The Garden: Evolving Warriors in Core Wars

David G. Andersen

August 23, 2001

Abstract

The Garden is a new architecture for evolving Core Wars programs, short, assembly-language like creatures which battle in a simulated computer environment. With an agressive direct-compilation scheme, the Garden avoids nearly all external overhead for evaluating Core Wars programs. The resulting efficiency allows us to explore and compare different evolutionary methods at a scale (thousands of warriors) and number of generations (hundreds) that is nearly an order of magnitude more in-depth than previous research. By harnessing the speed of the Garden architecture, we are able to compare a number of different evolutionary strategies (initial seeding, evaluation, and mutation methods and parameters) in a quantitative manner.

1 Introduction

The Core Wars system [4] is a simulated linear array memory computer designed originally by A. K. Dewdney in 1998. In basic core wars, two assembly language programs attempt to survive the longest in this array, by moving, replicating, and forcing the other program to execute an invalid instruction. The program which survives the most out of some number of runs is declared the "winner." Core wars programs are written in an assembly-like language called Redcode [5]. Redcode is a reasonably simple, complete assembly language, with a few addressing modes and a small set of arithmetic and data manipulation operations.

Core Wars provides an interesting middle-ground for evolving artificial life. It is far more fragile than a system like Tierra [1], but much more robust than the assembly languages found in real computer systems, and it provides a built-in fitness function (winning).

1.1 Core Wars details

It is necessary to understand a bit about the Core Wars system to understand the way in which evolved creatures behave. Core Wars takes place in a simulated computer with a linear array of memory which wraps around, usually with 8000 cells. The programs are put into the array at random locations, and take turns executing one assembly instruction each. A basic but complete set of instructions are provided, like `MOV, DIV, MOD, ADD, SUB`, as well as some flow control instructions `CMP, SEQ` (split if equal), etc.

Programs may also "fork" control with the split (`SPL`) instruction, so that they have multiple independently executing processes. These processes share time in round-robin fashion within one warrior's turn, so forking doesn't get you any more time to run your instructions - but if one process dies, then your other processes will keep running.

A warrior "wins" a battle by killing all of its opponents processes. A process dies by executing an invalid instruction, typically a `DAT` (data) instruction. Common strategies in Core Wars, then, involve placing `DAT` instructions into memory at locations inside the opponent's code. Some common, simple strategies used often in core wars include:

- **Imps** have a single instruction, `MOV 0, 1` which copies the imp to the next location in memory, which executes next. They move through memory and turn other programs into imps. They cannot win, but frequently tie.
- **Bombers** blindly "bomb" `DAT`s through memory, avoiding themselves but rapidly hitting many memory locations.
- **Scanners** examine memory for the telltale instructions of othe programs, and then overwrite large chunks of memory in that region.
- **Papers** replicate themselves into other parts of memory and have multiple copies running at a time. They disrupt other program's code when they hop in to them.

Most evolved programs are **bombers** or **papers**, two strategies which evolve quite simply and require little coordination between functional units in the program.

1.2 Evaluating warriors

There exists no official standard for how to evaluate the "fitness" of a Core Wars program, or "warrior." However, two unofficial methods have emerged: The Wilkies Benchmark, and the King of the Hill competition. In almost all Core Wars competition, warriors are awared 0 points for losing a battle, 1 point for a tie, and 3 points for winning a battle. (This helps encourage winning strategies rather than simple survival strategies).

1.2.1 The Wilkies Benchmark

The Wilkies Benchmark consists of a set of 12 hand-selected warriors from past core wars competitions. A test warrior fights 100 battles against each of these warriors, and their resulting score is averaged (divided by 12) to determine a number between 0 and 300. Most decent human coded warriors have little trouble scoring over 100 on the Wilkies benchmark, but to date, no evolved program has done so.

1.2.2 King of the Hill

King of the Hill is a semi-realtime on-line competition of Core Wars programs. Authors submit their warriors to the competition, and the new warrior fights against each warrior on the hill. From these battles, it is given a "hill score," and if its hill score is high enough, it is inserted into the hill, displacing the lowest-scored program on the hill.

In times past, there was a "beginners" hill where people frequently tested evolved warriors, but this hill no longer exists. *No* purely evovled core warrior is good enough to take even the bottom rung of the standard core wars hills, though one warrior on the standard hill has several parameters which were determined via a genetic algorithm.

2 Related Work

Other experiments in this area are have been performed, but all report only moderate success. The approaches taken thus far fall into certain categories. We can categorize the Core Wars evolution methodologies on a few axes.

2.0.3 Initial seeding

A core-wars program can be seeded completely randomly (as is done in GA-war [2]), or it can be seeded with non-functional patterns (For instance, [7] seeds the individuals with 4 initial "split" instructions to increase the modularity and robustness of the evolved individuals).

Finally, the evolution program may be seeded with fully functional warriors. One warrior in the current "King of the Hill" tournament features a human-coded warrior whose parameters (spacing, number of children, etc) were genetically evolved.

2.0.4 Mutation and reproduction methods

Change in the code can be induced by point mutations, point crossovers, multpoint crossovers, or a block-based crossover scheme (in which the code is divided up into blocks of specific size, and crossovers occur between those blocks). Each of these results in changes occuring at different granularities, and may result in different types of offspring.

Similarly, the space into which a creature may breed will affect how new code spreads. Reproduction may be localized to adjacent neighbors (as in Redmaker [8]), allowed to propagate throughout the entire pool of organisms, or constrained to a specific "pool" of organisms.

2.0.5 Evaluation methods

Evaluation can occur by internal or external competition. With external competition, each warrior in the soup is tested against an external battery of (usually human-coded) warriors, and the score that results is their fitness.

Internal competition can occur in a variety of ways. In general, warriors are tested against other evolved warriors. It can be handled in an exhaustive competition (all warriors against all warriors), a random subset competition, or a "king of the hill" style competition where the best warrior from each generation moves onto the hill. Exhaustive internal competition requires $O(N^2)$ tests, but in theory provides the best fitness score. The other mechanisms attempt to approximate the best fitness score with much lower effort.

2.1 Categorization of related work

Name	**Seeding**	**Mutation**	**Reproduction**	**Evaluation**
Perry	Hand-coded	Point, Multipoint crossover	Single generation	Internal pairs
GA-War	Random	Point, Line	Global replacement	Internal Subset
Sys4	Random	Point	Neighbor replacement	Neighbor combat
RedMaker	Random	Point, Line	Global replacement	Random global
Hillis	4 SPL at start	point, crossovers: 1-point, 1-point-block	Global replacement	External battery, Internal KOTH
Garden	Random N SPL at start Real code	Point, multipoint crossover	Global replacement	External battery, internal exhaustive internal KOTH, or combination

The initial work in this area, by John Perry [9], used only a single generation brewed from existing warriors to attempt to improve them by mutation and crossover. Unfortunately, his generations took a full day to run with a few hundred warriors, and so the work was not continued or explored from purely random warriors.

The GA-war system allows point mutations to existing code, and the insertion or deletion of random lines. It uses global replacement - warriors fight a subset of other warriors and the winners replace the losers. It and the Sys4 tool are the most "artificial life" like of all of the programs discussed herein - they both have some form of localization. Sys4 goes even farther, allowing warriors to compete only with their neighbors in a two-dimensional grid. Unfortunately, both of these programs suffer from somewhat limited mutation and evaluation capabilities.

Hillis has what is likely the most sophisticated evolution program to date. It's derived from GA-war, but allows crossover mutations, and adds a "block" granularity to the programs and their mutations: programs are started with an initial set of "SPL" instructions to the block boundaries within the program.

These blocks are then subject to block-based crossovers. (The whole warrior is also subject to single-point crossovers). Through the imposition of this amount of external structure, Hillis reports the most fit evolved warriors in the literature.

3 The Garden Architecture

The Garden is unique among Core Wars evolutionary schemes in that it does not use external invocations of the popular `pmars` core wars simulator for evaluating its warriors. Instead, the Garden incorporates the execution code from `pmars` into its own system. Programs used for evaluation in the external or internal batteries are only assembled once, and are then stored in already-assembled form in memory. Instead of writing evolved programs to files and then evolving them, the Garden assembles them directly from their internal genetic representation to `pmars`-executable warriors, and again, only performs this assembly step once per warrior per generation.

Surprisingly, though everything the Garden does is stored in memory, its storage requirements are very modest - only 2.5MB for an instance which is evolving a pool of 1000 warriors, several times larger than other systems have handled in the past.

3.1 Initial Seeding

Initial seeding in the garden can occur in one of three ways. The most basic mechanism is purely random generation of individuals. This results in a pool with a reasonably low level of fitness, though occasionally an individual will make a lucky series of SPLits or JMPs that allow it to survive a battle by pure luck.

The next mechanism emulates the modular evolution in Hillis, with a twist. Individuals are randomly generated, but their first N instructions (N is typically 4 in the experiments I've run) are `SPL` instructions to other locations in the warrior. At present, I use only random locations, and do not emulate the block-based crossover which goes along with this approach in Hillis' work, but it provides a way to determine if the benefit in [7] comes more from the block-based modularity or the simple presence of initial splits.

Finally, we can take input from whole or fractional existing warriors. These inputs can be combined in part; for instance, chunks of fractional existing warriors may be combined with 4-split initial warriors. Unfortunately, combinging entire warriors with randomly generated input simply results in the entire warriors rapidly taking over the entire pool, since their fitnesses are vastly greater than those of the evolved creatures, even after several generations.

3.2 Genetic Representation

The genetic representation used in the Garden is deliberately simplistic, and is conceptually very close to the actual phenotype of garden warriors. This was a deliberate choice to avoid adding accidental structure to Garden warriors via a biased choice of genetic representations. In addition, the slight fragility of an assembly language makes it a particularly interesting environment in which to attempt to evolve programs; as we will see later, many of the resulting programs are surprisingly robust to mutation even though they consist of unsophisticated assembly instructions.

A gene in the Garden has an instruction component, a first argument and mode, and a second argument and mode – just like the corresponding assembly instructions. However, instruction and mode representations are packed densly in the gene, so that any single bit change will always result in a valid (though perhaps nonsensical!) instruction.

Creatures consist of a fixed array of genes. While the creature may load many instructions into the Core Wars array, it does not necessarily execute these instructions; therefore, the "useful" length of an evolved Core Warrior may in fact be considerably shorter than its "observed" length. However, the extra length can frequently be useful - many human-coded opponents take up considerable extra space with dummy instructions designed to hide them from warriors which scan the memory space for opponents.

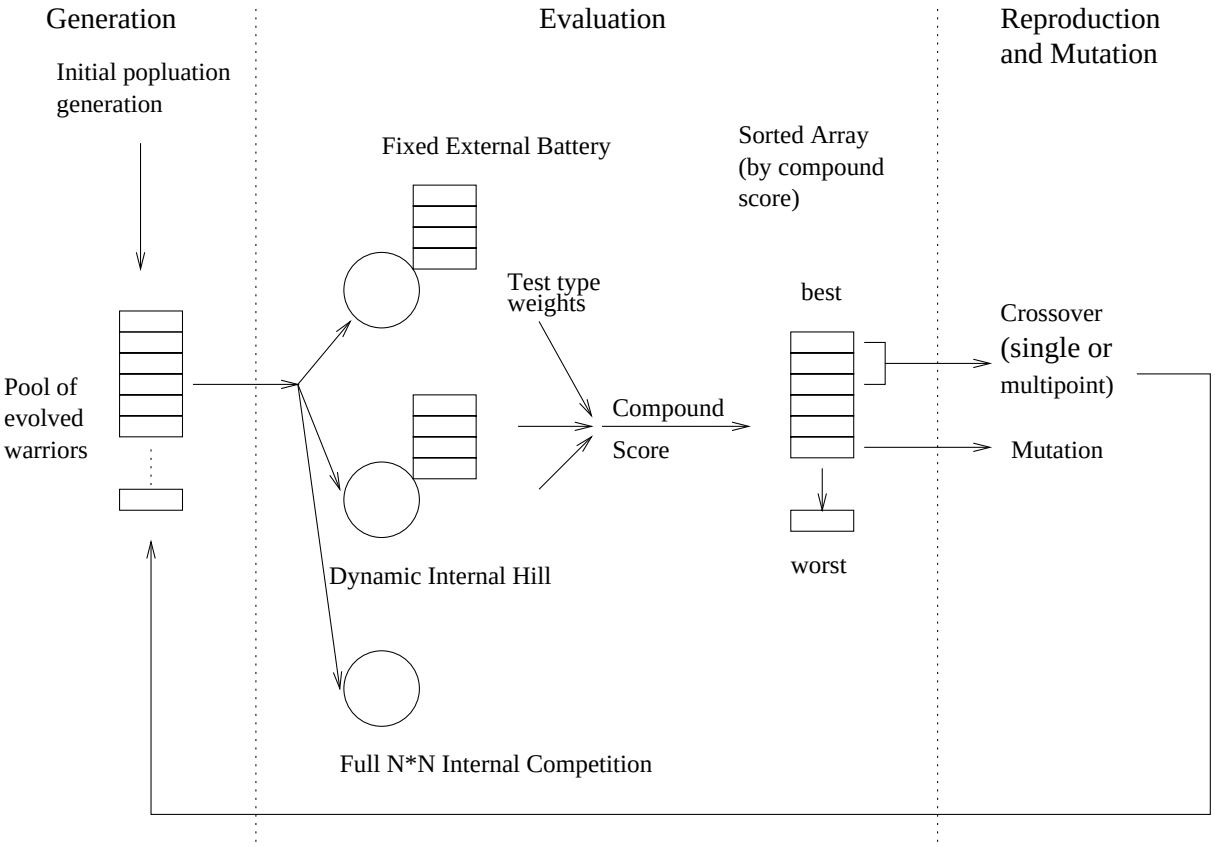

Figure 1: An overview of the Garden architecture.

3.3 Overall Architecture

The general pattern taken by the Garden is identical to most genetic evolution programs:

1. Generate an initial population
2. Evaluate the population by some metrics
3. Reproduce (with mutation) in a way that rewards more fit individuals.
4. Repeat with evaluation.

The actual structure of the Garden is shown in figure 1.

3.4 Mutation methods

Mutation in The Garden is handled by one of three methods:

- "Cosmic-ray" style point mutations to all individuals
- Point mutations upon reproduction
- Single or multipoint crossovers upon reproduction

The probabilities of each of these types of mutation is controllable, but the exact settings matter little - the random noise between runs results in far larger differences than do surprisingly large parameter changes.

Some common mutation parameters I use:

parameter	**value**
Chance of sexual reproduction (crossovers)	10%
Mutating a subgene on reproduction	$.1 - .9\%$ (1% - 4.5% for whole gene)
Cosmic ray random mutation probability	0 %

On average, these probabilities result in between a 50 and 80% chance that a child of reproduction will be mutated, but parents typically survive unscathed. (In addition, it provides a relatively gaussian distrubtion for the number of mutations a child gets, centered around three. Rare children receive up to 10 mutations, allowing for significant change, but most children receive only a small number of mutations).

3.5 Evaluation methods

The Garden has an extremely flexible evaluation infrastructure, with ideas borrowed from many existing systems. It allows a variety of both internal and external competition.

External competition is provided in an obvious "battery" form, where individual warriors fight a specified number of rounds against each warrior in the external battery. The number of battles per individual per generation is thus N (the number of warriors in the battery) * B (the number of battles per per warrior). Typical values for these numbers are 12 to 17 for different sets of benchmark warriors, and 10 to 20 battles per warrior.

Internal competition is provided by one of two methods. First, the Garden supports a "King of the Hill" style for internal competition, similar to that found in Hillis. In this mechanism, a smaller number of individuals are pulled together to form an initial "hill," and engage in an N*N competition between each other to establish their hill scores (the sum of their score against each other warrior on the hill). To evaluate warriors, they battle each member of the hill, and receive a tentative hill score, used to sort the warriors. The best warrior of the pool is then re-tested against the hill with a larger number of battles, to reduce initial placement noise. If their score is sufficiently high, they are placed on the hill, displacing the worst previous warrior on the hill. The king of the hill is a nice approximation to combat against the entire pool, because it maintains sufficient history that it doesn't always fill up with "the best" warriors.

Finally, the Garden can compete warriors against all other warriors in an exhaustive competition. Due to the $O(N^2)$ scaling of this, this is only practical for populations of 100-200 individuals.

3.6 Reproduction

Reproduction in the Garden is handled on a global basis. The most successful individuals reproduce into random spots in the population, probabalistically displacing the less-fit individuals. The actual reproduction is handled by *copying* the top F individuals (where $F \leq$ the number of individuals in the population), and starting from the *least* fit of these individuals, each warrior has K children which are inserted into the population. The best warriors reproduce last, meaning their children have a higher probability of remaining in the population; the worst warriors may not get a chance at all to breed. This results in the expected drift towards individuals with higher fitness, as we show in figure 2.

4 Performance

The Garden architecture achieves extremely high performance relative to other, similar systems. On a PII-600, we perform a single-pair check in about 15ms, including time to assemble. The fastest earlier work [7] used a Sun ultrasparc 2, and performed a single-pair check in over 400ms. The ultrasparc 2 is 2-3x slower than the PII-600, meaning that the Garden architecture is about an order of magnitude faster than previous architectures.

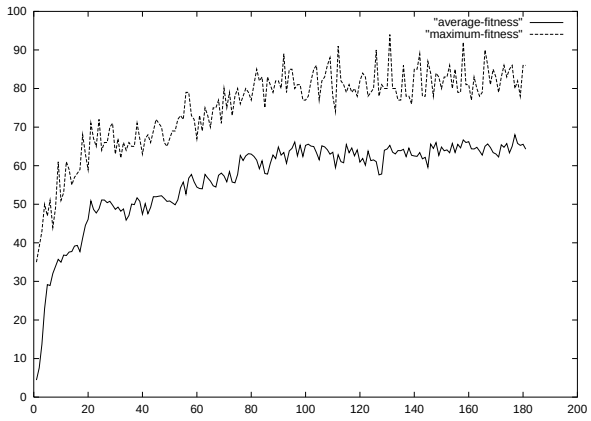

Figure 2: Average and maximum fitness by generation. Scores are for 10 iterations against the Wilkies benchmark. Note the considerable noise in the scores with such a relatively small number of tests.

The time to perform a single round battle varies with the warriors involved; the worst-case scenario is a tie, where all 8000 turns must be played out. This worst-case behavior is seen most often when competing evolved warriors against other evolved warriors, because they frequently resort to simple survival strategies and tie.

Unfortunately, this performance is still relatively slow compared to the number of executions required to evaluate a generation. The smallest realistic simulation of 50 individuals and a static battery of 12 individuals (the Wilkes benchmark) requires about 80 seconds to evaluate a generation.

Profiling experiments with the Garden show that over 98% of the execution time of the Garden is spent in actually evaluating core warrior instructions. Within this time, there are no glaring bottlenecks. Unfortunately, this means that to achieve even higher performance than the Garden, future architectures will need a vastly different Core Wars instruction emulation architecture.

5 Results

With the Garden, we evolved several hundred different warriors with moderate degrees of fitness. Figure 3 shows one early-stage evolved warrior, and compares it with an advanced scanner/bomber warrior. There are structural similarities in the bombing loops used by the evolved warrior and Rave, but Rave is much more optimized. In time, the evolved warrior's bomb loop will become tighter as extra instructions are deleted, but it will likely not evolve the advanced scanning behavior which allows Rave to perform so well.

5.1 Robustness of Genetic Algorithms

The most interesting result of writing the Garden was not a warrior: It was a lesson about the robustness of genetic algorithms. I ran experiments for about three weeks with a *major* error in the mutation procedure which resulted in a 99.99% mutation rate for individuals not subject to crossover (at that point, 33% of the individuals were instead subject to crossover). Therefore, evolution due to point mutation was virtually nil. However, the crossover functionality, combined with a great deal of random genes available in the population, the multipoint crossover functionality served to approximate point mutation surprisingly well: Individuals were able to achieve scores on my custom benchmark of about 110, compared to 120 without the bug.

This is, of course, a rather embarassing discovery to write about, but the results are rather fascinating.

```
Evolved                Ev2                    Rave

>  SPL #-52, $109      SPL #-52, $109      scan:   SUB.F incr, comp
   DIV #-48, >-5       MOV *-96, #59       comp:   CMP.I $125, $113
   SEQ $127, @-101     SUB @37, *-88               SLT.A #24, $1
   SEQ *25, #59      > SPL #-5, *5                 DJN.F $-3, <-308
   SUB @37, *-88     > DJN >-9, >-70               MOV #14, $2
>  SPL <-52, *-42      DIV #17, @-74       split:  mov.i bomb, >comp
>  MOV >-9, >109       MOD >-9, >-70       count:  djn.b split, #0
   DIV #-50, *6        DIV #-50, *8                sub.ab #BOMBLEN, comp
   SEQ >21, *17        SEQ <21, *17                jmn.b   scan,scan
   MOD @-89, @114      DAT @-4, @114        bomb:  spl.a 0,0
                                                   mov.i incr, <count
                                            incr:  dat.f <-42, <-42
```

Figure 3: Two warriors of short length, one evolved (on the left) and one high-performance hand-coded warrior, Rave. Note the similarities between the "bomb" mechanism in Rave (SPL, mov, die) and the SPL, MOV pair in the evolved warrior which has very similar functionality. The evolved warrior bombs through memory. Only the three instructions marked with a > are important to the evolved program's functionality; the rest are effectively no-ops. This program is in an early stage of evolution. Ev2 shows it 20 generations later, where its bomber loop has moved closer towards the beginning of the program, and it bombs much more quickly.

Similarly, varying the mutation rate within a reasonable range (a few mutations per organism per generation) had very little effect on the fitness of the resulting organisms, and little effect (e.g. it was in the noise between runs) on the time required to achieve a particular fitness. While these parameters may in fact have some effects, the noise in the system because of the "luck" required to evolve good warriors is far larger than the changes from different mutation rates.

5.2 Importance of instruction mix

As has been widely suggested about the Tierra system, the choice of operators is quite critical to the success, or lack thereof, of an evolutionary system. In particular, certain instructions in the Core Wars system provide a "short path" to a relatively successful strategy. The most stunning example of this are the autoincrement and autodecrement addressing modes, in which executing an instruction has a side-effect of incrementing or decrementing one of its operands.

These two operands make it much easier for a Core Wars program to perform iterative operations, because it vastly reduces the complexity of a loop required. The effects of these instructions can be seen clearly in figure 4, which was run with 1000 individuals for 30 generations. The difference in peak starting fitness is extremely noticable.

5.3 Utility of modularity: Initial SPL instructions

Hillis' idea of inserting additional SPLs at the beginning of a warrior has immediate benefits for the robustness of the resulting organism: With 4 immediate program "forks", the program is immediately much more likely to jump to a "stable" piece of code. However, without the additional structure imposed by the block-based crossover, the initial SPLs do not seem to have an overwhelming effect upon the long-range scores achievable by the evolved organisms.

In figure 5, we show a comparison of four runs of the Garden, two with initial SPL seeding, and two without. It is immediately obvious that the SPL instructions provide a higher survival rate for

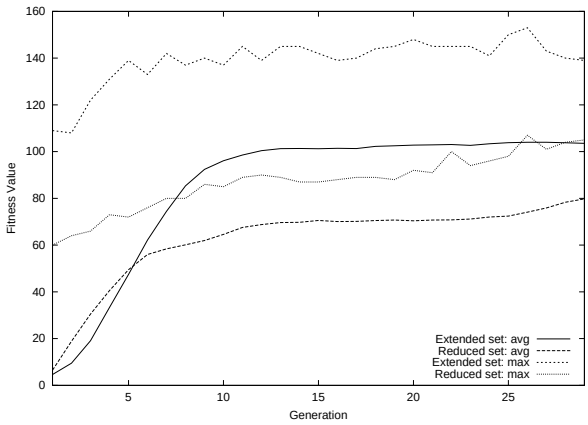

Figure 4: Avg. and Max fitness by generation for two runs of the program. The first has a normal instruction mix with no increment addressing, and the second (extended) has the preinc and postdec operators, and achieves much higher fitness.

early populations, but the data becomes too noisy due to random evolutionary leaps to draw conclusions about their long-term effects. The graphs are for populations of 1000 individuals, run for 24 hours on a quad-processor 200Mhz Pentium Pro.

The more formal structure used by Hillis, with jumps to boundaries along which crossover occured, results in considerably higher fitnesses than I achieved without this structure, even with the SPL instructions. In effect, Hillis was specifying that his creatures consisted of 4 independent modules which evolved separately. While a similar SPL-based structure often *evolved* in my experiments, the mutation was not keyed to these block units, so they did not evolve their functionality separately.

5.4 Importance of Evaluation Criteria

The evaluation criteria used are quite important, and interact with the instruction mix choices. Experiments run with full internal competition and "King of the Hill" (KOTH)-style competition *without* an external battery and no access to the extended addressing modes frequently failed to evolve anything past the "survive" stage (e.g. warriors which simply executed the same instruction over and over with no external effects). In contrast, in a run of 50 warriors bred with the extended addressing modes, a stable warrior evolved in the first generation which bombed some locations of the core. In the second generation, the "winner" was a simple single-instruction JMP loop (totally stable but boring). However, by generation 3, it had evolved into a combined "stable looper" and core-bomber, and was able to achieve a Wilkies benchmark score of 14 - quite low, but higher or equal to the scores of individuals evolved through over a CPU-week of internal competition without the extended instruction mix. This compares extremely favorably with a population of 100 warriors bred for **four days** (88 generations), the best warrior of which was completely stable, but achieved a Wilkies benchmark score of only 4.5 - far less than 10 minutes of evolution of a smaller population with a better instruction mix.

In contrast, warriors evolved using an external battery *were* able to achieve sophisticated attack strategies *without* access to the extended addressing modes. This interplay between the initial sets and the evaluation function is indicative of the different evolutionary "humps" a warrior needs to get over to survive. With an internal battery, simple survival represents a huge point gain, with additional wins occuring rarely and resulting in small gains. In contrast, a simple "sit around" strategy does *not* perform well with an external battery, so organisms must evolve more advanced strategies to survive.

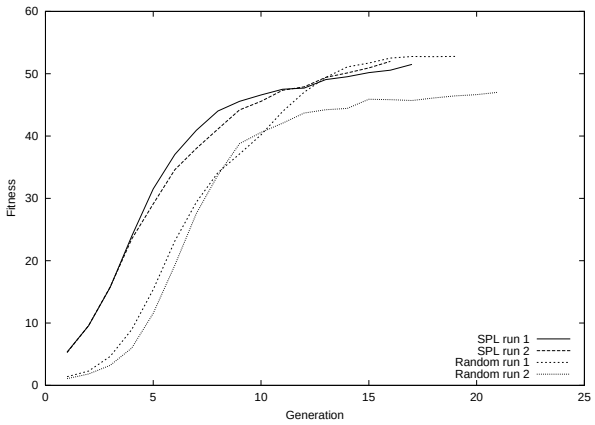

Figure 5: Average generational fitness for populations seeded with 4 SPL instructions, and without. The initial benefits of the SPL instructions are clear, but the long term benefits without block-based crossover are in doubt.

5.4.1 Directed Evolution: A single competitor

Not surprisingly, evolutionary methods can be extremely effective when targeting a single opponent. For this experiment, I picked the "rave" warrior, an advanced "scanner" type warrior which looks through the core to find its opponent, and then blasts the area where it finds it. Nearly all of the warriors evolved in either internal competition or with a larger extenal battery fail completely against Rave, losing 80-100% of the battles against it. However, a warrior evolved from purely random initialization designed to fight rave can do very well - winning 50% of the time, and tying another 30% of the time. The evolution of *this* warrior is shown in figure 6.

Interestingly enough, though this warrior was evolved specifically to fight against a single competitor, it still performs reasonably well overall, achieving a Wilkies benchmark score of 57. Its strategy, however, is simple, and its adaptations are primarily in the form of matching the length of the executing Rave code, and bombing in a pattern designed to rapidly detect Rave. The execution of the program can be seen in Figure 7.

6 Discussion

In this section, we discuss briefly the suitability of genetic evolution within the Core Wars framework. We list those features of Core Wars which facilitate evolution, and those which push the system towards the evolutionary dead-ends frequently encountered in this work and others.

On the positive side, it is possible – and even desirable – to construct organisms which are robust to changes in their instructions. Because combat in Core Wars operates by changing the code executed by an enemy (sometimes to purely invalid instructions like `DAT` and sometimes simply to unexpected code), an organism which can handle changes in its code – its gene sequence, if you will – is also likely to be more robust against certain types of attacks. Evolutionary approaches are likely to lead to organisms with this quality. In almost all runs, mutating an individual rarely causes it to immediately terminate, but instead simply reduces its fitness by some amount (e.g. its fitness does not drop immediately to zero simply by zapping a few instructions). This is likely an offshoot of the "split" based loops that frequently evolve, since they're as likely to come about as a JMP based loop.

On the downside, Core Wars is a reasonably sequential environment. While the SPL operation allows multiple processes to continue executing, the basic structure of Core Wars provides nothing which makes

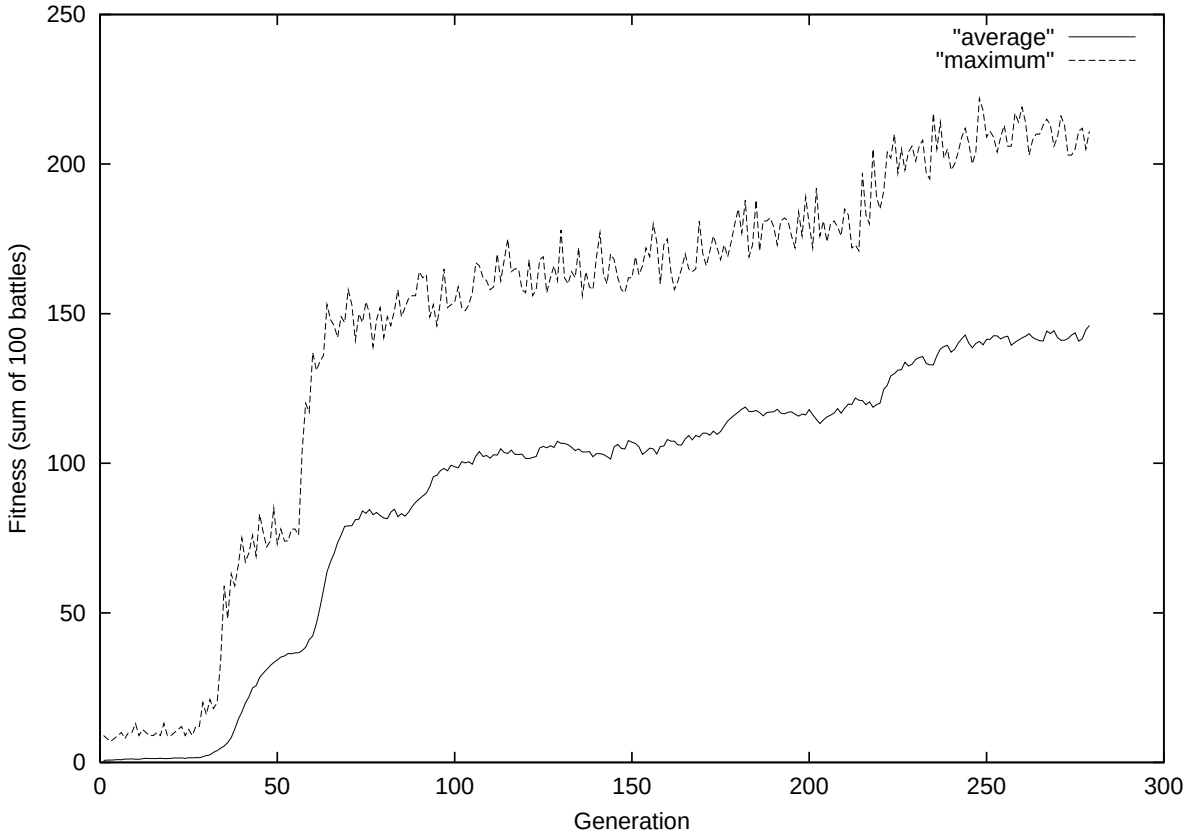

Figure 6: Directed evolution against the "rave" warrior

the execution of parallel, independent processes easy - they must be synchronized by their execution time requirements, which are predictable. Since evolutionary approaches work well with more modular systems (extending the length of an arm without inducing red-green color blindness, for instance), and successful Core Wars warriors are frequently very tightly integrated, it is not too surprising that *pure* evolutionary approaches fail to produce excellent warriors.

All current approaches to Core Wars evolution are somewhat lacking in emulating the rich diversity of environments available in the real world. From evolutionary biology [6], we know that geographic separation is a strong factor in speciation and the resulting genetic strength and diversity of the population. Additionally, without breeding barriers or other forms of balancing selection, current Core Wars frameworks frequently fail to evolve the kinds of "evolutionary arms races" [3] which fuels intense interpopulation growth.

7 Conclusions and future work

Core Wars provides an interesting middle ground for evolutionary systems. It is far more fragile than most systems designed to explore evolution, but less fragile than true computer programs. The Garden architecture for Core Wars evolution is more than an order of magnitude faster than previous evolutionary systems, and by taking advantage of this speedup – and advances in processor speed – we were able to explore several different evolutionary approaches to generating Core Wars warriors.

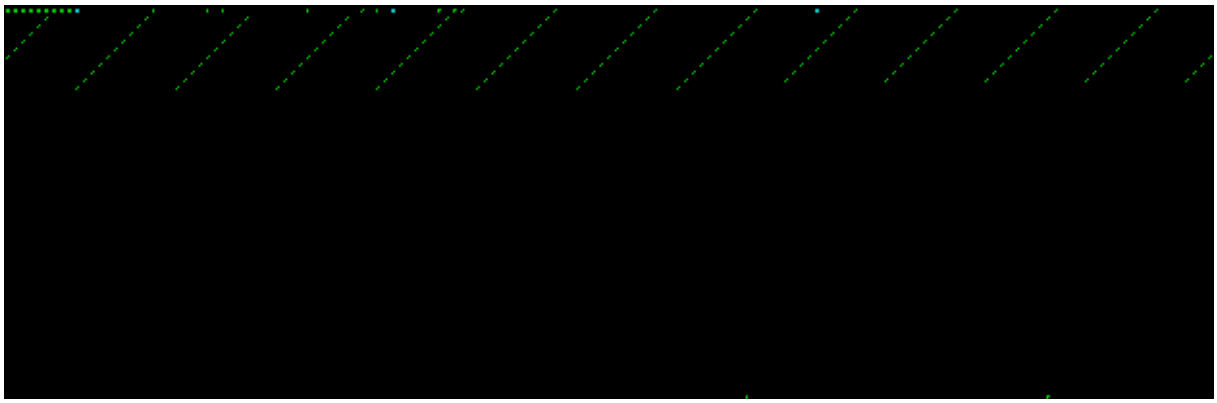

Figure 7: The middle of bombing by the anti-rave evolved warrior. The warrior is executing in the upper left hand, and the bombing is shown by the lines slanting down and to the left. The bombed lines crisscross the memory array with a pattern which kills the Rave warrior quickly.

In this work, we show the importance of the instruction mix made available to Core Wars programs: the addition of addressing modes and instructions can vastly change the fitness of the evolved warriors, and in fact makes a much larger difference than larger pools of creatures or more generations. In a similar vein, we also show the effect of encouraging a modular approach with initial SPL instructions, a middle course between pure evolution and more structure-based evolution.

As a next step, we plan to extend the Garden's evolutionary mechanisms to directly support the block-based evolution of Hillis, and produce similar side-by-side comparisons of this approach. In addition, it would be valuable to explore other mixes of instructions than those discussed herein. As the block-based approach produces the strongest known warriors – better, even, than those produced by pure evolution with nearly 100x more creature-time (population size and length of run), this comparison should prove quite enlightening, since the basic SPL approach does *not* yield huge long-term benefits on its own.

Finally, as an accidental byproduct of bugs during the development of The Garden, our results show the surprising resilience of evolutionary approaches: even when we accidentally killed 66% of the children each generation and failed to provide point mutations, evolution proceeded and resulted in entities nearly as strong as those generated by the correct algorithm, though they took longer to evolve.

References

[1] An approach to the synthesis of life. In Langton, Farmer, and Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, 1991.

[2] J. Boer. Ga war source code. `http://www.avalon.net/~jboer/projects/corewar/ga_war.c`, 1997.

[3] R. Dawkins. *The Blind Watchmaker*. W. W. Norton and Company, Inc., 1986.

[4] A. K. Dewdney. *The Armchair Universe: An Exploration of Computer Worlds*. H. Freeman, 1988.

[5] M. Durham. Annotated draft of the proposed 1994 core war standard. http://www.koth.org/info/icws94.html, 1994.

[6] D. J. Futuyma. *Evolutionary Biology*. Sinauer Associates, Inc., 1986.

[7] D. Hillis. Evolving core warriors. `http://nc5.infi.net/~wtnewton/corewar/evol/evolving.txt`, 1998.

[8] T. Newton. Evolved core-warriors. http://nc5.infi.net/~wtnewton/corewar/evol/index.html, 1998.

[9] J. Perry. Core wars genetics: The evolution of predation. http://www.cs.ucla.edu/ jperry/corewars.html, 1991.