Improving genome compiler's performance

André Dias, Penousal Machado,
Amílcar Cardoso

CISUC – Center for Informatics and
Systems, Univ. de Coimbra
Pinhal de Marrocos – Pólo II,
3030 Coimbra, Portugal

Abstract

In this paper we propose the extension of current genome compiler systems through the inclusion of two techniques, and study the effects on the performance of the system. The goal is to reduce the number of operations necessary to evaluate the individuals, hence reducing the computational weight of the evaluation step.

1 INTRODUCTION

Genetic Programming (GP) has known a great increase of utility during last years. From initial problem domains GP application has been extended to real-world problems that tend to be complex. Unfortunately, GP tends to imply a considerable computational cost.

In this paper we study the effects of the inclusion of two techniques intended to reduce to reduce the computational weight of GP applications. We will refer to these techniques as: caching of sub-trees and simplification of the individual's code.

The paper has the following structure: First we will describe the addressed problem; Then, on Section 3, introduce current approaches to reduce the computational cost of GP; On Section 4 we make an overview of our system; On Section 5 we describe the simplification technique, and on Section 6 the caching mechanism; Section 7 is dedicated to the presentation experimental results and their discussion; Finally, on Section 8, we draw some conclusions.

2 SPEED PROBLEM

GP is a computational demanding task and its application to new, complex, problem domains is limited by the inexistence of such resources.

On a typical GP run, 99% of total time is spent evaluating individuals over the training set [Nordin 94]. For each node of the individual's tree a function is called and the arguments must be copied into the stack. When the functions belonging to the function set are simple (e.g. arithmetic operations), most of the time is spent on function calling, and pushing and popping arguments to/from the stack. Thus, most of the time is spent on operations that do not directly contribute to the execution of the individual[1].

Evaluating every individual over each fitness case makes GP not viable in domains with large training sets. Using GP for programmatic compression of an image with 256×256 pixels can take 50 years if implemented in Lisp or 2 years if using C [Nordin 96].

3 STATE OF THE ART

Two different approaches have been proposed for improving the performance of the evaluation step: direct evolution of machine code [Nordin 94], and online compilation of individuals [Fukunaga 98].

In Nordin's approach [Nordin 94] there isn't an interpretation step. The evolutionary process is performed on linear individuals representing machine code operations. The genetic operators manipulate machine instructions, their opcodes, register designation and immediate values. The resulting machine code is then executed for each fitness case in the training set. This approach eliminates all the function calling overheads and represents the most effective solution for evaluation. According to [Nordin 94] speed improvements of 100 times over standard C implementations of GP can be obtained. Typically this approach is 60 times faster than a C implementation.

[1] It's important to notice that this only applies to certain domains. If the evaluation of the individuals requires the simulation of their behavior on a complex environment the time spent on the execution of the individuals may be negligible when compared with the time spent by the simulator. In these situations a genome compiler won't improve the performance significantly.

The system proposed by A. Fukunaga [Fukunaga 98] is based on a typical GP implementation, using a tree representation of the individuals. The evaluation process implies compiling the tree into machine code, which is then executed for each fitness case. The overhead of function calling and stack manipulation is reduced, since the tree is traversed only once. Improvements up to 50 times can be expected with this approach. This type of system is usually designated by Genome Compiler (GC) [Keith 94].

The performance of Nordin's and Fukunaga's systems tends to degrade when complex functions are present on function set. In this type of situation the time consumed on interpretation, function calling and passing of arguments is relatively small when compared to the time spent in function calculus (i.e. inside the function). Since the time gains result from the elimination of those steps and not from faster function calculus, and since the number of nodes to be evaluated remains the same, the speed improvement tends to be small.

4 REDUCING CODE SIZE

Our goal is to build a GC system able to give significant speed improvements even when complex functions are present in the function set.

Considering the work described [Fukunaga 98] we implemented a base GC system with two additional mechanisms for speed improvement. Both rely on reducing the number of operations necessary to evaluate the individuals, without affecting fitness calculus. These techniques don't alter the genotype of the individuals and, therefore, don't have any type of side effect on evolution.

The first of these mechanisms is caching of sub-trees. In a GP algorithm a population is generated from previous one. This means that a great amount of the genetic code, sub-programs, of the current population was already present on previous ones, and, therefore, has already been evaluated. If we are able to store the results of their evaluation, and retrieve them efficiently from memory, we can avoid recalculating these sub-trees.

In [Machado 99] a sub-tree caching mechanism was implemented on top of a conventional GP making it 17 faster. We adapted this technique to our GC system.

The simplification mechanism aims at eliminating unnecessary operations. It has been observed the emergence of genetic chunks of code, which are not expressed on individual phenotype (usually called introns) during normal GP runs. The emergence of introns seems to be linked with protection against genetic operations and its role on evolution is still under debate. The growth of introns can be exponential and represents a great drawback on system performance. In [Machado 01] the authors propose identifying and eliminating pieces of code that are not necessary for the individual's evaluation as a way to improve the system's performance.

5 CACHING

Caching is implemented as an array of pre-calculated values, representing a static version of the solution. The user defines a level for the cache. All possible sub-trees, with depth equal or lower than this level, are calculated, at the beginning of the run, for each fitness case and the result are stored on the array. During the compilation step a unique signature identifies each individual's sub-tree with depth equal or lower than the cache level. Instead of calculating the sequence of instructions represented by the tree, we generate machine code to fetch the appropriate value from the cache. Later, during execution of code, only one access to memory is needed, avoiding a sequence of instructions. If this sequence is complex enough, a significant speed improvement is achieved.

When the function set includes complex functions, the improvement introduced by cache tends to increase. Having complex and computational demanding instructions replaced by a simple memory access represents the best-case scenario for the caching mechanism.

Caching is limited to problems where input vector does not change during the run. Additionally, caching can only be applied to deterministic functions.

6 SIMPLIFICATIONS

Removing introns from compiled code represents a direct reduction on code size. In our approach the genetic code is kept intact. However, identified introns are filtered during compilation step, and are not translated to machine code, hence not being part of the code that is going to be executed.

In order to achieve speed improvements one must detect a significant amount of introns, moreover the techniques for intron detection must have low computational cost.

In our approach we resort to basic simplification techniques. For instance: if the children of a node are all constants and the function is deterministic, its result won't vary with the fitness cases, in this situation we can calculate the result once and replace the node and it's children by a constant. In some situations, although, the arguments are not constants, the result is, for instance the protected division of a variable by itself. In our system the user can specify a set of simplification rules, e.g.:

$$MUL(1,EXPR) \rightarrow EXPR$$

$$MUL(0,EXPR) \rightarrow 0$$

$$DIV(EXPR,EXPR) \rightarrow 1$$

...

When an expression that can be simplified is found, it is replaced by the simplification. This is done in a way that allows these simplifications to propagate.

The time overhead introduced by the simplification step is negligible, even when a relatively small training set is

used. It's important to notice that every improvement induced by simplifications on final machine code will be explored several times (as much as the number of fitness cases).

7 EXPERIMENTAL RESULTS

We tested the system with two symbolic regression problems: of an image and of the function $f(x)=x^9$. The symbolic regression of images was also used by Nordin [Nordin 96] to access the performance of his system, while Fukunaga used the $f(x)=x^9$ function to evaluate his. We chose the "lenna" image because it is the most popular one in the image compression field.

In the regression of $f(x)=x^9$ we considered two variants. In the 1st the f-set is composed by {+, -, *, %} and the terminal set by {X}, the functions used are predefined compiler functions. On the 2nd we used a similar f-set however, the functions are not predefined compiler ones, instead they are C function supplied by the user.

On the regression of the "lenna" image we used as f-set {+,-,*,%} and as terminal set with {X,Y}. The image regression was implemented using integer values and the other problem using floating-point values.

For all the problems we used a population size of 500, tournament selection, of size 5, a crossover rate of 90%, reproduction rate of 10%, an initial tree depth of 2-6, maximal depth of 17, and performed 30 generations. We made 10 independent runs for each configuration. To allow comparison the problems where also implemented using a standard GP shell, lil-gp [Zongker 96].

In our tests we used a Pentium III 800 Mhz computer with 256MB of RAM, running windows 2000 Pro.

7.1 RESULTS

The charts presented in this section show the improvements obtained by our GC system using cache and simplifications over a standard lil-gp implementation. The values refer to the relation between the time consumed on evaluation by lil-gp and the time spent on executing the code produced by our GC system. This relation indicates the maximum improvement possible for the approach, when compilation time is negligible compared to execution time [Fukunaga 98].

The charts on figures 1 and 2 concern the regression of the $f(x)=x^9$ function using predefined compiler functions.

The chart on figure 1 indicates that our GC system significantly improves the performance of the evaluation step, being approximately 90 times faster than lil-gp. When we use cache the speedup is even higher: 147 for a level 1 cache, 152 for a level 2, and 170 for level 3.

It also shows that level 3 caching leads to performance degradation when the number of fitness cases exceeds 500, this result was expected since caching consumes a significant amount of memory leading to trashing.

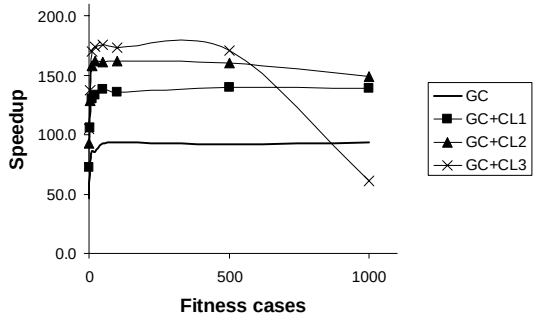

Figure 1: Speed improvements using cache only

In figure 2 we show the speed improvements achieved by the inclusion of the simplification mechanism. This induces a significant performance boost, attaining a maximum speedup of 200 over lil-gp. It's interesting to notice that, when using simplifications, the speedup achieved by caching is marginal.

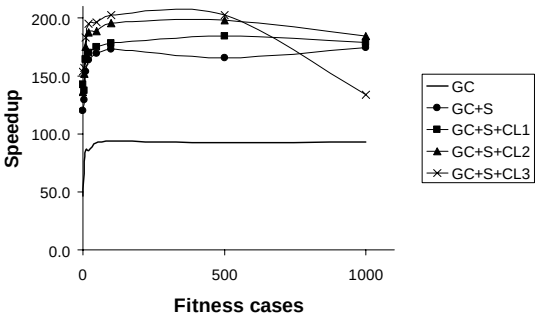

Figure 2: Speed improvements using cache and simplifications

The charts on figures 3 and 4 concern the regression of the $f(x)=x^9$ function using user defined functions.

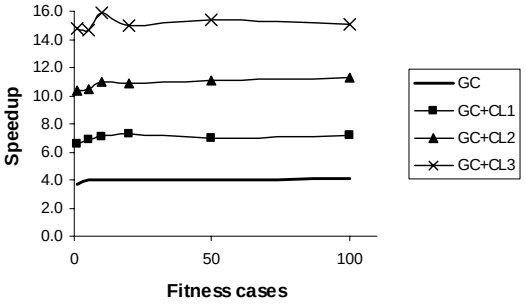

Figure 3: Cache performance using user functions

As expected the increase of performance isn't as significant as in the previous situation, due to the overhead of function calling.

The GC alone achieves a speed improvement of 4 times over lil-gp, which can be explained by the fact that the tree is traversed only once. Using a level 1 cache yields a speedup of 6, level 2 of 10, and level 3 of 16.

The chart on figure 4 relates to the speed improvements when using simplifications and caching, which leads to a maximal speedup of 25 over lil-gp, simplifications alone leading to 10.

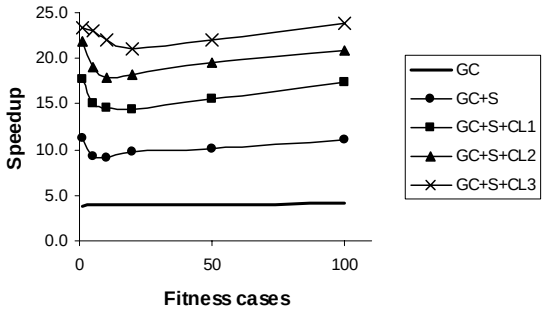

Figure 4: Speed improvements using cache and simplifications.

The charts on figures 5 and 6 concern the regression of the "lenna" image.

A maximum speed improvement of 200 was achieved using a level 2 cache. This is slightly better than the results achieved on the $f(x)=x^9$ function. This is mainly due to use of integer values instead of floating point. In these tests we decided not to use a level 3 cache since it would necessarily lead poor performance.

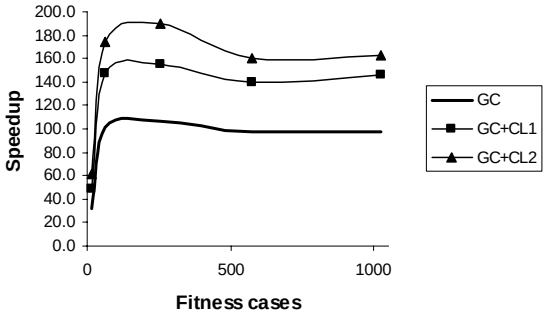

Figure 5: Speed improvements using cache, integer mode

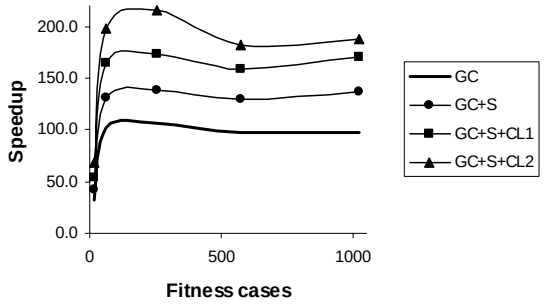

Figure 6: Speed improvements using cache and simplifications, integer mode

Using simplifications and caching simultaneously results in a maximum speed up of 215 over lil-gp.

The simultaneous use of caching and simplifications doesn't give a high improvement of performance when compared to the use of caching alone. However, it's important to notice that the simplification doesn't cause any type of memory overheads, which allows it's use even when the number of fitness cases is very high without leading to degradation of performance.

8 CONCLUSIONS

We explored the use of two well-know techniques: intron detection and removal and caching, to improve the performance of our Genome Compiler system.

Although these techniques are not new, their use as a way to improve the performance of GP, more specifically of Genome Compiler systems, is to our knowledge novel.

The experimental results clearly indicate that significant improvements can be achieved by their use. Moreover, they can be used in situations where genome compilers perform badly.

Our approach has potential to outperform systems like the ones presented in [Fukunaga 98] and [Nordin 94] in situations where the function set includes complex functions or with a high number of fitness cases.

The simplification mechanism is of particular interest since it proved to be highly effective and since it doesn't imply any type of memory overhead.

Additionally, it's important to notice that these techniques can be applied even when the functions of the function set are defined by the user, and thus do not belong to the set of primitives of the Genome Compiler.

Acknowledgments

This work was partially funded by the Portuguese Ministry of Science and Technology and FEDER under contract POSI/34756/SRI/2000

References

[Fukunaga 98] Fukunaga, A. Stechert, A. Mutz, D. *A Genome Compiler for High Performance Genetic Programming*, Genetic Programming Conference, GP'98, 1998.

[Keith 94] Keith, M. Martin, M. *Genetic programming in C++: Implementation Issues*. Advances in Genetic Programming, 1994.

[Koza 92] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. 1992.

[Machado 99] P. Machado and A. Cardoso, *Speeding up Genetic Programming*. Procs. 2nd Int. Symp. AI and Adaptive Systems, CIMAF'99, La Havana, Cuba, pp. 217-222, 1999.

[Machado 01] Machado, P., Dias, A., Cardoso, A., *GenCo – A project Report.* Proceedings of the Third International Symposium on Artificial Intelligence and Adaptive Systems (ISAS'2001), La Havana, Cuba, 2001.

[Nordin 94] Nordin, P. *A compiling genetic programming system that directly manipulates the machine-code*. Advances in Genetic Programming, 1994.

[Nordin 96] Nordin, P.; Programmatic Compression of Images and Sound; In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 345-350; MIT Press, 1996

[Zongker 96] Zongker, D. Punch, B. *lil-gp1.01 User's Manual*, 1996.