Courage in Profiles

Kenneth R. Anderson*

July 28, 1994

This paper challenges the myth that Lisp programs are slow, and C programs are fast, by comparing two 2,000 line programs. The two programs are popular machine learning programs, Fahlman's Cascade Correlation program, CASCOR1, and John Koza's Genetic Programming program, referred to here as GP. CASCOR1 is floating point intensive, while GP is floating point and structure manipulation intensive.

These programs not only provide medium sized interlanguage benchmarks, but also provide an opportunity to study performance tuning. One man-day of tuning was done on each of the Lisp versions of the programs, using commonly available tools such as compiler advice and profilers. Much less than 10% of the code was changed.

The performance of CASCOR1 (already highly optimized) was improved by almost 40%, comparable to or slightly faster than the C version. The main difference between Lisp and C is the quality of compiling a dot product loop.

GP (not optimized, but well written pedagogical software) was improved by a factor of 30, and is over twice as fast as the C version which was advertised as "highly optimized". Lisp's advantages are due, at least in part, to automatic garbage collection and dynamic typing.

In real programs, performance has more to do with choice of algorithm, and quality of compiler than on choice of language.

1 Performance Myths Persist

It is a persistent myth that low level languages, such as C, are faster than higher level languages, such as Lisp or Smalltalk. The argument is basically

*KAnderson@BBN.com, BBN STC, 10 Moulton St., Mail Stop 6/4c, Cambridge, MA, 02138

that C is really just a cleverly disguised, easy-to-use, assembly language. You have plenty of low level access to the machine when you need it. You can almost count the machine instructions just by looking at the code.

Actually, this is much less true with the current RISC hardware than it was, for example, with PDP-11's. Today's hardware can make efficiency issues difficult to expose. For example, [BKV] points out that experts (people who write compilers and operating systems) can differ by a factor of 5 on their performance estimates of C constructs.

This myth has two unstated premises. One is that you can easily construct any additional support you might need, that the language doesn't provide, either by getting a library, or writing software yourself. This remains to be seen. While there may be many libraries, they may not all be compatible. Also, creating a good library is almost as hard as creating a new language [BS].

Another premise is that higher level features conspire in a way that tends to make programs slower. This need not be the case. Current Common Lisp has performance that is as good as C [G91], [F91], or Fortran [FBWR].

Modern generational garbage collectors are quite fast, and can even be used in real time applications [AEL] [B78]. If enough memory is available, garbage collection becomes almost free [AWA]. Automatic garbage collection is even quite competitive with hand tailored application specific collectors, and can even be faster[Zorn].

Language features can combine in ways to provide unexpected performance advantages. For example, in Lisp one can declare numbers to be a certain type and in certain ranges, such as `(single-float 0.0 *)`, a single floating point number greater than or equal to zero. In C, one can only declare a number to be a certain type, such as `int`, or `float`. Finer grained declarations can allow a compiler to generate more efficient code, such as using a hardware `sqrt` function without a check for a possible complex result.

Such small advantages might not seem important, but they can add up. For example, Pare and Lee [PL94] report that replacing a single recursion with iteration improved the performance of a C program by 7% (10% in the final system). Such optimization is provided automatically by compilers that treat tail recursive calls as iteration.

2 Good interlanguage Benchmarks are Hard to Find

To quantify the effects that various language features have on performance of real programs, one needs a set of high quality comparable benchmarks in each language. Such benchmarks have been extremely hard to come by.

Most benchmarks, such as the Gabriel benchmarks [G85], are relatively short which makes them relatively easy to port and to analyze. However, they have been criticized for not being representative of real programs, especially when they suggest a flaw in a particular language implementation.

Benchmarks may not always represent current programming practice. For example, several of the Gabriel benchmarks have been rewritten and made many times faster [B92A] [B92B] [B92C], [B94] [F91]. Presumably, such improvements would be made to real production quality software as well. Such vast speed improvements make the new versions useless as benchmarks.

The act of translating a program from one language to another can have profound effects on its performance. For example, when porting a program to a multiprocessor, sometimes the uniprocessor version of the program becomes faster because of insights gained in the port. Since C requires detailed type information while Lisp doesn't, a straightforward translation from Lisp to C could make the C program faster. For example, Aletan reports a 10 to 100 time speed up when porting 3 programs from Lisp to C [A89]. In a more even-handed port the two languages perform equally well [P94].

Library routines, such as a sort routine, might be a good source of benchmark material since these are available in most languages, and are building blocks of real programs. However, care must be taken to ensure that two implementations of the library routine are comparable. For example, `quicksort` is a widely used sorting algorithm. Bentley [BM] has shown that several improvements to the basic algorithm are possible, and that a widely used Berkeley 4.2 BSD implementation has $O(n^2)$ behavior when sorting sequences containing only a few distinct values (such as alternating 0's and 1's) rather than the expected $O(n * log(n))$ behavior. Sort routines provided by some Lisp implementations have this problem while others do not. In fact, in some cases sorting a list can be faster than sorting a vector because the list sorting algorithm avoids this problem, while the array version does not.

Two programs have been identified that seem to meet the minimum

requirements for a C/Lisp benchmark suite. They are medium sized benchmarks (2,000 lines of Lisp) though still quite small by programming standards. However, they are typical of components of larger programs, and make a reasonably wide use of the underlying language. The remainder of this paper presents a performance analysis of these programs.

3 Both Benchmarks are Machine Learning Algorithms

The performance of Lisp and C implementations of two programs, CASCOR1 and GP was analyzed on a Sun Sparc 10/30 with 190 Mbytes of memory, running the Sun OS 4.1.3 operating system. The Lisp and C implementations of each will be referred to as CASCOR1-LISP, CASCOR1-C, and GP-LISP and GP-C, respectively. The original versions of the software used in these study can be FTP'ed from the Internet as follows.

```
CASCOR1-LISP
 Author: Scott Fahlman
 FTP: pt.cs.cmu.edu:afs/cs/project/connect/code/cascor1.lisp

CASCOR1-C
 Author: R. Scott Crowder, III
 FTP: pt.cs.cmu.edu:afs/cs/project/connect/code/cascor-v1.0.3.shar

GP-LISP
 Author: John Koza
 FTP: ftp.aic.nrl.navy.mil:pub/galist/src/ga/koza.gp

GP-C
 Author: Walter Alden Tackett and Aviram Carmi
 FTP: ftp.aic.nrl.navy.mil:pub/galist/src/ga/sgpc1.01.tar.Z
```

The following subsections describe each program.

3.1 Cascade Correlation Constructs Neural Networks

CASCOR1 is an implementation of Scott Fahlman's Cascade Correlation method for constructing a neural network [F90]. A neural network is a

function that maps a vector of inputs to a vector of outputs. This function is constructed out of a simple computational unit called a "neuron".

A neuron is a function that takes a vector of inputs and produces an output value. For example, the output of a neuron might be computed as (first in Lisp, and then in C):

```
;;; Lisp:
(defun activation(sum)
  (- (/ 1.0 (+ 1.0 (exp (- sum)))) 0.5))

(defun output (input weight)
  (activation (dot input weight)))

/* C */
double activation (double sum)
 {return 1.0/(1.0 + exp(-sum)) - 0.5;}

double output (double * input, double * weight)
 {return activation(dot(input, weight));}
```

Where `input` is a vector of input values, `weight` is the neuron's weight vector, `dot` is the vector dot product, and `activation` is the neuron's activation function. Here, a nonlinear activation function is used, as is typically done when a neural network is applied to a classification task.

Given a set of training data, consisting of input/output examples, the CASCOR algorithm constructs a neural network by adding one neuron at a time. The first neuron is connected to all the input variables and a constant 1.0, used as a bias term. Each additional neuron is connected to all the input variables and to the output of the previous neurons.

The weights of the added neuron is chosen to maximize the magnitude of the correlation between the new neuron's output and the residual error in the output of the current network. To compute these weights, several candidate neurons are trained in parallel, using gradient ascent, and the best candidate is added to the network.

Each step of gradient ascent is referred to as an "epoch". During each epoch, a pass over each example is made to compute the gradient for each candidate neuron. For each example/candidate pair one computes a dot

product (of length equal to the number of previous neurons), and the derivative of the activation is computed using `activation`. This epoch loop accounts for about 85% of the computation.

The Lisp and C programs are very similar. A neuron is represented as a floating point array of its weights. The type of Lisp array used is `(simple-array single-float (*))` which is like the `float[]` declaration in C. These arrays allow floating point numbers to be read or written to them without floating point consing. They are essentially the same size as C arrays created with `calloc`.

The main difference between the programs is that the Lisp version inlines five functions (`activation`, `activation-prime`, `output-function`, `output-prime`, `quickprop-update`) to reduce floating point number consing across function call boundaries. Lisp compilers can avoid consing floating point number internal to computations. However, a function that returns a floating point number must cons that number, even if it is used as a temporary value [1]. Inlining these function allows the compiler to "see" that they are temporary and not cons them.

3.2 Genetic Programming Breeds Software

The GP benchmark programs are based on software provided in John Koza's book, "Genetic Programming"[Koza]. Genetic programming is a genetic algorithm with s-expressions representing the genetic material.

The basic idea of genetic programming is to evolve a program that accomplishes a task. One starts with an initial set of randomly generated programs (s-expressions) called a "population". A task-specific fitness function measures how well a specific program performs the task. The population is allowed to evolve over many generations as follows: The population members of the next generation are computed from the current population using genetic operators such as mutation and crossover. The mutation operator takes a program and randomly removes a program fragment and replaces it with a randomly generated program fragment. As it makes a relatively small local change to a program, it acts like a random walk through the solution space. The crossover operator operates on two programs, referred to as "parents" to produce two new programs. Each parent is split into two random pieces. Each new program is constructed from one piece of each

[1] Allegro Common Lisp 4.2 provides a facility to declare functions in such a way that float consing can be avoided

parent. The goal of crossover is for good features of two parents to combine to produce a higher performing individual.

Individuals are chosen for mutation and crossover based on their fitness, so that programs of higher fitness are more likely to be involved in producing offspring for the next generation. Thus the average fitness of the population tends to improve over time.

Since Lisp programs are represented as trees, they are a natural representation for these programs. The mutation and crossover operations are simply operations on these trees.

To make genetic programming problems more tractable, a problem-specific subset of Lisp is used. For example, our benchmark uses Koza's `REGRESSION` problem. In this problem, one is given a set of x,y data pairs and the objective is to compute a program that computes a value for y given a value of x. The data comes from a quadratic function so only the binary operators `+`, `-`, `*`, and `/`, the symbol `x`, and random floating point numbers are required to construct trial programs.

The main differences between the Lisp and C versions are in the representation and evaluation of programs in the population. GP-LISP represents programs as s-expressions – a list of symbols, numbers, or other s-expressions. This is a natural representation of programs in Lisp. The s-expressions are generated using `cons`. The built-in function `eval` is used to evaluate the s-expression in an environment where its free variable symbols are bound globally.

In GP-C, the program is also represented as trees, but tree nodes are 6 word `structs`. There are two types of tree nodes. Terminal nodes represent random floating point numbers and symbols. Function nodes represent function applications. Function nodes take two additional arrays, one to hold the expression representing each argument, and one to hold the value of each argument during evaluation.

4 Lisp Source Code is Compact

Figure 1 shows some statistics measured from the text of the four programs.

`Total Lines` is the number of newline characters. `Lines of Code` is `Total Lines` with comments and blank lines removed. Lines with single non-blank characters were also removed. In both languages, roughly half the software is comments.

The number of functions, macros, and global variables is provided to

	CASCOR1		GP	
	Lisp	C	Lisp	C
Total Lines	1467	3290	2029	5147
Lines of Code	799	1827	1335	2456
Functions	32	74	87	113
Macros	11	42	7	70
Globals	65	90	24	6
Lines/function	24.9	24.7	15.3	21.7

Figure 1: Software Statistics

give some sense of the complexity of the programs. Macros tend to be used for slightly different purposes in C than in Lisp. The C programs use macros (`#define`'s) mostly to define constants. CASCOR1-C does not use macros to inline functions, while GP-C does. CASCOR1-LISP uses macros to inline functions as well as to provide declarations for numerical operations. This reflects an older programming style that is unnecessary in modern Lisps[F94]. GP-LISP makes almost no use of macros.

Both languages have roughly the same number of lines of code per function. The GP programs have slightly fewer lines of code per function on average than the CASCOR1 programs. This is at least in part because recursion (on lists) is more commonly used in GP than in CASCOR1 where iteration (on vectors) dominates.

The Lisp version of the programs are considerably smaller than the C programs, both in terms of the number of lines of code and number of functions. The C programs require about twice as many lines of code and 30 to 100% more functions. This is because C functions must have their arguments redundantly declared, while Lisp functions don't. Also the C programs must provide more support for input/output and other facilities which are provided by Lisp as part of the language (such as a command loop and garbage collection). This ratio may go down somewhat as a program becomes larger, but this difference in program size suggests that Lisp programs may be easier to develop and maintain than their C counterparts.

5 Profiling Can Improve Performance

Profiling is an important activity for fine tuning the performance of an existing program. This is particularly true in Lisp because the performance implications of certain language features are not always obvious. For example, Lisp provides many functions for operating on elements of a sequence, such as `map`, `member`, `position`, and `find`, which take several optional keyword arguments. While these functions may be inlined when given appropriate arguments, it isn't always clear when they are. Thus performance can vary based on which sequence function is used.

Benchmarking provides an important opportunity to study profiling. The reason for this is that, as Gabriel has pointed out [G85], one should not quote benchmark measurements without also analyzing what one is measuring. Profiling is a perfect tool to aid this analysis.

In the process of studying the benchmark programs, one man-day of profile based optimizations were applied to each Lisp program. A backtrace profiler was used. Such a profiler periodically records the state of the execution stack (or a portion of it) during the execution of the program. It then produces a report on the fraction of the time spent in each function in each particular execution context.

Advice generated by the compiler was also used to indicate where a change, such as adding a declaration or rewriting some software, might be advantageous. Such advice is commonly provided by Lisp compilers. The CMU Common Lisp compiler even provides a relative estimate of the cost of not following its advice.

The following subsections describe this process.

5.1 CASCOR1 Tries Too Hard to be Fast

The benchmark problem used for the CASCOR1 programs is the classic "two spiral problem" that is provided with both the Lisp and C versions. In this problem, one is given two sets of 2-D points representing two classes of data. The points of each class are laid out in interlocking spirals. CASCOR1 must construct a neural network that assigns any 2-D point to the appropriate class. This is a relatively hard problem, but CASCOR1 solves it easily less than 20 neurons.

CASCOR1-Lisp was already highly optimized, so it was expected that profiling would reveal relatively few additional opportunities for optimization.

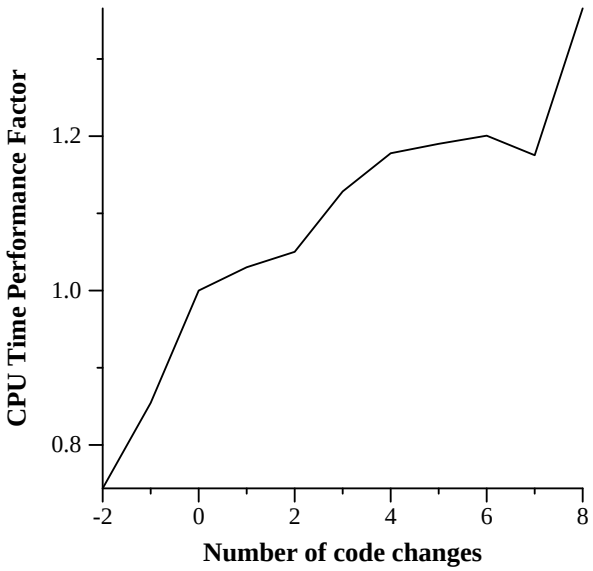

Figure 2: Time Performance Improvement for CASCOR1-LISP

Figure 2 shows the results of a series of optimizations suggested by profiling. To make this graph, the optimizations were applied sequentially to the software, and the resulting performance was measured. The figure shows the resulting performance factor, defined as the ratio of the time of the original benchmark time to the time of the optimized benchmark.

To see the effect of the inlined functions, the first two changes removed the `inline` declaration from first one, and then both of the functions `activation-prime` (essentially a subtraction and a multiplication), and `activation`. These two changes are plotted to the left of 0 on the x axis to better reflect the order in which they might be made in the course of normal optimization.

Not inlining each function resulted in a huge increase in floating point consing (75 and 122 megabytes respectively). Each change lead to about a 17% drop in performance, thus these inline decarations are extremely valuable.

The first small rise in performance (changes 1 to 3) is due to replacing `defvar` declarations with `defconstant` declarations. The major effect of this is to allow Lisp to eliminate several tests in the inner loop and the corresponding dead code.

To some extent, CASCOR1-Lisp tried too hard to be fast. Several inner

loops reference global variables. Each global variable reference takes two load instructions. Pulling these references out of the loop made them accessible as registers and lead to almost a 20% improvement (changes 4 to 8).

The final change was to replace the call to the foreign function `exp` in the function `activation` with a call to a Lisp version of `exp`. This led to a final cumulative improvement of 37%. Execution time went from 37.88 seconds to 27.74 seconds.

The original software was declared with the highest optimization settings, `(optimize (speed 3) (safety 0))` this has the effect of removing function argument counting overhead (typically two instructions) for each function call. Since crucial inner loop functions were already inlined, this optimization accounts for only a 0.6% (less than 1% in the final version) improvement in performance, and thus is not very important.

Unfortunately the Lisp and C programs are not directly comparable because different random number generators are used to set the initial set of weights. Thus each program would take a different number of epochs each time the network is grown by one neuron, and might produce a different number of neurons. However, the amount of time per epoch in the function `train-inputs` is comparable. This function accounts for 85% of the time of the benchmark.

Figure 3 shows the time per training epoch versus the number of neurons currently in the network[2]. The area under each curve is essentially the total runtime, so a lower curve and shallower slope indicate higher performance.

The solid lines are for `gcc`, the GNU C compiler, for highest and lowest optimizations settings. The dashed and dotted lines are for 3 Lisps (Allegro 4.2, Lucid 4.1, and CMU 17c). For each Lisp, two lines are plotted. The thicker one corresponds to the original CASCOR1-LISP software, and the thinner one corresponds to the optimized version described above.

Two of the original Lisp lines, Lucid and CMU, fall between the unoptimized and optimized `gcc` curves, while the Allegro falls mainly above the unoptimized one. The reason for this is that Allegro Common Lisp does not honor `inline` declarations, and thus is impacted by the floating point consing described earlier.

After the optimizations described above, the performance for the 3 Lisp implementations are similar to the optimized C curve. The corresponding intercepts and slopes for the lines are show in Figure 4. The slope is con-

[2]Each point in each curve is a median of 3 runs.

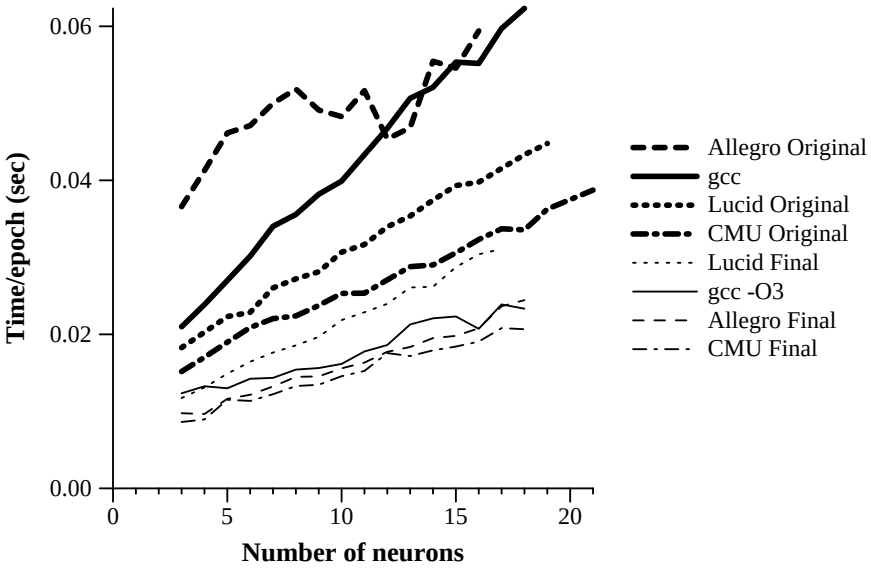

Figure 3: CASCOR1 Benchmark

trolled by the time of two dot product loops, and thus influenced by the quality of the compiled code. The number of instructions in the dot product loop is shown in the third column, labeled "NDOT" in the figure. All the Lisps do a bit worse than the optimized C indicating one or more optimization are not being applied in this case. This does not reflect the quality of the compiler in general, only the optimization of this particular loop.

5.2 GP-LISP Has Vast Performance Potential

The benchmark used for the GP program was the REGRESSION benchmark provided by both the Lisp and C versions of the program. This benchmark starts with 10 x,y pairs from the equation $y = x^2/2$ and evolves a program that best matches this equation in the least squares sense. The benchmark was modified to terminate after a fixed number of generations, rather than terminate early if a perfect solution was found.

It is generally a good idea to use a suite of benchmark parameter settings or input data that stress different parts of a program differently [BM]. Thus two versions of this benchmark were used. One ran for 20 generations with a population size of 100 individuals, while the other ran for 40 generations with a population size of 50. Both of these versions did roughly the same

```
Intercept   Slope    NDOT
 (msec)  (msec/unit)
 38.70     1.170         Allegro Original
 13.30     2.750      26 gcc
 13.60     1.660         Lucid original
 12.60     1.230         CMU original
  9.06     0.795       7 gcc -03
  7.64     1.380      14 Lucid optimized
  6.74     0.909      10 Allegro optimized
  6.53     0.803       9 CMU Optimized
```

Figure 4: `train-inputs` Line fit, intercepts and slopes

amount of work, but distributed the load between inner and outer loops differently.

GP-LISP was written to accompany Koza's book. It was written for clarity and portability, not for performance. Thus it was expected that profiling could provide substantial benefit. In fact, GP-C, was written in part to overcome the performance problems in GP-Lisp. It is advertised as being 20 to 50 times faster than the Lisp version.

The space and time performance results from profiling are shown in Figures 5 and 6.

Since evaluating programs is at the core of the algorithm, the speed of `eval` is crucial. While it is convenient to use Lisp's `eval`, Koza provides a faster specialized version. This version improved performance by a factor of three.

After that, profiling uncovered several surprises. It turned out the input data points were rational numbers. Replacing them with floats, produced an additional 25% improvement. Also random number generation was very slow and consful. In fact, a specialized version was provided for Lucid Common Lisp, which made performance even worse. A better random number generator improved performance by an additional factor of two.

Optimizations in the central part of the figure (changes 6 to 22) were suggested by inspecting the profiling results for potential optimization candidates. For example, if a routine spent a lot of time in `aref`, then a declaration might help. Declarations were only added to routines for which

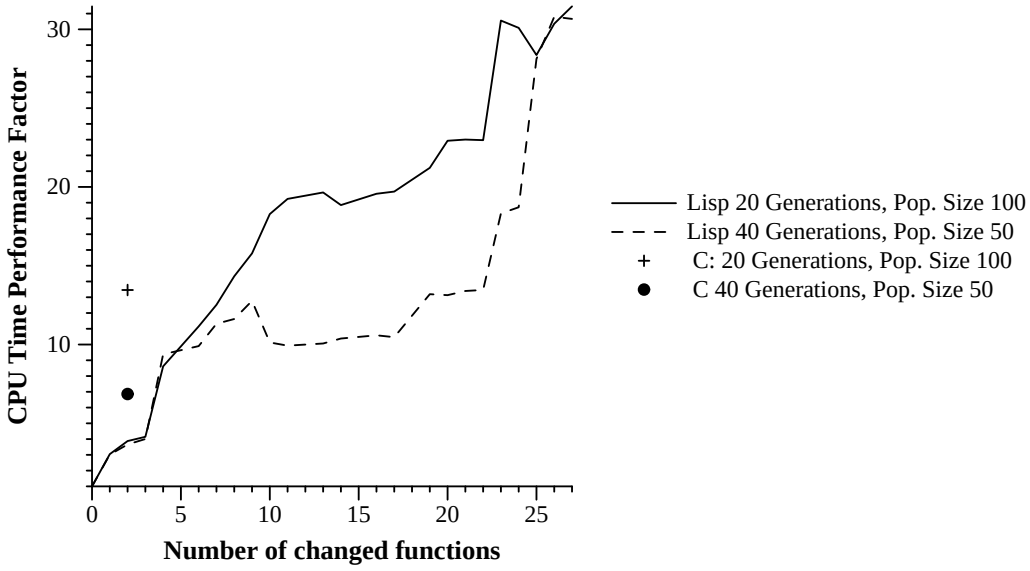

Figure 5: Time Performance Improvement of GP-LISP

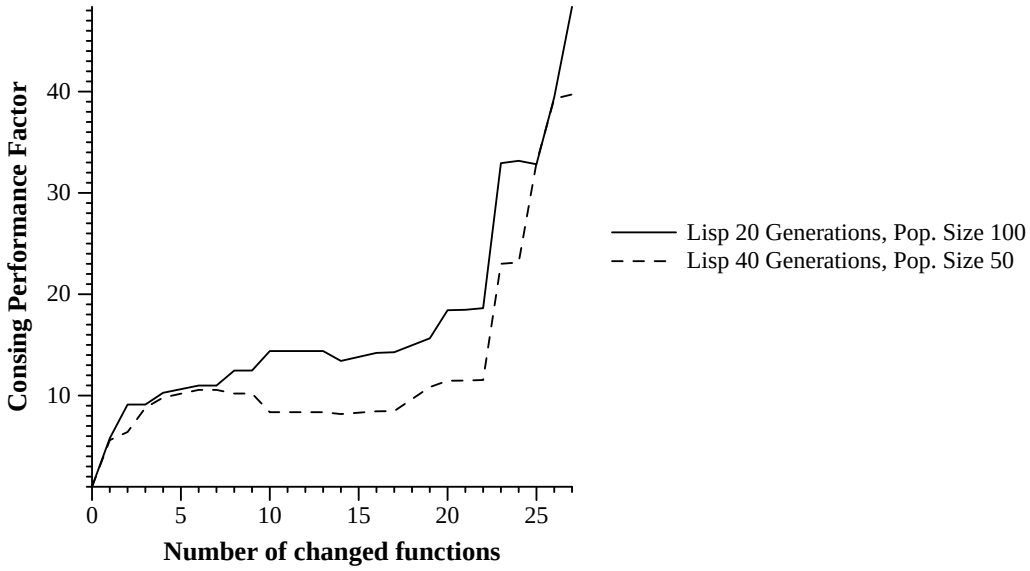

Figure 6: Space Performance Improvement of GP-LISP

profiling suggested a benefit. This led to a series of experiments which led to general improvements with occasional setbacks. In all, only about 15 functions were changed, but no inlining was done.

The final major improvement came when the application specific `eval` was rewritten. Unrolling the evaluation loop allowed a dramatic reduction in the amount of floating point number consing [3].

The final result was a factor of 30 improvement. Execution time went from 138.73 seconds, to 4.57 seconds. As execution time is now dominated by things like printing, its usefulness as a benchmark has been destroyed. Other improvements are possible but would require changing the structure of the software. At least another factor of two improvement seems likely.

As with the CASCOR1 benchmark, the Lisp and C are not directly comparable because different random number generators were used. However, taking the median of five runs of GP-C with different random number seeds we can get a reasonable estimate of relative performance. These medians are plotted as points in the figures. The performance improvement over the original GP-Lisp is a factor of 6 for the 40 generation case, and a factor of 14 for the 20 generation case, a factor of two slower than the final Lisp version.

One difference between GC-C and GC-LISP is the evaluation strategy. GC-C does more function calling but no number consing, while the final GC-LISP is more inlined but does some number consing.

Another factor is storage management. GP-C uses more space to represent its programs than GP-LISP does. For example to represent the program fragment

```
(+ (* x 3.0) 2.0)
```

GP-LISP requires 6 two-word `cons` cells, and two four-word `single-float`'s. for a total of 20 words. The space for the symbols used to represent functions and variables is shared among all programs so it can be ignored.

GP-C, on the other hand, requires five six-word structures and four two-word arrays for a total of 38 words, about twice as much as required by GP-LISP[4].

However, this is only part of the story, since both programs use different storage management strategies. GP-C explicitly allocates and deallocates memory using C's `malloc` and `free`. GC-LISP on the other hand uses Lisp's automatic garbage collector. The `malloc`/`free` approach requires

[3] Alternatively, using a floating point stack was equally successful.

[4] There may be other storage overhead added by `malloc`

	C	Lisp
EVAL	34.8	46.8
MALLOC/CONS	5.8	4.0
FREE/GC	25.4	9.7
OTHER	34.0	39.5

Figure 7: Percentage of execution time for GP

effort proportional to the number of objects created, while the GC approach requires effort proportional to only the number of live objects. Since few objects survive from one generation to the next, the automatic GC approach can be quite effective.

Since GC-C takes about twice as long and uses twice as much storage as GC-LISP, the percentage of time spent in different program phases is roughly comparable[5] as shown in Figure 7. The `malloc/cons` times are comparable, but since Lisp spends only 10% of its time in `GC` compared to C's 25% in `free`, Lisp can use the 15% difference to do useful work.

6 Things We've Learned

We have shown that Lisp and C programs can go head to head on performance. Well written Lisp programs are more compact than C programs, and can be just as fast, even faster. Performance of real programs has more to do with choice of algorithm than with choice of language.

We have also seen an important point about benchmarking. Often, a suite of benchmark programs is used, but only a single number is quoted for each program in the suite. This could led to false impressions. For example, from Figure 3 we see that several lines cross, such as with the original Allegro and unoptimized C data. In such cases, one language would seem faster than the other for certain sized problems and not for others.

The slope of these curves is controlled by two dot products, about 10 lines of code. Thus, if one concluded that one language was faster than the other, one would be condemning the language based on only 10 lines of code. The difference is in the abilities of the compilers, not the languages.

[5] In GC-C, a special version of `malloc` was used that allocates large blocks of storage at once and then quickly allocates smaller objects from it.

Lisp programs can be slow, otherwise one would not be able to speed them up by a factor of 30. One of the powers of using Lisp is that one can just focus on programming, and let Lisp worry about the details. However, one should be prepared to pay for this convenience in increased execution time and space.

Luckily, commonly available tools, such as compiler advice and profilers can help. A small amount of profiling can go a long way. We have shown that as little effort as a man day of profiling, and minor adjustments to less than 20% of the code can provide substantial improvements. This is not specific to Lisp. The significant optimizations uncovered during the profiling above can be applied in either language.

However, profiling can never make up for an inadequate design. For example, while it is possible to write GP-LISP or GP-C to do only a fixed amount of consing, since programs have a fixed maximum size, it would require a major rewrite of both programs.

Using a profiling tool requires some expertise. However, it is not unreasonable to imagine an optimizing expert system that could suggest declarations or small code changes based on a combination of software analysis (such as done by current advising compilers) guided by profiling. It could even suggest replacing something like the `activation` function by a faster one generated by a machine learning algorithm such as the ones studied here.

7 References

References

[A89] Samuel Aletan, "Current and Future Trends in Artificial Intelligence Architectures and Programming Languages", IEEE International Workshop on Tools for Artificial Intelligence: Architectures, Languages, and Algorithms, Fairfax, VA, Oct 23-25, 1989, p. 215-221.

[AEL] Appel, A.W., Ellis, J.R., and Li, K. "Real-time concurrent garbage collection on stock multiprocessors", SIGPLAN PLDI, June, 1988.

[AWA] Andrew W. Appel, Garbage collection can be faster than stack allocation, Information Letters, 25 (1987) 275-279.

[B78] Baker, H.G. "Lisp processing in real time on a serial computer", CACM, 21,4, (April 1978), 280-294.

[B92A] Baker, H.G., "The Gabriel 'Triangle' Benchmark at Warp Speed", ACM Lisp Pointer, V, 3, (July-Sept, 1992), 15-17

[B92B] Baker, H.G., "Seeding up the 'Puzzle' Benchmark a 'Bit"', ACM Lisp Pointer, V, 3, (July-Sept, 1992), 18-21

[B92C] Baker, H.G., "A tachy 'TAK"', ACM Lisp Pointer, V, 3, (July-Sept, 1992), 22-23.

[B94] Baker, H.G. A 'Linear Logic' Quicksort. ACM Sigplan Notices 29,2 (Feb 1994), 13-18.

[BKV] Bentley, Kernighan, and VanWyk, "An Elementary C cost model", Unix Review, Vol. 9, No. 2, p. 38-48.

[BM] J.L. Bentley, and M.D. McIlroy, "Engineering a sort function", Software - Practice and Experience", Vol. 23(11), 1249-1265 (Nov. 1993).

[BS] Bjarne Stroustrup, "Library design using C++", C++ Report, June, 1993, p. 14-22.

[F90] Fahlman, Scott, "The Cascade-Correlation Learning Architecture" Carnegie Mellon University Computer Science Technical Report CMU-CS-90-100, ftp: archive.cis.ohio-state.edu /pub/neuroprose fahlman.cascor-tr.ps.Z

[F94] Fahlman, Scott E., Email to comp.lang.clos from sef+@cs.cmu.edu, 1994.

[F91] R.J. Fateman, "Endpaper: FRPOLY: A benchmark revisited", Lisp and Symbolic Computation, 4, 155-164, 1991.

[FBWR] R.J. Fateman, K.A. Broughan, D.K. Willcock, and Duane Rettig, "Fast floating-point processing in Common Lisp", ftp from peoplesparc.berkeley.edu ftp/pub/papers/fastlisp.ps.Z

[G85] R.P. Gabriel, "Performance and Evaluation of Lisp Systems", MIT Press, Cambridge, MA, 1985.

[G91] R.P. Gabriel, "Lisp: good news, bad news, how to win big", AI Expert, June, 1991, p. 31-39. Also ftp from cs.utexas.edu pub/garbage worse.ps.

[Koza] John R. Koza, "Genetic Programming, On the Programming of Computers by Means of Natural Selection", The MIT Press, Cambridge, MA, 1992.

[P94] Poeck, Karsten, Personal Email from poeck@informatik.uni-wuerzburg.de, June, 1994.

[PL94] Dave Pare and Jonathan Lee, "Speed Generator", Unix Review, March 1994.

[Zorn] Benjamin Zorn, "The Measured Cost of Conservative Garbage Collection", Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1992, revised August 1992, Available PRocessingby anonymous FTP and e-mail from ftp.cs.colorado.edu in the file pub/cs/techreports/zorn/CU-CS-573-92.ps.Z