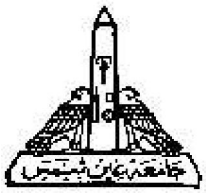

Computer Science Department
Faculty of Computer and Information Sciences
Ain Shams University

Compiler Optimization Techniques

THESIS
Submitted in partial fulfillment of the requirements for the degree
of Master of Science in Computer and Information sciences

To the department of Computer Science, Faculty of Computer
and Information Sciences, Ain Shams University.

BY
MUHAMMED HAGGAG MUHAMMED ZAYAIN
B.Sc., Faculty of Computer and Information Sciences,
Mansoura University.

SUPERVISED **B**Y

Prof. Dr. **MOSTAFA MAHMOUD AREF**
Faculty of Computer and Information Sciences,
Ain Shams University.

Asst. Prof. Dr. **TAHER TAWFEEK HAMZA**
Faculty of Computer and Information Sciences,
Mansoura University.

Dr. **SHAIMAA MUHAMMED ARAFAT**
Faculty of Computer and Information Sciences,
Ain Shams University.

Cairo-2009

CONTENTS

List of Figures

List of Tables

ABSTRACT

Compilers fail to deliver satisfactory levels of performance on modern processors, due to rapidly evolving hardware, fixed and black-box optimization heuristics, simplistic hardware models, inability to fine-tune the application of transformations, and highly dynamic behavior of the system. This analysis suggests revisiting the interactions of the compiler optimizations. Building on the empirical knowledge accumulated from previous iterative optimization prototypes; we select the best sequence in which the optimization techniques are applied. We design a tool that can access and select the compiler optimizations involved in the compilation process.

The purpose of this tool is to select the best sequence of compiler optimizations that reduce the consumption of execution resources such as processor time, memory size and power supply watts and joules that leads to enhance various efficiency metrics such as execution time, program size, power peak, energy consumption, power smoothing and energy-delay product.

The problem of defining the optimal optimization techniques for compilers consider as an optimization search problem. The problem arises from the fact that the enormous number of possible combinations of optimizations creates a search space which cannot adequately be searched. Further, different results can be achieved by switching optimization techniques on and off during the optimization process. Register allocation only complicates matters, as the interactions between different optimizations can cause more spill code to be generated. Although it has been shown that compiler settings can be found that outperform the built-in compiler selection of optimizations for a single application, it is not known how to find such settings that work well for sets of applications.

In this thesis, we solve our optimization search problem with two methods namely a genetic algorithm and a statistical method using the Mann-Whitney test. Both methods focus on the interaction among the compiler optimizations through applying iterative compilations and they are experienced for defining the "best" optimization techniques. "Best", in this context, is defined as those compiler optimization techniques that produce the optimal result of specific efficiency metric such as performance and energy consumption. The solutions generated by both methods are compared to solutions found using the best fixed optimization method used in the GNU Compiler Collection (GCC). We apply both methods against collection of computer benchmarks maintained by Standard Performance Evaluation Corporation (SPEC), the results are found compiler settings that perform better than the standard –Ox settings of GCC, based on the results, the genetic algorithm is better than the statistical method.

Published Work

- Muhammed H. Zayain, Taher T. Hamza, Zaki T. Fayed, Shiamaa M. Arafat, **Selection of Compiler Optimizations Sequence using Genetic Algorithm**, The International Journal of Intelligent Computing and Information Sciences (IJICIS), Cairo, Egypt, 2008.

- Taher T. Hamza, Zaki T. Fayed, Shiamaa M. Arafat, Muhammed H. Zayain, **Compiler Optimizations Tuning Methodologies**, Proceedings of Al-Azhar Engineering International Conference (AEIC), Cairo, Egypt, 2008.

ACKNOWLEDGMENTS

First and foremost, I thank ALLAH for giving me the incentive to go on and finish this work and giving me glimpses of heavenly hope in the darkest times.

I would like to express sincere thanks to my supervisor, *Dr. Mostafa M. Aref* for his time, knowledge, friendly encouragement and invaluable advices. I also thank *Dr. Taher T. Hamza* for his great support and invaluable guidance. I would also like to thank *Dr. Shaimaa M. Arafat* who has provided useful ideas for the duration of this work.

Finally, and most especially, I would like to dedicate my thesis to my parents, for their unlimited love, support and encouragement.

Muhammed Haggag Zayain

CHAPTER 1

INTRODUCTION

1.1 Motivation

A new generation of advanced embedded products has emerged. These products, ranging from laser printers to network routers to video games, increasingly call for enormous levels of processing power - levels which can only be achieved through the use of high-speed RISC and CISC microprocessors.
While developers can now choose from a wide variety of low-cost, high performance processors, market demands for better, faster products have forced developers to seek increased performance in a variety of ways. However, since all developers have similar access to the latest high-speed hardware components, competitive advantage is difficult to achieve through hardware alone. As a result, competitive product advantage is increasingly being sought and achieved through superior software.

The shift in emphasis to superior software performance has raised the importance of software development tools dramatically. This is especially true for the compiler - the one software development tool with the greatest impact on a product's ultimate performance. With the widespread use of high-level languages such as C and C++ for embedded software development, compiler optimization technology plays a more critical role than ever in helping developers to achieve their overall design goals.

Innovations and improvements have long been made in computer and system architectures to essentially increase the computing power truly observing the Moore's Law for more than three decades. Improvements in semiconductor technology make it possible to incorporate millions of transistors on a very small die and to clock them at very high speeds. Architecture and system software technology also offer tremendous performance improvements by exploiting parallelism in a variety of forms. While the demand for even more powerful computers would be hindered by the physics of computational systems such as the limits on voltage and switching speed [1], a more critical and imminent obstacle is the power consumption and the corresponding thermal and reliability concerns [2]. This applies not only to low-end portable systems but also to high-end system designs.
Since portable systems such as laptop computers and cell phones draw power from batteries, reducing power consumption to extend their operating times is one of the most critical product specifications. This is also a challenge for high-end system designers because high power consumption raises temperature, which deteriorates performance and reliability.

Why care about the compiler?

C and C++ compilers have come a long way from the simple code translators of the past. Although the emergence of standards such as ANSI C have led some developers to treat compilers as commodity products, two forces have combined to create notable differences in optimization technology from one compiler to the next. One stems from the architectural nature of today's high-speed processors. With complex instruction pipelines and on-board

structure instruction and data caches, today's advanced processors are highly dependent on the compiler to structure object code for optimal performance. Creating this optimal structure is difficult and highly CPU-specific, causing large differences in program performance depending on which optimization techniques are employed. The second force creating these compiler differences is market pressure. With developers demanding ever-higher performance and/or denser code, certain compiler vendors have committed themselves to devising increasingly sophisticated optimization techniques.

Compiler suites that employ the latest optimization techniques offer many benefits to embedded system developers. With challenging real-time performance goals, cost constraints, and the pressures to deliver complex products in less time, developers increasingly rely on a compiler's intimate knowledge of a processor's instruction set and behavior to produce optimal code. The benefits help developers to achieve fundamental project goals, such as:

- **Higher Performance :**
  Compilers employing the latest optimization technology routinely produce code 20-30% faster than standard compilers, and in some cases, two to three times faster. This kind of performance boost is critical for maximizing system performance. In addition, since a large majority (often more than 90%) of program execution time is spent in application code rather than in the operating system, compiler performance is much more significant than Real-Time Operating System (RTOS) performance for overall application speed.

- **Lower Cost :**
  Superior optimization enables developers to reduce system cost and improve products in a number of ways. First, a higher performance product could be produced without having to resort to a higher-speed and higher cost processor. A slower processor also permits the use of lower-speed, less costly memory. Compiler optimization techniques also have a very significant effect on code size. With decreased memory requirements, developers can further reduce production costs by using less memory, or they may opt to add additional features to make their product more competitive.

- **Reduced Development Time :**
  Embedded software applications have become much more complex. As a result, code reuse is often essential to delivering new products on time. The emphasis on code reuse has led to a major shift from assembly code to high-level languages such as C and C++.

However, effective code reuse and maintenance requires more than just programming in a high-level language. It is important to use techniques such as structured and object-oriented programming tools and techniques to ensure modular and readable code. In the past, less sophisticated compiler optimization technologies forced developers to avoid such programming practices and hand-tune the source code for the target architecture. Today, a highly optimizing compiler enables developers to write the most readable and maintainable source code with the confidence that the compiler can generate the optimal binary implementation.

1.2 Research Goal

Compiler optimizations or transformations attempt to produce the best machine code from source code. For example, constant propagation optimization 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.

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. 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 thesis is discovering optimization techniques sequences that result in best values of efficiency metrics such as energy consumption, code size and peak value. Since the software define the process of resources dissipation like processor, read-only memory and power supply, so these issues are put into consideration while building the software via metric-aware compiling.

We implement two techniques to solve our problem. The first is a genetic algorithm (GA) is 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. The idea is generally attributed to Holland [3]. The second is a statistical compiler tuning methodology, described earlier in [4] is adapted to our experiments. The idea is to detect the significant optimization compiler options.

The experiments show that GA is well suited to the problem of finding good optimization techniques sequences. For each program in the set of benchmarks, the GA discovered a sequence that produces the best result for each metric than the fixed - default - sequence used in the GCC compiler. Also it proves that the GA gives solutions better than statistical ones. Experiments are tested by optimizing several SPEC Int95 benchmarks for integer processing.

1.3 Thesis Outline

This thesis is organized into six chapters, including this chapter (Chapter 1), and one appendix.

- **Chapter 2: COMPILER OPTIMIZATION**

  This chapter gives details about different types of compiler optimization techniques. The first part of the chapter focuses on code optimization, while the other one focuses on the relationship between performance and energy optimizations.

- **Chapter 3: EVALUATION FRAMEWORK**

  This chapter describes the evaluation framework for assessing program efficiency metrics. The conclusions and results presented in this thesis have been obtained with the benchmarks, the simulators and other tools that are presented in this chapter.

- **Chapter 4: GENETIC ALGORITHM**

  This chapter gives an overview about the general idea of the genetic algorithm and illustrates how to adapt the general GA to be used in the optimization problem. Finally it shows the results from our GA implementation for all efficiency metrics

- **Chapter 5: MANN-WHITNEY TEST**

  This chapter gives an overview about the non-parametric statistic and illustrates how to use it in the problem of finding the best compiler optimization sequence. The following sections cover the following: Mann-Whitney test is illustrated by a generic example, apply the method to compiler optimization sequence problem and finally results obtain by Mann-Whitney test.

- **Chapter 6: CONCLUSION & FUTURE WORK**

  This chapter presents the conclusion of our proposed framework, comparison between Genetic Algorithm, Mann-Whitney test and GCC default optimization level '-O2' results. Finally list some points as a future work.

- **Appendix A: Installation Issues**

  This appendix contains the installation instructions for our framework and the other tools that are shared in it.

CHAPTER 2

COMPILER OPTIMIZATION

This chapter gives details about different types of compiler optimization techniques. The first part of the chapter focuses on code optimization, while the other one focuses on the relationship between performance and energy optimizations.

2.1 Code Optimization

Optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. For instance, a computer program may be optimized so that it executes more rapidly, or is capable of operating with less memory storage or other resources, or draw less power. In practice it is impossible to achieve optimal performance but we can design computer programs so that they become more (or less) efficient and use programming constructs that can be efficiently executed by the processor.
Performance should be a concern in all stages of the development from the choice of solution method to the executable program and it's easiest to improve the performance of a program in the early stages of design (at the highest level of abstraction).

2.1.1 Why we need to optimize the code?

Optimizing the code aims lowering the amount of the consuming hardware resources such as processor, memory and power supply units.

- Execution time
  - Not important if the program execution time is very short.
  - Important in High Performance Computing where execution times may be very long (days, weeks or months).
- Memory usage
  - Part of normal algorithm design.
  - Corresponds directly to the variables allocated in the program.
- Disk space
  - Very important for embedded systems which the code burned into smaller ROM.
- Power consumption
  - Very important in mobile systems and embedded systems.

2.1.2 What to optimize (find the stuff)

- Find out where the program spends its time
  It will be unnecessary effort to optimize code that is seldom executed.

- The 90/10 rule
  A program spends 90% of its time in 10% of the code. So the focus is directed for optimizations in this 10% of the code.

Several tools are used to find out where a program spends its time. These tools such as

- The time command – user and system time.
- Measuring with timer functions in the code.
- Profilers – gprof.

2.1.3 Speed factors of the program execution

Four components determine the speed of a program execution:

- The architecture (and clock speed) of the processor.
- The compiler used to produce the executable program.
- The source code.
- The algorithm.

The processor and the compiler are often fixed so they can be both replaced by more efficient ones. The programmer chooses the algorithm, the data structures and implements these in the source code.

2.2 Compiler Optimization

The compiler translates programs written in a high-level language to assembly language code. Assembly language code is translated to object code by an assembler. Object code modules are linked together and relocated by a linker, producing the executable program. Code optimization can be done in all these stages.

2.2.1 Optimization categories

Optimizations fall into two general categories:

- Machine-independent optimizations.
  are usually implemented in the intermediate code phase.

- Machine-dependent optimizations.
  include register allocation and utilization of machine idioms.

2.2.2 Optimization places

Places for the potential optimizations. *Figure 2.1*, illustrates their places visually.

- Algorithmic level optimizations – Design phase
  e.g., the user can choose a sorting algorithm. Quick sort algorithm is faster than bubble sort algorithm.

- Source code optimizations
  e.g., the user transforms loops so that they run efficiently. Loop unrolling is an example of such loop transformation.

- Intermediate code optimizations
  in this level the compiler can improve loops, address references etc.

- Target machine level optimizations
  in this level the compiler uses the machine specific information to make good use of the machine resources.

The first three places are related to the machine-independent optimizations.

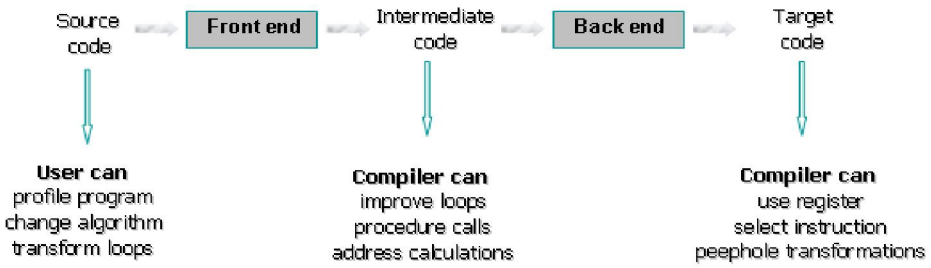

Figure 2.1: Places for potential optimizations by the user and the compiler.

2.2.3 Optimization scope

Transformations can be performed at:

- **Local level** (within the same basic block).
  Basic block is represented with DAG (Directed Acyclic Graph).

- **Global level** (among the basic blocks).
  Global data-flow analysis used to collect information of the program, e.g., the lifetime of the used variables.

2.2.4 Optimization criteria

The criteria of code-improving transformations:

- Preservation of the correctness of the program
  - may not do transformations that can alter the behavior of the program.
  - only applies transformations that are known to always produce similar results as the original code.
- Satisfactory performance
  - improving the execution speed, shrinking code size, power dissipation reduction.
  - depends strongly on the architecture of the processor.
- The transformation should be worth the effort
  - A non-optimizer compiler is more helpful during debugging.

2.2.5 Intermediate level optimization techniques

Different classes of compiler optimization techniques can be categorized upon their uses.

- Optimizations that improve assembly language code
  - reduces the number of instructions and memory references.
  - uses more efficient instructions or assembly language constructs.
  - instruction scheduling to improve pipeline utilization.

- Optimizations that improve memory access
  - reduces cache misses.
  - Pre-fetching of data.
- Loop optimizations
  - builds larger basic blocks.
  - removes branch instructions.
- Function call optimization.

2.2.5.1 Copy propagation

- Propagates values of constants or variables into the expressions where they are used.
- Example:
  - the second statement depends on the first.
  - copy propagation eliminates the dependency.
  - if x is not used in the subsequent computation, the assignment x = y can be removed (by dead code elimination).
- Applying this optimization aims to reduce register pressure and eliminates redundant register-to-register move instructions.

Before	*After*
x = y; z = c+x;	x = y; z = c+y;

2.2.5.2 Constant folding

- Expressions consisting of multiple constants are reduced to one constant value at compile time.
- Example:
  - two constants Pi and d.
  - *tmp* = Pi/d is evaluated at compile time.
  - the compiler uses the value *tmp* in all subsequent expressions containing Pi/d.
- Explicitly declaring constant values as constants help the compiler to analyze the code also improves code readability and structure.
- Applying this type of optimization reduces the number of instructions.

Before	*After*

const double Pi = 3.15149;
...
d = 180.0;
...
t = Pi*v/d;
	...
	t = v*tmp;

2.2.5.3 Dead code removal

- Removes code that has no effect on the computation
  - often produced as a result of other compiler optimizations.
  - may also be introduced by the programmer.
- Two types of dead code
  - instructions that are unreachable.
  - instructions that produce results that are never used.

  result in a non-global variable that is not live immediately after its definition.