Automatic Selection of GCC Optimization Options Using A Gene Weighted Genetic Algorithm

San-Chih Lin, Chi-Kuang Chang, Nai-Wei Lin
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
{lsch94,changck,naiwei}@cs.ccu.edu.tw

Abstract

Compilers usually provide a large number of optimization options for users to fine tune the performance of their programs. However, most users don't have the capability to select suitable optimization options. Compilers hence usually provide a number of optimization levels. Each optimization level is a pre-selected group of optimization options and produces good efficiency for most programs. However, they exploit only a portion of the available optimization options. There is still a large potential that an even better efficiency can be gained for each specific source code by exploiting the rest of the available optimization options. We propose a gene weighted genetic algorithm to search for optimization options better than optimization levels for each specific source code. We also show that this new genetic algorithm is more effective than the basic genetic algorithm for a set of benchmarks.

1. Introduction

Compilers usually provide a large number of optimization options for users to fine tune the performance of their programs. For example, the GCC version 4.1.2 provides more then one hundred optimization options [7]. This large number of optimization options gives users a good potential to generate more efficient code.

These optimization options are, however, rarely used because of two reasons. First, most users are not familiar with compiler optimizations so that they don't have the capability to select suitable optimization options. Second, the interactions between compiler optimizations are so complex that it is almost impossible to predict the efficiency of a group of optimization options even for compiler experts.

Compilers hence usually provide a number of optimization levels each of which pre-selects a group of optimization options for users. For example, the GCC version 4.1.2 provides optimization levels: O1, O2, O3, and Os. These optimization levels greatly aid users in generating code with good efficiency for most programs.

Although optimization levels provide good efficiency for most programs, they exploit only a portion of the available optimization options. There is still a large potential that an even better efficiency can be gained for each specific source code by exploiting the rest of the available optimization options. This effective exploitation of all available optimization options for each specific source code is worthy to investigate carefully.

Because of the large number of available optimization options and the complex interactions among optimizations, finding the optimal optimization options for a specific source code is a very hard and time consuming problem. Most previous works apply genetic algorithms to search for the optimal optimization options for a specific source code [3, 4, 5, 8, 9, 11]. This article proposes a new genetic algorithm, called a gene weighted genetic algorithm, to search for the optimal optimization options for a specific source code. This article also investigates the effectiveness of three optimization algorithms: the random algorithm, the basic genetic algorithm, and the gene weighted genetic algorithm.

The remainder of this article is organized as follows. Section 2 describes the framework for automatic selection of optimization options. Section 3 reviews the basic genetic algorithm. Section 4 presents the gene weighted genetic algorithm. Section 5 shows the experimental results of using the random algorithm, the basic genetic algorithm, and the gene weighted genetic algorithm. Section 6 reviews related work. Finally, Section 7 gives the conclusion of this article.

2. The option selection framework

The framework for automatic selection of optimization options is depicted in Figure 1. The option selector first selects a set of optimization options from the list of available optimization options. This first set of optimization options can be selected randomly or can be a system-provided optimization level. The source program is next compiled with this set of optimization options to generate the target code. The target code is then simulated with the performance profiled. The profiled performance is then sent back to the option selector as a feedback to select hopefully a better set of optimization options. This select-compile-profile process continues until we run out of a predetermined number of iterations. In this article, the profiled performance is the cycle counts of the program on a simulator.

The option selector is mainly an optimization algorithm that tries to search a set of optimization options for an input source program with the optimal performance. There are three issues that make this optimization problem a very hard and time consuming problem. First, the large number of optimization options makes the search space extraordinarily huge.

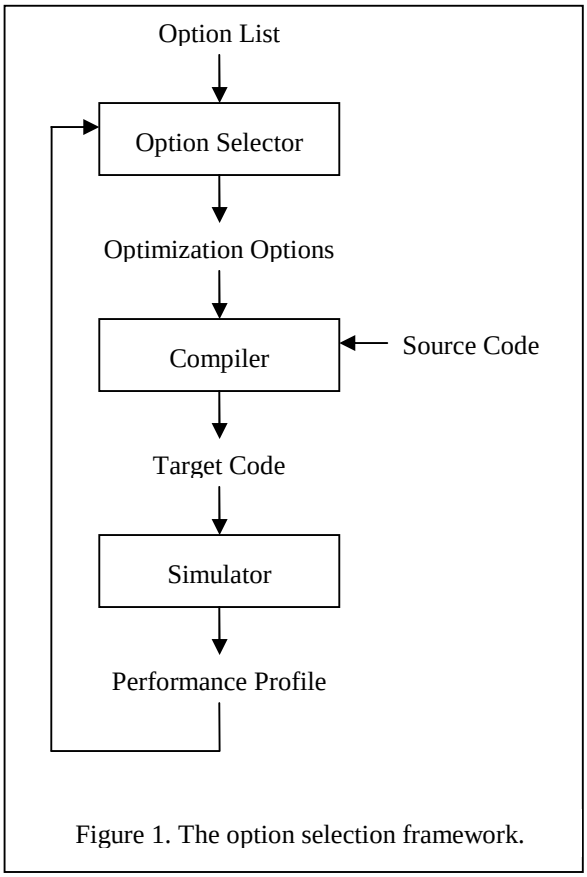

Figure 1. The option selection framework.

Second, the performance of a set of optimization options for such an input source program can only be known by compiling this input source program with this set of optimization options and then profiling the performance of the generated target code via directed or simulated execution. This makes determining the cost of a set of optimization options very time consuming.

Third, the interactions between optimization options depend on the input source code in such a complex way that it is very hard to find any heuristics to speed up the search process. Even worse is that the interactions between optimization options also depend on other selected optimization options in a very complex way. This article proposes a gene weighted genetic algorithm to address the third issue.

3. The basic genetic algorithm

Before we present the gene weighted genetic algorithm, we first review the basic genetic algorithm. Genetic algorithms are inspired by Darwin's theory about evolution [12]. The traits of an organism are basically encoded by the chromosomes of the organism. Each chromosome consists of a string of genes. Each gene encodes a particular trait, for example color of eyes. Possible values for a trait (e.g. blue, brown) are called alleles. Each gene has its own position in the chromosome. This position is called locus.

During the reproduction of an organism, parent chromosomes may recombine to generate new offspring chromosomes. The new offspring chromosomes may also mutate. This means that some genes of the offspring may be a bit different from that of both parents. The traits of the new organism should be fit to the environment of living more successfully than that of the original organism.

Genetic algorithms try to solve an optimization problem evolutionally. The parameters of the optimization problem correspond to the genes of an organism and are usually encoded as a vector corresponding to a chromosome. The values of a parameter correspond to the alleles of a gene. The objective function of the optimization problem corresponds to a fitness function that evaluates a value, called the fitness of a chromosome, representing how successfully the chromosome is fit to the environment of living. In the GCC optimization option selection problem, each gene corresponds to an optimization option in GCC. The alleles for all genes are binary values 1 and 0 representing the on and off of the corresponding optimization option. In GCC 4.1.2, there are a total of 128 optimization options. Therefore, each

chromosome corresponds to a binary string of length 128. The fitness of a chromosome corresponds to the performance of the target code compiled using the corresponding optimization options. The GCC optimization option selection problem is equivalent to finding a chromosome with the optimal fitness.

A basic genetic algorithm is given in Figure 2. The algorithm maintains n suitable solutions for the optimization problem at any time. This collection of n suitable solutions is called a population of chromosomes and represents one generation of the evolution. The algorithm aims to improve such a population evolutionally.

Initially, a population of n chromosomes is generated. This initial population is usually generated randomly. We also put the optimization levels of GCC into the initial population. In step 2, the fitness of each chromosome is evaluated via the fitness function.

Steps 3 to 8 form an evolution process. Step 3 selects two chromosomes c_1 and c_2 from the population according to their fitness. The chromosomes with better fitness have bigger chance to be selected. Step 4 performs a recombination on c_1 and c_2 to generate two new chromosomes rc_1 and rc_2. A locus in the chromosome is first randomly selected. The two front portions before the selected locus of c_1 and c_2 are then swapped to generate rc_1 and rc_2. For example, suppose c_1 and c_2 are as follows.

$a_1a_2a_3a_4a_5a_6a_7a_8$

$b_1b_2b_3b_4b_5b_6b_7b_8$.

If the selected locus is 4, then the generated new chromosomes rc_1 and rc_2 after the recombination are

$b_1b_2b_3a_4a_5a_6a_7a_8$

$a_1a_2a_3b_4b_5b_6b_7b_8$.

Step 5 performs a mutation on rc_1 and rc_2 to generate two new chromosomes mc_1 and mc_2. Each gene in the chromosome is mutated with a low probability, usually less than 1%. For example, if the gene in locus 6 of rc_1 is selected to be mutated to allele x, then the generated new chromosome mc_1 after the recombination is

$b_1b_2b_3a_4a_5xa_7a_8$.

Step 6 evaluates the fitness of mc_1 and mc_2. Step 7 replaces mc_1 and mc_2 for chromosomes in the population according to their fitness to generate a new population. The chromosomes with worse fitness have bigger chance to be selected. This step generates one new generation of the evolution.

It is assumed that after a sufficient number of generations, the population will contain some approximate optimal solutions to the optimization problem. The evolution stops after a predetermined number of generations and returns the best solution in the final population.

1. Generate the initial population.
2. Evaluate the fitness of each chromosome in the population.
3. Select two chromosomes c_1 and c_2 from the population according to their fitness.
4. Perform a recombination on c_1 and c_2 to generate two new chromosomes rc_1 and rc_2.
5. Perform a mutation on rc_1 and rc_2 to generate two new chromosomes mc_1 and mc_2.
6. Evaluate the fitness of mc_1 and mc_2.
7. Replace mc_1 and mc_2 for chromosomes in the population according to their fitness to generate a new population.
8. If the end condition isn't satisfied, go to step 3.
9. Stop and return the best fitness in the population.

Figure 2. The basic genetic algorithm.

4. The gene weighted genetic algorithm

It is very hard to find any heuristics to speed up the searching of the optimal optimization option because of the following two reasons. First, the interactions between optimization options depend on the input source code in a very complex way. Second, the interactions between optimization options also depend on other selected optimization options in a very complex way. This section presents a gene weighted genetic algorithm to address these issues.

The gene weighted genetic algorithm is based on the following two observations. First, in the basic generic algorithm, the probability for mutation is the same for all genes. In other words, the probability is independent of the input source code. However, in reality, the effectiveness of each optimization option may highly depend on the input source code. Hence, we propose to associate a weight to each gene to indicate an estimated fitness of the corresponding optimization option to the input source code. During each generation of the evolution, the gene weights are modified according to the actual fitness, namely the performance, of the new chromosomes. We expect that the maintenance of gene weights can alleviate the first issue mentioned above.

Second, in the basic generic algorithm, the mutation is performed independently on all genes. In other words, the probability is independent of the genes. However, in reality, the effectiveness of an optimization option may highly depend on the on or off

of other optimization options. Hence, we propose to fix the on of a group of positively correlated genes for a number of generations. During each generation of the evolution, the mutation of some genes may be passed. We expect that the fixing of a group of positively correlated genes for a number of generations can alleviate the second issue mentioned above.

The gene weighted genetic algorithm is given in Figure 3. We only explain the steps that are distinct from the basic genetic algorithm. In step 3, the weight of each gene is initialized to 0. The weight of a chromosome is the sum of the weights of genes in the chromosome. The weight of a chromosome is viewed as an estimated fitness. In step 11, for the four new generated chromosomes mc_1 , mc_2, gc_1, and gc_2, if the allele of a gene changes from 0 to 1 and the fitness of chromosome improves, then the weight of that gene increases by 1. If the allele of a gene changes from 1 to 0 and the fitness of chromosome degrades, then the weight of that gene also increases by 1. Otherwise, the weight of that gene decreases by 1.

1. Generate the initial population.
2. Evaluate the fitness of each chromosome in the population.
3. Initialize the weight of each gene to 0.
4. Set the initial group G of genes fixed to be on as the empty set and the candidate list C to be the empty set.
5. Select two chromosomes c_1 and c_2 from the population according to their fitness.
6. Perform a recombination on c_1 and c_2 to generate two new chromosomes rc_1 and rc_2.
7. Perform a mutation on rc_1 and rc_2 to generate two new chromosomes mc_1 and mc_2.
8. Perform a mutation on rc_1 and rc_2 to generate two new chromosomes gc_1 and gc_2. by fixing the genes in G to be on.
9. Evaluate the fitness of mc_1 , mc_2, gc_1, and gc_2.
10. Insert mc_1 , mc_2, gc_1, and gc_2 into the candidate list C if appropriate.
11. Update the weight of each gene.
12. Replace mc_1 , mc_2, gc_1, and gc_2 for chromosomes in the population according to their fitness to generate a new population.
13. If this is a regrouping iteration, update the group G of genes fixed to be on.
14. If the end condition fails, go to step 5.
15. Stop and return the best fitness in the population.

Figure 3. The gene weighted genetic algorithm.

The weights of genes are used to identify a group of genes that are positively correlated. In step 4, this group is initialized to be the empty set. In Step 10, if the weight of mc_1 or gc_1 is less than the weight of rc_1, but the fitness value of mc_1 or gc_1 is greater than the fitness value of rc_1, then mc_1 or gc_1 is inserted into a list, called a candidate list. The candidate list will be used to determine the group of genes fixed to be on. This indicates that the genes set to on are positively correlated. The same operation is performed on mc_2, gc_2, and rc_2. The candidate list is initialized to be the empty set in step 4. The group of genes fixed to be on is updated every specific number of generations, e.g. every 20 generations. This regrouping is performed in step 13 as follows. Assume that the candidate list contains n chromosomes. A gene g is put into the group of genes fixed to be on if it is on in $n - d$ chromosomes, where d is a very small integer. This implies that the gene g is highly positively correlated with other genes set to on.

5. Experimental results

We have run an experiment on a set of benchmarks using three optimization algorithms: the random algorithm, the basic genetic algorithm, and the gene weighted genetic algorithm. We select the following seven benchmarks from the MiBench 1.0 [10].

1. CRC32: detect file errors.
2. dijkstra: compute the shortest paths.
3. qsort: quick sort.
4. stringserach: search strings.
5. basicmath: basic mathematic operations.
6. blowfish: document encoding and decoding.
7. sha: safe hash functions.

The compiler is gcc 4.1.2. The gcc 4.1.2 contains 128 optimization options. The simulator is ADS 1.2 [2].

The number of iterations in the random algorithm is 100 and the number of evolution generations in both genetic algorithms is 100. The size of the population in both genetic algorithms is 20. The mutation probability in both genetic algorithms is 0.1. The group of genes fixed to be on is updated every 20 generations in gene weighted genetic algorithm. The difference d used to determine whether a chromosome should be inserted into the candidate list is 2 in gene weighted genetic algorithm.

The experimental results are shown by displaying two sets of diagrams. The diagrams are shown in Figures 4 to 17. The first set shows the trace of cycle counts for the 100 iterations or generations. This set demonstrates the evolution of performance during the optimization process. The second set shows the

maximal performance improvement rate for the 100 iterations or generations. This set demonstrates the evolution of performance improvement rate during the optimization process.

The first set of diagrams that display the trace of cycle counts shows that the two genetic algorithms significantly outperform the random algorithm and the gene weighted genetic algorithm notably outperforms the basic genetic algorithm in most benchmarks. This demonstrates that the gene weighted genetic algorithm evolves notably faster than the basic genetic algorithm in most benchmarks.

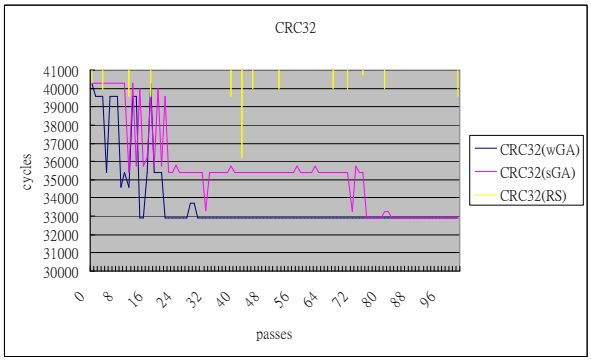

Figure 4. Trace of performance for CRC32.

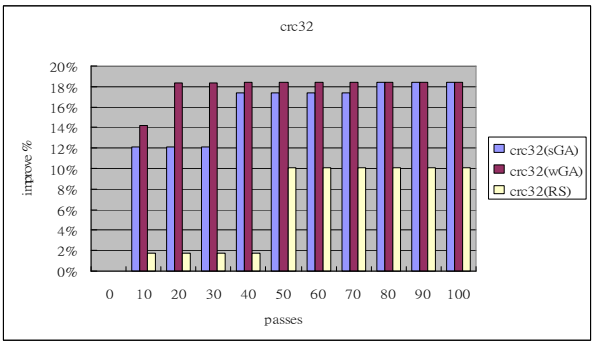

Figure 5. Improvement rate for CRC32.

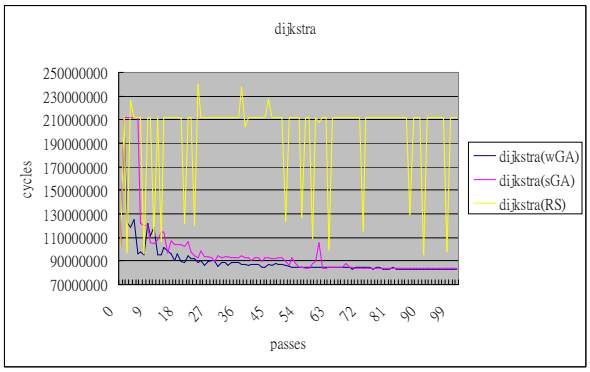

Figure 6. Trace of performance for Dijkstra.

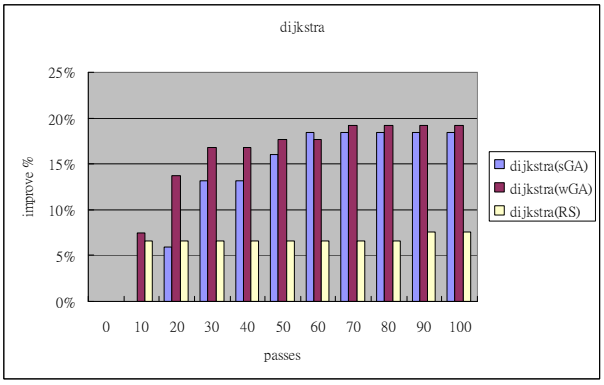

Figure 7. Improvement rate for Dijksta.

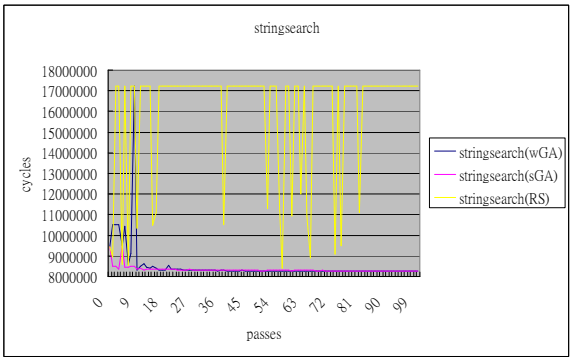

Figure 8. Trace of performance for stringsearch.

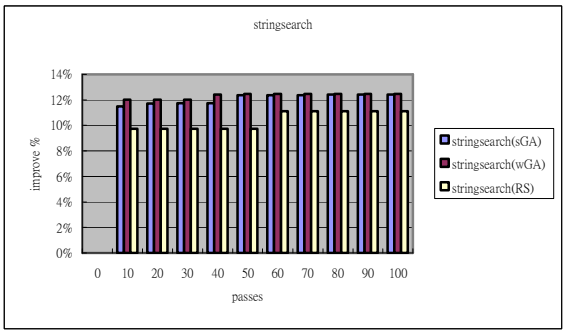

Figure 9. Improvement rate for stringsearch.

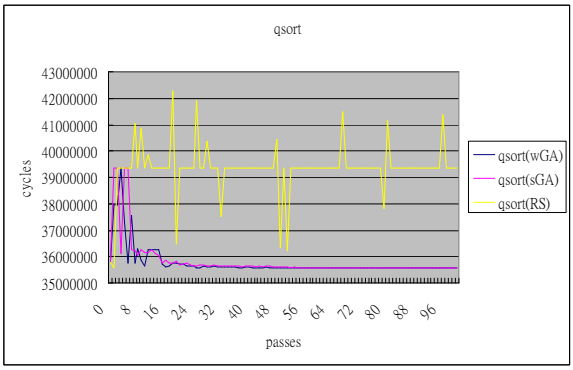

Figure 10. Trace of performance for qsort.

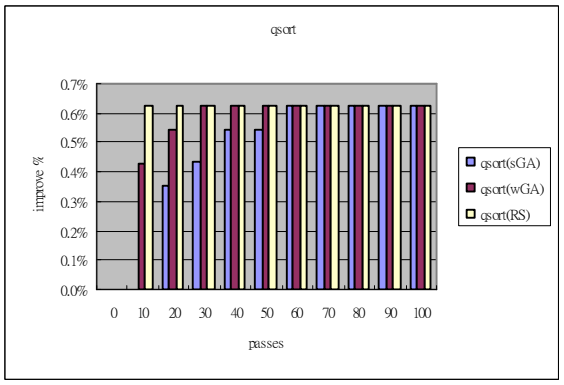

Figure 11. Improvement rate for qsort.

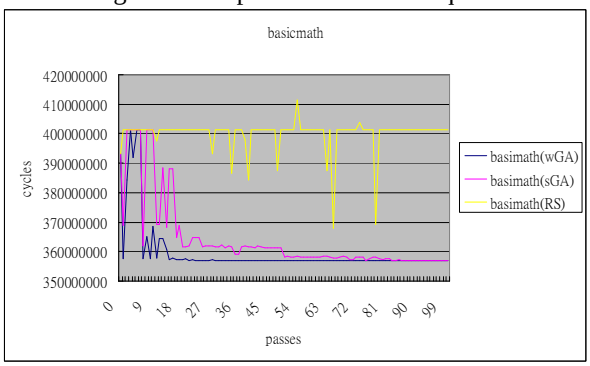

Figure 12. Trace of performance for basicmath.

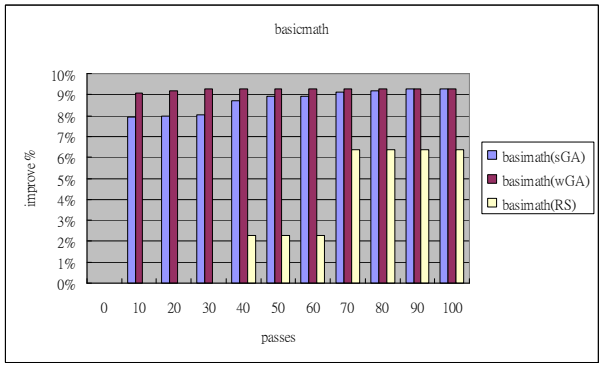

Figure 13. Improvement rate for basicmath.

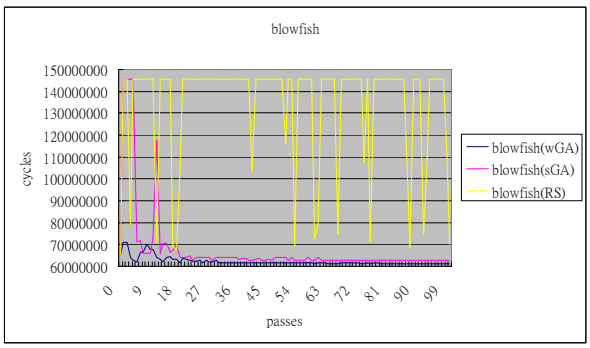

Figure 14. Trace of performance for blowfish.

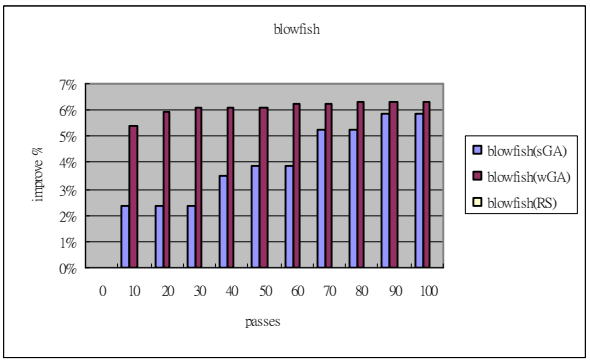

Figure 15. Improvement rate for blowfish.

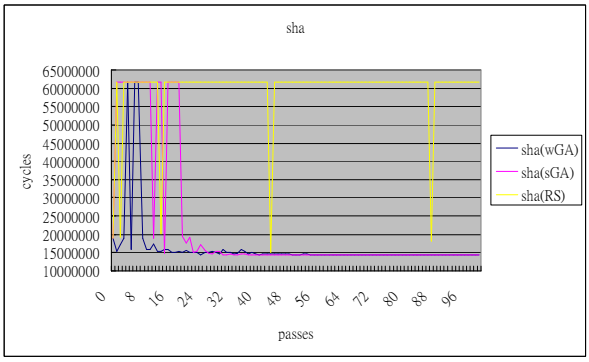

Figure 16. Trace of performance for sha.

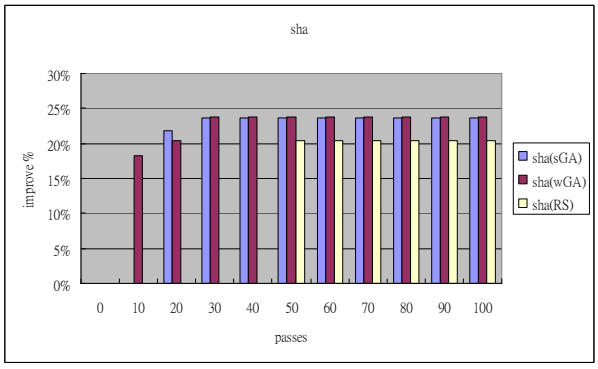

Figure 17. Improvement rate for sha.

The second set of diagrams shows that the performance improvement rates of the seven benchmarks can reach 18%, 18%, 12%, 0.6%, 9%, 6%, and 24%, respectively with respect to the best of O1, O2, and O3. Except for the qsort benchmark, the other benchmarks show notable performance improvement over system provided optimization levels. In all seven benchmarks, the gene weighted genetic algorithm reaches a very good performance improvement rate within 20 passes.

This experiment demonstrates that the automatic optimization option selection is very valuable for the GCC compiler.

6. Related work

Automatic optimization option selection can be performed statically or dynamically. Childers, Davison, and Soffa presented a static method for automatic optimization option selection. The interactions among optimizations need to be available in order to do automatic optimization option selection statically [3]. Currently, the knowledge about the interactions among optimizations is still very primitive.

There are several works on performing automatic optimization option selection dynamically [4, 5, 8, 9, 11]. Most dynamically methods are based on genetic algorithms. Except for the optimization of execution time, there is work on optimization of code size [6]. In addition to automatic optimization option selection, there is work on reordering of optimizations [1]. The search space in reordering of optimizations is much larger than automatic optimization option selection.

7. Conclusion

Compilers usually provide a large number of optimization options for users to fine tune the performance of their programs. However, most users don't have the capability to select suitable optimization options. Compilers hence usually provide a number of optimization levels. However, they exploit only a portion of the available optimization options. There is still a large potential that an even better efficiency can be gained for each specific source code by exploiting the rest of the available optimization options.

It is very hard to find any heuristics to speed up the searching of the optimal optimization option. We propose a gene weighted genetic algorithm to search for optimization options better than optimization levels for each specific source code. We propose to associate a weight to each gene to indicate an estimated fitness of the corresponding optimization option to the input source code. We propose to keep a group of positively correlated optimization options as on for a number of generations. Finally, we also show that this new genetic algorithm is more effective than the basic genetic algorithm for a set of benchmarks.

8. Acknowledgements

This work was supported in part by the National Science Council of R.O.C. under grant number NSC-95-2221-E-194-041.

9. References

[1] Almagor, L., K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, T. Waterman, "Finding effective compilation sequences," *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004, pp. 231-239.

[2] ARM Developer Suite, http://www.arm.com/products/DevTools/ADS.html.

[3] Childers, B., J. W. Davidson, M. L. Soffa, "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation," *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003, pp. 205-214.

[4] Cooper, K. D., D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century", *Journal of Supercomputing*, Vol. 23, No.1, 2002, pp. 7–22.

[5] Cooper, K. D., A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, T. Waterman, "ACME: adaptive compilation made efficient", *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 69-77.

[6] Cooper, K. D., P. J. Schielke, D. Subramanian, "Optimizing for Reduced Code Space Using Genetic Algorithms", *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999, pp. 1-9.

[7] GNU Consortium, *GCC Online Document*. http://gcc.gnu.org/onlinedocs/.

[8] Haneda, M., P. M. W. Knijnenburg, W. A. G. Wijshoff, "Optimizing General Purpose Compiler Optimization", *Proceedings of the 2nd Conference on Computing Frontiers*, 2005, pp. 180-188.

[9] Haneda, M., P. M. W. Knijnenburg, H. A. G. Wijshoff, "Generating New General Compiler Optimization Settings", *Proceedings of the 19th International Conference on Supercomputing*, 2005, pp. 161-168.

[10] Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 3-14.

[11] Triantafyllis, S., M. Vachharajani, N. Vachharajani, D. I. August, "Compiler Optimization-Space Exploration", *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003, pp. 204 – 215.

[12] Whitley, D., "A Genetic Algorithm Tutorial", *Statistics and Computing*, Vol. 4, 1994, pp. 65-85.