When Lisp is Faster than C

Børge Svingen
Fast Search & Transfer
bsvingen@fast.no

ABSTRACT

This paper compares the performance of the program evaluation phase of genetic programming using C and Common Lisp. A simple experiment is conducted, and the conclusion is that genetic programming implemented in Common Lisp using on-the-fly compilation of the evolved programs can be faster than an implementation in C, also when the compilation time is taken into consideration. The deciding factor is the number of times that each evolved program is evaluated.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning.

General Terms

Performance, Experimentation, Languages.

Keywords

Genetic Programming, Lisp, C, Implementation, Performance

1. INTRODUCTION

In most situations, efficient C produces faster code than Common Lisp. Common Lisp does, however, have one big advantage when it comes to performance — the ability to do on-the-fly compilation of dynamically created code. When a program generates code that is to be evaluated many times, this can lead to a significant increase in performance.

One such situation is the implementation of genetic programming ([5]). Most of the processing power is here spent on evaluating the programs in the population, so the efficiency with which that is done is therefore vital.

When genetic programming is implemented in C or C++ ([2, 8, 4, 7]), some data structure is typically used to represent the programs, and this data structure is then traversed in order to do fitness evaluation. In Common Lisp, it is possible to represent the evolved programs as anonymous functions — Lambda expressions — which are then compiled before they are evaluated.

This paper performs an experiment in order to compare the performance of C and Common Lisp on a simple symbolic regression problem. There are two obvious parameters that influence how beneficial dynamic compilation using Common Lisp will be: The compilation time, and the number of times that each program is evaluated. Similar comparisons have been done before ([1]), the purpose here is to expand upon this by looking at how these two factors influence the relative performance.

2. THE EXPERIMENT

Since the focus of this experiment is on the evaluation phase of genetic programming, only the actual evaluations will be performed. Therefore, a number of random programs will be created, and timed evaluations will be carried out using both Common Lisp and C. Everything is implemented as straight forward as possible, in order to reduce possible error sources.

The chosen problem domain is symbolic regression on a single variable, meaning that the evaluation will consist of applying the evolved programs to a number of variable values. In a complete genetic programming settings, the results of these calculations would then be used to calculate program fitness.

The function set consists of the arithmetic operations addition and subtraction, and the terminal set consists of the variable x and the ephemeral constants $-5 \leq n \leq 5, n \in N$. x is given 5 times the selection probability of any single integer.

All created programs are full trees, and three separate runs are performed, with tree depths of 4, 6 and 8, respectively. In each experiment a population of 1000 programs of the specified depth is created. Furthermore, sets of random values $0 \leq n \leq 9, n \in N$ are created on which the programs should be evaluated. A total of 25 such sets are created, consisting of from 1000 to 25000 random values.

Finally, all the programs in the populations of each depth are evaluated for all the value sets using both C and Common Lisp. The implementations are kept as simple as possible — in the C version the programs are represented by pointer trees that are traversed as they are evaluated, while in the Common Lisp version a compiled anonymous lambda function is used.

3. RESULTS

For each of the three tree depths, the evaluations are timed for each of the 25 value sets. The measured values consist of the evaluation time for the C version, the evaluation time for the Common Lisp version, and the compilation time for the Common Lisp version.

The results are shown in Figures 1 and 2 (the results for programs of depth 6 are not shown). In all cases, C is faster

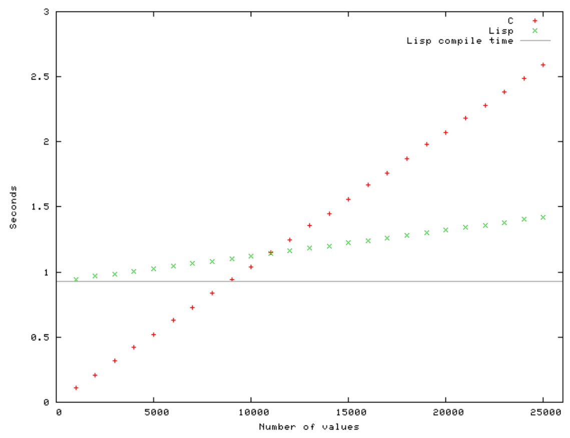

Figure 1: Performance of Programs of Depth 4

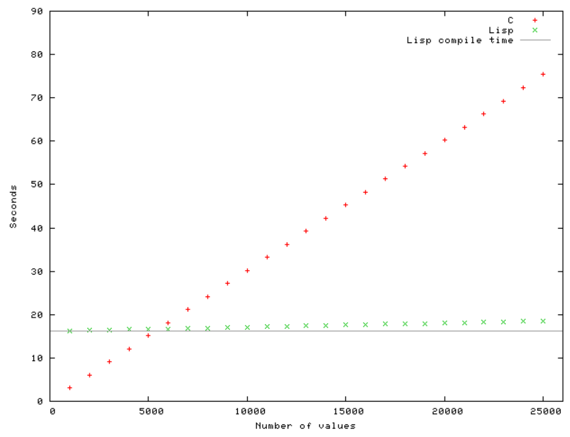

Figure 2: Performance of Programs of Depth 8

for small value sets, while Common Lisp is faster for large value sets.

The dotted curves in the figures show the measured evaluation time for each of the value sets for the C and the Common Lisp version, respectively. The numbers for the Common Lisp version include the compilation time, which is also indicated separately by the continuous line.

To start with Figure 1, for programs of depth 4 , the compilation time is 0.93 seconds. The point where Common Lisp gets the advantage is at between 10000 and 11000 values. At 25000 values, Common Lisp is almost 1.8 times as fast as the C version, and the compilation time is 65% of the total evaluation time.

In Figure 2, for programs of depth 8, Common Lisp passes C at between 5000 and 6000 values, and the compilation time is 16.2 seconds. At 25000 values, Common Lisp is 4.1 times as fast as the C version, and the compilation time is 88% of the total evaluation time.

4. DISCUSSION

The actual numbers obtained by this experiment are obviously both implementation and problem dependent. It should be possible, however, to draw some generic qualitative conclusions.

The most important result from these experiments is that Common Lisp may be faster than C for a high number of evaluations, while the situation is the opposite for low numbers of evaluations. This is not unexpected, since the compilation time must be compensated for — in all three cases, the compilation time makes up most of the total Common Lisp evaluation time.

Secondly, it is clear that the advantage of Common Lisp can be very large for many evaluations, but also that the required number of evaluations is fairly high, for many cases too high.

Third, the larger the programs to be evaluated are, the bigger the advantage for Common Lisp. This is not an obvious result, since compilation time is not normally linear with program size. Also, this is particularly interesting due to the fact that for larger programs the compilation time takes up an increasing portion of the total evaluation time.

When ADFs ([6]) are used, they may be compiled separately from the programs using them. The compiled version of unchanged programs could also be kept between generations. Both of these measures will decrease the total compile time, and lead to a bigger advantage for Common Lisp.

Although it is clear that Common Lisp may be considerably faster than C, future work should examine things in more detail, both for other types of problems, and with different implementations. A comparison with approaches that dynamically compile C code by calling external compilers ([9, 3]) would be another interesting direction for future work.

5. REFERENCES

[1] K. R. Anderson. Courage in profiling. Technical report, BBN, 28 July 1994.

[2] M. Brameier, W. Kantschik, P. Dittrich, and W. Banzhaf. SYSGP – A C++ library of different GP variants. Technical Report CI-98/48, Collaborative Research Center 531, University of Dortmund, Germany, 1998.

[3] K. Cranmer and R. S. Bowman. PhysicsGP: A genetic programming approach to event selection, Feb. 05 2004. Comment: 16 pages 9 figures, 1 table. Submitted to Comput. Phys. Commun.

[4] M. J. Keith and M. C. Martin. Genetic programming in C++: Implementation issues. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[6] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[7] A. Singleton. Genetic programming with C++. *BYTE*, pages 171–176, Feb. 1994.

[8] B. Svingen. GP++ an introduction. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 231–239, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.

[9] T. Weinbrenner. Genetic programming techniques applied to measurement data. Diploma Thesis, Feb. 1997.