruction set simulator, and extensive simulations emphasize the effectiveness of *assertions* and other checks. Even more, simulation results may be recorded and used to check local optimizations and run regression tests in subsequent phases. For example, in [24], Bentley describes how formal techniques, together with simulation techniques, were successfully used to debug Pentium® 4 designs.

Devising effective test programs is becoming both more important and more challenging. Hand-written test programs are only a first line of defense against bugs, since they focus on basic functionalities and important but rarely-occurring corner cases. *Automatic* test-generation systems have been proposed [25–28]. Although impressive results were shown, they are far from being *fully automated*, requiring high amount of manual work performed by skilled experts, and are biased towards corner cases and not broadly usable.

In contrast, $\mu$GP is more automatic, broadly-applicable and does not rely on skilled experts. It generates a test program that is able to effectively maximize a metric based on the design description. Furthermore, since endogenous parameters are automatically self-adapted to their optimal values, human intervention is usually limited to the enumeration of all available instructions and their possible operands. In no experiment the users were required to modify the initial mutation strength $S$, nor the self-adaptation inertia $\alpha$.

*Table 1.* $\mu$GP parameters.

| Parameter | Value | Description |
|---|---|---|
| $\mu$ | 30 | Population size |
| $\lambda$ | 20 | Genetic operators applied in each generation |
| $\alpha$ | 0.8 | Self-adaptation inertia |
| $\tau$ | 2 | Tournament size |
| $M$ | 300 | Maximum number of generations |
| $S_t$ | 50 | Steady-state threshold |

The experimental evaluation was performed against a *DLX/pII*, a complex, 5-stage pipelined implementation of the DLX microprocessor [29]. $\mu$GP was given the goal to maximize a validation metric called *instantiated statement coverage* i.e., to generate a test program that, when its execution is simulated against the VHDL model of the processor, attains 100% on the selected code coverage metric. That is to say, to generate a test program able to stress the functionalities of the microprocessor.

To avoid confusion, in the following the term “statement” will refer to a *statement* in an RT-level description, while the term “instruction” will denote an *instruction* in an assembly program. Since RT-level descriptions closely resemble programs, the term “execute” is commonly used in both domains: statements in a VHDL description are *executed* when the simulator evaluates them to infer design behavior; instructions in a program are *executed* when the processor fetches them and operates accordingly.

The metric used in this paper measures the percentage of executed (evaluated.) RT-level statements over the total when the execution of a given test program is simulated. The metric is calculated against the *elaborated design*. Thus, if a component (e.g., an adder) is used 13 times in the design, 13 independent components are considered when computing the metric.

Statement coverage can be considered as a required starting point for any design validation process. Attaining complete coverage ensures that no part of the design missed functional test during simulation, as well as reducing simulation effort from “over-validation” or redundant testing.

### *4.1. The DLX/pII*

The DLX/pII is a 5-stage pipelined implementation of the DLX microprocessor [29]. It realizes 79 of the instructions described in [30], including arithmetic and logic ones, tests, branches, specials, and load/store. Floating point instructions are not supported. The DLX uses 3 addressing modes: *register*, *immediate*, *displacement* (*offset*, *register deferred* and *absolute*). The hardware architecture is described at the RT-level by 979 VHDL statements, while the synthesized core is composed of about 38000 logic gates and 650 memory elements.

The instruction library for the DLX consists of 91 entries: prologue, epilogue, 7 conditional branches and 82 sequential instructions. Listing instructions and their syntax was a trivial task. The prologue contains a routine for initializing RAM memory, while the epilogue contains an empty endless loop to allow the evaluator to stop the simulation. The adopted external evaluator is based on Modelsim v5.7a by *Mentor Graphics* a commercial VHDL simulator.

The DLX/pII takes advantage of a pipelined architecture. A pipeline contains several independent units, called *stages*. Each stage executes concurrently, feeding its results to the following units. Instruction execution steps are arranged so that the CPU does not have to wait for one operation to finish before starting the next: consecutive instructions are likely to have their execution overlapped in time.

The first interesting consequence of this architecture is that the behavior of the DLX/pII is not determined by *one* instruction and its operands, but by a *sequence* of instructions and all their operands.

In general, the simultaneous execution of multiple instructions leads to several difficulties [31]. The set of pipeline peculiarities is prohibitively long to describe but it is insightful to sketch three types of potential problems: *data*, *control* and *structural* hazards.

Data hazards are caused by data dependency between instructions. For instance, one instruction may depend on the result of a previous one, already in the pipeline. Control hazards are caused by instructions that alter the usual flow of the program. For example, a conditional branching instruction invalidates the execution of all instructions following the incorrectly-predicted branch. Finally, structural hazards are produced by instructions contending non-sharable resources, such as the floating-point unit.

While in all cases the simplest solution is to "stall" the pipeline until the hazard is resolved, an excessive stalling may significantly degrade the overall performance. Thus, to reduce stalls, designers adopt mechanisms such as *data forwarding*. The basic idea of data forwarding is to pass a result directly to the functional unit that needs it, forwarding data from the output of one unit to the input of functional unit(s) requiring it.

Verifying the pipeline correctness is a complex task. It is not sufficient to check functionalities of all possible instructions with all possible operands. It is further necessary to check all possible interactions between instructions and operands inside the pipeline. Data forwarding and similar mechanisms may lead to complex interactions.

Table 2 compares 17 programs in terms of instance statement coverage (ISC). The list includes 15 functional test programs provided by microprocessor implementers (*arith_s, carry_su, except, fak, intrpt1, jump1, loadstore_s, loadstore_su, mul_su, set_s, set_su*), short applications (*mul32* and *div32*), system software (*system01*) and an exhaustive functional test that checks all possible instructions (*all_instr*). The last two rows report the result attained by induced test programs. Row [CCSS03] contains the results attained by the previous version of the algorithm and published in [18], while the performance of current version is shown in row [$\mu$GP].

To allow for the comparison to the previous results it is necessary to remember that the previous work exploited a different code-coverage metric. It counted *lines* in the source files, while *instantiated statements* are considered here (after the *elaboration* phase). Adopting the same setup of [18], the code-coverage of the proposed approach would be 97.96%, significantly above the previous results of 94.59%. However, with the more precise metric adopted in this paper, the old result is 96.20% and the new is 99.63%.

All experiments were run on the same Sun Enterprise 250 with two UltraSPARC-II CPUs at 400 MHz, and 2 GB of RAM. Remarkably, [18] required about two days (elapsed time), while the improved $\mu$GP only took about one single day.

Reasonably, *mul32*, a 32-bit multiplication performed through shifts and sums, and test benches, like *set_s*, attained very low statement coverage. It is well known that general application code is seldom effective to fully validate a design.

Results are further detailed by examining the different pipeline stages: *instruction fetch* (IF), *decode* (DEC), *execution* (EXE), *memory access* (MEM), *write-back* (WB). For the sake of completeness, also the two spare blocks of *input/output pads* (PADS) and *special-register control logic* (SREG) are considered. Remarkably, the induced test programs outperform all other programs in all stages. The *all_instr*, a program able to test *all* possible instructions, is unable to thoroughly verify pipeline stages, attaining a coverage around 90%. The statement coverage attained by the test program generated by our automatic method is more than 10% higher.

*Table 2.* DLX Summary.

| | Instance statement coverage (%) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Program | TOT | IF | DEC | EXE | MEM | WB | PADS | SREG |
| *arith_s* | 56.29 | 80.72 | 54.14 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *carry_su* | 57.17 | 80.72 | 55.13 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *except* | 60.02 | 89.76 | 57.01 | 100.00 | 50.00 | 100.00 | 100.00 | 90.72 |
| *fak* | 58.41 | 80.72 | 56.54 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *intrpt1* | 52.22 | 80.72 | 49.54 | 100.00 | 31.06 | 100.00 | 96.55 | 78.35 |
| *jump1* | 55.56 | 80.72 | 53.33 | 100.00 | 31.06 | 100.00 | 96.55 | 78.35 |
| *loadstore_s* | 58.06 | 86.75 | 55.13 | 100.00 | 52.27 | 100.00 | 100.00 | 79.38 |
| *loadstore_su* | 56.59 | 86.75 | 53.46 | 100.00 | 52.27 | 100.00 | 100.00 | 79.38 |
| *mul_su* | 51.44 | 74.10 | 49.07 | 100.00 | 31.06 | 100.00 | 96.55 | 73.20 |
| *set_s* | 55.53 | 80.72 | 53.28 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *set_su* | 57.00 | 80.72 | 54.95 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *div32* | 60.80 | 80.72 | 59.26 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *mul32* | 59.59 | 77.71 | 57.98 | 100.00 | 31.06 | 100.00 | 96.55 | 79.38 |
| *system01* | 59.19 | 89.76 | 56.07 | 100.00 | 50.00 | 100.00 | 100.00 | 90.72 |
| *all_instr* | 89.14 | 89.76 | 89.87 | 100.00 | 56.06 | 100.00 | 100.00 | 90.72 |
| *[CCSS03]* | 96.20 | 93.98 | 96.37 | 100.00 | 94.70 | 100.00 | 100.00 | 90.72 |
| $\mu$*GP* | 99.63 | 97.59 | 99.92 | 100.00 | 100.00 | 100.00 | 100.00 | 90.72 |

The execution of multiple instructions in the pipeline leads to intricate interactions, and corner cases hardly appear *by chance*. For instance, the *instruction-fetch* stage handles jump destinations, data hazards and delay slots, and it is not sufficient to execute all instructions with all operands in order to cover it.

For the sake of comparison, random test programs of about 1000 instructions each have been generated and simulated for one week. Random programs were generated exploiting the instruction library developed for the $\mu$GP, but no evolutionary operators. Cumulatively, all random programs attain an effective instantiated statement coverage of about 99%. This means that the adoption of the $\mu$GP evolutionary mechanisms reduces by 7 times the time required to reach comparable statement coverage with respect to a random approach. It must be noticed that the maximum attainable coverage may be lower than 100%, and that the proposed method leads to a sensible reduction of the amount of simulation output that needs to be analyzed by validation engineers.

After a detailed analysis of the covered statements, $\mu$GP appears able to stress the forwarding logic more than other approaches: the generated test set completely covers multiplexers and registers in the instruction fetch unit, exciting different types of data collisions. Moreover, test programs are able to generate several exception types in delay slots (e.g., a new test immediately after a branch-if-not-equal-to-zero instruction). Finally, the $\mu$GP-induced test set fully covers memory module, accessing aligned and non-aligned data of different size, while other programs only exploit standard (and faster) memory accesses.

As an example, there are two lines in an instruction decoder that are triggered only when a **JR** (an absolute jump to the address contained in a register) is found immediately after a **TRAP** (a software trap). They are essential since, differently from *delayed control-transfer* (delay slots), the microprocessor must handle the exception before executing any subsequent instructions. However, neither hand-written programs, nor random ones are able to stress them. Thus, after simulating the 15 programs in the test set and all the random programs, a few RT-level lines are still not verified. In contrast, the $\mu$GP is able to automatically induce a fragment of code for testing the behavior.

A closer examination shows that uncovered statements in the *instruction-fetch* control the program counter overflow. Since $\mu$GP was given the additional goal of program parsimony, it never managed to generate sufficiently long code to trigger a program counter overflow.

In the *decode* stage, most of the 113 uncovered statements control the $32 \times 32$-bit register file and the output multiplexers. Specifically, two possible configurations in the output multiplexers are never selected. This result indicates that $\mu$GP was not able to fully test all possible operands, however it turns out that some of the registers are not accessible to the programmer or $\mu$GP. Also the forwarding logic was not fully tested, showing that $\mu$GP was not able to generate sufficiently elaborate tests. Finally, in the *execution* unit, the analysis shows that a logic operation was not used, so the 32-bit adder was not fully checked. Moreover, $\mu$GP was not able to cover some statements controlling carry and overflow conditions.

## 5. Conclusions

$\mu$GP was originally devised to help microprocessor engineers in the automatic generation of test programs, a challenging task due to the difficulty and dimension of the problems involved. After two years of development, the internal individual representation and the instruction library allow assembly programs to be efficiently devised and impose very few syntactic restrictions. The flexible, modular architecture permits easy exploitation within different contexts for different goals. Moreover, the self-adaptation mechanisms effectively enhance the population-based search.

$\mu$GP has been used to devise a test program for validating a complex 5-stage pipelined microprocessor, maximizing a suitable code-coverage metric. Induced test programs have outperformed other approaches and surpassed the (supposedly) exhaustive test benches provided by designers. More than 99% of the executable statements were covered by the automatically generated test program. The few limitations of the approach were identified and are currently under study.

The experiments showed that engineers could get high-quality test programs with acceptable computation effort and reduced human effort. The evolved programs include combinations of instructions and branches that are likely to excite corner-case events that would be unlikely to be detected by manually-written targeted test programs. Thus, the tool could be adopted during the design cycle and used to add new content to the existing set of test programs.

The main difficulties with the approach are caused by the nature of the goal: the evolutionary fitness landscape is completely unknown and evaluating each candidate test program is computationally very intensive. Consequently, in practical applications, it is usually more useful to reach a local optimum as soon as possible. The design of the evolutionary core

may have been biased by this consideration. An external evaluator (based on a hardware accelerator) which will allow the execution of massive simulations that could reveal a better understanding of the mechanisms of the evolutionary process is currently under development.

## Acknowledgments

The author needs to acknowledge: Fulvio Corno, Ernesto Sanchez and Matteo Sonza Reorda for their inestimable support and insightful comments; Fabio Salto for the valuable debugging, and for running most of the experiments; Gianluca Cumani for his work implementing the earliest prototype of the tool.

This work has been partially supported by Intel Corporation through the grant "GP Based Test Program Generation."

## Note

1. http://www.cad.polito.it/research/microgp.html

## References

1. R. M. Friedberg, "A learning machine: Part I," IBM Journal of Research and Development, vol. 2, no. 1, pp. 2–13, 1958.
2. R. M. Friedberg, B. Dunham, and J. H. North, "A learning machine: Part II," IBM Journal Research and Development, vol. 3, pp. 183–191, 1959.
3. J. R. Koza, "Genetic programming," in Encyclopedia of Computer Science and Technology, A. Kent and J. Williams (Eds.), Marcel Dekker Publisher: New York, NY, USA, vol. 39, 1998, pp. 29–43, ISBN 0-8247-2292-2.
4. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT-Press: Cambridge, MA, USA, 1992, ISBN 0-262-11170-5.
5. J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT-Press: Cambridge, MA, USA, 1994, ISBN 0-262-11189-6.
6. A. Fukunaga, A. Stechert, and D. Mutz, "A genome compiler for high performance genetic programming," in Genetic Programming 1998: Proceedings of the 3rd Annual Conference, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo (Eds.), Morgan Kaufmann Publishers: San Francisco, CA, USA, 1998, pp. 86–94.
7. S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs," in Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE: New York, NY, USA, 1994, pp 154–159.
8. R. Poli, "Evolution of graph-like programs with parallel distributed genetic programming," Genetic Algorithms: Proceedings of the 7th International Conference, Thomas Back (Ed.), Morgan Kaufmann: San Francisco, CA, USA, 1997, pp. 346–353.
9. P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in Advances in Genetic Programming, K. Kinnear (Ed.), MIT-Press: Cambridge, MA, USA, 1994, pp. 311–331.
10. P. Nordin and W. Banzhaf, "Evolving turing-complete programs for a register machine with self-modifying code," in Genetic Algorithms: Proceedings of the 6th International Conference, L. Eshelman (Ed.), Morgan Kaufmann: San Francisco, CA, USA, 1995, pp. 318–327.
11. P. Nordin, W. Banzhaf, and F. Frantone, "Efficient evolution of machine code for CISC architectures using blocks and homologous crossover," in Advances in Genetic Programming III, L. Spector (Ed.), MIT-Press: Cambridge, MA, USA, 1999, pp. 275–299.
12. E. Lukschandl, H. Borgvall, L. Nohle, M. Nordahl, and P. Nordin, "Evolving routing algorithms with the JBGP-system," in Evolutionary Image Analysis, Signal Processing and Telecommunications: First European

Workshop, EvoIASP'99 and EuroEcTel'99, R. Poli, H.-M. Voigt, S. Cagnoni, D. Corne, G. D. Smith, and T. C. Fogarty (Eds.), Springer-Verlag: Berlin, DE, 1999, pp. 193–202.

13. F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante, "On the test of microprocessor IP Cores," in IEEE Design, Automation & Test in Europe Conference, W. Nebel and A. Jerraya (Eds.), IEEE Computer Society: Los Alamitos, CA, USA, 2001, pp. 209–213.
14. F. Corno, G. Cumani, M. Sonza Reorda and G. Squillero, "Efficient machine-code test-program induction," in Congress on Evolutionary Computation, D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton (Eds.), IEEE: Piscataway, NJ, USA, 2002, pp. 1486–1491.
15. W. Kantschik and W. Banzhaf, "Linear-Graph GP: A New GP Structure," in Genetic Programming: 5th European Conference, J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi (Eds.), Springer-Verlag: Berlin, DE, 2002, pp. 83–92.
16. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Evolutionary test program induction for microprocessor design verification," in 11th Asian Test Symposium, IEEE Computer Society: Los Alamitos, CA, USA, 2002, pp. 368–373.
17. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Exploiting auto-adaptive $\mu$GP for highly effective test programs generation," in 5th International Conference on Evolvable Systems: From Biology to Hardware (ICES2003), A. M. Tyrrell, P. C. Haddow, and J. Torresen (Eds.), Springer-Verlag: Berlin, DE, 2003, pp. 262–273.
18. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Automatic test program generation for pipelined processors," in SAC03: 18th ACM Symposium on Applied Computing, H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose (Eds.), ACM: New York, NY, USA, 2003, pp. 736–740.
19. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores," in DATE: IEEE Design, Automation & Test in Europe, N. Wehn and D. Verkest (Eds.), IEEE Computer Society: Los Alamitos, CA, USA, 2003, pp. 1006–1011.
20. A. L. Samuel, "Some studies in machine learning using the game of checkers," IBM Journal of Research and Development, vol. 3, no. 3, pp. 210–229, 1959.
    Also available as: A. L. Samuel, "Some studies in machine learning using the game of checkers," in Computers and Thought, E. A. Feigenbaum, and J. Feldman (Eds.), MIT Press: Cambridge, MA, USA, 1995, pp. 71–105.
21. G. Cabodi, S. Nocco, and S. Quer, "Improving SAT-based bounded model checking by means of BDDb-based approximate traversals," in IEEE Design, Automation & Test in Europe Conference, N. Wehn and D. Verkest (Eds.), IEEE Computer Society: Los Alamitos, CA, USA, 2003, pp. 898–903.
22. H. H. Kwak, I. H. Moon, J. H. Kukula, and T. R. Shiple, "Combinational equivalence checking through function transformation," in International Conference on Computer Aided Design, IEEE: Piscataway, NJ, USA, 2002, pp. 526–533.
23. J. Harrison, "Formal verification at Intel," in IEEE Symposium on Logic in Computer Science, IEEE Computer Society: Los Alamitos, CA, USA, 2003, pp. 45–54.
24. B. Bentley, "Validating the Intel Pentium 4 microprocessor," in Design Automation Conference, ACM: New York, NY, USA, 2001, pp. 244–248.
25. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau, "Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units," in DATE: IEEE Design, Automation & Test in Europe, C. D. Kloos and J. da-Franca (Eds.), IEEE Computer Society: Los Alamitos, CA, USA, 2002, pp. 36–43.
26. N. Utamaphethai, R. D. Blanton, and J. P. Shen, "Superscalar processor validation at the microarchitecture level," in 12th IEEE International Conference on VLSI Design, IEEE Computer Society: Los Alamitos, CA, USA, 1999, pp. 300–305.
27. A. Chandra, V. Iyengar, D. Jameson, R. Jawalker, I. Nair, B. Rosen, M. Mullen, J. Yoor, R. Armoni, D. Geist, and Y. Wolfstal, "AVPGEN—a test generator for architecture verification," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, IEEE: New York, NY, USA, vol. 3, no. 2, 1995, pp. 188–200.
28. A. Aharon, A. Ba-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator," IBM Systems Journal, vol. 30, no. 4, pp. 527–538, 1991.
29. D. A. Patterson and J. L. Hennessy, Computer Architecture—A Quantitative Approach, 2nd edition, Morgan Kaufmann: CA, USA, 1996, ISBN 1-55860-329-8.

30. P. M. Sailer, and P. M. Sler, DLX Instruction Set Architecture Handbook, Morgan Kaufmann: CA, USA, 1996, ISBN 1-55860-371-9.
31. D. Van Campenhout, T. N. Mudge, and J. P. Hayes, "High-level test generation for design verification of pipelined microprocessors," in Design Automation Conference, IEEE: Piscataway, NJ, USA, 1999, pp. 185–188.