Optimizing for Reduced Code Space using Genetic Algorithms

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian
Department of Computer Science
Rice University
Houston, Texas, USA
{keith | phisch | devika}@cs.rice.edu

Abstract

Code space is a critical issue facing designers of software for embedded systems. Many traditional compiler optimizations are designed to reduce the execution time of compiled code, but not necessarily the size of the compiled code. Further, different results can be achieved by running some optimizations more than once and changing the order in which optimizations are applied. Register allocation only complicates matters, as the interactions between different optimizations can cause more spill code to be generated. The compiler for embedded systems, then, must take care to use the best sequence of optimizations to minimize code space.

Since much of the code for embedded systems is compiled once and then burned into ROM, the software designer will often tolerate much longer compile times in the hope of reducing the size of the compiled code. We take advantage of this by using a genetic algorithm to find optimization sequences that generate small object codes. The solutions generated by this algorithm are compared to solutions found using a fixed optimization sequence and solutions found by testing random optimization sequences. Based on the results found by the genetic algorithm, a new fixed sequence is developed to reduce code size. Finally, we explore the idea of using different optimization sequences for different modules and functions of the same program.

1 Introduction

For years the compiler community has been developing "optimizations", or transformations, whose ultimate goal is to improve the quality of the code generated by the compiler. For example, constant propagation attempts to find all registers whose values at run time are provably constant. The computations of these values can be replaced by a less costly load immediate operation. Later, an optimization to eliminate useless code can remove all the load immediate operations whose result is never used.

With so many optimizations available to the compiler, it is virtually impossible to select the best set of optimizations to run on a particular piece of code. Historically, compiler writers have made one of two assumptions. Either a fixed optimization order is "good enough" for all programs, or giving the user a large set of flags that control optimization is sufficient, because it shifts the burden onto the user.

Interplay between optimizations occurs frequently. A transformation can create opportunities for other transformations. Similarly, a transformation can eliminate opportunities for other transformations. These interactions also depend on the program being compiled, and they are often difficult to predict. Multiple applications of the same transformation at different points in the optimization sequence may produce further improvements. With all these possibilities, it is unlikely that a single fixed sequence of optimizations will be able to produce optimal results for all programs.

The focus of this work is discovering optimization sequences that result in small code sizes. Small code sizes are increasingly important as the number of embedded applications increases. Since code for these systems is typically burned into a ROM, the embedded systems software designer may be willing to tolerate substantially longer compile times (on the "final" compile) in the hopes of reducing the size of the object code.

A genetic algorithm (or GA) is essentially a biased sampling search technique. Instead of merely choosing solutions at random, the GA evolves solutions by merging parts of different solutions and making small mutational changes to solutions, to produce new solutions. Genetic algorithms have a rich history in the literature on artificial intelligence. The idea is generally attributed to Holland [14]. Other excellent sources of information on GA's are books by Goldberg [13], Davis [9], and Mitchell [17].

Genetic algorithms have been applied to other compiler problems with some success. In his thesis, Beaty tried genetic algorithms for instruction scheduling [1]. Nisbet is currently investigating genetic algorithms that discover transformation sequences for parallelizing FORTRAN loops [19].

GA's appear to be a good fit to the problem of finding compiler optimization sequences to minimize code size. First, the space of possible sequences is quite large when several optimizations are at the compiler's disposal. This space is too large to search completely. Second, we have a very good evaluation function to assess the quality of a solution. We simply perform the optimizations and compute the size of the compiled code. Our objective function is discrete and nonlinear, making the problem difficult to address with more classical combinatorial optimization techniques. Finally, the amount of time spent by the GA is flexible. More computation time may result in better solutions, but the algorithm can be interrupted at any time to return the best solution found.

Our experiments show that GA's are well suited to the problem of finding good optimization sequences. For each program in our set of benchmarks, the GA discovered a sequence that produces smaller codes than the hand-designed sequence used in our compiler for many years. In many cases, the resulting code is also faster. These results showed us how to design a different fixed sequence that produces smaller code than our default sequence. Finally, we show that additional savings in space of up to five percent can be obtained by running the GA on a program-by-program basis versus a single fixed sequence for all programs. This suggests the potential of designing optimization sequences that are custom-fitted for a program, rather than constructing a universal, one-size-fits-all sequence.

In Section 2, we give the necessary details of our research compiler and describe the optimizations used in our experiments. Section 3 gives an overview of genetic algorithms and provides the details of the GA we used in our experiments. In Section 4, we present some experimental results for several benchmark programs. We compare code sizes for compiling with no optimization, fixed optimization sequences, and optimizing using sequences found by the genetic algorithm. Finally, in Section 5, we present some conclusions, observations, and directions for future work.

2 Our compiler framework

Our research compiler includes front ends for C and FORTRAN. The front ends translate the source code into our low-level intermediate language, called ILOC. ILOC is an assembly-level language and is RISC-like; operands for all computational operations must be in registers. Several flavors of load and store operations transfer values between registers and memory.

Once the code has been transformed into ILOC, we have various optimization passes that can be applied. Each optimization is an ILOC-to-ILOC transformation designed to improve the code in some way. In the experiments for this paper, the following 10 program transformations were used:

1. Constant propagation (`cprop`): This pass is an implementation of Wegman and Zadeck's sparse conditional constant propagation technique [22]. We run cprop with an option (`-m`) that disables conversion of multiplies into shifts and adds. That conversion forestalls some subsequent optimizations and increases code size. We also enable the option (`-f`) that performs propagation on symbolic constants.

2. Dead Code Elimination (`dead`): Our dead code eliminator is based on Kennedy's work [15] and uses SSA form [8].

3. Empty Basic Block Removal (`clean`): This pass removes basic blocks that contain only a single control-flow operation.

4. Global Value Numbering (`valnum`): Value numbering is done using the SCC-based value numbering described by Cooper, Briggs, and Simpson [6, 3]. The `valnum` implementation has many options. The work in this paper used the following settings: perform constant folding (`-f`), trace values through memory (`-m`), remove some redundant computations (`-a`), perform algebraic simplification (`-s`), and perform value driven code motion (`-c`).

5. Lazy Code Motion (`lazy`): This pass performs code motion using techniques described by Drechsler and Stadel [11], and Knoop, *et al.* [16].

6. Partial Redundancy Elimination (`partial`): This pass implements Morel and Renvoise's technique for partial redundancy elimination [18], following Drechsler and Stadel [10].

7. Peephole Optimization (`combine`): This pass is an SSA-based peephole optimizer that combines ILOC operations according to a Power-PC 601 architectural model.

8. Reassociation (`shape`): This pass reorders expressions using commutativity and associativity to expose opportunities for other optimizations [2]. It reassociates complex expressions to expose loop invariant subexpressions.

9. Register Coalescing (`coalesce`): This pass coalesces register-to-register copies using the interference graph of a Chaitin style register allocator [5].

10. Operator Strength Reduction (`strength`): This pass performs operator strength reduction using an SSA-based algorithm due to Cooper, Simpson, and Vick [7]. The algorithm differs from prior work in that it finds secondary effects in a single pass—it need not be repeated in an iterative fashion.

It should be noted that a simple global value-numbering pass that renumbers registers is run before `partial` and `lazy` to ensure that the input ILOC conforms to the naming requirements of these passes. Additionally, `shape` is run before `partial` to enforce other code shape requirements.

To gather execution statistics and test for correctness. the ILOC code is translated into C code. This C code is instrumented to tabulate statistics, including the number of operations executed during a run of the program. The C code is then compiled and run.

This work focuses on finding optimization sequences that result in reduced code size. Typically, this would refer to the number of bytes required to store the object code. The analog to this in our ILOC framework is the *static operation count*. That is, the number of ILOC operations in the optimized code. A reasonable approximation to the number of bytes the code requires can be made by assuming each ILOC operation occupies a four byte word.

3 The genetic algorithm

Genetic algorithms are basically search algorithms designed to mimic the process of natural selection and evolution in nature. To begin, we need to define a few terms. A *population* consists of a fixed number of members, or *chromosomes*. Each chromosome is a fixed-length string of *genes*. The *fitness* of a chromosome is some measure of how desirable it is to have that chromosome in the population. A *generation* is a time step in which several events occur. Some of the most "unfit" chromosomes die and are removed from the population. To replace these, some number of *crossover* operations are applied to the population. A crossover is an operation analogous to mating or gene splicing. It combines part of one chromosome with part of another chromosome to create a new chromosome. Finally, some amount of *mutation* occurs in the population, in which genes are changed randomly with some (typically low) probability. It is also possible to have *elitism* in the population in which some of the most fit genes are immune from mutation between generations.

Using these definitions, we define a GA to find customized compiler optimization sequences. As described in the previous section, we have ten optimizations at our disposal. An optimization is analogous to a gene. We have selected a chromosome length of twelve genes. This corresponds to performing a sequence of twelve optimizations. (The selection of twelve is arbitrary. The fixed sequence used as a default in our system has length 12. We describe it in Section 4.) We have ten different genes and chromosomes of length twelve for a total solution-space size of 10^{12}. We set a population size of twenty chromosomes. Larger populations do not produce appreciably different results. The fitness value for a particular chromosome (optimization sequence) is the size of the object code produced when that optimization sequence is applied to the source code. Unlike typical genetic algorithms, we desire chromosomes with *low* fitness values since these correspond to smaller object codes.

Each generation in our experiments consists of the following steps:

1. Compute a fitness value for each chromosome. The code being compiled is passed through the front end of the compiler and then optimized according to the optimization sequence defined by the chromosome. After optimization, we remove empty basic blocks from the code (*i.e.*, a final pass of the `clean` optimization) and perform Briggs style register allocation with 32 integer and 32 floating-point registers [4]. The number of static ILOC operations in the final code is recorded as the fitness value for that chromosome.

2. The chromosomes are sorted by fitness values from lowest to highest. The population is split into a lower and upper half, based on fitness value, each half consisting of 10 chromosomes. The upper half consists of the 10 chromosomes with the lowest fitness values.

3. The chromosome with the highest fitness value (worst performance) is removed from the population. Three additional chromosomes are chosen at random from the lower half and removed from the population.

4. To fill the four vacancies in the population, new chromosomes are generated using the "crossover" operation. Two "parent" chromosomes are randomly chosen from the upper half of the population. The first six genes of one chromosome are concatenated with the last six genes of the other chromosome and vice versa, creating two new chromosomes.[1] This operation is performed twice, to produce four new chromosomes.

5. Next, fifteen chromosomes are subjected to mutation. The best performing chromosome is exempted from mutation, a form of elitism. The four chromosomes created by crossover in the previous step are also exempted. In the fifteen chromosomes subject to mutation, each gene is considered. For a chromosome in the lower half of the population, mutation occurs with a probability of 0.1 (or 10 percent). For a chromosome in the upper half of the population, the probability of mutation is reduced to 0.05 (or 5 percent). To mutate a gene, it is replaced with a randomly selected gene.

6. Duplicate chromosomes are removed from the population and replaced with random chromosomes.

This process is repeated for 1000 generations, and we keep track of the best chromosome found over the course of the run.

[1] The current version of our algorithm splits the chromosome in a random position. This change does not noticeably alter our results.

	Unoptimized		GA results					
Benchmark	static	dynamic	static	% red.	dynamic	% red.	gen. found
adpcm	438	17221981	351	19.9	12290460	28.6	6
compress	1753	8402188	1318	24.8	5545480	34.0	(77,79)
dfa	1744	842382	1107	36.5	496164	41.1	806
dhrystone	760	4920264	536	29.5	3200191	35.0	(22,920)
fft	2415	18339859	1757	27.2	14574279	20.5	2
fmin	374	2192	187	50.0	963	56.1	32
nsieve	353	761244374	202	42.8	539954218	29.1	(0, 189)
rkf45	1525	511251	745	51.1	201470	60.6	74
seval	1061	3594	288	72.9	842	76.6	39
solve	1023	2729	437	57.3	1029	62.3	(33,58)
svd	2087	13049	972	53.4	4760	63.5	26
tomcatv	2250	1379982621	551	75.5	232833969	83.1	90
urand	204	1563	93	54.4	613	60.1	(0,18)
zeroin	273	1815	150	45.1	809	55.4	(239,270)

Table 1: Optimizing with the GA.

When computing fitness values, we also simulate a run of the optimized code to test for correctness. If any error was encountered during the optimization or the execution, a fitness value of infinity is assigned to the chromosome.

Since we run the code each time we optimize it, we also keep track of the number of operations executed during the run. Each time any ILOC operation is executed, the *dynamic* operation counter is incremented. The number of dynamic operations is a crude estimate of the number of CPU cycles needed to execute the program. Operations that typically require more time to execute (like divide) are *not* weighted in this count. We use these dynamic operation counts as secondary fitness values. When sorting the chromosomes by fitness value (static operation count), we break any ties by looking at the dynamic operation count. Thus, the final optimization sequence returned by the algorithm has the lowest dynamic operation count of all optimization sequences of the given fitness.

Note that the runtime bottleneck of this genetic algorithm process is the computation of the fitness values. Also notice that at least one chromosome (the one with the best fitness value) remains unchanged from one generation to the next. Thus, there is no need to recompute the fitness value for that chromosome. It is likely that many of the other chromosomes may not be mutated. Over the course of 1000 generations many chromosomes get "regenerated". On average, over the whole run of 1000 generations, we found that about 45 per cent of chromosomes already had fitness values computed previously. To improve the runtime performance of the algorithm we keep a hash table of fitness values indexed by chromosome. When computing fitness values, we first check the hash table to see if the fitness value for that chromosome is already available. If so, there is no need to recompute it and we move on to the next chromosome.

4 Experimental Results

To test the efficacy of our GA, we used it to find optimization sequences for several benchmark programs. The FORTRAN programs used in the experiments were `fmin`, `rkf45`, `seval`, `solve`, `svd`, `urand`, and `zeroin` from the FMM benchmark suite [12] and `tomcatv` from SPEC. The C codes used were `adpcm`, which performs adaptive differential pulse code modulation, `compress`, which is the UNIX file-compression utility, `fft`, a fast Fourier transformation algorithm, `dfa`, an implementation of the Knuth-Morris-Pratt string-matching algorithm, `dhrystone`, a common hardware benchmark, and `nsieve`, code that implements the Sieve of Eratosthenes as written by Al Aburto.

The GA described in the previous section is implemented with a Perl script. The script determines the optimization sequence according to the algorithm and then calls the program `ctest`, which is the optimization-testing program for our research compiler. `Ctest` runs the corresponding optimization passes on the ILOC code, including one additional pass of `clean` and the register allocator. Finally, `ctest` has the resultant ILOC code converted to C, compiled, and run. The number of static and dynamic operations are recorded and reported back to the Perl script for use as fitness values and secondary fitness values, respectively.

4.1 Solutions found by the GA

Table 1 presents the results of running our GA on the benchmark codes. Column two shows the unoptimized code size (number of static operations), and column three the unoptimized dynamic operation count. (Unoptimized code has been run through one pass of `clean` and the register allocator.) Column four reports the smallest code size returned by the GA, and column six reports the lowest dynamic oper-

Benchmark	GA sequence	reduced sequence
adpcm	tonosdnzscno	osnzc
compress	ncsnzosvndvs	nczvnds
dfa	rztonvncodvs	rztcodvs
dhrystone	covolcnocdvc	covlcodvc
fft	ovocdvscnnos	ovcdvs
fmin	tdcotcscnnvo	dcsvo
nsieve	snzdvvcdsvss	nzdvcs
rkf45	ovndtdcvsndd	ovndtcvsd
seval	rnldoncvconv	ldovco
solve	dondvnsdsdcv	ovndcv
svd	odvdvdsnnnss	odvs
tomcatv	dntvvccocvdv	tvcodv
urand	cnottcdtvooc	nodvc
zeroin	rsdosvncnsss	rsosvcs

Table 2: Optimization sequences found by the GA.

ation count found for codes having the size reported in column four. (For example, several solutions resulting in object code of size 187 were found for `fmin`. The best dynamic operation count of those solutions was 963.) Columns five and seven report the percentage reduction of static operations and dynamic operations, respectively, over the unoptimized case. Recall that the algorithm is tuned to find the solution with the lowest static size and is only secondarily concerned about execution time results. The last column labeled "gen. found " reports the generation in which the GA found the reported solution. A pair of numbers in this column indicates that the first occurrence of a solution producing the result in column four was found in a different generation than the solution resulting in the best dynamic instruction count as reported in column six. Thus, the first number in the pair is the first generation where a solution resulting in minimal code size appeared. The second number is the generation where the final solution appeared.

The results in Table 1 show the overall impact of optimization on code size and speed for our compiler. Improvements in code space range from 20 to 75 percent. Improvements in speed range from 20 to 83 percent. These numbers show the potential for improvement in our compiler. The size and speed of the unoptimized code depends heavily on the quality of code emitted by the compiler's front end. In our compiler, the code is carefully shaped to expose opportunities for optimization. This may produce slower unoptimized code than necessary. Other authors, however, have reported total improvements that are comparable [20].

Table 2 reports the optimization sequences found by the GA which give the results reported in Table 1. Each letter in the sequence corresponds to one ILOC optimization. The letters correspond to `ctest` options indicating `ctest` should perform that optimization. The mapping from these letters to their corresponding optimizations is given in Table 3.

gene	optimization
c	cprop
d	dead
l	partial
n	clean
o	combine
r	shape
s	coalesce
t	strength
v	valnum
z	lazy

Table 3: Mapping of `ctest` options (genes) to corresponding optimizations.

The last column of Table 2 is a reduced sequence that produces the same results as the sequence found by the GA. These smaller sequences were found by an automatic tool, starting from the sequence returned by the GA and removing optimizations from the sequence one at a time. A simple work-list algorithm is used, the work-list being initialized with the sequence returned by the GA. When a sequence s of length l is removed from the work-list, l sequences of length $l-1$ are generated by removing a single optimization from s. Each shorter sequence is tested and if it produces the same results as the original sequence it is added to the work-list. A hash table of tested solutions is maintained to avoid testing a solution more than once. The process continues until the work-list is empty, and the shortest solution found is returned. Smaller sequences producing the same results may exist.

It is interesting to note which optimizations appear frequently in the sequences (especially the reduced sequences) and which ones do not. Peephole optimization (`combine` or `o`) appears in twelve of the fourteen reduced sequences. Register coalescing (`coalesce` or `s`) and dead code elimination (`dead` or `d`) appear in most sequences. This is not surprising, since these optimizations almost always remove operations from the code. Global value numbering (`valnum` or `v`) was also present in most sequences.

Other transformations were less prevalent. Not surprisingly, partial redundancy elimination (`partial` or `l`) is seen in very few sequences (only twice in `seval`, and `dhrystone`) This is due to the fact that `partial` frequently adds operations to the code in order to reduce the lengths of some paths through the code. The effects of `partial` can reduce run times but tend to increase code size. Another code motion technique, lazy code motion (`lazy` or `z`), appeared in only four cases. `Lazy` can increase code size but tends to produce fewer long register live ranges than `partial`. It should be noted that `valnum` was run with an option that performs value driven code motion as described in Simpson's thesis [21]. Operator strength reduction (`strength`

	rvzcodtvzcod		nodvcodvs	
Benchmark	operations	%	operations	%
adpcm	362	3.0	356	1.4
compress	1412	6.7	1325	0.5
dfa	1168	5.2	1145	3.3
dhrystone	574	6.6	544	1.5
fft	1973	10.9	1757	0.0
fmin	203	7.9	198	5.6
nsieve	227	11.0	202	0.0
rkf45	832	10.5	751	0.8
seval	313	8.0	297	3.0
solve	609	28.2	438	0.2
svd	1641	40.8	973	0.1
tomcatv	770	28.4	565	2.5
urand	93	0.0	93	0.0
zeroin	158	5.1	154	2.6

Table 4: Comparison of GA solutions and fixed sequence solutions – static operation counts. Percentages indicate improvement of GA solution over the fixed sequence.

	rvzcodtvzcod		nodvcodvs	
Benchmark	operations	%	operations	%
adpcm	11965360	-2.7	12440360	1.2
compress	7494982	26.0	5548268	0.1
dfa	546396	9.2	511349	3.0
dhrystone	3390200	5.6	3270193	2.1
fft	15088485	3.4	14574279	0.0
fmin	955	-0.8	947	-1.7
nsieve	570608938	5.4	554450986	2.6
rkf45	214146	5.9	215620	6.6
seval	809	-4.1	985	14.5
solve	1099	6.4	1033	0.4
svd	4731	-0.6	4758	-0.0
tomcatv	186355888	-24.9	239463612	2.8
urand	631	2.9	613	0.0
zeroin	829	2.4	811	0.2

Table 5: Comparison of GA solutions and fixed sequence solutions – dynamic operation counts. Percentages indicate improvement of GA solution over the fixed sequence.

or `t`) appeared in only three reduced sequences (`dfa`, `rkf45`, and `tomcatv`), but appeared more frequently in non-reduced sequences. This suggests that the optimization does not typically increase code size, but tends not to reduce it either. Reassociation (`shape` or `r`) appeared in only two cases (`dfa` and `zeroin`).

There was some repetition of optimizations, but it was not very consistent across benchmarks. Repeated optimizations occurred in the reduced sequences only a few times: `valnum` in `dhrystone`, `fft`, `rkf45`, `solve`, and `tomcatv`; `combine` in `dhrystone` and `seval`; `clean` in compress, `dead` in `rkf45`; `coalesce` three times in `zeroin`; `cprop` three times in `dhrystone`.

The GA required about 1 day (on a Sparc Ultra-10) to run 1000 generations for most benchmarks. This running time is dominated by the time required to compute fitness values, since that requires actually optimizing the code and running it. Some time could be saved by modifications to our testing program `ctest`. With the exception of `dfa` and `zeroin`, the best solution in terms of code size was found by the GA in less than 100 generations. This suggests that with some fine tuning of the GA parameters, the GA would need to be run for far fewer than 1000 generations to produce quality results. Also, we were able to find reduced sequences of nine or fewer optimizations in all cases. This suggests that a smaller chromosome size could be used by the algorithm, further improving total running time.

4.2 Performance of GA against a fixed sequence

Typical compilers present a variety of optimization levels to the user. Each of the optimization levels represents some fixed sequence of optimizations that the compiler applies to the code.

In this section, we investigate the performance of the GA against fixed optimization sequences. We compare the solutions found by the GA to two fixed sequences. The first is used as the default in our system for producing highly optimized code. This sequence was designed to produce code with low *dynamic* operation counts, not necessarily low static code size. This sequence is `rvzcodtvzcod`. The second sequence is based on our observations of optimization sequences returned by the GA. We tried several sequences using optimizations that were prevalent in the reduced sequences of GA solutions and repeated some of them. The best fixed sequence we developed was `nodvcodvs`. `Clean` and the register allocator were run after each optimization sequence. Static code size numbers are presented in Table 4 and dynamic operation counts in Table 5. The percentages indicate the percent reduction given by the GA compared to the fixed string. A negative value indicates that the fixed string performed better. Notice that the GA performed very well against the first sequence in terms of static code size. The second sequence we developed performed quite well against the GA but still lost by as much as 5.6 percent in terms of static code size.

The first sequence (our default sequence) found several solutions resulting in code executing fewer operations at run time, but also had quite a few losses. The GA did slightly better than the second sequence in terms of dynamic operation counts with only a couple of losses, and won by as much as 14.5 percent. Remember that the GA is specifically tuned to find sequences that result in small code. Improved running times are only a secondary goal of the algorithm.

Small changes in the fixed sequence can have dramatic impacts. For example, when the `shape` transformation was inserted into the sequence we developed above, the code size for `dfa` was reduced to 1109 operations from 1145. However, the same sequence *increased* the code size for `tomcatv` to 737 operations, from 565. This emphasizes the potential improvement of using customized optimization sequences for different programs.

4.3 The GA vs. random probing.

We would like some assurance that the GA is doing something useful. That is, can the same results be found quickly by simply trying random optimization sequences? To test this, we performed extensive testing on two benchmarks, `fmin` and `adpcm`. We ran 25 experiments using the GA with different random number seeds. We forced the GA to stop after it found a solution of the same quality as reported in Table 1. The number of unique solutions tested was reported. We also ran 25 experiments where we randomly choose an optimization sequence consisting of 12 optimizations from the pool. Again we stopped after we found a solution of the quality reported in Table 1. The number of unique solutions tested was reported. One would hope that average number of solutions tested by the GA would be less than the average number tested by random sampling to find the same quality solution.

The average number of solutions tested by the random algorithm was 351.8 for `adpcm` with a standard deviation of 311.8. The GA tried an average of 198.7 solutions with a standard deviation of 129.1. At the .05 level of significance the random algorithm tests on average more than 42 solutions more than the GA. On average 3330.9 random solutions were tested for `fmin` with a standard deviation of 2513.0. The GA tested on average 431.6 solutions on `fmin` with a standard deviation of 176.0. At the .05 level of significance we conclude that the random algorithm tests on average greater than or equal to 2070 more solutions than the GA.

Since the overhead required to perform the GA algorithm vs. random testing is insignificant compared to the time required to actually test solutions, we conclude that the GA converges on quality solutions faster than mere random probing of the solution space. That is, given the same fixed amount of run time, the GA on average would produce a better solution than random probing.

4.4 Module specific optimization sequences

Different programs have different optimization requirements. Extending that idea, we could conclude that different modules or functions of the same program may have different optimization needs. As a preliminary test of that hypothesis, we ran our GA on the separate modules of several of our benchmarks. That is, the GA was allowed to produce different optimization sequences for each module. This test was performed on `rkf45`, `adpcm`, `fft`, and `dhrystone`. `Rkf45` has three ILOC functions. Each one was optimized separately. The total code size for the three was 741 operations; a 1.5 percent reduction over the best fixed sequence and a 0.9 percent improvement over running the GA on the whole program. `adpcm` has six functions and we computed optimization sequences for each one. We saw no reduction in the static size of the code over running the GA on the whole program. However, separate optimization did lead to a 2.0 percent improvement in dynamic operation count. `fft` contains ten functions that we grouped into seven separate modules. Applying the GA to each module separately resulted in a total program size of 1743 operations; a 0.8 percent reduction from the whole program GA solution. A slight increase in dynamic operation count was observed. `Dhrystone` consists of two C modules. Optimizing them separately resulted in a total size of 533 operations or a reduction of 0.6 percent over running the GA on the entire program. No change in dynamic operation count was observed.

5 Conclusions and Observations

In this paper, we described a genetic algorithm that is designed to find compiler-optimization sequences resulting in reduced static code sizes. The algorithm was tested on a variety of benchmarks, and the solutions generated by the algorithm were compared to no optimization and fixed optimization sequences. The results of these experiments are summarized below:

- The GA found optimization sequences that dramatically reduced the code size (static operation count) compared with no optimization. The GA produced optimization sequences that resulted in smaller code than the compiler's default sequence. In most cases, the smaller code executed fewer instructions, as well.

- By observing patterns in the GA solution sequences, we were able to construct a fixed optimization sequence that generated up to 40 percent smaller codes than the standard sequence used in our compiler. In many cases, the resulting code is also faster, up to 26 percent faster. This suggests that the GA can be used to give a "tune-up" to an existing compiler by finding a better optimization sequence that uses the existing passes.

- When applied to a single program, the GA generated a program-specific optimization sequence that was often better than either the compiler's old default sequence or the new sequence described in Section 4.2. This shows that the "best" sequence differs from program to program. In particular, if a program is too large for the available space and the space cannot be enlarged, it may be worth running a GA for several hours to obtain a custom tailored optimization sequence.

We believe there are several avenues for continued research based on this work:

- The number and kind of optimizations available to the GA can be increased.
- Several of the optimizations allow options that force the optimization to use different algorithms or heuristics. These options could also be put under the control of the GA. For example, one "gene" could correspond to the `valnum` optimization as described. A separate gene could correspond to running `valnum` without performing value driven code motion.
- There are many parameters of the algorithm that can be modified. The parameterization presented was one of several tried and produced the best results of those tested. Further changes may produce better results or help the algorithm to converge more quickly.
- The preliminary work into module specific optimization showed modest improvements. More study needs to be done to see if separate optimization for different modules in the same program is a viable technique for reducing code size.
- We described how to use the GA to reduce code size. However, by simply changing the fitness function, the algorithm can be tuned to find solutions resulting in other desirable properties such as reduced running time or power consumption.

We believe that using a GA to find optimization sequences could be a valuable technique for embedded systems compilers. When compile time is plentiful, an embedded system software designer may want to consider a GA approach to final optimization, especially when code size is of critical importance. A GA can be used to find a high quality optimization sequence for particular set of optimizations.

6 Acknowledgements

This work was supported by DARPA through USAFRL grant F30602-97-2-298. The motivation for this work came from a talk by Andy Nisbet on his GAPS genetic algorithm for parallelizing FORTRAN loops. Many thanks to past and present members of the Massively Scalar Compiler Group for the development and programming of the optimizations used in this work. These include Preston Briggs, Tim Harvey, John Lu, Rob Shillner, Taylor Simpson, and Linda Torczon. We appreciate the helpful comments of the referees.

References

[1] Steven John Beaty. *Instruction Scheduling Using Genetic Algorithms.* PhD thesis, Colorado State University, Fort Collins, Colorado, 1991.

[2] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[3] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, June 1997.

[4] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[5] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[6] Keith D. Cooper and L. Taylor Simpson. Scc-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, 1995.

[7] Keith D. Cooper, L. Taylor Simpson, and Chris A Vick. Operator strength reduction. Technical Report CRPC-TR95635-S, Center for Research on Parallel Computation, Rice University, 1995.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] Lawrence Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[10] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[11] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *SIGPLAN Notices*, 28(5):29–38, May 1993.

[12] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesly, 1989.

[14] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[15] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[16] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[17] Melanie Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

[18] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[19] Andy P Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *Poster Session at The International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '98*, 1998.

[20] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, pages 660–676, November 1980.

[21] L. Taylor Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, May 1996.

[22] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.