Genetic Programming Applied to Compiler Heuristic Optimization

Mark Stephenson[1], Una-May O'Reilly[2],
Martin C. Martin[2], and Saman Amarasinghe[1]

[1] Laboratory for Computer Science
[2] Artificial Intelligence Laboratory
Massachusetts Inst. of Technology
Cambridge, MA, 02139
{mstephen,saman}@cag.lcs.mit.edu {unamay,mcm}@ai.mit.edu

Abstract. Genetic programming (GP) has a natural niche in the optimization of small but high payoff software heuristics. We use GP to optimize the priority functions associated with two well known compiler heuristics: predicated hyperblock formation, and register allocation. Our system achieves impressive speedups over a standard baseline for both problems. For hyperblock selection, application-specific heuristics obtain an average speedup of 23% (up to 73%) for the applications in our suite. By evolving the compiler's heuristic over several benchmarks, the best general-purpose heuristic our system found improves the predication algorithm by an average of 25% on our training set, and 9% on a completely unrelated test set. We also improve a well-studied register allocation heuristic. On average, our system obtains a 6% speedup when it specializes the register allocation algorithm for individual applications. The general-purpose heuristic for register allocation achieves a 3% improvement.

1 Introduction

Genetic programming (GP) [11] is tantalizing because it is a method for searching a high dimensional, large space of executable expressions. GP is widely applicable because its representation, a directly *executable expression*, is so flexible. Koza [11] argues that most problems can be reformulated to accept program-style solutions.

Yet, even without reformulation, there are a vast number of problems for which a program or codelet is a *direct* solution. Consider substantially sized software systems such as compilers, schedulers, text editors, web crawlers, and intelligent tutoring systems. Current GP knowledge and practice certainly cannot generate such large scale efforts. Yet, GP can act remedially. For example, Ryan et al. [18] used GP to convert serial code to parallel. There was a sizeable payoff in updating legacy software.

When large scale software systems are developed, they inevitably acquire shortcomings. We believe that GP can address many problems associated with complex systems– either at development time or later. Large software systems have 'admitted' shortcomings that arise from necessity. Real world problems of complex nature often offer NP-complete sub-problems. Since these problems demand solutions to be delivered within practical time limits, the employment of heuristics is necessary. Heuristics, by definition, are supposed to be good enough, but not necessarily perfect.

The genesis of our idea for using GP came from dissatisfaction with compiler efficiency and design, combined with our realization that compiler designers are overwhelmed with countless nonlinearly complex considerations. We examined different *passes* within a compiler and their heuristics. We found a common, easily learned feature in the heuristics that we term a 'priority function'. Put simply, priority functions prioritize the options available to a compiler algorithm. Could GP generate more effective priority functions than currently exist in compilers? We use Trimaran– a freely downloadable research compiler and simulator– to answer that question [19]. This paper offers a proof of concept: we use genetic programming to optimize the priority functions associated with register allocation as well as branch removal via predication. Our contention is that priority functions are most certain to also lurk in heuristics within other similar software systems. Genetic programming is eminently suited to optimizing priority functions because they are best represented in GP terms: as directly executable expressions. Plus, GP offers the scalable means of searching through priority function space.

2 Related Work

Both GP[15] and Grammatical Evolution (GRE)[13] have been used to optimize a caching strategy. A caching strategy, in essence, has a priority function. It must determine which program memory locations to assign to cache or move to main memory in order to minimize 'misses'. A miss occurs when main memory must be accessed rather than the cache. One human designed priority function is Least Recently Used (LRU). While LRU is intuitive, results evolved via GP and GRE outperform it.

Many researchers have applied machine-learning methods to compilation, and therefore, only the most relevant works are cited here. By evolving compiler heuristics, and not the applications themselves, we need only apply our process once. This contrasts with Cooper et al. who use genetic algorithms (GA) to solve compilation phase ordering problems [7] and the COGEN(t) [10] compiler. Calder et al. use supervised learning techniques to fine-tune static branch prediction heuristics [4]. Since our performance criteria is based on execution time it requires an unsupervised technique such as the one we present in this paper.

3 Compilation, Heuristics and Priority Functions

Compiler writers have a difficult task. They are expected to create effective and inexpensive solutions to NP-hard problems such as instruction scheduling and register allocation for intractably complex computer architectures. They cope by devising clever heuristics that find good approximate solutions for a large class of applications.

A key insight in alleviating this situation is that many heuristics have a focal point. A single *priority* or *cost* function often dictates the efficacy of a heuristic. A priority function, a function of the factors that affect a given problem, measures the relative value or weight of choices that a compiler algorithm can make.

Take register allocation, for example. When a graph coloring register allocator cannot successfully color an interference graph, it 'spills' a variable from a register to memory and removes it from the graph. Choosing an appropriate variable to spill is crucial. For many allocators, this decision is handled by a single priority function. Based on an evaluation of relevant data (*e.g.*, number of references, depth in loop nest, etc.), the allocator invokes its priority function to assign a weight to each uncolored variable. Examining the relative weights, the allocator determines which variable to spill.

Compiler writers tediously fine-tune priority functions to achieve suitable performance [2]. Priority functions are widely used and tied to complicated factors. A non-exhaustive list of examples, just in compilation, includes list scheduling [9], clustered scheduling [14], hyperblock formation [12], meld scheduling [1], modulo scheduling [17] and register allocation [6]. GP's representation appears ideal for improving priority functions. We have tested this observation via two case studies: predication and register allocation.

4 Predication

Studies show that branch instructions account for nearly 20% of all instructions executed in a program [16]. The control dependences that branch instructions impose decrease execution speed and make compiler optimizations difficult. Moreover, the uncertainty surrounding branches makes it difficult (and in many cases impossible) to parallelize disjoint paths of control flow. The data and control dependences may preclude instruction level parallelism.

Unpredictable branches are also incredibly costly on modern day processors. The Pentium® 4 architecture invalidates up to 120 in-flight instructions when it mispredicts. When a branch is mispredicted, not only does the processor have to nullify incorrect operations, it may have to invalidate many unrelated instructions following the branch that are in the pipeline.

The shortcomings of branching have led architects to rejuvenate *predication*. Predication allows a processor that can execute and issue more than one instruction at a time to simultaneously execute the taken and fall-through paths of control flow. The processor nullifies all instructions in the incorrect path. A predicate operand guards the execution of every instruction to ensure only correct paths modify processor state.

Trimaran's predication algorithm identifies code *regions* that are suitable for predication. It then enumerates paths (*i.e.*, sequences of instructions that it must merge into a predicated hyperblock). Merging depends on the compiler's confidence that a path is processor efficient. The priority function assigns the confidence value of a path.

Trimaran's priority function is shown in Equation 1. In addition to considering the probability of path execution, this priority function penalizes paths that have hazards (*e.g.*, pointer dereferences), relatively large dependence height, or too many instructions.

$$priority_i = exec\_ratio_i \cdot h_i \cdot (2.1 - d\_ratio_i - o\_ratio_i)\, where \tag{1}$$

$$h_i = \begin{cases} 0.25 & : \text{ if } path_i \text{ contains a hazard.} \\ 1 & : \text{ if } path_i \text{ is hazard free.} \end{cases}$$

$$d\_ratio_i = \frac{dep\_height_i}{\max_{j=1\to N} dep\_height_j}, \qquad o\_ratio_i = \frac{num\_ops_i}{\max_{j=1\to N} num\_ops_j}$$

The variable *exec_ratio*, which is based on a runtime profile, is the probability that the path is executed; num_ops_i refers to the number of operations in $path_i$, and *dep_height* is the extent of control dependence.

4.1 Predication Primitives

In addition to the path properties used in Equation 1, there are other salient properties that could potentially distinguish good paths from useless paths. We created a GP terminal corresponding to each property. Tables 1 and 5 contain lists of the primitives we use.

5 Priority-Based Coloring Register Allocation

The gap between register access times and memory access times is growing. Therefore, register allocation, the process of assigning variables to fast registers, is an increasingly important compiler optimization. Many register allocation algorithms use cost functions to determine which variables to spill when spilling is required.

Property	Description
dep_height	The maximum instruction dependence height over all instructions in path.
num_ops	The total number of instructions in the path.
exec_ratio	How frequently this path is executed compared to other paths considered (from profile).
num_branches	The total number of branches in the path.
predictability	Path predictability obtained by simulating a branch predictor (from profile).
avg_ops_executed	The average number of instructions executed in the path (from profile).
unsafe_JSR	If the path contains a subroutine call that may have side-effects, it returns *true*; otherwise it returns *false*.
safe_JSR	If the path contains a side-effect free subroutine call, it returns *true*; otherwise it returns *false*.
mem_hazard	If the path contains an unresolvable memory access, it returns *true*; otherwise it returns *false*.
max_dep_height	The maximum dependence height over all paths considered for hyperblock inclusion.
total_ops	The sum of all instructions in paths considered for hyperblock inclusion.
num_paths	Number of paths considered for hyperblock inclusion.

Table 1. GP Terminals for Predication Experiments. These properties may influence predication. Some are extracted from profile information while others do not require program execution. We also include the min, mean, max, and standard deviation of all paths to provide macroscopic information.

For instance in priority-based coloring register allocation, the priority function is an estimate of the relative benefits of storing a given variable[3] in a register [6]. The algorithm then assigns variables to registers in priority order. The success of the register allocation algorithm depends on the priority function.

Priority-based coloring first associates a *live range* with every variable. This range simply denotes the portion of code in which a variable is *live*. More specifically, a live range is the composition of code segments (basic blocks), through which the associated variable's value must be preserved. The algorithm then prioritizes each live range based on the estimated execution savings of register allocating the associated variable:

$$savings_i = w_i \cdot (LDsave \cdot uses_i + STsave \cdot defs_i) \tag{2}$$

$$priority(lr) = \frac{\sum_{i \in lr} savings_i}{N} \tag{3}$$

Equation 2 is used to compute the savings of each code segment. $LDsave$ and $STsave$ are estimates of the execution time saved by keeping the associated variable in a register for references and definitions respectively. $uses_i$ and $defs_i$ represent the number of uses and definitions of a variable in code segment i. w_i is the estimated execution frequency for the segment.

Equation 3 sums the savings over the N code segments that compose the live range. Thus, this priority function represents the savings incurred by accessing a register instead of resorting to main memory.

[3] For ease of explanation, our description of priority-based register allocation is not precisely accurate. A single variable may actually be assigned to several different registers. See [6] for details.

Property	Description
spill_cost	The estimated cost of spilling this range to memory. See Equation 2.
region_weight	Number of times the basic block was executed (from profile).
live_ops	The number of live operations in the block.
num_calls	The number of procedure calls in a basic block.
callee_benefit	The callee's 'benefit' of allocating the range.
caller_benefit	The caller's 'benefit' of allocating the range.
def_num	The number of definitions in the block.
use_num	The number of uses in the block.
STsave	Estimate of the execution time saved by keeping a definition in a register.
LDsave	Estimate of the execution time saved by keeping a reference in a register.
has_single_ref	If the block has a single reference, return *true*, otherwise return *false*.
is_pass_through	If the number of live references in the block is greater than 0, return *true*, otherwise return *false*.
ref_op_count	The number of references in the block.
reg_size	The number of registers available for the register class of the live range.
forbidden_regs	The number of registers that are not available to the live range (because it interferes with an allocated live range).
GPR, FPR, PR	If the live range belongs to the class GPR, FPR, or PR respectively, return *true*, otherwise return *false*.

Table 2. GP Terminals for register allocation experiments.

5.1 Register Allocation Primitives

Trimaran's register allocation heuristic essentially works at the basic block level. To improve register allocation we evolved an expression to replace Equation 2. Since Equation 3 simply sums and normalizes the priorities of the individual basic blocks, we leave it intact. Table 2 shows the quantities we used as GP terminals for the priority-based coloring register allocator.

6 Experimental Parameters

6.1 Infrastructure

Our experimental infrastructure is built upon Trimaran [19]. Trimaran is an integrated compiler and simulator for a parameterized EPIC (Explicitly Parallel Instruction Computing) architecture. Trimaran's compiler, which is called IMPACT, performs code profiling. Table 3 details the specific architecture over which we evolved. This model is similar to Intel's Itanium architecture. We enabled the following Trimaran compiler optimizations: function inlining, loop unrolling, backedge coalescing, acyclic global scheduling [5], modulo scheduling [20], hyperblock formation, register allocation, machine-specific peephole optimization, and several other classic optimizations.

We built a GP loop around Trimaran and internally modified IMPACT by replacing its predication priority function (Equation 1) with our GP expression parser and evaluator. The predication algorithm provides variable bindings for the primitives, and most of these were already available in IMPACT. We modified the compiler's profiler to extract branch predictability statistics. We added the minimum, maximum, mean, and standard deviation of all path-specific characteristics, which together encapsulate some global knowledge. In addition, we added a 2-bit dynamic branch predictor to the simulator.

Feature	Description
Registers	64 general-purpose registers, 64 floating-point registers, and 256 predicate registers.
Integer units	4 fully-pipelined units with 1-cycle latencies, except for multiply instructions, which require 3 cycles, and divide instructions, which require 8.
Floating-point units	2 fully-pipelined units with 3-cycle latencies, except for divide instructions, which require 8 cycles.
Memory units	2 memory units. L1 cache accesses take 2 cycles, L2 accesses take 7 cycles, and L3 accesses require 35 cycles. Stores are buffered, and thus require 1 cycle.
Branch unit	1 branch unit.
Branch prediction	2-bit branch predictor with a 5-cycle branch misprediction penalty.

Table 3. Characteristics of the EPIC architecture.

Similarly, to study register allocation we modified Trimaran's Elcor register allocator by replacing its priority function (Equation 2) with another expression parser and evaluator. To more effectively stress the register allocator, we only use 32 general-purpose registers and 32 floating-point registers.

6.2 GP Run Parameters

For each run of 50 generations, the initial population consists of 399 randomly initialized expressions, as well as Trimaran's original priority function (Equation 1 for hyperblock formation, and Equation 2 for register allocation). Tournament selection with a tournament size of seven is used. We *randomly* replace 22% of the population every generation with offspring adapted by mutation and crossover. Roughly 5% of the offspring are mutated, and the remainder result from crossover. Only the *single* best expression is guaranteed survival. In addition to the specialized primitives in Table 1 and Table 2, we add the standard arithmetic and boolean logical and comparison primitives listed in Tables 4 and 5.

Real-Valued Function	Representation
$Real_1 + Real_2$	(add $Real_1$ $Real_2$)
$Real_1 - Real_2$	(sub $Real_1$ $Real_2$)
$Real_1 \cdot Real_2$	(mul $Real_1$ $Real_2$)
$\begin{cases} Real_1/Real_2 & : \text{if} Real_2 \neq 0 \\ 0 & : \text{if } Real_2 = 0 \end{cases}$	(div $Real_1$ $Real_2$)
$\sqrt{Real_1}$	(sqrt $Real_1$)
$\begin{cases} Real_1 & : \text{if} Bool_1 \\ Real_2 & : \text{if not} Bool_1 \end{cases}$	(tern $Bool_1$ $Real_1$ $Real_2$)
$\begin{cases} Real_1 \cdot Real_2 & : \text{if} Bool_1 \\ Real_2 & : \text{if not} Bool_1 \end{cases}$	(cmul $Bool_1$ $Real_1$ $Real_2$)
Returns real constant K	(rconst K)
Returns real value of *arg* from environment	(rarg *arg*)

Table 4. General real-valued functions included in the primitive set.

6.3 Evaluation

The results presented in this paper use total execution time (reported by the Trimaran system) for either one or two sets of input data to assign fitness. This approach rewards the optimization of frequently executed procedures, and therefore,

Boolean-Valued Function	Representation
$Bool_1$ and $Bool_2$	(and $Bool_1$ $Bool_2$)
$Bool_1$ or $Bool_2$	(or $Bool_1$ $Bool_2$)
not $Bool_1$	(not $Bool_1$)
$Real_1 < Real_2$	(lt $Real_1$ $Real_2$)
$Real_1 > Real_2$	(gt $Real_1$ $Real_2$)
$Real_1 = Real_2$	(eq $Real_1$ $Real_2$)
Returns Boolean constant $\{true, false\}$	(bconst $\{true, false\}$)
Returns Boolean value of arg from environment	(barg arg)

Table 5. General purpose GP primitives. Both experiments use the primitives shown in this table.

it may be slow to converge upon general-purpose solutions. However, when one wants to specialize a compiler for a given program, this evaluation of fitness works extremely well. Our system rewards *parsimony* by selecting the smaller of two otherwise equally fit expressions [11, p. 109].

Our experiments select training and testing programs from a suite of 24 benchmarks listed in Table 6. We run GP on nine benchmarks to examine specialized predication priority functions and six benchmarks for register allocation priority functions.

To find a general-purpose priority function (*i.e.*, a function that works well for multiple programs), we run GP on a set of 'training' programs, each with one set of input data. To avoid the computational expense of a large training set, we use dynamic subset selection (DSS) [8]. DSS essentially selects different subsets (size 4,5, or 6) of the benchmark training set that is used for fitness evaluation. Subset selection is based on how poorly the current best expression performs. Thus, hard to optimize training benchmarks are more likely to appear in the training set. The training set consists of twelve and eight benchmarks for predication and register allocation, respectively.

We present the best results of all runs completed to date. This illustrates our focus on application performance. We used the recognized benchmarks in Table 6 to evaluate evolved priority functions. The set includes all of the Trimaran certified benchmarks[4] [19] and most of the Mediabench benchmarks.

7 Results

7.1 Predication: Specialized Priority Functions

Specialized heuristics are created by optimizing a priority function for a particular benchmark evaluated with one set of input data. Figure 1 shows that GP is extremely effective on this basis. The dark bar shows the speedup of each benchmark, over Trimaran's baseline heuristic, when run with the same data on which it was trained. The light bar shows the speedup when alternate input data is used. We obtain an average speedup of 23% (up to 73%) for our evaluation suite.

As we would expect, in most cases the speedup on the training data is greater than that achieved on the test data. The alternate input data likely exercises different paths of control flow—paths which may have been unused during training.

In most runs, the *initial* population contains at least one expression that outperforms the baseline. This means that by simply creating and testing 399 random expressions, we were able to find a priority function that outperformed Trimaran's

[4] We could not get 134.perl to execute correctly, though [19] certified it.

Benchmark	Suite	Description
codrle4 decodrle4	See [3]	RLE type 4 encoder/decoder.
huff_enc huff_dec	See [3]	A Huffman encoder/decoder.
djpeg	Mediabench	Lossy still image decompressor.
g721encode g721decode	Mediabench	CCITT voice compressor/decompressor.
mpeg2dec	Mediabench	Lossy video decompressor.
rasta	Mediabench	Speech recognition application.
rawcaudio rawdaudio	Mediabench	Adaptive differential pulse code modulation audio encoder/decoder.
toast	Mediabench	Speech transcoder.
unepic	Mediabench	Experimental image decompressor.
085.cc1	SPEC92	gcc C compiler.
052.alvinn	SPEC92	Single-precision neural network training.
179.art	SPEC2000	A neural network-based image recognition algorithm.
osdemo mipmap	Mediabench Mediabench	Part of a 3-D graphics library. similar to OpenGL.
129.compress	SPEC95	In-memory file compressor and decompressor.
023.eqntott	SPEC92	Creates a truth table from a logical representation of a Boolean equation.
132.ijpeg	SPEC95	JPEG compressor and decompressor.
130.li	SPEC95	Lisp interpreter.
124.m88ksim	SPEC95	Processor simulator.
147.vortex	SPEC95	An object oriented database.

Table 6. Benchmarks used. The set includes applications from the SpecInt, SpecFP, Mediabench benchmark suites, and a few miscellaneous programs.

for the given benchmark and input data. In many cases, GP finds a superior priority function quickly, and finds only marginal improvements as the evolution continues. In fact, the baseline priority function is often quickly obscured by GP-generated expressions. Note, however, that human designed priority functions may have been designed for more generality than can be evaluated in our investigative setup.

Once GP has homed in on a fairly good solution, the search space and operator dynamics are such that most offspring will be worse, some will be equal and very few turn out to be better. This seems indicative of a steep hill in the solution space. In addition, multiple runs yield only minuscule differences in performance. This might indicate the search space (determined by our primitive set) has many possible solutions associated with a given fitness.

7.2 Predication: Finding General Purpose Priority Functions

We divided the benchmarks in Table 6 into two exclusive sets[5]: a 12 element training set, and a 12 element test set. We then applied the resulting priority function to all 12 benchmarks in the test set. Since the benchmarks in the test set are not related to the benchmarks in the training set, this is a measure of the priority function's generality.

Figure 2 shows the results of applying the single best priority function to the benchmarks in the training set. The dark bar associated with each benchmark is the speedup over Trimaran's base heuristic when the training input data is used.

[5] We chose to train mostly on Mediabench applications because they compile and run faster than the Spec benchmarks. However, we randomly chose two Spec benchmarks for added coverage.

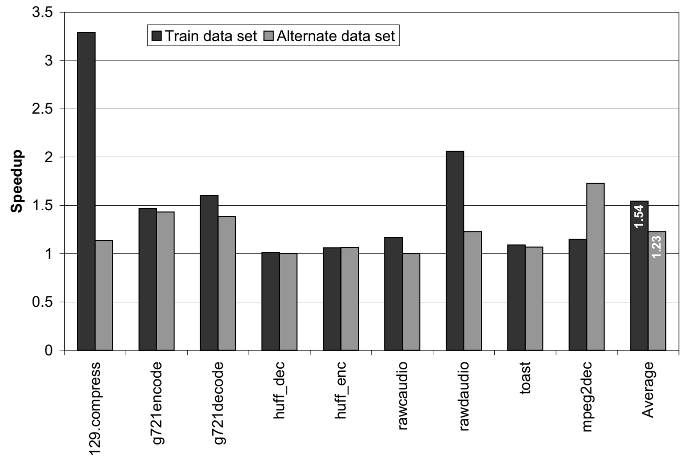

Fig. 1. GP Evolved Specialized Predication Priority Functions. Dark colored bars indicate speedup over Trimaran's baseline heuristic when using the same input data set on which the specialized priority function was trained. The light colored bars show speedup when alternate input data was tested.

This data set yields a 44% improvement. The light bar shows results when alternate input data is used. The overall improvement for this set is 25%.

It is interesting that, on average, the general-purpose priority function outperforms the application-specific priority function for the alternate data set. The general-purpose solution is less susceptible to variations in input data precisely because it is more generally applicable.

Figure 3 shows how well the best general purpose priority function performed on the test set. The average speedup over the test set is 9%. In three cases (unepic, 023.eqntott, and 085.cc1) Trimaran's baseline heuristic marginally outperforms the GP-generated priority function. For the remaining benchmarks, the heuristic our system found is better.

7.3 Register Allocation: Specialized Priority Functions

Figures 4 shows speedups obtained by specializing the Trimaran register allocator's priority function for specific benchmarks. The dark bar associated with each benchmark represents the speedup obtained by using the same input data that was used to specialize the heuristic. The light bar shows the speedup when an alternate input data set is used. GP evolved register allocation functions that improve the heuristic described in Section 5 by up to 13%.

Once again, it makes sense that the relative performance on training input data is better than that achieved on the alternate input data. In contrast to predication, however, with register allocation, we see that the difference between the two is less pronounced. This is likely because predication is extremely data-driven and thus vulnerable to diverse input data. An examination of the general-purpose predication heuristic reveals two dynamic factors (*exec_ratio* and *predictability*) that are critical components in the hyperblock decision process.

Figure 5 plots the best individual's speedup over generations. The fairly constant improvement in speedup over several generations seems to suggest that this problem is harder to optimize than predication. Additionally, unlike the predication algorithm, the baseline heuristic was typically retained (*i.e.*, it remained in the population) for several generations.

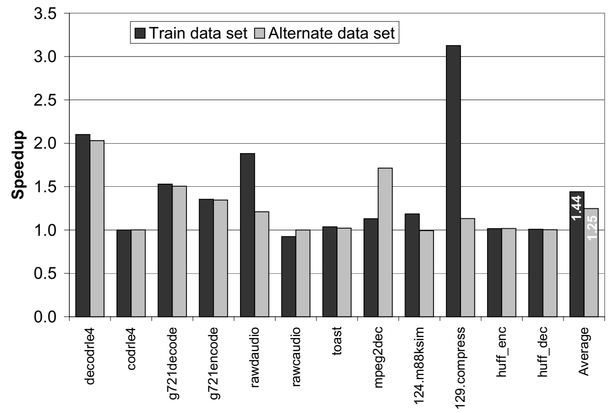

Fig. 2. GP Performance with General Purpose Predication Priority Functions. Training on multiple benchmarks. A *single* priority function was obtained by training over all the benchmarks in this graph. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to an alternate data set.

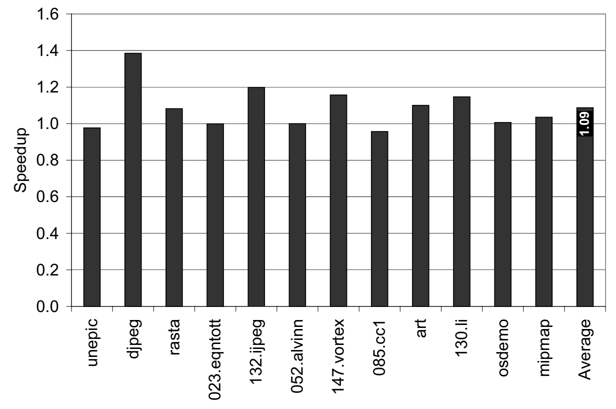

Fig. 3. GP Performance with General Purpose Predication Priority Functions. Cross validation of the general-purpose priority function. The best priority function found by training on the benchmarks in Figure 2 is applied to the benchmarks in this graph.

7.4 Register Allocation: General Purpose Priority Functions

Just as we did in Section 7.2, we divide our benchmarks into a training set and a test set[6].

The benchmarks in Figure 6 show the training set for this experiment. The figure also shows the results of applying the best priority function (from our DSS run) to all the benchmarks in the set. The dark bar associated with each benchmark is the speedup over Trimaran's base heuristic when using the training input data. The average for this data set is 3%. The light bar shows results when alternate data is used. An average speedup of 3% is also attained with this data.

Figure 7 shows the test set for this experiment. The figure shows the speedups (over Trimaran's baseline) achieved by applying the single best priority function to all the benchmarks. Even though we trained on a 32-register machine, we also

[6] This experiment uses smaller test and training sets due to preexisting bugs in Trimaran. It does not correctly compile several of our benchmarks when targeting a machine with 32 registers.

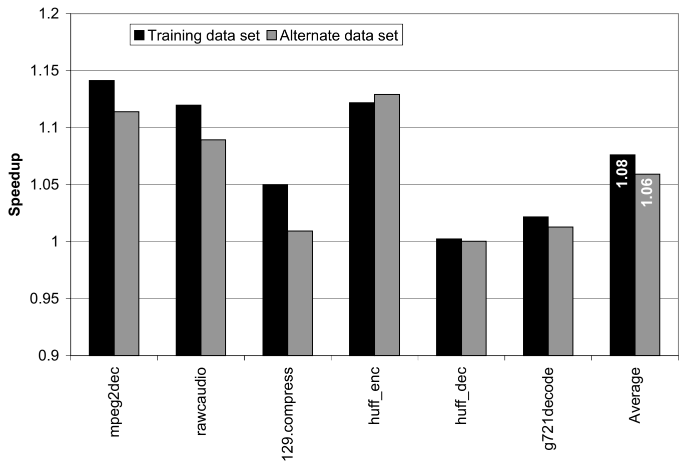

Fig. 4. GP Performance on Specialized Register Allocation Priority Functions. The dark colored bars are speedups using the same data set on which the specialized priority function was trained. The light colored bars are speedups that use an alternate data set.

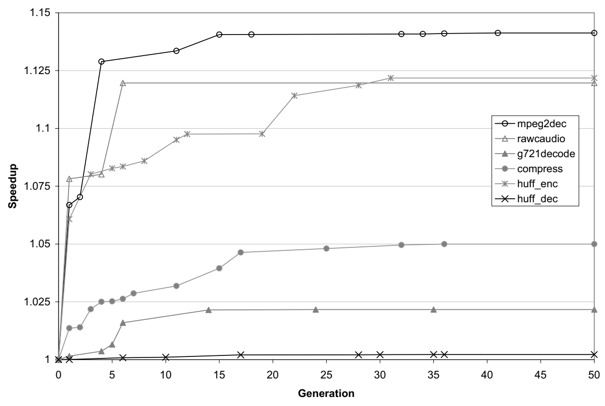

Fig. 5. GP Evolution on Specialized Register Allocation Priority Functions. This figure graphs fitness over generations. Unlike the hyperblock selection evolution, these fitnesses converge slowly.

apply the priority function to a 64-register machine. It is interesting that the learned priority function is not only stable across benchmarks, it is also stable across similar platforms.

8 Conclusions

We used GP in a straightforward fashion to optimize priority functions in compiler heuristics and observed impressive results that improve on existing ones. These results are valuable to the compiler development community because the speedup comes in code sections that are difficult to hand-optimize because of nonlinearities within the compiler as well as overwhelming complexity in the target processor architecture. GP is especially appropriate for this application because it offers a convenient representation; priority functions are executable expressions. In addition, GP proved capable of searching the solution space of the two compiler problems that we investigated in this paper. In general, GP is valuable to the compiler development community because priority functions are prevalent in compiler heuristics.

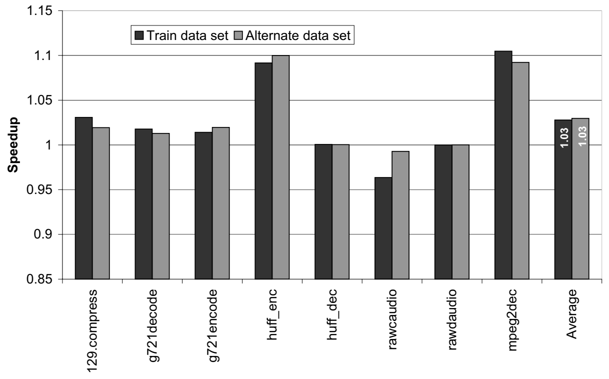

Fig. 6. GP Performance with General Purpose Register Allocation Priority Functions. Training on multiple benchmarks. Our DSS evolution trained on all the benchmarks in this figure. The single best priority function was applied to all the benchmarks. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to an alternate data set.

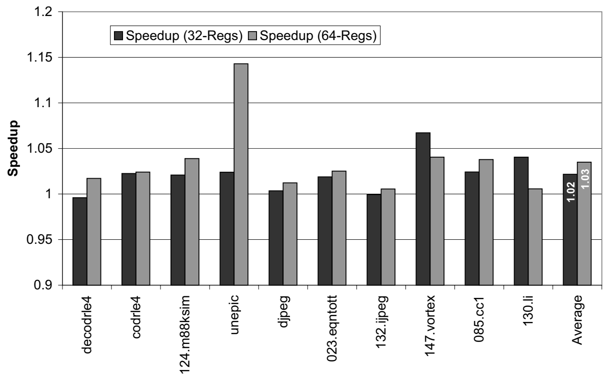

Fig. 7. GP Performance with General Purpose Register Allocation Priority Functions. Cross validation of the general-purpose priority function. The best priority function found by the DSS run is applied to the benchmarks in this graph. Results from two target architectures are shown.

Our results suggest that GP offers both great potential and convenience in the niche of optimizing isolatable, smaller-sized code sections that perform heuristics within a larger software system base.

9 Future Work

We have presented our research as a proof of concept and as such, to derive general conclusions, we have used a very standard ('off the shelf') version of GP in terms of operators and the selection algorithm used. We have not added any 'bells and whistles' that might offer more improvement. In order to improve the results for a specific compiler, we must strive to better understand our system. To this end we plan to study the priority functions' fitness landscapes via hillclimbing, simplify and better understand the GP evolved expressions, and conduct additional runs with some parameter experimentation.

References

1. S. G. Abraham, V. Kathail, and B. L. Deitrich. Meld Scheduling: Relaxing Scheduling Constaints Across Region Boundaries. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 308–321, 1996.
2. D. Bernstein, D. Goldin, and M. G. et. al. Spill Code Minimization Techniques for Optimizing Compilers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, 1989.
3. D. Bourgin. *http://hpux.u-aizu.ac.jp/hppd/hpux/Languages/codecs-1.0/*. Losslessy compression schemes.
4. B. Calder, D. G. ad Michael Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (ToPLaS-19)*, volume 19, 1997.
5. P. Chang, D. Lavery, S. Mahlke, W. Chen, and W. Hwu. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined processors. In *IEEE Transactions on Computers*, volume 44, pages 353–370, March 1995.
6. F. C. Chow and J. L. Hennessey. The Priority-Based Coloring Approch to Register Allocation. In *ACM Transactions on Programming Languages and Systems (ToPLaS-12)*, pages 501–536, 1990.
7. K. Cooper, P. Scheilke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.
8. C. Gathercole. *An Investigation of Supervised Learning in Genetic Programming.* PhD thesis, University of Edinburgh, 1998.
9. P. B. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, volume 21, pages 11–16, 1986.
10. G. W. Grewal and C. T. Wilson. Mappping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). In *International Symposium on Microarchitecture*, volume 34, pages 192–202, 2001.
11. J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.
12. S. A. Mahlke. *Exploiting instruction level parallelism in the presence of branches.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1996.
13. M. O'Neill and C. Ryan. Automatic generation of caching algorithms. In K. Miettinen, M. M. Mkel, P. Neittaanmki, and J. Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134, Jyvskyl, Finland, 30 May - 3 June 1999. John Wiley & Sons.
14. E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register Filee Microarchitectures.
15. N. Paterson and M. Livesey. Evolving caching algorithms in C by genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 262–267, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
16. D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.
17. B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1994.
18. C. Ryan and P. Walsh. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *Proceedings of ParCo'95*. North-Holland, 1995.
19. Trimaran. *http://www.trimaran.org*.
20. N. Warter. *Modulo Scheduling with Isomorphic Control Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1993.