Evolving Assembly Programs: How Games Help Microprocessor Validation

F. Corno    E. Sanches    G. Squillero

Abstract

This presentation covers several sections of the 2005 paper by Squillero et al. for the *Advanced Topics in Evolutionary Algorithms* seminar.

Ben-Gurion University of the Negev, fall 2005.

— Typeset by FoilTEX —
Setting 1: microprocessor validation

- modern microprocessors are very complex (caches, parallelism/pipelining/branch prediction, interrupts/exceptions)

- complex to conceive and design, even worse to validate and test

- system integration testing only by test programs

- developing and executing all possible instruction sequences combinatorically unpractical

- $\mu$GP — a 2002 (Squillero) GP test generator, simple/med. CPUs
Setting 2: corewars

- programs compete for survival in a virtual machine

- simple assembler (*Corewars* or *Redcode*) allows read/write, arithmetic, data blocks, jumps, forking etc.

- scoring by cells owned, number of kills, and clones alive

- international competitions; human/machine-generated/mixed

- peak of interest: 80s to early 90s
Results

- GP-produced warriors
  - extended $\mu$GP to assimilate existing code and to detect clones
  - highly competitive (and human-competitive)

- once the framework excelled with corewars, it was applied to chip validation

- promising results were observed
Motivation for the approach

Instead of fighting the barrier between the simple processors (easy to test, but of no practical use) and the complex ones (the real target), switch to a different problem (corewars) with similar characteristics, but simpler definition and a faster execution engine for fitness computation.
Outline

• Core War
  – the game; virtual machine; assembly syntax
  – similarity to CPU testing; evolutionary programming

• μGP

• Results
Core War: history (as a game)

1960 *Darwin* Bell Labs game (Vyssotsky, Morris, Ritchie)

1984 Core War popularised by A.K. Dewdney *Scientific American* article

International Core War Society: standards/tournaments (KOTH)

1990 So far: 6 annual tournaments, 2 new standards (ICWS ’86, ’88)

Slow decline of interest thereafter.

1994 ICWS ’94 proposed, never advanced due to lack of enthusiasm.
Core War virtual machine: MARS, corewars(1)

- VM addresses modulo SIZE, all code is PIC
- round-robin scheduler (1 opcode per process and time slice)
- forking a thread possible; #threads limited
- a cell is owned by whoever wrote to it last
- no SEGV: it’s OK to execute others’ code on your behalf
- random initial base addresses; min. distance configurable
- only instructions executable; crash on jump to a data cell
An example program: `dwarf.cw`

```assembly
L:     add 5, A
      move 0, [A]
      jump L
A:     data &A

Note: suicide potential, unless SIZE is divisible by 5.
```
Another example program: mover.cw

title "Self mover"
author "Walter Hofmann <walterh@gmx.de>"

START: move START, NEXT  # move on
NEXT: data 0
Dwarf vs. Mover!
Core War: appeal for evolutionary algorithms

- competitive environment with struggling for survival

- *Redcode* is a simple and orthogonal assembly
  - all addressing modes utilisable with all opcodes
  - all instructions exactly in the same format
Outline

- Core War
  - the game; virtual machine; assembly syntax
  - similarity to CPU testing; evolutionary programming
- $\mu$GP
- Results
Goals similarity: Core Wars & modern CPU testing

- target is an assembly-level program
- evaluation by running in a partially unknown env.
- evaluation (e.g. by syntax) impossible w/o explicit simulation
- multi-threading in MARS like speculative execution in CPUs
- Core Wars: use all opcodes aspects and weird side-effects
- memory and time are limited resources
Core War: previous EA history highlights

1991 Perry: random code to successful warriors in a few generations

1998 Terry Newton: RedMaker

2000 Martin Ankerl: Yet Another Corewar Evolver (losers replaced with modifications of winners)

   Dave Hillis: Redrace (all warriors in the population compete against all on the target hill; Valhalla and Resurrection)

2002 Blaha and Wunsch: automatic assembly-level optimization based on Core War
Outline

† Core War

• $\mu$GP
  – original $\mu$GP, deficiencies w.r.t. CPU testing
  – advanced $\mu$GP: clone scaling, delta entropy

• Results
$\mu$GP (Squillero 2003)

- based on the standard GP techniques
- assembler programs stored as trees, node $\sim$ macro(params)
- generates (by design) programs with
  - correct syntax
  - variable length
  - full exploitation of available features
- unable to reuse existing test libraries material
\[ \mu \text{GP components} \]

- evolutionary core
- instruction library (mapping genotypes to assembly phenotypes)
- external evaluator (running a simulator, feeding back the core)
Outline

† Core War

• $\mu$GP
  † original $\mu$GP, deficiencies w.r.t. CPU testing
    – advanced $\mu$GP: clone scaling, delta entropy

• Results
Advanced $\mu$GP

- assimilating existing code (coaxing into generatable)

- diversity in a test-program generation problem is critical
  - clone scaling
  - entropy fitness holes
Clone Scaling in $\mu$GP

• Before evaluating an individual, check for its clones in the pool.

• Multiply actual fitness by $S^C$

$C$ number of clones
$S$ clone-scaling factor, from 0 (no scaling) to 1 (clone extermination)

NB identical genotype might map to different phenotype (labels, unique tags)
Entropy and Delta-Entropy in $\mu$GP

We need to detect whether the amount of genetic diversity in a set of individuals is increasing or decreasing.

We’ll consider a symbol $s$ from an individual as an instance of a macro — i.e., a macro and a value of its parameters.

For the entropy computations, an individual is a set of such symbols.

$$Q = \{I_1, \ldots, I_n\} — \text{set of individuals in the population}$$
\[ \Sigma_Q = \{ s \mid \exists I : s \in I \in Q \} \quad \text{— } Q\text{-universe of symbols} \]

\[ f_{\Sigma_Q}(s) = \frac{\text{occurrences}(s)}{\sum_{s' \in \Sigma_Q} \text{occurrences}(s')} \quad \text{— frequencies} \]

Similar to the classical Shannon’s

\[ H(x) = \sum_{i=1}^{n} p(i) \log_2 \left( \frac{1}{p(i)} \right) = - \sum_{i=1}^{n} p(i) \log_2 p(i) \]

we define

\[ H(Q) = - \sum_{s \in \Sigma_Q} f_{\Sigma_Q}(s) \cdot \log \left( f_{\Sigma_Q}(s) \right) \]
Let now the individuals in the population be ordered according to their fitness, with $I_0$ being the fittest. The partial entropy of the population, measured for the fittest $x$ individuals, is defined as

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

and the *delta entropy*

$$\begin{align*}
\Delta H_0 &= PH(0) \\
\Delta H_i &= PH(i) - PH(i - 1) \quad (i > 0)
\end{align*}$$

giving a qualitative measure of the amount of new genetic information brought by each successive individual into the population.
Delta entropy as grounds for selection

High $\Delta H_i$

- $I_i$ brings effective fragments of code
- should be preserved

Low/negative $\Delta H_i$

- $I_i$ could be obtained by applying the genetical operators to some (fitter) parents
- could be discarded
Delta-Entropy Fitness Holes

• $\mu$GP selection — tournament of size $\tau$

• when 2 individuals are compared, then with probability $h$ their delta entropy values are considered instead of their fitness values

• this creates a hole in the fitness function

• fitness holes bias evolution w/o affecting the fitness calculation, promote genetic variety

• disfavour *almost identical* individuals, reduce overall code bloat
Outline

† Core War

† μGP

● Results
Experimental evaluation: Core War achievements

- finite and infinite hills
- hill core sizes (8,000, 8,192, 55,400, 800, 80)
- source code open and closed hills
- manual, evolved, and mixed hills
- minor VM and scoring variations

Powerful winning code, just 20 redcode instructions, leaves human experts at a loss during reverse engineering!
Preliminary Results on DLX/PII

Test programs hand-developed by designers enhanced to sets maximizing one of the following coverage verification metrics given:

- statement (percentage of the executable statements in the model exercised)
- branch (percentage of branch conditions evaluated to both true and false)
- condition (percentage of Boolean subexpressions evaluated to both true and false)
- expression (same, against concurrent signal assignments)
- toggle (percentage of bits having toggled both 0 to 1 and 1 to 0)
Outline

† Core War

† $\mu$GP

† Results
Conclusions

- playing Core Wars to help devise effective CPU testing programs
- for effective Core War warriors breeding, $\mu$GP was enhanced with
  - the ability to assimilate existing code
  - detect clones
  - selection mechanism for promoting diversity
- in this way, $\mu$GP-produced warriors aced the four main int’l hills (the first ever machine-written champion with such result!)
- same framework can generate CPU tests under various coverage metrics
Thank you...