Evolving Assembly Programs: How Games Help Microprocessor Validation

F. Corno, E. Sanchez, G. Squillero

Politecnico di Torino – DAUIN
Corso Duca degli Abruzzi 24
10129 TORINO – ITALY

http://www.cad.polito.it/

Contact Author:
Giovanni Squillero
giovanni.squillero@polito.it
Politecnico di Torino – DAUIN
Corso Duca degli Abruzzi 24
10129 TORINO – ITALY
Tel: +39-011564.7092
Fax: +39-011564.7099

Evolving Assembly Programs: How Games Help Microprocessor Validation

F. Corno, E. Sanchez, G. Squillero

Politecnico di Torino – DAUIN
Corso Duca degli Abruzzi 24
10129 TORINO – ITALY

http://www.cad.polito.it/

Abstract

Core War is a game where two or more programs, called warriors*, are executed in the same memory area by a time-sharing processor. The final goal of each warrior is to crash the others by overwriting them with illegal instructions. The game was popularized by A. K. Dewdney in his column on Scientific American in mid-1980s. In order to automatically devise strong warriors, µGP, a test program generation algorithm, was extended with the ability to assimilate existing code and to detect clones; furthermore, a new selection mechanism for promoting diversity independent from fitness calculations was added. The evolved warriors are the first machine-written programs ever able to become King of the Hill (champion) in all the four main international Tiny Hills. The paper shows how playing Core War may help generate effective test programs for validation and test of microprocessors. Tackling a more mundane problem, the described techniques are currently being exploited for the automatic completion and refinement of existing test programs. Preliminary experimental results are reported.*

1 Introduction

The incredible advances in microelectronics technologies that allow semiconductor manufacturers to deliver chips with ever-shrinking form factors and ever-increasing switching frequencies, enabled hardware architects to develop extremely complex integrated circuits and required the invention of sophisticated architectures to be able to best exploit the available silicon and computing power.

Only few years ago, testing and verification costs represented a small percentage of the total cost in the whole manufacturing budget, but these costs approach 70%. These problems are especially critical in the case of microprocessors. Such devices contain extremely complex architectures taking full advantage of the latest technological advances, and competitive pressure enforces this trend.

This paper tackles the problem of *automatic test program generation* for microprocessor test and validation. The innovative approach consists in using *competitive* *games* to emulate the complexity of real-world problems. In particular, a game where the goal is to produce assembly-level programs, and these programs have to exploit very peculiar characteristics of a (multithreaded) execution engine, and execution times and program length are limited resources.

The goal of this paper is twofold. We seek to explain the similarity between the microprocessor validation and test problem and the chosen game, and to justify the reasons why an optimizer devised for winning a game can be successful in a different and more down to earth domain.

This paper is organized as follows: the next section details the test program generation problem. Section 3 introduces the game, motivating the choice and identifying similarities with microprocessor validation and test. Section 4 describes the original µGP algorithm; readers already familiar with µGP may skip this section. New algorithms and strategies that empowered the µGP are described in detail in Section 5, while Section 6 describes the experimental evaluation and comments on the performance reached by µGP-generated warriors. Section 7 concludes the paper, and sketches the initial promising results observed on real microprocessor test program generation. Section 8 outlines some possible future directions of the research.

2 Automatic Test Program Generation

Chips of a hundred million transistors and running at clock frequencies of several GHz pose exceptional design challenges [1]. Designers are currently exploiting these chips in two directions: building *systems on chip* (SOC) or building more powerful microprocessors. In a SOC [25], the available on-chip area is used to integrate more and more functions (subsystems) in the same chip, resulting in a net decrease of the number of different chips in a system (and thus of the system cost) and in faster communications between subsystems, since no slow off-chip data exchange is needed. On the other hand, modern microprocessors [30], on which this paper focuses, use the available silicon headroom to increase the number of instructions executed per clock cycle, through the adoption of highly parallel computation and the integration of on-chip cache memories. In particular, higher parallelism is achieved by replicating several execution units, each capable of executing a subset of the available instructions, and by letting an instruction scheduler assign to different units the assembly instructions to be executed in the program. In this execution model, instructions are often executed speculatively (i.e., an instruction is executed before knowing if it needs to be executed, usually because it is beyond a conditional branch that has not been evaluated yet) and their results might be invalid (i.e., the computed value depends on the result of other instructions executed in parallel, and if it turns out to be invalid the instruction must be re-executed). Exact details of instruction execution (i.e., predicting the execution unit to which an instruction will be assigned, or predicting the completion time of an instruction) is becoming nearly impossible, due to the speculative nature of execution, to the latency time of the memory subsystem, whose values depend on the current status of the caches, and to the interrupts or exceptions that can happen during the normal execution flow, to service external peripherals or to handle context switches or demand paging.

These kinds of advanced architectures are very complex to conceive and design, but even more complex to validate and test. Validation [29] [11] is a step in the design process where the design is checked against its specification. In the case of a microprocessor, the specification is usually expressed by the "Instruction Set Architecture," i.e., the intended behavior of all assembly instructions, and validation consists in verifying that the correct result is always obtained for any sequence of valid instructions. Test [10] [19] is the final step, where a just-produced microprocessor chip is tested to check possible production errors, i.e., defects in the production process (diffusion, packaging, handling, soldering, etc.). Also in this case, the microprocessor under test must be shown to execute all instructions correctly, and to operate within the timing constraints given by the specification. validation and test are related activities, because they both consist in checking the result of a step in the design process (the circuit architecture, or the produced chip) against the specifications, aiming at detecting possible errors (design errors in validation, production errors in test). Manufacturers report that more than 60% of the chip cost can be attributed to validation and test, and it is evident that the quality of the shipped products is related directly to the quality of the checks during validation and test. As an example, the famous Pentium FDIV bug was uncovered due to insufficient Validation [22].

While single sub-components in the microprocessor may be validated or tested individually (by accessing their inputs and outputs directly through the simulator or through specific *test buses* built in the chip, and by applying specific *test patterns*), the most critical verification level is the system integration test, where the whole processor is checked. At this level, the only practical possibility is to let the processor execute carefully crafted *test programs*. A test program is a valid sequence of assembly instructions, that is fed to the processor through its normal execution instruction mechanism (i.e., the processor executes it as it would execute any other "normal" program), and whose goal is to uncover any possible design or production flaw in the processor under test.

The quality of the validation and test process thus relies heavily on the quality of the utilized test programs. We should also point out that, in this context, the quality of a test program is measured by its "coverage" of the design errors or production defects, by its code size, and by the text execution time.

When considering the problem of test program generation, we should recall the complexity of current microprocessors: architectural solutions are pipelined, superscalar, speculative, hyper-threaded, emulate several virtual processors, rely on several memory caching layers, and new features appear every quarter. Each of these keywords implies a complexity degree in the processor architecture, and test programs should be able to test all these advanced features. Incidentally, it makes no sense to test individual *instructions*, since the context in which an instruction executes (i.e., its preceding and following instructions) modify the processor state and modify the execution path taken by the instruction. This observation rules out exhaustive test programs, since developing and executing all possible *sequences* of instructions is combinatorially unpractical.

Manual generation of test programs is also impossible with current processors, due to the number of specific cases and instruction interactions. Manually written instructions are useful to check some specific behavior that is known to be critical and is known to be difficult to be covered by test programs built with other techniques. In particular, one class of manually developed test programs is that of *systematic* test programs that execute an array of similar operations with small variations (e.g., to test an arithmetic unit with different values of the operands).

The only general solution for a high-quality test program is to devise an automatic generation method. In 2002, a new test program generation algorithm called µGP [31] was proposed; later, it has been shown to be very effective for testing simple microprocessors like the i8051 [14], medium-size ones like the SPARC v8 [15], and even an Intel® Pentium® 4 microprocessor [24]. µGP is a general and versatile assembly-level test program generator, and may be used for different microprocessors as long as their Instruction Set Architecture is described in the form of an *instruction library*, and as long as a fitness function can be defined and computed.

As with all optimization approaches, taking the initial µGP algorithm and improving it until it generates useful and high-quality test programs for complex microprocessors requires several intermediate research steps. However, the difference between simple processors (where little or no parallelism is present) and current chips prevents generalizing the results. Improving the µGP performance on simple processors simply does not produce improvements to the results on complex ones, due to the intrinsically different architecture. On the other hand, developing µGP improvements or testing new techniques directly on complex processors is impossible: the computational effort required to simulate the processor models and extract the results coupled with their architectural complexity, would make an experimental analysis of proposed innovations practically infeasible.

A complementary approach was used to reach the same goal: instead of fighting with the barrier between simple processors (easy to test, but of no practical use) and complex ones (the real target, but unusable during algorithm development), find a *different* problem, with characteristics similar to microprocessor verification, but with a simpler definition and with a faster execution engine for fitness computation.

A game whose high-level characteristics resemble clearly the arduous microprocessor test program generation problem is the *Core War* game popularized by Dewdney [20]. In Core War, two or more programs are executed in the same memory area by a time-sharing processor, and the goal of each program is to crash the others by having them execute illegal instructions. Programs are written in an assembly language called *redcode*.

The *redcode* interpreter is fast enough to simulate the battles fought by two programs in seconds, and is perfectly suitable to quickly compute the metrics needed by a fitness function.

A set of µGP strategies was optimized to generate a strong program for the Core War game (in Core War terminology, "a strong warrior"). The simple definition of the problem, the quick evaluation, the availability of Internet servers where other powerful warriors could be met, allowed for greatly improving the µGP system, and in fact the latest versions of µGP evolved warriors are able to defeat both machine-generated and hand-written warriors. Indeed, µGP-generated warriors are the first machine-written programs ever able to top some Core War international competitions.

Using the Core War game, we could conceive, implement, evaluate, and fine tune several mechanisms that assist the evolutionary core in µGP. Such techniques include assimilation of existing code, detection of clones, and analysis of gene-level entropy to improve the selection operator and favor diversity. The combination of these new techniques, coupled with the previous µGP architecture, which already featured generic *instruction library*, weighted instruction fragments, and self-adaptive endogenous parameters, allowed µGP to generate strong Core War warriors.

3 Core War

Core War is a game played by two or more programs written in an assembly language called *redcode* running in a virtual computer called *memory array redcode simulator* (MARS). The object of the game is to cause all processes of the opposing programs to terminate, leaving the winner in sole possession of the machine. This is eventually accomplished by overwriting the opponents' code with illegal instructions. MARS memory is named "core", and, to stress the aggressive nature of the task, *redcode* programs are commonly called "warriors."

Core War was popularized by Dewdney in his column in *Scientific American* [20]. The "Core War Guidelines," a formalization of the rules, were written in the same year by Jones and Dewdney himself. It must be recalled that the idea of programs fighting in a computer memory, trying to overwrite the opponents, dates back to the early 1960s with the game *Darwin* devised at *Bell Labs* by Vyssotsky, Morris, and Ritchie.

Since its appearance, Core War attracted a huge interest from both the scientific community and from hobbyists. The *International Core War Society* (*ICWS*) was established in 1984 for the creation and maintenance of Core War standards, and for running tournaments. In the following years, there have been six annual tournaments and two new standards (ICWS'86 and ICWS'88). Several enthusiasts devised impressive warriors and developed subtle strategies, most labeled with evocative names such as *scanner, vampire*, *dwarf*, and *stoner*. A big community appeared suddenly.

In 1994, the ICSW proposed the new Core War standard, named ICSW'94. However, the golden era of Core War was almost ended and interest was swiftly decreasing. Today, despite several Core War tournaments run every year, the official Core War FAQ still maintains that the ICSW'94 "is currently being evaluated."

3.1 Core War and Validation

As stated in the Introduction, the choice of Core War was inspired by the necessity of finding a simple enough problem, with strong similarities to the microprocessor test program generation.

The MARS virtual machine is extremely abstract and atypical when compared with current microprocessors' Instruction Set Architectures. However, striking similarities can be identified to motivate our choice:

- In both cases the goal is to generate an assembly-level program, whose "performance" may be assessed by running it on a (virtual) processor in a partially unknown environment.
- In both cases it is not possible to evaluate a program without explicit simulation. Except in trivial cases, the fitness of an individual cannot be inferred with a syntactical analysis.
- The MARS virtual machine is a multithreaded concurrent environment, with some undetermined behavior due to the ignorance about the opponent's memory location and functionality. This resembles the concurrent speculative execution in microprocessors.
- Successful Core War warriors must exploit all aspects of the available instruction set, and usually must rely on weird side-effects specified by the *redcode* language. This resembles the need for test programs to find corner cases to be able to test all functionalities of the microprocessor.
- In MARS, memory size and execution time are limited resources: the memory is of fixed size, and faster warriors have a competitive advantage over slower ones. This polarization towards compact and fast programs resembles the quality requirements for microprocessor test programs.

3.2 Evolutionary Core War

Core War also attracted the interest from the evolutionary algorithm community. The idea of competing entities struggling in an artificial environment for survival is appealing, and it is illustrated clearly by the imaginative terminology developed. Moreover, *redcode* is a simple and completely orthogonal assembly language (all addressing modes are utilizable with all instructions, and all instructions are exactly in the same format), and the implementation of genetic operators is simplified.

In 1991, Perry [27] showed how random code can evolve into successful Core War warriors in only a few generations. In the following years, several interesting approaches were devised. Major contributions include: Newton's *Redmaker* [8], an experimental warrior evolver based on a grid-shaped evolution pool; Ankerl's *Yace* (*Yet Another Corewar Evolver*) [4]; and Hillis's *RedRace* (*Red Queen's Race*) [5]. Both the latter approaches start with a random population. In *Yace*, warriors fight against each other and the losers are replaced by slightly modified versions of the winners. In *RedRace*, on the other hand, all warriors in the population compete against all warriors on a target hill. *RedRace* also includes sharp techniques for speeding-up the search process, saving and restoring effective warriors (*Valhalla* and *Resurrection*), and handling multiple populations.

In 2002, Blaha and Wunsch [12] presented a study on automatic assembly program optimization using Core War as a case study. They analyzed different techniques and attained interesting results, although, in the authors' own words, the investigation was unable to devise really effective warriors.

In 2003, we [17] started using Core War as a test bench to enhance µGP, a generic assembly-level program generator. The game was exploited to evaluate the effectiveness of new evolutionary methodologies: the results attained by evolved warriors were used as a feedback for the adopted selection schemes and operators. The experience yielded an effective modified island model. However, despite the effort, no evolved warrior was ever able to attain good results on international Core War competitions.

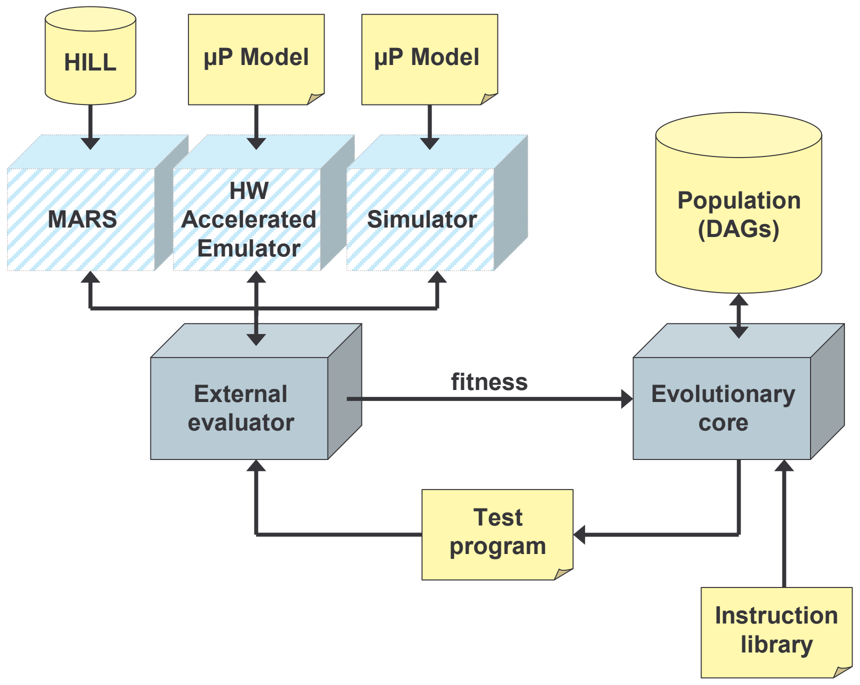
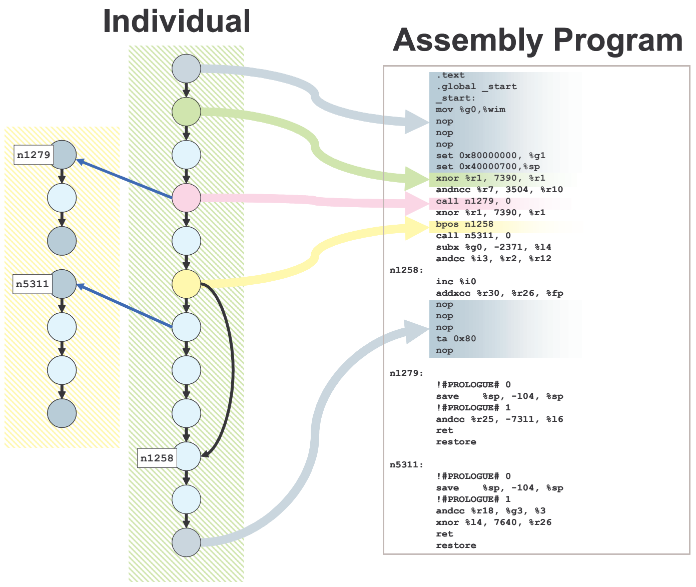


Figure 1: *µGP system architecture. The characteristics of the assembly language are stored in the* instruction library. *The* evolutionary core *evolves a population and may exploit different fitness functions for different problems through the* external evaluator.

In 2004, the approach was improved with a new migration model that exploits the polarization effect and a new hierarchical coarse-grained approach applicable whenever the final goal can be seen as a combination of semi-independent subgoals [18]. The µGP warrior eventually managed to be ranked 18th in an international competition among other evolved warriors.

4 µGP

This paper exploits µGP [31], a system for automatically devising and optimizing a program written in assembly-like languages. While features of µGP stem from standard genetic programming [23], µGP was deigned for generating syntactically correct assembly programs of variable size and fully exploiting the available assembly syntax (e.g., different addressing modes, instruction set asymmetries, subroutines, interrupt calls). Moreover, µGP was planned specifically to be versatile and usable with different microprocessors and different goals, and it has already been used for many different tasks [14] [15][16] [24].



```
        .text
        .global _start
        _start:
        mov %g0,%wim
        nop
        nop
        nop
        set 0x80000000, %g1
        set 0x40000700,%sp
        xnor %r1, 7390, %r1
        andncc %r7, 3504, %r10
        call n1279, 0
        xnor %r1, 7390, %r1
        bpos n1258
        call n5311, 0
        subx %g0, -2371, %l4
        andcc %i3, %r2, %r12
n1258:
        inc %i0
        addxcc %r30, %r26, %fp
        nop
        nop
        nop
        ta 0x80
        nop

n1279:
        !#PROLOGUE# 0
        save    %sp, -104, %sp
        !#PROLOGUE# 1
        andcc %r25, -7311, %l6
        ret
        restore

n5311:
        !#PROLOGUE# 0
        save    %sp, -104, %sp
        !#PROLOGUE# 1
        andcc %r18, %g3, %3
        xnor %l4, 7640, %r26
        ret
        restore
```

Figure 2: *A genotype (left) and the corresponding phenotype (right). Arrows show how nodes are mapped to different fragments of assembly code. From top to bottom: the prologue; a single instruction; a node referring to an outer label (subroutine call); a node referring to an inner label (conditional branch); the epilogue.*

µGP is composed of clearly separated blocks (Figure 1): an *evolutionary core*, an *instruction library*, and an *external evaluator*. The *evolutionary core* cultivates 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 a highly concise description of the assembly syntax or more complex, parametric fragments of code. The *external evaluator* evaluates the assembly program exploiting a simulator or other tools and eventually provides the necessary feedback to the evolutionary core.

Briefly, programs are represented internally as directed graphs (Figure 2). Graphs are composed of nodes, each one mapped to a macro (a generic fragment of code, possibly with parameters) (Figure 3).

Test programs are generated by an evolutionary algorithm implementing a (µ+λ) strategy modifying graph topologies and mutating parameters inside nodes. A population of µ individuals is cultivated, each individual representing a test program. In each step, λ new individuals are generated. Parents are selected using tournament selection with tournament size τ (i.e., τ individuals are selected randomly and the best one is picked). Each new individual is generated by applying one or more genetic operators. After creating λ new individuals, the best µ programs in the population of (µ+λ) are selected to survive to the next generation.

The fitness evaluation is intentionally external to µGP. The *external evaluator* is requested to compute a fitness value for each program, exploiting all the required tools. To further extend the usability, µGP was enhanced to handle a fitness value $\mathbf{F} = (f_1, f_2, \ldots, f_T)$ composed of T terms of strictly decreasing importance, i.e., $F^1 > F^2$ if $\exists i \le T : \left(\left(j < i \Rightarrow f_j^1 = f_j^2 \right) \wedge f_i^1 > f_i^2 \right)$. The parameter T must be selected at the beginning of the evolution process.

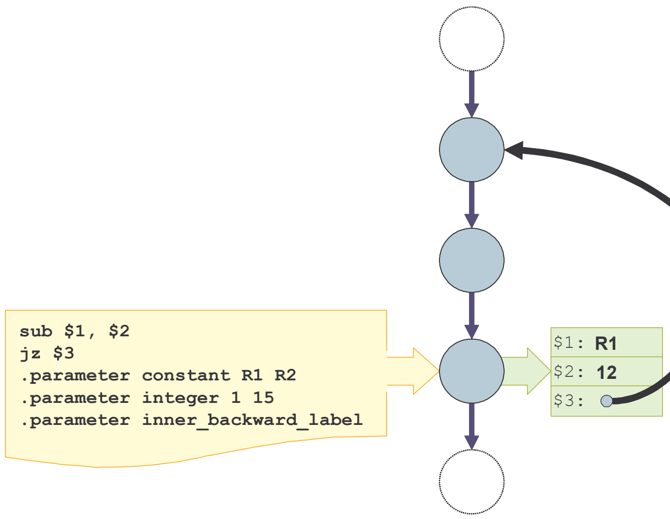


Figure 3: *A short program with a one node implementing a 2-line, 3-parameter macro. The highlighted node contains the actual values of the parameters: the constant string* R1*, the integer number* 12*, and a reference to a preceding node.*

The average strength of mutations and the activation probabilities of all genetic operators are endogenous parameters and are self-adapted by the algorithm. More details on µGP may be found in [31].

5 Advanced µGP

5.1 Exploitation of Existing Program

A significant limitation of µGP was its inability to reuse of existing material. In the current industrial practice, designers and test engineers write a relevant number of test programs for functional testing and for checking specific corner-case events. It would be useful to modify them automatically, possibly extending and enhancing the efficacy of existing test cases. Moreover, it would be extremely beneficial to start the evolution from a "soup" containing fragments of code already able to excite infrequent behaviors.

To allow the reuse existing code, µGP was extended by adding the ability to analyze a set of existing programs and, according to the *instruction library*, to translate them back to a possible internal representation. However, since µGP was given the ability to generate almost any fragment of code, translating generic programs (phenotypes) back to their originating graphs (genotype) leads to several, non-trivial practical problems. Moreover, the mapping from genotypes to phenotypes is problem-dependent: depending on the *instruction library*, a single fragment of code may correspond to different sets of macros (the representation may be non-univocal), or a given fragment of code may be even impossible to generate.

The whole assimilation process is sketched in Figure 4. It is composed of four main phases. First of all, the *instruction library* is read and each macro is translated into a regular expression [21]. Concurrently, the program is analyzed to identify different blocks (sections) showing syntactical uniformity. In this phase, different heuristics are exploited to handle generic hand-written code. Then, in each section, the set of potential *labels* (reference points) is identified. Finally, the section is partitioned in a list of macros. The last phase massively exploits the regular expressions built from the Instruction Library. After a preliminary pattern matching operation, the macros are validated assessing each parameter and label. Whenever a fragment of code cannot be generated by the original *instruction library*, the *instruction library* itself is modified.

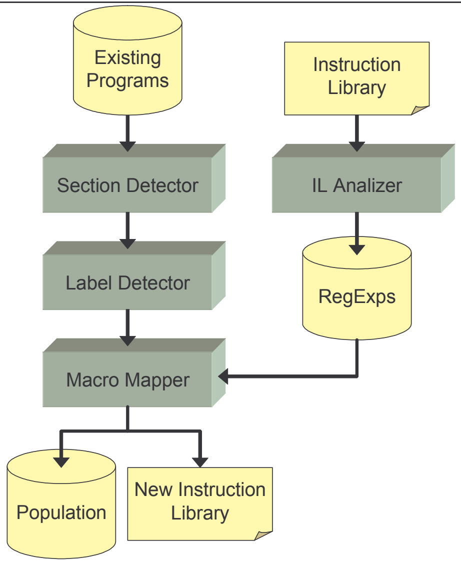


Figure 4: *Assimilation Process. The existing programs are analyzed using different heuristics, and the* Instruction Library *transformed in a set of* regular expressions *in order to discover a possible mapping. The result is a* Population *and a new* Instruction Library.

As a result of the whole process, the existing code is fully *assimilated* by µGP. The evolutionary core is able to modify the code, or mix fragments of codes originally in different programs exploiting a process that is akin to sexual recombination.

The result of assimilating a test set is a population of individuals and a new instruction library.

5.2 Diversity in µGP

While diversity is a key element of the biological theory of natural selection and maintaining high diversity is supposed to be generally beneficial [13], in a test-program generation problem, maintaining a high degree of diversity in the population is critical.

First of all, in a test-program generation problem the fitness function is requested to condense the whole behavior of a program into a manageable numeric amount and the loss of information may be relevant. While µGP allows the use of fitness values with an arbitrary number of terms, the result is always highly succinct compared to a hypothetical execution trace with all the values stored in memory and registers at different times. For example, the loss of information is evident when a coverage metric such as statement coverage or expression coverage is used: there is no guarantee that two programs are related by any means simply because they both excite the same percentage of the functionalities of a microprocessor. Thus it is possible that two programs attaining the same result on a code-coverage metric share no common parts and exploit different functionalities in the microprocessor. Indeed, it could be highly beneficial to combine such programs with a crossover operator, and conventional wisdom suggests that the diversity of the programs should not be overlooked during evolution. Such a situation is common to a large number of different scenarios and metrics.

The problem is intensified by the assimilation mechanism: adding hand-written code into a population of random individuals is likely to produce a very unbalanced situation where few individuals are significantly fitter than the vast majority. Such a population can be invaded quickly by individuals that are identical or almost identical to the fittest ones, reaching a premature steady-state condition.

The genetic programming literature consistently cites the importance of maintaining diversity as being crucial in avoiding convergence toward local optima and there are several different possible strategies to promote diversities, including non-standard selection, mating, or a replacement strategy. Indeed, µGP performances were already enhanced exploiting coarse-grained approaches, and such geographical distributions of individual are known to promote diversity [17] [18].

A qualifying aspect of µGP is the loose relationship between evolutionary core, phenotypic representation (controlled by the *instruction library*), and fitness calculation (performed through the *external evaluator*). Therefore, the adopted method for promoting diversity must not be based on the phenotypic appearance of individuals (the assembly programs), nor their fitness.

Two approaches are proposed to tackle the problem: *clone scaling* and *delta-entropy fitness holes*. The former technique is used to detect identical individuals and assigning them a fitness value without the need of running the evaluation. Such an assigned fitness value may be scaled down by a predefined factor. The latter technique is a mechanism to promote diversity in the population acting on the selection process.

5.2.1 Clone Scaling

Before evaluating an individual, µGP checks if identical individuals (clones) are already present in the population. When the number of clones C is greater than zero, the actual fitness assigned to the individual is multiplied by S^C. The parameter S is called the *clone-scaling factor*. While a continuous range of values is possible, usually S is set either to one (no clone scaling) or to zero (clone extermination). The clone-scaling factor is not self-adapted.

It must be noted that identical individuals at the genotypic level may not be mapped to identical programs at the phenotypic level. All labels, for instance, are translated to unique strings. Also, *unique tags* are designed intentionally to be distinctive. Moreover, different individuals at the genotypic level may be clones since the actual value of the constants is considered and not the index.

To speed-up the search for clones, a hash value is computed for each individual. µGP exploits state-of-the-art algorithms for calculating the hash functions of different portions of the graph such as string constants, pointers, and integers. The explicit comparison of two individuals is performed only if two individuals have the same hash value, making the process extremely efficient.

Clone scaling is particularly useful to discard duplicates after assimilating a large number of existing test sets or merging different populations.

5.2.2 Entropy and Delta-Entropy

Before defining the delta-entropy fitness hole, this section introduces some background definitions, in particular the concept of *entropy* in µGP. The purpose of the entropy value is not to rank a population in absolute terms, but to detect whether the amount of genetic diversity in a set of individuals is increasing or decreasing.

Let us define a symbol s as an *instance of a macro*, i.e., a macro and the value of its parameters (Figure 3). As for clone detection, the actual value of the constants is considered (the order of the list of possible alternatives has no influence); inner labels (labels inside the same section) are transformed to relative offsets; and outer labels are ignored completely. With this definition, just for the sake of entropy computation, the individuals in the population may be represented by the corresponding set of symbols.

The number of possible symbols, according to the given definition, is quite large: integer and hexadecimal parameters easily bloat the space of possible values. Moreover, the calculation may include tricky choices: some macros may be listed in the *instruction library* with a null probability, and some of them may be present in the actual population; the number of possible values for inner labels depends on the size of programs.

To overcome these difficulties, µGP simply measures the frequency of a given symbol in the universe of actually used symbols. Given a set **Q** of n individuals, I_k, in the population, $\mathbf{Q} = \{I_{k_1}, I_{k_2}, \ldots, I_{k_n}\}$, we define the **Q**-Universe of Symbols, $\Sigma_\mathbf{Q}$, as the set of all symbols appearing in at least one individual in **Q**. Formally,

$\Sigma_{\mathbf{Q}} = \{ s \mid \exists I : s \in I \wedge I \in \mathbf{Q} \}$. The frequency $\mathrm{f}_{\Sigma_{\mathbf{Q}}}(s)$ of a symbol s in the set $\Sigma_{\mathbf{Q}}$ is $\mathrm{f}_{\Sigma_{\mathbf{Q}}}(s) = \dfrac{\mathrm{occurrences}(s)}{\sum_{s' \in \Sigma_{\mathbf{Q}}} \mathrm{occurrences}(s')}$. With a formula similar to the calculation of entropy in information theory, the entropy of the set of individuals **Q** is defined as

$$\mathrm{H}(\mathbf{Q}) = -\sum_{s \in \Sigma_{\mathbf{Q}}} \mathrm{f}_{\Sigma_{\mathbf{Q}}}(s) \cdot \ln\left(\mathrm{f}_{\Sigma_{\mathbf{Q}}}(s)\right) \tag{1}$$

In a population, $\mathbf{P} = \left(I_0, I_1, \ldots, I_\lambda\right)$, individuals are considered ordered according to their fitness ($\mathrm{fitness}(I_0) \geq \mathrm{fitness}(I_1) \geq \ldots \geq \mathrm{fitness}(I_\lambda)$). The partial entropy function PH(x) (with $0 \leq x \leq \lambda$) is defined as

$$\mathrm{PH}(x) = \mathrm{H}\left(\left\{I_0, I_1, \ldots, I_x\right\}\right) \tag{2}$$

i.e., the entropy of the subpopulation composed of the x fittest individuals of the original population. Clearly, PH(λ) = H(**P**).

Given the above definition, each individual I_i ($i = 0,1,\ldots,\lambda$) can be associated with a *delta entropy* value ΔH_i:

$$\begin{cases} \Delta H_0 = \mathrm{PH}(0) \\ \Delta H_i = \mathrm{PH}(i) - \mathrm{PH}(i-1) \qquad (i > 0) \end{cases} \tag{3}$$

The delta entropy associated with an individual is an approximate and qualitative measure of the amount of new genetic information brought by the individual in the actual population.

Intuitively, a high delta entropy value, ΔH_i, indicates that the individual i brings effective fragments of code in the population. Thus, the individual is valuable and should be preserved. On the contrary, a low or negative ΔH_i suggests that individual i does not introduce enough unique symbols considering its fitness value. Theoretically, such an individual could be generated merely by applying the appropriate genetic operators to fitter parents, and thus may be safely discarded.

5.2.3 Delta-Entropy Fitness Holes

µGP selection is based on tournament selection of size τ. When two individuals are compared, with a probability h their delta entropy values are considered instead of their fitness values. This corresponds to a hole in the fitness function, where individuals are not chosen according to their direct ability to solve the specified task, but to a different measure. Thus, fitness holes bias evolution without affecting the fitness calculation. Delta-entropy fitness holes favor individuals containing *new* genetic material, and promote genetic variability in the population.

The fitness holes technique has been proposed by Poli [28] for solving the bloating problem. It must be noted that delta-entropy values are deeply related to fitness (in the definition of ΔH_i, the order in which individuals are considered is significant).

Moreover, the fitness calculation is external to the evolutionary core leading to additional difficulties. While a mathematical analysis is out of the scope of this paper, from a practical point of view the main effect of delta-entropy fitness holes is to disfavor *almost identical* individuals. If a genetic operator slightly mutates an existing individual enhancing it, while the fitness values of the two individuals may be similar, the delta-entropy value of the descendant is likely to be much higher than the delta-entropy value of the parent (almost all symbols of the parent are also present in the offspring).

6 Experimental Evaluation

The experiments report two lineages of warriors: *RedBorg* and *White Noise*. The former were the first Core War warriors evolved exploiting an assimilation process and are reported here for completeness. *White Noise*, on the other hand, was cultivated exploiting all the techniques reported in this paper.

6.1 Core War Hills

Core War competitions are usually called *hills*, and the champion of a Core War competition is therefore called *king of the hill* (KOTH). Most hills are a repositories of *N* warriors. When a new program is submitted, it plays a certain number of one-on-one games against each of the *N* programs currently on the hill. In each game the two opponents are put in random positions in the core. The new warrior gets *W* (usually 3) points for each win and *T* (usually 1) points for each tie. In each game, the two warriors are put at a pseudo-random distance inside the *MARS* memory. The pseudo-random sequence is usually seeded with a number calculated from the source code of the warriors themselves. Thus all matches are reproducible, but distances are unpredictable by warriors' authors as long as sources are undisclosed.

Existing programs never replay each other. On some hills their previous battles are recalled and the score *s* updated exactly, while on other hills the new score *s* is calculated with a formula such as:

$$s_{new} = \frac{s_{old} \cdot (N-1) + W \cdot victories + T \cdot ties}{N} \quad (4)$$

Both these scoring systems are called *flat*. Some hills instead of ranking warriors with flat scores use a so-called *recursive scoring*, where flat scores are weighted, and these weights modified repeatedly until a steady state is reached. The least warrior is usually pushed off the hill, except from *infinite hills* which grow indefinitely (no warrior is ever discarded from an infinite hill). Typically, infinite hills use recursive scores.

Several hills are available on the Internet, each one accepting a specific *redcode* style (e.g., instruction set or program length) and running games with certain parameters (e.g., number of matches, maximum number of concurrent warriors or scoring systems). The oldest and most famous server is simply named KOTH [6] and still hosts seven hills with different settings, including two multi-warrior melee hills and two hills using the older *redcode* '88 standard. New servers appear and close frequently.

The core size *c* (the dimension of the MARS memory) is a crucial parameter and may profoundly influence warrior strategies. The most common core size is $c = 8{,}000$, followed by $c = 8{,}192$, $c = 55{,}400$, and $c = 800$ (all of them divisible by 4). Hills running core of size $c = 800$ are called *tiny hills*, and usually do not accept warriors containing more than 20 instructions.

Due to this limited search space, most approaches exploiting evolutionary techniques have focused on climbing tiny hills. These hills are also practical for the purpose of this paper: test programs are required to be extremely short, and being able to optimize the size of a test program is a required capacity. While unnecessary code in the validation process may lead to overlong simulations and force checking irrelevant data, it could cause significant loss of money in the test process. Moreover, some test procedures, such as *on-line testing*, require firm limits to test programs size.

The most active tiny hills are available currently on *SAL* [3], *Sourceforge* [2], and *Koenigstuhl* [7]. *SAL* is hosted at the Department of Mathematical and Statistical Science of University of Alberta, Canada. The server runs five different hills, including a tiny hill ($c = 800$) and a nano hill ($c = 80$). *Sourceforge* server runs nine hills, two of which are tiny: the *Tiny (evolved) Hill* limited to evolved (non-handwritten) warriors, and the *Tiny (all) Hill* accepting all types of code. The *Koenigstuhl* (King's Chair) server hosts ten infinite hills, one of them, called *Tiny-Koenigstuhl*, using a core size of $c = 800$. The infinite tiny hill lists more than 250 strong programs collected over the years.

Differently from other hills, the source code of warriors posted to *SAL* is not visible to all users. Thus, many authors who are not willing to expose their strategies send their latest warriors to this server only, making its hills particularly active and hard.

6.2 RedBorgs

For devising *RedBorgs*, µGP was set to evolve a population of $\mu = 20$ and $\lambda = 80$ individuals through 100 generations. Cultivation did not exploit clone scaling and delta-entropy fitness holes. The adopted instruction library is described in [18].

RedBorg v1.0r6 started from a population composed of all the warriors in the *Tiny (evolved) Hill* of *Sourceforge* and easily attained the selected goals, getting KOTH of both *Sourceforge* tiny hills in May 2004. When new warriors became KOTH on *Tiny (all) Hill*, they were assimilated and a new *RedBorg* (*RedBorg v1.0r7*) was quickly cultivated to defeat them. This warrior was not submitted to the *Tiny (evolved) Hill* since it is likely to exploit handwritten code.

Rank	Warrior	Score
1	***RedBorg v1.0r6***	**8,029**
2	*Maria's Kiss 947651080 x 500*	7,175
3	*Evolution Strikes Back*	7,138
4	***PanGenetic Gargle Blaster III***	**5,414**
5	*2119-6757-xt430-0-tiny-eve86*	5,289
6	*I, Robot*	5,279
7	*rdrc: Dementia Minnie*	5,191
8	*Redrace*	5,130
9	*SjHn (2/22/03)*	5,092
10	*211.red 09-05-2003*	5,060

Table 1: *Top 10 warriors of Tiny (evolved) Hill at Sourceforge. Programs devised by µGP are shown in boldface.*

Table 1 shows the top 10 warriors on the *Tiny (evolved) Hill*, while Table 2 shows the top 10 warriors on *Tiny (all) Hill* in August 2004. Warriors devised by µGP are shown in **boldface** (*PanGenetic Gargle Blaster III* is a warrior previously evolved by µGP without exploiting the assimilation process, the details are out of the scope of this text).

Rank	Warrior	Score
1	***RedBorg v1.0r7***	**6,077**
2	*Tiny Blowrag*	5,746
3	*Digital Swarm*	5,707
4	*Snufkin*	5,580
5	*Easter Egg*	5,532
6	***RedBorg v1.0r6***	**5,447**
7	*Where's Giles?*	5,443
8	*Dark Skies*	5,359
9	*801-948-xt642-3-tiny-eve86*	5,333
10	*E-clear*	5,321

Table 2: *Top 10 warriors of Tiny (all) Hill at Sourceforge. Programs devised by µGP are shown in boldface.*

RedBorg v1.0r6 initially ranked 4th on SAL *Tiny Hill*, neither *RedBorgs* was posted to infinite hills.

6.3 White Noise

A more interesting genetic experiment started assimilating all possible warriors from all accessible tiny hills, such as *Tiny-Koenigstuhl* and *Sourceforge*. Several additional warriors were taken from the *usenet* or enthusiasts' web pages over the Internet. Overall, µGP assimilated over 2,000 different warriors. Relying on clone scaling and delta-entropy selection, no pre processing of any type was performed on the initial set of programs. All programs previously evolved by µGP in [17] and [18] were also assimilated. A minimal *instruction library*, limited to a single macro was used (Figure

5). It must be noted that due to the simplicity of both *redcode* language and *instruction library* only a subset of the features in the assimilation mechanism was exploited effectively.

```
.type INST constant add sub mul div mod jmz jmn djn seq sne slt mov dat nop spl jmp
.type INST_MOD constant a b ab ba f x i
.type INT integer 0 799
.type ADDR constant # $$ @ * { } < >

.macro
       $1.$2 $3$4+$5, $6$7+$8
.parameter type INST
.parameter type INST_MOD
.parameter type ADDR
.parameter inner_generic_label
.parameter type INT
.parameter type ADDR
.parameter inner_generic_label
.parameter type INT
.endmacro
```

Figure 5: *The instruction library exploited for evolving White Noise. A single macro is sufficient to describe the whole Redcode syntax since instructions are completely orthogonal to addressing modes, and there are no subroutines, nor interrupts.*

After the assimilation, μGP was started with a population of $\mu = 200$ and $\lambda = 800$ individuals. The maximum number of generations was set to 1,000. No problem-specific enhancements of any kind were exploited, since the goal was to test the enhancements of the evolutionary core.

Warriors were cultivated against a hill of 50 programs taken from the *Tiny-Koenigstuhl* and exploiting different strategies. To remove the bias introduced by the pseudo-random distance calculation, all matches starting at a valid distance (i.e., *all possible matches*) for each couple of warriors were run.

Moreover, to favor an aggressive behavior, the main contribution to the fitness value was the number of victories followed by the score attained on the hill ($W = 3$, $T = 1$). Practical considerations helped in setting the dimension to the fitness hole to $h = 0.3$, thus 30% of the parents were chosen by comparing their delta-entropy contributions rather than their fitness values.

The evolved warrior was called *White Noise* (Figure 6) and on August 2004 became the first KOTH ever devised by a machine on the *SAL Tiny Hill*. It was also submitted to the *Koenigstuhl Infinite Tiny Hill*, and became KOTH of the infinite hill. The evolved warrior was posted immediately on the newsgroup *rec.games.corewar* asking the community for comments, and exposing it to targeted attacks.

```
;redcode-tiny
;name White Noise (RBv1.5r10)

org n3262515
n3262514:
    add.f $n3262517+0, @n3262518+0                  ;scan loop
n3262515:
    sne.i }n3262515+733, $n3262515+727
n3262516:
    djn.f $n3262514+0, {n3262516+50
n3262517:
    spl.b #n3262517+774, {n3262517+774              ; core clear (endgame routine)
n3262518:
    mov.i @n3262521+0, >n3262515+0                  ; first bombing on detected opponent
n3262519:
    mov.i @n3262521+0, >n3262516+0                  ; more bombing
    mov.i @n3262521+0, >n3262516+0
n3262521:
    djn.x $n3262519+0, {n3262523+0
n3262522:
    dat.f #n3262522+619, }n3262530+0                ; bomb (to kill the opponent)
n3262523:
    spl.b #n3262523+775, $n3262518+17               ; bomb (to stun the opponent)
n3262524:
    spl.x #n3262524+1, $n3262524+1                  ; probably a decoy
    mov.i @n3262527+0, >n3262522+148
    spl.b #n3262516+480, >n3262524+396
n3262527:
    spl.b #n3262517+248, >n3262527+396              ; mysterious code
n3262528:                                           ; may become effective when attacked
    spl.x $n3262528+1, $n3262528+1                  ; by other warriors
n3262529:
    spl.x #n3262529+1, $n3262529+1
n3262530:
    mov.i @n3262532+0, >n3262527+148
n3262531:
    spl.x #n3262531+1, $n3262531+1
n3262532:
    spl.x $n3262532+1, }n3262532+1
n3262533:
    spl.x $n3262533+1, $n3262533+1
    end
```

Figure 6: *White Noise source code. Comments have been inserted by hand.*

White Noise resembles *Tiny BiShot 2.0* by Schmidt: it scans the MARS memory seeking for its opponent, and starts the attack as soon as a suspicious memory location is detected. This approach is called *one-shot scanning*, and it is possible because the memory is initialized with a special value before each match.

More precisely, *White Noise* compares pairs of memory locations and conjectures the presence of an opponent if they are not equal, regardless of their actual content. Since the attack starts as soon as a difference is found, it can be mystified by spurious code or randomly modified memory locations. This is a drawback common to most one-shot scanners. Indeed, *White Noise* does modify the memory while scanning creating a decoy track against other scanners.

The aggression is performed in two steps. First, *White Noise stuns* the other warrior by throwing inside the opponent code special instructions that, although legal, have the effect of slowing down the execution. Then, it crushes the other program by completely overwriting it with illegal instructions. The second attack phase, called *core clear*, is particularly robust, and could remain operative even if damaged.

White Noise scans the memory backwards using uncommonly big and irregular steps. Remarkably, when it finds a suspicious memory location, it focuses its attack both on it and 50 locations away. The overall effects of this strategy are unclear: commentators suggest that it could help finding certain type of opponents (the so-called *papers*) faster, but it may also be a penalizing factor against different ones, especially small warriors, or frontward one-shots. As a result, some authors wrote short, frontward one-shots and submitted them to the *SAL* tiny hill, but did not manage to defeat *White Noise*.

Rank	Warrior	Score
1	***White Noise***	**173.5**
2	*Tinyshot*	170.1
3	*Provenance*	170.0
4	*Tiny BiShot 2.0*	169.2
5	*Holograph*	168.7
6	*Seek and Destroy*	168.6
7	*Four Winds*	166.9
8	*Cream and Chocolate*	165.7
9	*Betadine*	165.5
10	*Tinyboss III*	165.3

Table 3: *Top 10 warriors of Infinite Tiny Hill at Koenigstuhl.*
The program devised by µGP is shown in boldface.

Table 3 reports the top of *Tiny-Koenigstuhl*. *White Noise* scores more than 4 points higher than the closest opponents.

Rank	Warrior	Score
1	***White Noise***	**149.4**
2	*Endless pain*	148.0
3	*Digital Swarm*	145.7
4	*Muskrat*	144.9
5	*Sneaky Spike 2*	144.3
6	*Hired Sword*	142.9
7	*tiny Blowrag*	142.8
8	*Four Winds*	141.0
9	*Moomintroll*	139.8
10	*Soft as Silk*	139.7

Table 4: *Top 10 warriors of Tiny Hill at SAL.*
The program devised by µGP is shown in boldface.

Table 4 reports the top of the *Tiny Hill* at *SAL* after challenge #393, nine months after the submission of *White Noise*. Even if its source code was publicly available, the evolved warrior defeated more than 70 human-written challengers, remaining KOTH on *SAL*. Indeed, *White Noise* also entered the *Tiny Hall of Fame* at *corewar.co.uk* [9], the record of the warriors that have survived longest on *SAL*, and, in the May 2005 list, it ranks 25th. Remarkably, *White Noise* is also the only evolved warrior on such list.

Challenge #394 eventually modified the situation, pushing *Endless pain* by Labarga to the top. Recall that this is probably the hardest and most active tiny hill available today, and all scores are updated whenever a new warrior challenges it.

Not being able to access the source code of warriors is a double handicap for µGP. Firstly, as showed by *RedBorg* warriors, assimilating a hill is an easy and simple way to get KOTH. Secondly, when all warriors on a hill are exposed, it is possible to use the real results that the programs would attain as fitness values, focusing the optimization process. Differently, to challenge *SAL* the aim was to evolve *generic* warriors.

The ending instructions of *White Noise* are unclear, but, since the behavior of the program cannot be studied against the opponents, a detailed analysis is difficult. It can be maintained, however, that they are effective against other warriors since removing them reduces the effectiveness of the warrior: a version of *White Noise* without such code has been submitted to the hill under the name of *Blue bubble*, and its score was 3.8 points lower than the original one.

It should be noticed that *Tiny BiShot 2.0* is the second strongest warrior on *Tiny-Koenigstuhl*. Indeed, since *White Noise* was evolved against a subset of this hill, it looks reasonable that µGP favored strategies similar to Schmidt's. Differently, on SAL (after challenge #393), *Tiny BiShot 2.0* was ranked only 23rd.

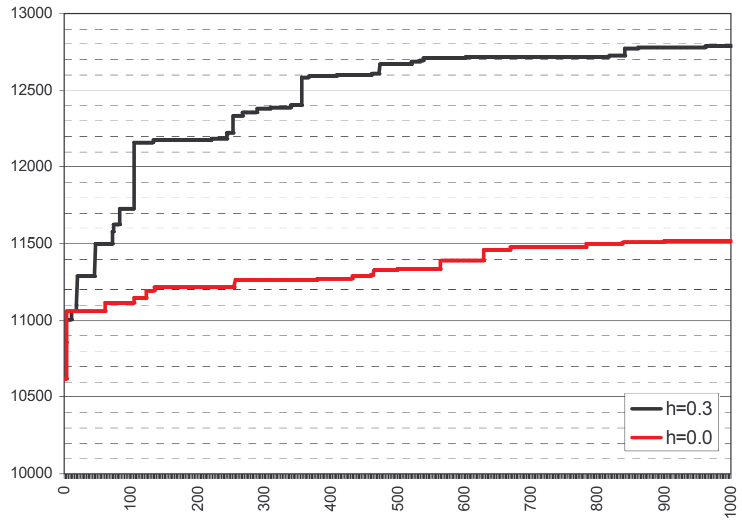


Figure 7: *Effects of delta-entropy fitness holes in a* Core War *experiment*

Lastly, Figure 7 shows the score attained by the best warrior against the test hill during the 1,000 generations of the experiment. The effect of a delta-entropy fitness hole (h=0.3) was compared against the standard selection mechanism (no hole, h=0.0). All the other settings were identical and are described above. The practical effect of the delta-entropy fitness hole is evident.

7 Preliminary Results on DLX/pII

The described techniques are also being exploited for the automatic completion and refinement of existing test programs.

In the design cycle of a microprocessor core, the unit is usually refined through a series of subsequent steps. To deliver a flaw-free unit at the end of the process, a verification step is required in each stage, and it would be useful to automatically develop the set of test programs for verification concurrently to the design, the automatic process exploiting assimilation and µGP would require both less human intervention and less computational resources. Preliminary experiments targeted the DLX/pII, a pipelined microprocessor [26] and showed how the test programs developed by designers can be enhanced automatically and completed to build a set able to maximize a given verification metric.

An iterative process was then run: a set of test programs was cultivated automatically by the µGP to maximize a given coverage metric. It was assimilated and used as a starting point to maximize a more complex metric. The assimilation process allows tweaking the *instruction library* in each step, modifying it, and permits the inclusion of hand-written code for covering rare corner cases.

The five considered coverage metrics were: *statement coverage* (the percentage of executable statements in the model that have been exercised during the simulation); *branch coverage* (the percentage of Boolean expressions evaluated to both true and false); *condition coverage* (the percentage of Boolean subexpressions evaluated to both true and false); *expression coverage* (as condition coverage, but calculated against concurrent signal assignments); *toggle coverage* (the percentage of bits that toggle at least once from 0 to 1 and at least once from 1 to 0 during the simulation).

Program Set	**Evaluation metric**				
	Statement	**Branch**	**Condition**	**Expression**	**Toggle**
Initial	68.91%	52.40%	43.01%	26.93%	30.34%
µGP-*Statement*	**99.65%**	98.43%	77.20%	36.96%	43.28%
µGP-*Branch*	99.23%	**98.59%**	76.17%	33.50%	39.39%
µGP-*Condition*	77.29%	68.69%	**77.20%**	25.11%	26.33%
µGP-*Expression*	93.07%	88.09%	55.44%	**43.71%**	44.02%
µGP-*Toggle*	96.31%	92.29%	62.18%	38.72%	**78.28%**
Total	99.65%	98.81%	79.79%	46.32%	80.31%

Table 5: Experimental Results on DLX/pII.

Results are summarized in Table 5. The set of functional programs devised by the designer to validate the main functionalities of the core is used as a starting point. The different rows contain the results for the different set of test programs: the initial one and the 5 generated by the µGP maximizing specific metrics. The values attained by each set on all possible metrics are also reported. The grayed cells represent the value on the metric that the set was intended to maximize. The last row contains the results of the complete set including the manually generated program.

Significantly, the µGP was always able to maximize the coverage it targeted (the grayed cells), but it is interesting to notice the relationship between the different verification metrics. For instance, maximizing toggle coverage yields the second highest statement coverage, showing that a high activity on data can cause a large number of statements to be exercised. On the other hand, maximizing the condition coverage does not yield impressive results compared with the branch coverage, while, in theory, the former metric is an extension of the latter.

While the number of assimilated test programs for the DLX/pII is considerably smaller than the number of warriors assimilated to cultivate *White Noise*, the complexity of the assembler is higher. New experiments on more complex microprocessors are currently being performed.

8 Conclusions

This paper showed how playing the computer game called Core War may help devise effective test programs for validating and testing microprocessors. To devise effective warriors, new techniques were integrated in µGP: the ability to assimilate existing code, detect clones, and a selection mechanism for promoting diversity completely independent from fitness calculations.

Using such techniques, µGP evolved the strongest Core War warriors in the four main international competitions: the *Tiny (evolved) Hill* and the *Tiny (all) Hill* at *Sourceforge*; the tiny hill at *SAL*; the infinite tiny hill at *Koenigstuhl*. µGP-generated warriors are the first machine-written programs ever able to become *King of Hill* (champion) in all these competitions.

Described techniques are now being applied on the original real-world problem. First results are very promising, showing that the improvements that could be developed to the optimization algorithm while working with a game problem are equally useful in a totally different domain. Thus, we legitimized the use of games as useful conceptual and practical tools to foster the advance in the evolutionary computation community to tackle industry-strength problems.

We are currently extending the evolutionary core of µGP enhancing the self-adaptation mechanism and adding new genetic operators. More specifically, we are turning μ, λ, and τ into endogenous parameters, and we are analyzing the effects of extremely small mutations to fine tune the programs.

9 Acknowledgments

Authors wish to thank Giuseppe Trovato for implementing the assimilation mechanism, and Germán Labarga, known in the Core War community as *Neogryzor*, for studying *White Noise* behavior.

A special acknowledgment is due to all members of the *μGP Core War Collective*, who enthusiastically helped μGP playing Core War: Fabio Salto, Massimiliano Schillaci, Luca Sterpone, and Pier Paolo Ucchino.

10 References

[1] *International Technology Roadmap for Semiconductors*, 2003 edition

[2] http://corewars.sourceforge.net/

[3] http://sal.math.ualberta.ca/

[4] http://students.fhs-hagenberg.ac.at/se/se00001/yace.html

[5] http://users.erols.com/dbhillis/

[6] http://www.koth.org/

[7] http://www.ociw.edu/~birk/COREWAR/koenigstuhl.html

[8] http://www.tl.infi.net/~wtnewton/corewar/evol/

[9] http://corewar.co.uk/tinyhof.txt

[10] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital Systems Testing & Testable Design*, Wiley-IEEE Press, 1994

[11] B. Bentley, R. Gray, "Validating The Intel® Pentium® 4 Processor", *Intel Technology Journal*, 1Q 2001

[12] B. Blaha, D. Wunsch, "Evolutionary programming to optimize an assembly program", *Proceedings of the 2002 Congress on Evolutionary Computation*, pp 1901-1903, 2002

[13] E. Burke, S. Gustafson, G. Kendall, "Diversity in Genetic Programming: An Analysis of Measures and Correlation With Fitness", *IEEE Transactions on Evolutionary Computation*, Feb 2004, Vol 8, No I, pp. 47-62

[14] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores", *Design, Automation and Test in Europe*, 2003, pp. 1006-1011

[15] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero, "Automatic Test Program Generation — a Case Study", *IEEE Design & Test, Functional Verification and Testbench Generation*, Volume: 21, Issue 2, March-April 2004, pp. 102-109

[16] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero, "Code Generation for Functional Validation of Pipelined Microprocessors", *Journal of Electronic Testing*, vol. 20, issue 3, June 2004, pp. 269-278

[17] F. Corno, E. Sanchez, G. Squillero, "Exploiting Co-Evolution and a Modified Island Model to Climb the Corewar Hill", *Proceedings of the 2003 Congress on Evolutionary Computation*, 2003, pp. 2222-2229

[18] F. Corno, E. Sanchez, G. Squillero, "On The Evolution of Corewar Warriors", *Proceedings of the 2004 Congress on Evolutionary Computation*, 2004, pp. 133-138

[19] A. L. Crouch, *Design for Test for Digital Ic's and Embedded Core Systems*, Prentice Hall PTR, 1999

[20] A. K. Dewdney, "Computer recreations: In the game called Core War hostile programs engage in a battle of bits", *Scientific American*, 250(5), pp. 14-22, 1984

[21] Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 2002

[22] T. R. Halfhill, "The Truth Behind the Pentium Bug", *Byte*, March 1995

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

[24] W. Lindsay, E. Sanchez, M. Sonza Reorda, G. Squillero, "Automatic Test Programs Generation Driven by Internal Performance Counters", to appear on: *IEEE Microprocessor Test and Verification,* 2004

[25] F. Nekoogar, F. Nekoogar, *From ASICs to SOCs*, Prentice Hall PTR, 2003

[26] D. A. Patterson and J. L. Hennessy, *Computer Architecture - A Quantitative Approach,* (2nd edition), Morgan Kaufmann, 1996.

[27] J. Perry, "Core Wars Genetics: the evolution of predation", http://www.soberit.hut.fi/tik-76.115/96-97/palautu-set/groups/DSM/ma/documents/cwbasics.html

[28] R. Poli, "A Simple but Theoretically-Motivated Method to Control Bloat in Genetic Programming", EuroGP 2003, pp. 204-217

[29] I. Silas, I. Frumkin, E. Hazan, E. Mor, G. Zobin, "System-Level Validation of the Intel® Pentium® M Processor", *Intel Technology Journal*, May 2003

[30] J. P. Shen, M. Lipasti, *Modern Processor Design*, McGraw-Hill, 2002

[31] G. Squillero, "MicroGP — An Evolutionary Assembly Program Generator", to appear on: *Genetic Programming and Evolvable Machines*, 2005