A Genome Compiler for High Performance Genetic Programming

Alex Fukunaga, Andre Stechert and Darren Mutz

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr., M/S 126-347
Pasadena, CA 91109-8099
alex.fukunaga@jpl.nasa.gov, andre.stechert@jpl.nasa.gov, darren.mutz@jpl.nasa.gov

ABSTRACT

Genetic Programming is very computationally expensive. For most applications, the vast majority of time is spent evaluating candidate solutions, so it is desirable to make individual evaluation as efficient as possible. We describe a genome compiler which compiles s-expressions to machine code, resulting in significant speedup of individual evaluations over standard GP systems. Based on performance results with symbolic regression, we show that the execution of the genome compiler system is comparable to the fastest alternative GP systems. We also demonstrate the utility of compilation on a real-world problem, lossless image compression. A somewhat surprising result is that in our test domains, the overhead of compilation is negligible.

1 Introduction

Genetic programming [Koz92] (Figure 1) is a very computationally intensive task. It is well-known that in many applications to which genetic programming is applied, the vast majority of computational resources is used by the *evaluate* step in Figure 1, which evaluates candidate solutions with respect to an objective function. Thus, one of the challenges in implementing a high-performance GP system is speeding up the evaluation step as much as possible.

We were made acutely aware of the need for an efficient individual evaluation process when we attempted to apply GP to image compression (see Section 4.2). Initially, we implemented the application using lil-gp [PZ96], a standard GP system used by numerous researchers, and found that it was prohibitively slow to study genetic programming-based image compression. We therefore sought to significantly improve the speed of execution of the GP system.

In standard GP, s-expressions are recursively evaluated, and each evaluation of an atom requires a recursive function call. This means that even though many atoms in the set of primitives can be evaluated by a single machine instruction (e.g., add, multiply, independent variables, etc.), much time is spent in unnecessary function call overhead such as pushing/popping values onto the stack. This problem is not unique to lil-gp: Keith and Martin [KM94] observed that function call overhead can overwhelm the time actually required to evaluate nodes, even with a very efficient, linear (non-recursive) s-expression representation.

We therefore sought to eliminate as much of this function call overhead as possible, and implemented a *genome compiler*[1] which compiles s-expressions into SPARC machine language instructions. In applications where the same tree is evaluated many times (e.g., symbolic regression, image compression), the benefits of eliminating function call overheads outweighs the overhead of compilation.

```
$t := 0$
**initialize** $P(t)$;
**evaluate** $P(t)$;
**while not terminate do**
  $P'(t) :=$ **recombine** $P(t)$;
  $P''(t) :=$ **mutate** $P'(t)$;
  **evaluate** $P(t)$;
  $P(t+1) :=$ **select** $(P''(t) \cup Q)$;
  $t := t + 1$;
**end while**
```

Figure 1 Algorithm schema for Genetic Programming. P is a population of candidate solutions; Q is a special set of individuals that has to be considered for selection, e.g., $Q = P(t)$.

The rest of the paper is organized as follows: In Section 2, we review related work on high-performance evaluation mechanisms for genetic programming. Section 3 describes our genome compiler in detail. Section 4 presents some empirical evaluations of genome compiler based genetic programming, comparing its performance with standard lil-gp and other proposed methods for high-performance GP implementations. We conclude in Section 5 with a discussion of results and directions for future work.

2 Related Work

A number of researchers have addressed the problem of highly efficient implementations of genetic programming.

Keith and Martin [KM94] observed that the recursive evaluation of the standard tree representation of s-expressions and used in GP systems widely used by the GP research community such as SGPC [TC93] lil-gp [PZ96] was inefficient because much time is used parsing the type token for each node in the tree. They showed that a linear, stack-based internal representation of s-expressions resulted in significant speed improvement over a tree representation. However, they observed that even with the efficient linear representation, function calls still posed a significant overhead, and suggested that the implementation of a genome compiler would be an interesting direction for future research.

Perkis [Per94] first demonstrated the use of genetic operators (crossover and mutation) on a linear individual representation using a stack-based virtual machine (as opposed to the standard s-expression representation).[2] The idea of directly evolving stack-based linear programs was also used in HiGP, a high performance, parallel GP system developed by Stoffel and Spector [SS96], which, like Perkis' system, works directly on a population of linear programs for a virtual stack machine.

An interesting contrast between s-expression based approaches and stack-based approaches is in the enforcement of closure property (i.e., that guarantees all programs generated are valid and executable by the interpreter). In stack-based approaches, it is possible that a virtual machine instruction which takes arguments off the stack will not have enough values available on the stack. In this case, both of the above systems maintain closure by ignoring the instruction (i.e., they treat it as a NOOP). In s-expression based genetic programming, the genetic operators assure that the arity of all functions is correct. This was cited as an advantage of using Lisp s-expressions as the program representation by Koza [Koz92]. Given the good performance behavior reported for stack-based GP [Per94, SS96] in comparative experiments with standard s-expression based GP, it is now unclear whether there is any advantage to the ease of maintaining the closure property that the s-expression based approaches offer.

[1] As far as we know, this term was coined by Keith and Martin in [KM94].

[2] In contrast, Keith and Martin used a linear representation as a representation of s-expressions, and their genetic operators worked at the level of s-expressions, not directly on the linear representation (e.g., "tree crossovers" were simulated on the linear representation).

While the previous stack-based approaches used a linear representation internally, Juille and Pollack implemented a system which applies genetic operators to s-expressions, but previous to execution, *compiles* them into a linear representation for execution on a stack-based virtual machine [JP96]. Note that in this scheme, there is no problem of handling possible stack underflow during execution, because the linear programs are directly translated from s-expressions that guarantee that the arity of the functions is correct.

Nordin developed the Compiling Genetic Programming System (CGPS) [Nor94, NB95], which manipulates linear arrays of SPARC machine language instructions. Crossover and mutation are applied at the instruction boundaries, to ensure that the machine code resulting from the operations are valid. Note that despite its name, the Compiling Genetic Programming System does not apply a compilation procedure to its individuals at any time – CGPS is unique in that it directly manipulates machine-specific code, as opposed to the other approaches, which apply genetic operators to s-expressions or linear code for a virtual stack machine.

The genome compiler described below combines 1) the idea in Juille and Pollack's work of applying genetic operators to s-expresssions, but compiling s-expressions into a representation that can be executed more efficiently, and 2) the machine code representation used by Nordin, which results in the fastest[3] possible execution (as opposed to a virtual machine).

3 The Genome Compiler

The motivation for examining the possibility of converting a LISP s-expression into a form that is more efficient to evaluate comes primarily from the observation that the standard method of recursive evaluation involves much more computational effort than simply applying arithmetic operators in sequence. That is, since simple arithmetic operations can be executed with a single instruction at the hardware level, our intuition tells us that the arithmetic portion of the computation is probably dwarfed by the overhead associated with pushing and popping arguments and return values on the program's stack during recursive s-expression evaluation. This observation led us to conclude that translating the s-expressions evolved by the GP into a more terse machine language equivalent would greatly improve performance. Before each individual s-expression is evaluated, the genome compiler compiles it (at run-time) to SPARC machine language [WG94] code as described below.

The method of generating machine executable code proceeds naturally from the standard recursive evaluation procedure. The post order traversal of the graph corresponding to the given s-expression is analogous to the order of operations one would perform if the computation were carried out in postfix form with a stack. That is, traversing the tree representation in Figure 2 in post order gives us the stack-executable code in Figure 3.

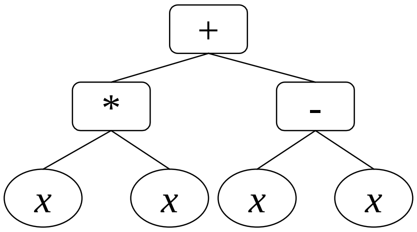


Figure 2 An example individual.

Translating this stack-executable code to machine code, where values are pushed and popped from locations in memory, is a clear speed improvement over recursive tree evaluation, which involves maintaining the program's stack in addition to these operations.

An additional speed improvement is realized when one considers the register file itself a stack,

[3] In this paper, when we say *fast*, we refer to execution speed for evaluating individuals, and not to the efficiency of GP search algorithms.

```
push(x)
push(x)
t1 = pop()
t2 = pop()
push(t2 * t1)
push(x)
push(x)
t1 = pop()
t2 = pop()
push(t2 - t1)
t1 = pop()
t2 = pop()
push(t2 + t1)
```

Figure 3 Stack machine code that computes the value of the s-expression in Figure 2.

albeit one of limited depth. In addition to a reduction in data access times we also gain the ability to effectively pop two operands, perform an arithmetic operation and push the result, all in a single machine instruction[4]. This is due to the fact that in many modern architectures arithmetic instructions allow both source registers and a destination register to be specified.

Representing floating point registers as $f_0, f_1, \ldots, f_{31}$, with the constant value x stored in f_{31}, Figure 4 gives the assembly-level code corresponding to the s-expression above. Three breaks in the generated assembly are labeled (1, 2, and 3); the breaks correspond to the code generated so far when the traversal has progressed to the node indicated in the tree diagram.

Our compiler directly generates the machine executable code that corresponds to this assembly-level code. A key point to note is that the C programming language provides the necessary flexibility here: it allows the programmer to create jumps to code that is generated at runtime by casting an integer array to a function pointer [NB95]. This eliminates the the overhead of invoking an external compiler.

The computational complexity of compiling each s-expression down to machine executable code is linear in the number of nodes in the tree corresponding to the s-expression, the same as that of recursive tree evaluation. Both procedures involve visiting each node in the tree exactly once and executing a constant number of operations at each node.

It was previously noted that some function primitives in s-expressions generated by the GP cannot be executed in a single machine instruction. This is particularly true of conditionals, as Figure 5 illustrates. For the purposes of this example, floating point register f_{31} contains the independent variable x, as before, and f_{29} and f_{30} contain the constants 0 and 5, respectively. The arity four primitive `ifgte` is defined such that if its four arguments are a, b, c, and d it returns c if $a \geq b$ and d otherwise.

The genome compiler approach is similar to [NB95] only in that both methods involve runtime machine code generation and execution; in our approach individuals are not manipulated at the machine code level. Like our compiler, The HiGP system described in [SS96] takes the approach of converting s-expressions into stack machine instructions, incorporating an extremely space efficient memory representation of individuals as well. However, HiGP performs evolution at this level, using a string-based genetic algorithms approach. Our system acts instead as a means of speeding up the execution of standard tree-based GP systems. Our method is perhaps most similar to [JP96] in that population members are "pre-compiled" down to a stack-executable form; the genome compiler takes that approach one step further and compiles the stack-executable instructions down to the machine code level.

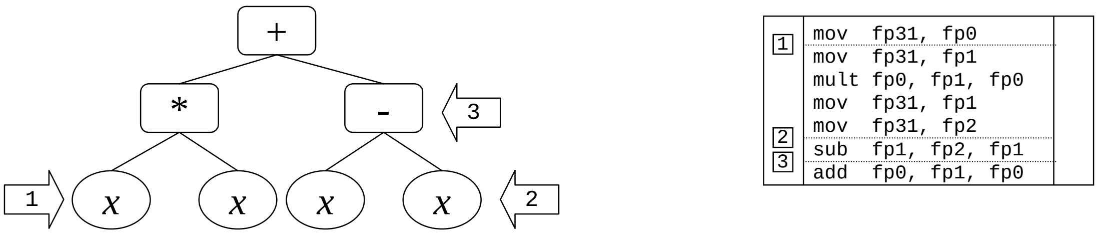
4 Empirical Evaluation

We evaluated the performance improvements obtained using the genome compiler on two tasks, symbolic regression and lossless image compression. In both tasks, each candidate s-expression is evaluated many times, which potentially justifies the overhead of compilation.

4.1 Symbolic Regression

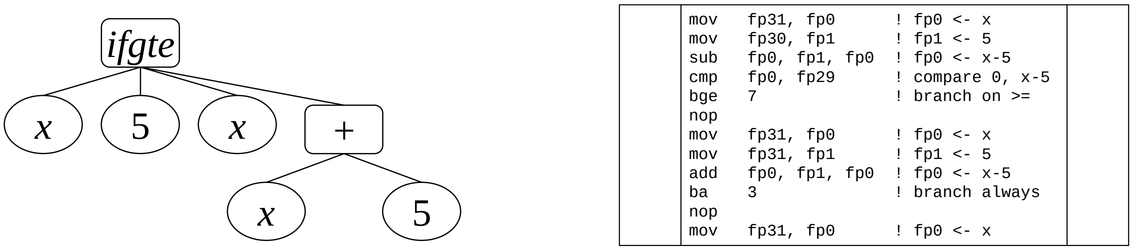
Symbolic regression is a canonical genetic program problem in which the task is to generate a program which approximates target function f_{target}. The ob-

[4] Note that not all function primitives in the individuals generated by the GP can be executed in a single instruction. For example, protected division requires a test for a denominator of zero.



```
1  mov  fp31, fp0
   mov  fp31, fp1
   mult fp0, fp1, fp0
   mov  fp31, fp1
   mov  fp31, fp2
2  sub  fp1, fp2, fp1
3  add  fp0, fp1, fp0
```

Figure 4 The s-expression from Figure 2 with corresponding assembly-level code. Numbered breaks in the code correspond to the code generated so far when the post order traversal has progressed to the node indicated in the tree diagram.



```
mov   fp31, fp0      ! fp0 <- x
mov   fp30, fp1      ! fp1 <- 5
sub   fp0, fp1, fp0  ! fp0 <- x-5
cmp   fp0, fp29      ! compare 0, x-5
bge   7              ! branch on >=
nop
mov   fp31, fp0      ! fp0 <- x
mov   fp31, fp1      ! fp1 <- 5
add   fp0, fp1, fp0  ! fp0 <- x-5
ba    3              ! branch always
nop
mov   fp31, fp0      ! fp0 <- x
```

Figure 5 An individual with a conditional and its assembly equivalent.

jective function to minimize is:

$$\sum_{i=1}^{numcases} (f_{gp}(x_i) - f_{target}(x_i))^2,$$

for *numcases* randomly generated *fitness cases* (test points), where f_{gp} is a candidate GP solution.

Here, we use symbolic regression to study the relative speed of the genome compiler compared with lil-gp, as well as other GP systems.

We used the test function $f_{target}(x) = x^9$. The genome compiler and standard lil-gp systems were configured as follows: population=500, generations=30, function set $= \{+, -, *, \%\}$ (where % is the protected division operator [Koz92]), terminal set=X, tournament selection (size=5), 90% crossover, 10% reproduction, no mutation, depth limit 5).

To observe the speed benefit of compilation as the relative overhead of compilation was varied, we varied the number of fitness cases between 1 and 1000. Figure 6 shows the runtimes (cpu time) of lil-gp and the genome compiler, averaged over 100 independent runs; Figure 7 gives a detail of the region where the number of fitness cases varies between 1 and 20.

As shown in Figure 6, the speedup of the genome compiler, $t_{lil-gp}/t_{genome\_compiler}$, improves as the

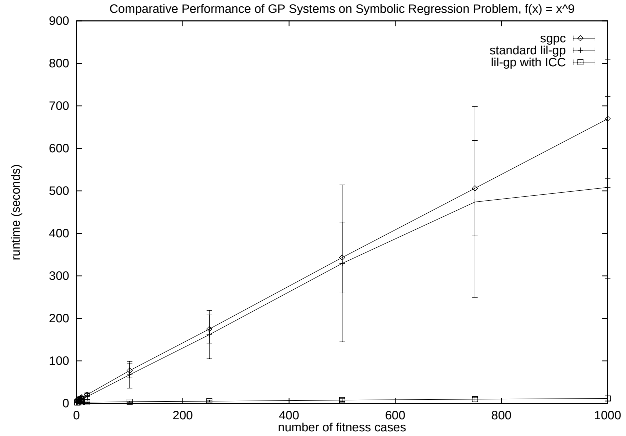


Figure 6 Time to complete 30 generations of GP on symbolic regression of $f_{target} = x^9$ for the Genome Compiler, lil-gp, and SGPC systems. Mean and standard deviation of 100 runs. All timings in this figure were measured on a 296 MHz UltraSparc 2.

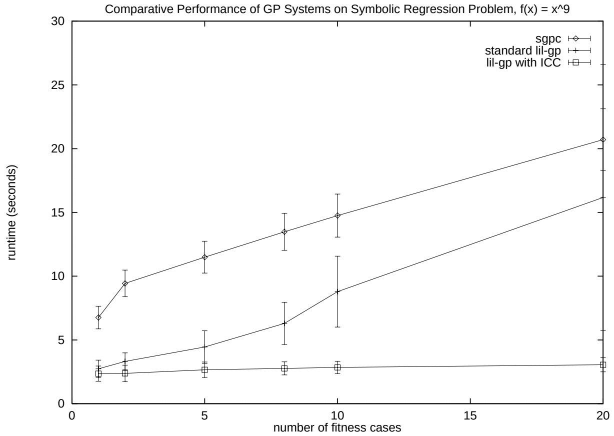


Figure 7 Performance on small numbers of regression test cases (a zoomed view of Figure 6):Time to complete 30 generations of GP on symbolic regression of $f_{target} = x^9$ for the Genome Compiler, lil-gp, and SGPC systems. Mean and standard deviation of 100 runs. All timings in this figure were measured on a 296 MHz UltraSparc 2.

number of test cases is increased (i.e., the relative overhead of compilation is decreased), reaching a maximum speedup of around 50 when the number of test cases is 1000.

A somewhat surprising result is that even when only a single test case was used for symbolic regression, the performance of the genome compiler is no worse than that of standard lil-gp. This is because even with one test case, both standard lil-gp and the genome compiler need to traverse the tree at least once (lil-gp traverses the tree once to evaluate it, the genome compiler traverses the tree once during compilation). The overhead of the function calls for this single traversal is quite significant, and dwarfs the differences between lil-gp's execution of the expression and the genome compiler's compilation of the expression. In comparison, the computational cost of actually executing the compact machine code translation is almost negligible – note in Figure 7 that the runtime for the genome compiler GP barely increases as the number of test cases is increased from 1 to 20. Thus, because both standard recursive evaluation and the compiler needs to traverse the expression at least once, compilation overhead is negligible.

4.1.1 Comparison with other high performance GP systems

To put the speedup enabled by genome compilation in perspective, we also briefly compare our symbolic results with other published results for high-performance GP systems.

Stoffel and Spector compared the speed of HiGP against lil-gp on symbolic regression of the target function x^9, where the configuration of lil-gp they used was: population=500, maxgenerations=30, function set = $\{+, -, *, \%\}$, terminal set=X, tournament selection (size=5), 90% crossover, 10% reproduction, no mutation). They compared the average time per generation of the two systems, and found that the maximum speedup (t_{lil-gp}/t_{HiGP}) measured was approximately 5, when the depth limit for lil-gp was set to 17 [SS96]. [5] The genome compiler, due to its use of machine code, achieves about an order of magnitude speedup over HiGP.

Because of the use of machine language instructions, the CGPS system of Nordin and Banzhaf [NB95] is expected to be closest to our genome

[5] Note that Stoffel and Spector stopped the lil-gp runs when the optimal solution was found, while in our experiments, the GP was run a full 30 generations.

compiler with respect to genome evaluation speed. Nordin reported that on a polynomial symbolic regression task, CGPS ran on average 60 times faster on symbolic regression than SGPC, a standard recursive tree evaluator based GP system written by Tackett and Carmi [TC93], where both CGPS and SGPC was running on a SPARC IPX. Although we were not able to directly compare the genome compiler with CGPS, we can perform an indirect comparison by comparing the genome compiler with measurements of SGPC speed on a 296 MHz SPARC Ultra 2, using the same symbolic regression problem ($f_{target} = x^9$) and the same control parameters (Figures 6,7). The genome compiler performs roughly 50-60 times faster than SGPC running on the same machine, which is comparable to the execution speeds for CGPS reported by Nordin and Banzhaf [NB95].

4.2 Lossless Image Compression

The impetus for the development of the genome compiler was the need to perform efficient s-expression execution for the task of lossless image compression using a nonlinear predictive coding algorithm for which the nonlinear model was automatically generated using a genetic programming system. We briefly describe the application below. Our compression system is detailed in another paper [FS98]. See [MW96, MW97] for more details on predictive coding based image compression.

Predictive coding is an image compression technique which uses a compact model of an image to predict pixel values of an image based on the values of neighboring pixels. A *model* of an image is a function $model(x, y)$, which computes (predicts) the pixel value at coordinate (x, y) of an image, given the values of some *neighbors* of pixel (x, y), where neighbors are pixels whose values are known. We process the image in raster scan order, and use the set of neighboring pixels {*(x-1,y-1), (x,y-1), (x+1,y-1),(x-1,y)*}. *Linear predictive coding* is a simple, special case of predictive coding in which the model simply takes a weighted average of the neighboring values. *Nonlinear* models assign arbitrarily complex functions to the models. Applying a model to an image results in an *error signal* (the differences at each pixel between the value predicted by the model and the actual value of the pixel in the original image. To complete the compression process, the error signal is compressed using a standard data compression technique such as Huffman coding.

If we transmit this compressed error signal as well as the model, then a receiver can reconstruct the original image by applying an analogous decoding procedure (see Figure 8).

```
Encoder(Model,Image)
  for $x = 0$ to $xmax$
    for $y = 0$ to $ymax$
      Error[x,y] = Image[x,y] - Model(x,y)
Decoder(Model)
  for $x = 0$ to $xmax$
    for $y = 0$ to $ymax$
      Image[x,y] = Model(x,y) + Error[x,y]
```

Figure 8 Algorithm schema for predictive coding. $Model(x, y)$ **is a function that takes the coordinates of a pixel and returns a predicted value of that pixel.** *Image* **and** *Error* **are two-dimensional arrays.**

Given an input image, our system uses GP to generate a nonlinear model for the predictive coding.

The terminals and functions used were:

- values of the four neighboring pixels *Image[x-1,y-1], Image[x,y-1], Image[x+1,y-1], Image[x-1,y].*
- selected constant values: 1, 5, 10, 100.
- arithmetic functions *+,-,*,% (protected division [Koz92])*
- the conditional operator *(IFLTE arg1 arg2 ret1 ret2)* which returns the value of *ret1* if *arg1* $\leq$ *arg2*, and the value of *ret2* otherwise.
- *(MIN a b)* and *(MAX a b)* functions which return the minimum and maximum values of their two arguments, respectively.

Since the model is applied to each pixel in the image this application would be expected to benefit from compilation.

We ran 5 runs each of 50 generations of both the genome compiler and lil-gp on a 64 pixel by 64 pixel image compression problem. On a 296MHz UltraSparc 2, the average runtime for lil-gp was 9177 seconds, and the average runtime for the genome compiler was 2071 seconds. [6]

Note that in these runs, a significant percentage of the current runtime (about 50% for the 64 by 64 images in the experiments) was spent by the adaptive Huffman coder which is run for each individual evaluation, and *not* in the execution of compiled machine code for individuals; this explains why the speedup obtained (4-5 times) is not as impressive as that for symbolic regression.

Although the runtime (several hours per image using the genome compiler) is still too slow for practical application of the technique, the significant speedup enabled by compilation makes it much more feasible to explore alternative search strategies, function/terminal sets, etc. for this problem (i.e., runs that took roughly a week using standard GP can be now be completed in about a day).[7]

5 Conclusion/Discussion and Future Work

We have described a genome compiler for speeding up individual evaluations in GP. Experiments with symbolic regression and image compression applications show that for applications in which individuals are repeatedly evaluated, the genome compiler provides a significant speedup over standard s-expression based GP systems as well as virtual stack machine based systems; the speedup over conventional GP systems written in C is comparable to CGPS, the fastest reported GP implementation in the literature. On extremely computationally expensive problems such as image compression, the speed improvement that the genome compiler offers makes the application of s-expression based GP to the problem much more feasible. Furthermore, we showed that the overhead of compilation can be negligible, so that the speed benefits of compilation can be significant even when the number of times individuals are repeatedly evaluated is small.[8]

Our results highlight the fact that the utility of genome compilation depends largely on the percentage of computational effort that can be saved by eliminating function call overhead during s-expression evaluation. Applications such as symbolic regression are well suited for performance improvements using compilation techniques because individual evaluations consist solely of s-expression evaluations where the functions and terminals are simple primitives, and the overhead of tree traversal is significant. On the other hand, the effect of compilation is less dramatic in our image compression application because much of the time for individual evaluation is spent in entropy coding, which can not be sped up by genome compilation.

Obviously, raw execution speed is not the only important factor in evaluating a GP system. The relative merits of s-expression based GP vs. alternatives such as stack-based GP and CGPS is still an open research problem – with respect to search effort, s-expression based GP seems to do better on some problems, while stack-based approaches do better on others (c.f., [Per94, Bru97]). Likewise, the dynamics of CGPS in comparison to traditional GP and stack-based GP are not well understood yet. Previous work had shown that alternative approaches such as stack-based GP and CGPS are capable of significantly faster execution of s-expressions than traditional s-expression GP. Our work shows that by using a compiler to remove function call overhead, s-expression based GP can be competitive with the fastest alternative approaches with respect to execution speed.

A disadvantage of the genome compiler approach is that the implementation is machine specific. In comparison, GP systems such as HiGP which use virtual stack-based machines are machine independent, while being significantly faster than traditional s-expression based GP systems. A genome compiler that compiles to a virtual machine code (like that of Juille and Pollack,) could possibly yield execution speeds comparable to virtual stack-machine GP. We have implemented an efficient virtual machine compiler, and preliminary results seem to indicate that we can obtain performance within a factor of 2-3 of the machine code compiler. Further details will be provided in a future paper.

Finally, another interesting direction in which to

[6] The compression ratios obtained by this system are promising, but are beyond the scope of the present paper (see [FS98]).

[7] [FS98] describes a faster implementation which removes much of the overhead of the entropy coding.

[8] Figure 7 shows that significant speed benefits can be obtained for symbolic regression even when only 5-10 test cases are used.

extend the genome compiler would be to implement compiler optimizations which use *editing* operations [Koz92] or standard compiler optimization techniques to collapse instructions together, remove redundant operations, reorder operations, etc., to further speed up execution. Although this would add additional compilation overhead, the benefits may be worthwhile for applications such as image compression in which the individual is evaluated many times.

Acknowledgments

The research described in this paper was performed at the Center for Integrated Space Microsystems, Jet Propulsion Laboratory, California Institute of Technology, and was supported by the National Aeronautics and Space Administration. Thanks to Bill Punch and Douglas Zonker for making lil-gp publically available, and to Walter Tackett and Aviram Carmi for making SGPC publically available.

References

[Bru97] W.S. Bruce. The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions. In *Proceedings of the Genetic Programming Conference*, pages 52–57, 1997.

[FS98] A. Fukunaga and A. Stechert. Evolving nonlinear predictive models for lossless image compression with genetic programming. In *Proceedings of the Genetic Programming Conference*, 1998.

[JP96] H. Juille and J.B. Pollack. Massively parallel genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[KM94] M.J. Keith and M.C. Martin. Genetic programming in C++: Implementation issues. In K. Kinnear, editor, *Advances in Genetic Programming*. MIT Press, 1994.

[Koz92] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[MW96] N. Memon and X. Wu. Lossless compression. In *CRC Handbook of Communication*. CRC Press, 1996.

[MW97] N. Memon and X. Wu. Recent progress in lossless image coding. *The Computer Journal*, 40:127–36, 1997.

[NB95] P. Nordin and W. Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In *Proceedings of the International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[Nor94] P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K. Kinnear, editor, *Advances in Genetic Programming*. MIT Press, 1994.

[Per94] T. Perkis. Stack-based genetic programming. In *Proc. IEEE International Conference on Evolutionary Computation*, 1994.

[PZ96] B. Punch and D. Zonker. lil-gp genetic programming system version 1.1 beta version. Michigan State University, http://GARAGe.cps.msu.edu/software/lil-gp/index.html, 1996.

[SS96] K. Stoffel and L. Spector. High-performance, paralle, stack-based genetic programming. In *Proceedings of the Genetic Programming Conference*, pages 224–229. MIT Press, 1996.

[TC93] W. Tackett and A. Carmi. sgpc: simple genetic programming in C. ftp://ftp.io.com/pub/genetic-programming, 1993.

[WG94] D. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9.* PTR Prentice Hall, 1994.