Statistical compiler tuning
Haneda, M.

Citation
Haneda, M. (2006, September 26). *Statistical compiler tuning*. *ASCI dissertation series*. Retrieved from https://hdl.handle.net/1887/4572

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/4572

Note: To cite this publication please use the final published version (if applicable).

Statistical Compiler Tuning

Statistical Compiler Tuning

proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 26 september 2006
te klokke 16.15 uur

door

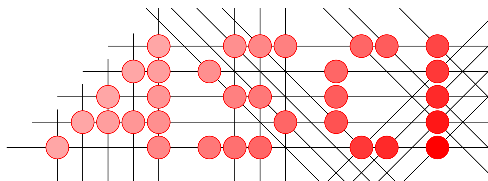
Masayo Haneda

geboren te Hyogo, Japan
in 1978

Promotiecommissie

Promotor:	Prof. Dr. H.A.G. Wijshoff
Co-promotor:	Dr. P.M.W. Knijnenburg
Referent:	Dr. M.F.P. O'Boyle (The University of Edinburgh, UK)
Overige leden:	Prof. Dr. Ir. H.J. Sips (Technische Universiteit Delft)
	Prof. Dr. H. Corporaal (Technische Universiteit Eindhoven)
	Prof. Dr. F.J. Peters
	Prof. Dr. S.M. Verduyn Lunel
	Prof. Dr. Ir. E.F. Deprettere

ISBN-10: **90-9020847-X**
ISBN-13: **978-90-9020847-3**



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 129.

Contents

1 Introduction **1**
1.1 The implementation of compiler optimizations . . . 1
1.2 Compiler optimizations for embedded systems . . . 5
1.3 Iterative compilation . . . 7
1.4 Our approach . . . 10
1.5 Related work on generating optimal code . . . 11
1.5.1 Generating optimal code for general purpose processors . . . 11
1.5.2 Generating optimal code for embedded systems . . . 12
1.6 Overview of this thesis . . . 13
1.7 Overview of publications . . . 14

2 Preliminaries **15**
2.1 Orthogonal arrays . . . 15
2.2 Main effect . . . 17
2.3 Inferential statistics . . . 18
2.3.1 The Mann-Whitney test . . . 19
2.3.2 Determining the sample size . . . 22

3 Using main effect to optimize the execution time of a single application **23**
3.1 The iterative search algorithm . . . 23
3.2 The experimental environment . . . 24
3.3 A case study . . . 27
3.4 Results . . . 28
3.5 Sensitivity of the algorithm to the threshold value . . . 32
3.6 Conclusion . . . 33

4 Using the Mann-Whitney test to optimize the execution time of a single application **47**
4.1 Our methodology . . . 47
4.2 The experimental environment . . . 48
4.3 Results . . . 49
4.3.1 Example: the iterations for mcf . . . 54
4.3.2 Settings for SPEC2000 . . . 54
4.3.3 Number of iterations . . . 61

4.4 The robustness of the methodology . . . 62
4.5 Conclusion . . . 66

5 Using the Mann-Whitney test to optimize the code size of a single application **67**
5.1 The experimental environment . . . 68
5.2 Results . . . 69
5.2.1 The optimization time requirements . . . 69
5.2.2 Code size reduction . . . 72
5.2.3 The compiler settings . . . 73
5.3 Conclusion . . . 73

6 The determination of compiler settings for multiple applications taking into account interaction between optimizations **81**
6.1 A methodology to define a compiler setting . . . 82
6.1.1 The detection of interaction between optimizations . . . 83
6.1.2 Defining interaction using a representative subset of the search space 85
6.2 The algorithm to find a compiler setting . . . 86
6.2.1 Step 1: Finding maximal subsets of positively interacting options . . 86
6.2.2 Step 2: Combining subsets . . . 87
6.2.3 Step 3: Selecting the best setting . . . 89
6.3 Results . . . 90
6.4 Observations . . . 92
6.5 Conclusion . . . 94

7 Using random search to determine a compiler setting **95**
7.1 The experimental environment . . . 95
7.2 The random generation of compiler settings for a single program . . . 96
7.3 Multiple program optimization . . . 101
7.3.1 The effectiveness of the new setting . . . 104
7.4 Conclusion . . . 104

8 Conclusion **107**

A Compiler Options **109**

B Benchmark Suites **111**

Samenvatting **127**

Acknowledgement **129**

Curriculum Vitae **131**

Chapter 1

Introduction

As Moore's law predicts, the number of transistors on a chip doubles about every two years [33]. This causes computer architectures to become more and more complex, so that it becomes a difficult task to optimize application codes for these architectures. Although researchers have claimed that compilers must be responsible to transform an application into the most suitable code for a target architecture automatically and many optimizations have been invented for that purpose [39], hand optimization is still necessary to achieve the highest performance. However, hand optimization cannot be the ultimate solution for this optimization problem since the size of applications grows rapidly and it is clear that programmers will not be able to oversee the complex task of optimizing their codes [21].

The problem of using compilers to optimize applications is two-fold. First, the effectiveness of compiler optimizations changes depending both on the target architectures, the target application, and the characteristics of the input data. Second, it is impossible to try out all possible compiler settings to find the best one since the number of available compiler optimizations causes the number of possible combinations of compiler settings to be sheer overwhelming. In this thesis, we describe methodologies to cope with the complex task of finding optimal compiler settings.

1.1 The implementation of compiler optimizations

A compiler can be generally defined as the software which is responsible for translating a high-level programming language into a machine language (object code) [5]. However, modern compilers do more than just translation. The translated code may contain redundancies and inefficiencies, which can be removed without changing the original semantics. A compiler, which is capable to perform these optimizations, is called an optimizing compiler. Figure 1.1 shows the general structure of such an optimizing compiler.

In Figure 1.1, an input program is translated into a syntax tree, which is a tree representation of the program structure where the internal nodes stand for operators and the leaf nodes represent their operands [5]. This syntax tree is translated into an intermediate representation, which enables code generation. Compiler optimizations are applied to the intermediate

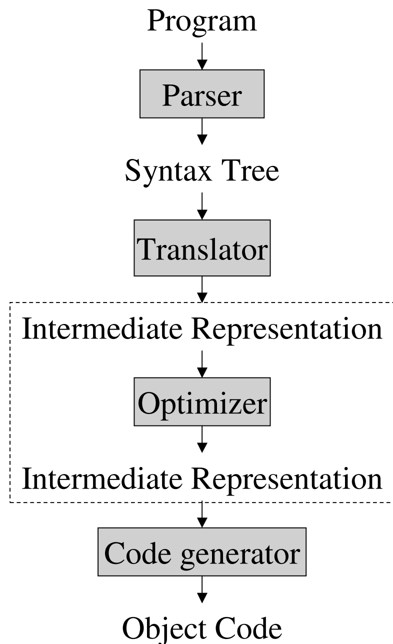


Figure 1.1: General Structure of Compiler

representation and transform the original program structure into another structure to improve the performance of the object code. It is very important to guarantee that the transformation preserves the original semantics of the program. Several analysis techniques are needed to understand the program and to examine the legality of these transformations [5][39][50].

Control-flow analysis is applied to understand the structure of a program [5]. A basic block is defined as a sequence of instructions which can be entered only at the beginning of the block and exited only at the end of the block. Basic blocks in a program are identified during control-flow analysis. Basic blocks enable us to construct a control-flow graph which reflects the dominance relation between basic blocks [5]. Assume that there are two basic blocks *A* and *B*. We say a basic block *A* dominates a basic block *B*, if every path to *B* from the entry of the control-flow graph includes *A*. According to this relation, it is possible to identify loop structures in the program [5].

Data-flow analysis provides information about how data is processed in a program. This analysis is important since the results can determine the legality of compiler optimizations. The analysis is based on the relationship between definition and use of variables. The results of the analysis can be expressed in various ways: def-use chains, use-def chains [5], and webs [38]. They represent definitions and uses for each variable. Static Single-Assignment (SSA) [17] is one of the intermediate representations which naturally expresses def-use chains.

As a result of data-flow analysis, it is possible to perform data dependence analysis. This enables us to perform instruction scheduling and data-cache optimization, namely, by apply-

```
do i=1,n
    do j=1,n
        b = b+a[i,j]
    enddo
enddo
```
(a) Original Code

```
do j=1,n
    do i=1,n
        b = b + a[i,j]
    enddo
enddo
```
(b) Transformed Code

Figure 1.2: Example of Loop Interchange

ing loop transformations. The best known and earliest data dependence test is the greatest common divisor (GCD) test [6]. Further, there exist the Power test [63] which is a combination of Fourie-Motzkin elimination and the GCD test, and the Omega test [54] which is a test for the existence of an integer solution to a set of equalities and inequalities which expresses the access pattern to the (array) variables in a loop. Most of the instruction scheduling and data-cache optimizations assume that the code to be transformed satisfies certain dependence patterns, which can be the absence of dependences, or the direction of dependences, or the distance of dependences [39]. Therefore, it is extremely important to determine the existence and characteristics of dependences. On the other hand, a strong dependence analysis, like the omega test, requires expensive computations.

An intermediate representation is defined by compiler developers, and its design depends on the way the entire compiler has been designed and is implemented. Many modern compilers maintain several intermediate representations, which are different in their level of abstraction of the program representation. For example, the DEC Alpha compiler has two kinds of intermediate representation, CIL (Compact Intermediate Language) and EIL (Expanded Intermediate Language). The Open Research Compiler (ORC) [3] implements 5 levels of intermediate representation. *gcc 3.3.1* implements a *tree* representation to be translated to RTL (register transfer language). *gcc 4.1* has RTL and higher level intermediate representations which are called GENERIC and GIMPLE [48] to implement SSA.

The intermediate representation on a high level of abstraction, which preserves procedures and loop structures, is most suitable for applying data-cache optimizations, i.e., loop transformations [50]. An example of a loop transformation is loop interchange. Loop interchange reverses the order of two adjacent loops in a loop nest. In Figure 1.2, one can see a nested loop that can be interchanged. This optimization increases the locality of data access. In Figure 1.2, the access to the array $a[i, j]$ becomes contiguous after the application of loop interchange.

Although loop transformations are suitable to be implemented on high level intermediate representations, it is also possible to implement them on a low level intermediate representation. The effect of a compiler optimization at different levels of intermediate representation may be different. In [50], major compiler optimizations are listed in their appropriate order of application based on the experience of the author. An appropriate abstraction level of

	(a) Original RISC Code		(b) Transformed RISC Code
L:	LOADF R1, A[t1]	L:	LOADF R1, A[t1]
	ADDF R2, R1, R2		ADDF R2, R1, R2
	...		...
	ADDF R4,R5,R4		CMP R2, R3
	CMP R2, R3		BLE L
	BLE L		ADDF R4,R5,R4

Figure 1.3: RISC Code for Delayed Branch

intermediate representation is also specified there.

Besides loop transformations, there are many target specific optimizations. They are implemented on the low level intermediate representation which is close to the representation of the target code since it uses more architecture specific operations. For example, strength reduction which replaces expensive instructions by equivalent cheaper instructions is usually implemented on the low level intermediate representation [5][50].

After optimization, the object code is generated from the intermediate representation by translation. This translation needs architecture-specific information, namely, the instruction set, the number of registers, etc. Efficient code can be generated when the final intermediate representation is well-suited for the target architecture. For example, pipelines which are implemented in most modern architectures, can work effectively when the executable code is scheduled to maximize the use of their computation utilities. For effective use of pipelines, some architectures implement delayed branching which execute the next instruction of a branch instruction. A scheduling strategy must consider such architectural features to generate effective code. Figure 1.3 shows an example of the RISC code for delayed branching, in which one can see that a ADDF instruction before the CMP instruction is moved after the BLE instruction. Software pipelining [42] is also an important technique for scheduling. This technique changes the order of instructions to hide memory latencies.

Multiple target compilers, for example *gcc*, have multiple code generators, one for each architecture. As might be expected, multiple target compilers cannot produce code as efficient as compilers which target a specific architecture. This is because optimizations are implemented without taking into account the implementation of the architecture. Additionally, most users only use standard optimization settings which are provided by the compilers. The most suitable form of intermediate representation can be different depending on the target architecture so that the appropriate optimizations also can be different.

Choosing an appropriate compiler optimization setting may improve the performance of the code produced by a multiple target compiler. However, compiler setting tuning is a difficult task since the search space is large, namely, the search space contains 2^N search points when the compiler has N independent optimizations with two states, on or off.

1.2 Compiler optimizations for embedded systems

In [21], embedded computing is defined as all computing which is not general purpose. General purpose processors, like the Pentium from Intel, can handle any kind of application "efficiently". However, there also exists a demand for processors which can achieve sufficient efficiency on special purpose applications utilizing only a limited number of resources. For example, it is of no use to have general purpose processors operating on gigahertz frequencies for mobile phones. For mobile phones, power consumption, cost, and memory size are much more important than high speed computation. Embedded systems target these requirements.

All embedded systems are unique since they are customized to achieve the best performance for a small number of applications. However, it is possible to categorize them into microprocessors, microcontrollers, and digital signal processors (DSPs) [21]. Microprocessors and microcontrollers are similar to general purpose processors but their computation resources are limited. Microprocessors are often employed as the core of embedded systems to control the processes on them. Microcontrollers are deployed to accomplish standalone operations in (industrial) electronics.

Digital signal processors are designed to perform well for signal processing. Matrix operations are heavily used in signal processing algorithms. So DSPs need special support for these matrix operations: multiply-accumulate (MAC) operations. To achieve a MAC operation in one clock cycle, DSPs use deep pipelines which make wrongly predicted branches costly. Also, DSPs sometimes have XY-memories, which are two separate local data memories. These DSPs have irregular instruction sets: some instructions must take one or both of their operands from the X or Y local memory. It is obvious that writing software for DSPs is difficult because of these peculiar features.

Traditionally, software development for embedded systems used mainly assembly language. This tradition has changed since high level language compilers have started to support embedded systems. Usually, these compilers are cross compilers, which enable one to develop applications in their familiar environment (e.g., a desktop). Next, the executable is loaded into a specific embedded system platform. A cross compiler with an assembler, a linker, a simulator, and a debugger is called the cross-development toolchain. For example, *gcc* provides toolchains for various embedded systems [13].

Standard C [4] is widely used as the high level language to write software for embedded systems [21]. Also there exist high level languages which are specialized for embedded systems. A DSP has many specific architectural features and it is impossible to make good use of them when programmers write their software in standard C. This causes programmers to use mostly assembly language for writing time critical sections of the code. Embedded C++ [1] and embedded JAVA [2] are also high level languages for embedded systems. Embedded C++ is a subset of C++. For example, embedded C++ does not allow using templates, exception handling, namespaces, etc. Embedded JAVA can target any architecture if it supports a JAVA virtual machine. Matlab [47] also provides an environment for the development of software for embedded systems.

Because embedded systems are mostly designed specifically for one task, the design criteria do not include backward compatibility and upgrade compatibility. The software has

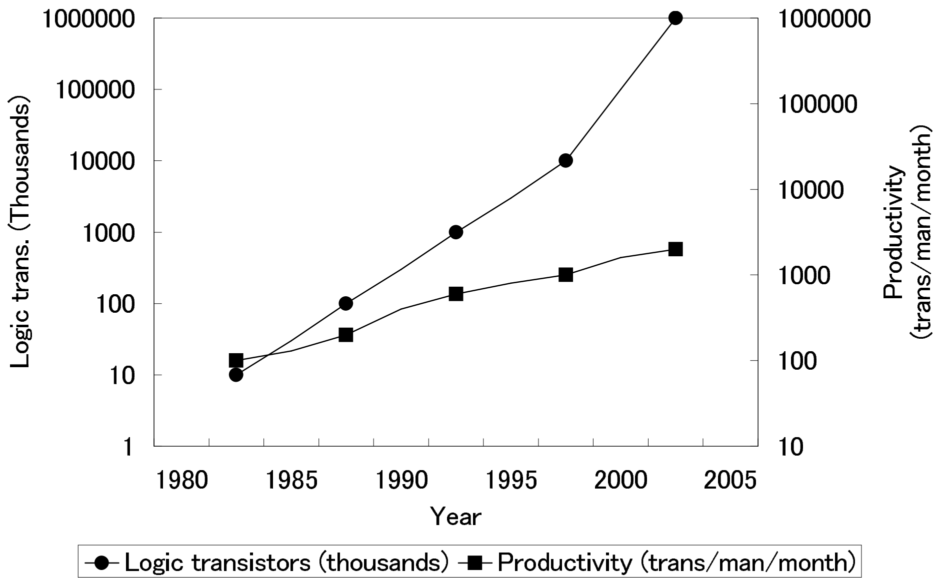


Figure 1.4: Growth in the Number of Logic Transistors

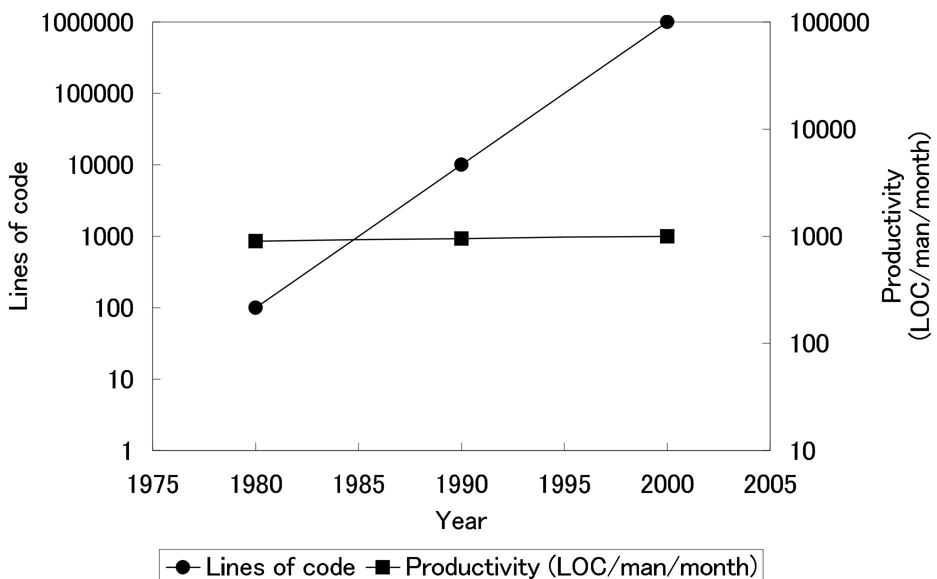


Figure 1.5: Growth in the Number of Lines of Software

to be strictly tuned to exploit instruction parallelism, to meet real-time constraints, to save power consumption, etc. This has been done by expert programmers who directly write their applications in assembly language. The trend of using high level languages to program the software for embedded systems causes this process to be done by compilers.

As already mentioned, software for embedded systems should satisfy multiple constraints. These constraints make the optimization strategy for compilers rather more complicated than a strategy for general purpose processors. For example, loop unrolling, which is a very common optimization for general purpose processors to exploit instruction level parallelism, cannot be applied bluntly for embedded processors because code size is increased, and this is not desirable for embedded systems which have a limited and expensive memory. Programmers need to decide whether the advantage of speedup is worth having despite of the disadvantage of code size increase. Such a tradeoff is difficult for compilers to understand since they have no dynamic information, like execution time or power consumption, of the resulting code. Therefore, programmers usually spend much time to configure compiler settings or to optimize the assembly code manually.

In most cases, the resulting codes from compilers are not as good as the ones which are generated by specialists manually. It is difficult to exceed the performance of code written by specialists, although it may be possible to generate codes which achieve reasonably acceptable performance by a compiler if we can tune the compiler correctly. Software for embedded systems used to be very small so that it was feasible that programmers write the corresponding assembler code manually. This situation has changed recently. Figures 1.4 and 1.5[1] show the results of a survey on the growth of the productivity of designers of hardware and software. From these graphs, one can see that analogous to the growth of number of logic transistors, the number of lines of code also grows (100 lines in 1980 and 1 million lines in 2000). Also, one can see that the productivity of a programmer has not changed for decades. Software for embedded systems also grows in size as embedded systems consist of more transistors. It is clear that generating and optimizing the code manually becomes more difficult since the productivity of a programmer does not change. Therefore, software for embedded systems should be written in a high level language to increase the reusability, and compilers should contribute to generate a highly optimized code.

1.3 Iterative compilation

Iterative compilation [8][40] has been proposed to maximally exploit the ability of a compiler for each architecture. Compilers usually apply a fixed order of compiler optimizations with the same heuristics. However, modern compilers implement many optimizations that can be configured by users. In general, this property is not being used effectively. The reason is that configuring compiler optimizations requires in depth knowledge of the implementation of the compiler and of the characteristics of target architectures and target applications. Most users do not have such knowledge so that the demand for tools that automatically tune compiler optimization has increased. The main characteristic of iterative compilation is the use

[1] These graphs are taken from [21]

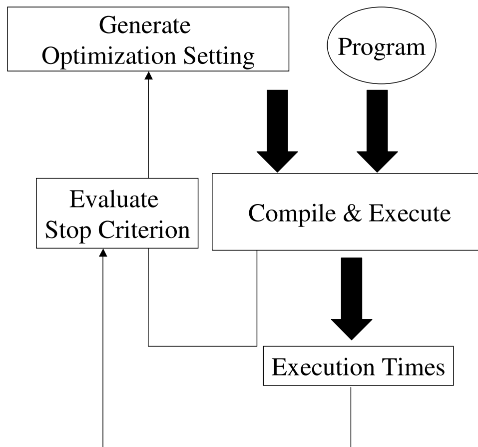


Figure 1.6: Iterative Compilation System

of execution times to tune compiler optimization. In other approaches, a model of the architecture is used to estimate the best compiler optimizations. The drawback of this approach is that architectures are too complex to be modeled completely so that the resulting settings are inaccurate.

A disadvantage of iterative compilation is that the cost of determining only one compiler setting that is only valid for one target application and one target architecture, is high. For instance, the iterative compilation system in [40] generates effective settings for two compiler transformations in several hundreds of iterations, which in general is too expensive. However, whereas iterative compilation is an expensive process to obtain better performance for just one application and one target architecture, using iterative compilation to obtain a general setting which can be used for many applications will spread the overhead over these applications, thereby making the cost acceptable.

Figure 1.6 illustrates the process of iterative compilation. In the first stage of the process, an arbitrary compiler setting is generated. The generated setting is compiled and executed in the next stage. Then the stop criterion is evaluated to determine whether an acceptable setting has been generated. The evaluation may involve profile information as well as direct feedback from the iterative mechanism.

The quality of the generated setting is crucial for the success of this approach. In [40], loop tiling and loop unrolling are applied to Matrix-Matrix multiplication, Matrix-Vector multiplication, and Forward discrete cosine transform. (In [24], the iterative compilation system is

also applied to optimize three benchmarks in the SPEC95FP.) The tile sizes of the loop tiling and the unroll factor of loop unrolling should be determined adequately. The best value of these parameters depends on the target architecture and application. Examples of loop tiling for a tile size of 64×64 and loop unrolling for an unroll factor of 2 on a small kernel are shown in Figures 1.7 and 1.8.

```
do i=1,n
    do j=1,n
        a[i,j]=b[i,j]+1
    enddo
enddo
```

(a) Original Code

```
do TI=1,n,64
    do TJ=1,n,64
        do i=TI,min(TI+63,n)
            do
            j=TJ,min(TJ+63,n)
            a[i,j] = b[i,j]+1
            enddo
        enddo
    enddo
enddo
```

(b) Transformed Code

Figure 1.7: Example of Loop Tiling

```
do i=1,n
    a[i] = a[i]+1
enddo
```

(a) Original Code

```
do i=1,n-1,2
    a[i] = a[i]+1
    a[i+1] = a[i+1]+1
enddo
```

(b) Transformed Code

Figure 1.8: Example of Loop Unrolling

Five different algorithms, a genetic algorithm, simulated annealing, grid search, window search, and random search, were employed in [40] to identify the best combination of tile sizes and unroll factors. Random search proved to be the quickest search algorithm for iterative compilation to generate the best settings.

[25] investigates the effectiveness of iterative compilation to reduce energy consumption. The authors conclude that several loop transformations, like loop unrolling and loop tiling, can effect the energy consumption of the optimized code, and therefore, iterative compilation is worthwhile to optimize a target code for energy reduction.

1.4 Our approach

Modern compilers are equipped with a large number of optimization switches and it is necessary to configure them carefully to obtain the best performance. Although the importance of setting the right compiler switches is evident, there exist few strategies to configure these compiler switches or flags. This is caused by the fact that the performance of a code is both dependent on the target architecture and the application. Therefore, it is extremely hard if not impossible to construct a generally applicable strategy. Additionally, the effect of the optimization switches can be dependent on other compiler switches causing the actual effect to be masked and not easily predictable. Therefore, it is inevitable for application developers to spend much time on hand optimization of their applications to obtain the best performance. The goal of our research is to find mechanism which determines the best set of compiler optimizations automatically. This thesis describes the algorithms with the experimental results of their application to the general compiler *gcc*. The results support our idea that searching for optimal compiler settings is feasible. This thesis introduces three approaches for identifying optimal compiler settings.

The first approach introduces the framework of Design of Experiments (DoE) [9]. DoE is proposed for effective data collection and analysis. In the framework of DoE, collected data is analyzed using statistical methodologies. In this thesis, we apply two kinds of statistical analysis, namely, main effect [32] and the Mann-Whitney test [35]. We employ Orthogonal Arrays (OAs) [32] to design our experiments. OAs are well known as suitable experimental plans which aim to qualify the effect of factors. In our approach, a column of an OA corresponds to a compiler option so that a row of an OA determines one entire compiler optimization setting. Main effect and the Mann-Whitney test are applied to the collected profiling data, and they identify effective compiler options with a large effect. Main effect and the Mann-Whitney test are conservative analyses so that they only identify few number of optimizations that have a large effect. Therefore, we propose an iterative algorithm to repeat the experiment by using the partial setting of compiler options which is identified in the previous iterations to complete the compiler setting. This algorithm can find compiler settings which outperform the standard *-O3* settings from the *gcc* compiler.

The second approach emphasizes the interaction between compiler options. In this approach, we measure the effect of combined compiler options while the first approach measures the effect of each compiler option separately. In this approach, we use a fixed number of compiler settings which are designed with an orthogonal array, and compute the effect of interaction of particular combination of compiler options. The definition of interaction is described in this thesis. This approach identifies an optimal compiler setting for multiple applications. The resulting compiler setting performs better than the *-O3* setting of the *gcc* compiler.

The third approach employs iterative compilation [8][40] using random search to find an optimal compiler setting. This approach shows that we can identify an optimal compiler setting in a limited number of iterations. The major difference between this approach and the previous approaches is that the previous methodologies only select optimizations which are significantly effective, while the compiler setting which is selected by iterative compilation

contains several no-effect optimizations because of the random generation of test settings. This approach enables us to estimate how much improvement we can obtain from the compiler optimizations. Therefore, the results from this approach can be used to evaluate compiler settings, for example, standard settings of the compiler (e.g., *-Ox*), or the compiler setting which we identified with the previous approach. We also applied this iterative compilation to find a compiler setting which is optimal for multiple applications. The resulting compiler setting is compared with the estimated optimal compiler setting for each application. The results show that the resulting settings achieve reasonably good performance for 7 applications.

1.5 Related work on generating optimal code

In this section, we describe related work to generate optimal code for an application. The problem is discussed for two domains, which are general purpose processors and embedded systems.

1.5.1 Generating optimal code for general purpose processors

General texts like the Dragon book [5], Waite and Goos [60], or Tremblay and Sorenson [58] simply give a collection of possible backend optimizations without much discussion about when to use them. Muchnick [50] simply sums up which optimizations should be employed one after the other based on experience.

A number of approaches to select best optimizations have been proposed by searching the optimization space. Iterative compilation [40] or the GAPS system that employs genetic algorithms [52] search for source level transformations. [16] use genetic algorithms to find optimal low level code sequences, paying attention to both performance and code size. [49] use machine learning techniques based on oblique decision trees to find compiler heuristics specific for a particular platform. [59] discusses an implementation of iterative compilation in the Intel IA-64 production compiler. In contrast to these efforts, our approach uses statistical analysis to systematically prune the search space and is focused on compiler switches. [56] uses an evolutionary algorithm to automatically find effective compiler heuristics for a particular application by searching for priority functions that drive optimizations. This work does not address the general problem of setting compiler switches.

Granston and Holler [26] propose a tool for automatic selection of compiler options, called *Dr. Options*. This tool uses information about the application supplied by the user and a set of tuning rules that have been created by interviewing tuning experts and analyzing optimization experiments. Hence, much compiler and application specific knowledge must be collected in order to create or even use such a tool. In contrast, our approach uses no knowledge at all but only statistical analysis to find an optimal setting. VISTA [65] is an interactive tool to assist the application programmer in finding optimizations and their phase order. However, it does not provide automatic optimization selection. [64] provides a framework

for predicting the impact of loop transformations to assist in selecting the optimal one. However, it is not clear how backend transformations can be incorporated in this framework and it does not provide automatic selection facilities. Whitfield and Soffa [61] propose a framework for specifying transformations and an automatic optimizer generator in order to experiment with transformations. However, this does not solve the problem of which transformations to enable.

Chow and Wu [12] approach the problem of determining which options to set for a given application as a fractional factorial experiment based on *aliasing* or *confounding* [9]. The effect of options is analyzed using a linear regression model. This approach has a number of drawbacks compared to the present work. First, it is not clear whether such a linear model can be used to accurately model compiler options. It is well known that many aspects of program execution are non-linear. Hence, it is not clear that such a model may be used and the authors do not give an argument for using it. Second, each alias actually is a generator for a collection of other aliases [9]. Generators that minimize the number of derived aliases are only known for a limited number of alias structures. It is far from clear how to alias many factors as we would need for modern compilers and we are likely to end up with many derived aliases that completely obscure what we are actually measuring. For this reason, Chow and Wu first restrict attention to 9 interesting options for which a minimal set of generators is available. However, this step requires detailed knowledge about the compiler. In contrast, our approach can start with all options available without any need to make an initial selection. The most important difference between [12] and our approach in this thesis, however, is that Chow and Wu use complex statistical analysis requiring many new experiments to resolve ambiguities and to find options with high main effects and interactions.

1.5.2 Generating optimal code for embedded systems

Although performance is obviously important for embedded systems and the previous approaches to optimization for performance can be applied to the embedded domain also, there also exists another optimization goal for embedded systems, namely code size.

Code size is a main cost factor for many high volume electronic devices. Reducing code size enables us to reduce the size of memory which has a high proportion of a product cost. Also, reducing code size enables us to put more features in the same ROM which may enhance the value of a product. It is therefore important to reduce the size of the applications in an embedded system.

Several methods have been proposed to deal with this problem, mostly based on compressing the binaries. Code compression using, e.g., Huffman coding, is a much studied approach to code size reduction [7]. Wolfe and Chanin [62] presented the first paper on this topic. In their approach, the code is compressed off-line and is executed by decompressing the code into the cache by a special hardware unit. The method is transparent to the CPU and only the cache refill engine needs to be modified. This approach was improved by Breternitz and Smith [10], and Lekatsas and Wolf [44]. Lefurgy et. al. [43] apply a dictionary based approach. Hoogerbrugge et. al. [36] approach code compression in a different way. They generate code for a virtual processor with a special instruction set. This code is the compressed version of the original code. It is then translated (i.e., decompressed) before execution on

the target processor by means of a dictionary. This approach allows macro-instructions that encode sequences of target instructions. Ernst et. al. [20] produce compressed low-level intermediate code that needs to be decompressed and interpreted on the native machine. Franz and Kistler [22] store a program in a platform independent intermediate representation that is moreover compressed. This compressed representation is called a slim binary. To execute a slim binary, it is decoded and native machine code is generated from the intermediate representation. Although compression can achieve a high reduction in code size, its obvious drawback is that it requires decompression at runtime. This can affect the performance of the code, although it has been shown that this overhead can in some cases be hidden. Moreover, ROM is required to store the decompression algorithm which reduces the gains achieved by compression.

There exist some papers that study how the compiler can be used to reduce code size. One important transformation that is specifically geared toward code size reduction is code factoring [23]. This transformation can be seen as the inverse to function inlining. The assembly code is searched for repeating patterns that are encapsulated in a new function and the patterns are replaced by a call to this function. Cooper and McIntosh [15] improved the original idea. The *squeeze* binary-rewriting tool uses aggressive inter-procedural optimization and code factoring [18]. Mathias et. al. [46] employ genetic algorithms to detect repeating patterns. The transformation reduces code size by 5 to 10%. As a drawback, code factoring can give rise to longer execution times by increasing the number of dynamic instructions and cache miss rates [57].

Instead of the compiler, the linker can also be adapted to produce size reduced binaries, as has done in the work by De Bus et. al. [11]. Instead of minimizing code size, code size constraints can also be enforced. Heydeman et. al. [34] use Integer Linear Programming to find loop unroll factors to maximize performance under code size constraints. Naik and Palsberg [51] phrase register allocation and code generation as an Integer Linear programming problem where the upper bound on the code size can be expressed as an additional constraint.

Cooper et. al. [16] have proposed to use genetic algorithms to search for short code sizes using a research compiler. However, they only employ 10 options, in contrast to the approaches in this thesis which use 53 options. It is not immediately clear that such large number of options will not lead to combinatorial explosion in their approach. Moreover, their compiler allows them to specify the order in which these optimizations are applied and the same optimization may occur several times in an optimization sequence. However, this order can generally not be manipulated in a production compiler.

1.6 Overview of this thesis

This thesis is organized as follows. Chapter 2 describes the statistical techniques which we employ in this thesis. Chapter 3 shows our algorithm using the main effect of a compiler option to determine compiler settings. Chapter 4 applies the Mann-Whitney test to determine the significance of compiler options for determining compiler settings. In Chapter 5, the Mann-Whitney test is applied to determine compiler settings which intend to reduce the code

size of executables. Chapter 6 shows our methodology to employ the definition of interaction between compiler options. Chapter 7 describes the application of iterative compilation to the problem of defining an optimal compiler setting for multiple programs. Finally, Chapter 8 summarizes this thesis.

1.7 Overview of publications

Some parts of this thesis have been previously published.

- Chapter 4: Using the Mann-Whitney test to optimize the execution time of a single application.
  This chapter is published in the proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05) [27]. Part of this chapter is also published in the proceedings of the workshop on Performance Optimization High-Level Languages and Libraries (POHLL'06) [31].

- Chapter 5: Using the Mann-Whitney test to optimize the code size of a single application.
  This chapter is published in the proceedings of the workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'06) [30].

- Chapter 6: Determine a compiler setting for multiple applications taking into account interaction between optimizations.
  This chapter is published in the proceedings of the second conference on computing frontiers (CF'05) [29].

- Chapter 7: Using random search to determine a compiler setting.
  This chapter is published in the proceedings of the 19th annual international conference on supercomputing (ICS'05) [28].

Chapter 2

Preliminaries

Design of Experiments (DoE) [9] provides a methodology to plan experiments from which we can obtain data which allow statistical analysis. Orthogonal Arrays have been used in the framework of DoE and are defined mathematically in [32]. In this chapter, we first explain the features of orthogonal arrays, and how the main effect is computed from experimental data. Next, we explain about inferential statistics and the Mann-Whitney test.

2.1 Orthogonal arrays

Orthogonal Arrays (OAs) have been proposed as an efficient design of experiments [32]. Each element of an array expresses the setting of an experimental factor (i.e., a compiler switch). The columns of an orthogonal array are associated with the experimental factors (compiler switches), hence each row of an orthogonal array represents one experimental compiler setting to be executed. In this thesis, we deal with binary compiler switches so that we use orthogonal arrays which are constructed with $+1$'s and -1's. In order to reduce the overall experimental set up, we have chosen to use OAs of strength 2. This means that for any two arbitrary columns, the patterns

$$+1+1 \qquad -1+1 \qquad +1-1 \qquad -1-1$$

occur equally often. This property is useful to avoid biased profiling data [32]. Figure 2.1 shows an example of an Orthogonal Array of strength 2. So, for instance, if we take the first and fifth column we can see that each combination $+1+1$, $-1+1$, $+1-1$, and $-1-1$ occurs exactly 3 times.

$$
\begin{pmatrix}
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
+1 & -1 & +1 & -1 & -1 & -1 & +1 & +1 & +1 & -1 \\
+1 & +1 & -1 & +1 & -1 & -1 & -1 & +1 & +1 & +1 \\
-1 & +1 & +1 & -1 & +1 & -1 & -1 & -1 & +1 & +1 \\
+1 & -1 & +1 & +1 & -1 & +1 & -1 & -1 & -1 & +1 \\
+1 & +1 & -1 & +1 & +1 & -1 & +1 & -1 & -1 & -1 \\
+1 & +1 & +1 & -1 & +1 & +1 & -1 & +1 & -1 & -1 \\
-1 & +1 & +1 & +1 & -1 & +1 & +1 & -1 & +1 & -1 \\
-1 & -1 & +1 & +1 & +1 & -1 & +1 & +1 & -1 & +1 \\
-1 & -1 & -1 & +1 & +1 & +1 & -1 & +1 & +1 & -1 \\
+1 & -1 & -1 & -1 & +1 & +1 & +1 & -1 & +1 & +1 \\
-1 & +1 & -1 & -1 & -1 & +1 & +1 & +1 & -1 & +1
\end{pmatrix}
$$

Figure 2.1: Example of an Orthogonal Array with 10 Columns

In order to derive these orthogonal arrays of strength 2, we make use of Hadamard matrices [32]. Hadamard matrices are square matrices of $+1$'s and -1's whose rows are orthogonal. A Hadamard matrix of order n is denoted by H_n, and it satisfies:

$$H_n H_n^T = nI_n \tag{2.1}$$

Hadamard matrices have the following features:

- When H_n exists, $n = 1$, $n = 2$, or n is a multiple of 4.
- If and only if there exists a Hardamard matrix of order 4λ where λ is a given integer, then a $4\lambda \times 4\lambda - 1$ orthogonal array of strength 2 can be constructed from it.

We can derive orthogonal arrays from Hadamard matrices, as follows [32]:

1. Symmetrically permute the rows and columns of an Hadamard matrix, so that the first row and first column consists of $+1$'s.
2. Drop the first column from this Hadamard matrix.
3. Negate this Hadamard matrix.

So, orthogonal arrays which are derived from Hadamard matrices have as first row all -1's. Since this feature is convenient for our experiments, we employ them throughout this thesis. Hadamard matrices have been found up to the order of 256, and they are available in [55]. We use these Hadamard matrices to produce the OAs that are used in this thesis. Later in this thesis, we denote $+1$ by 1 and -1 by 0 to make the notations more easily readable.

2.2 Main effect

The main effect represents the effect of an experimental factor (compiler optimization), without taking into account the other factors in the analysis. In general, an OA of strength t allows us to compute the interaction effect of $\lfloor \frac{t}{2} \rfloor$ factors for $t \geq 2$. Therefore, OAs of strength 2 can be used to compute the main effect of compiler options.

In order to explain how to compute the main effect of options, we need some notation. We use A to express an OA as a set of compiler settings. A is an $N \times k$ matrix, hence N settings of k compiler switches are determined by A. A row of A, which corresponds to one entire compiler setting, is denoted by s. The ith element of s (s_i) describes the setting of the ith option of the compiler setting, namely, 1 corresponds to on and 0 corresponds to off. Our methodology works independently of what is chosen as the optimization target, for example it can be execution time, code size, energy consumption, and so on. In this thesis, we use execution time as the optimization target. The execution time of given program that is obtained using of a compiler setting $s \in A$ is denoted by $T(s)$.

The main effect of the option O_i with respect to OA A, which is denoted by $E(O_i)$, is defined in Equation 2.2.

$$\begin{aligned} E(O_i) &= \frac{(\sum_{\{s \in A: s_i = 1\}} T(s))^2}{N/2} + \frac{(\sum_{\{s \in A: s_i = 0\}} T(s))^2}{N/2} - \frac{(\sum_{\{s \in A\}} T(s))^2}{N} \\ &= \frac{(\sum_{\{s \in A: s_i = 1\}} T(s) - \sum_{\{s \in A: s_i = 0\}} T(s))^2}{N} \end{aligned} \tag{2.2}$$

The main effect of an option is computed with respect to the entire OA A. Equation 2.2 uses the sum of squares of execution times when the option is switched on or off in *an arbitrary context of other options*. The OAs have the properties that

1. There are exactly the same number of rows that switch an option O_i on as there are rows which switch that option off, namely, $\frac{N}{2}$.

2. For an arbitrary other option O_j, in the set of rows in which O_i is switched on, there are exactly $\frac{N}{4}$ rows that switch O_j on and there are exactly $\frac{N}{4}$ rows that switch O_j off. Likewise for the set of rows that switch O_i off.

This last property of OAs ensures that main effect can be computed with high precision. However, only sufficiently large main effects should be taken into consideration because the main effect is calculated from measured data which might contain measurement errors. So, whenever the computed main effect is small, the noise of the experimental data is too large to ensure a correct setting of this option.

In order to determine the effects of the compiler options, we normalize the main effect using the sum of main effects of all options. We call the normalized main effect *relative effect*, and it is denoted by $RE(O_i)$. $RE(O_i)$ is given by

$$RE(O_i) = \frac{E(O_i)}{\sum_{j=1}^{k} E(O_j)} \cdot 100\% \tag{2.3}$$

Since the effects are expressed as squares in Equation 2.2, the actual effect of an option can be both positive and negative. The main effect simply expresses whether an option does

or does not affect execution time and not whether it improves or degrades performance. To distinguish between these two possibilities, we define the *improvement* of option O_i with respect to an OA A, denoted by $I(O_i)$, as follows

$$I(O_i) = \frac{\sum_{\{s \in A : s_i = 0\}} T(s) - \sum_{\{s \in A : s_i = 1\}} T(s)}{\sum_{\{s \in A : s_i = 0\}} T(s)} \tag{2.4}$$

This equation can be used to decide whether an option is beneficial for achieving better performance or not.

2.3 Inferential statistics

Most experiments in the field of scientific computing aim to support a prediction or an estimation of a phenomenon. The prediction or estimation is called an *experimental hypothesis*, and the study of inferential statistics aims to predict whether an experimental hypothesis is likely to be true. For our purpose where we want to decide whether compiler option A is beneficial for application B, the experimental hypothesis reads

> **Experimental Hypothesis** Compiler option A is effective to optimize application B.

The following three steps describe the basic idea of inferential statistics. First, we define a *null hypothesis* which negates an experimental hypothesis. When we want to know about the effectiveness of compiler option A for application B, the null hypothesis is

> **Null Hypothesis** Compiler option A is not effective to optimize application B.

We want to conduct an experiment that either confirms or rejects this null hypothesis. In the latter case, the experimental hypothesis which is the negation of the null hypothesis is accepted.

Second, we conduct an experiment which contains two groups which are respectively called the *control group* and the *experimental group*. The control group consists of the experimental runs that do not use compiler option A. The experimental group consists of the experimental runs which use compiler option A. The null hypothesis implies that the execution times from these two groups are the same. Hence, if we can conclude that the execution times from these two groups are significantly different, then we may reject the null hypothesis.

Third, we need to see the difference between two groups. A well known method is to compare the average values of the two groups. When these two values are significantly different, it may be appropriate to reject the null hypothesis and to conclude that compiler option A is effective for application B. Inferential statistics provides a so-called test statistic to evaluate the difference. The test statistic enables us to assess a confidence rate to support an experimental hypothesis. Inferential statistics knows many tests that can be applied to various experimental data, and a test statistic is defined for each such test [19].

O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	Time	Rank
0	0	0	0	0	0	0	0	0	0	20	8
1	0	1	0	0	0	1	1	1	0	25	12
1	1	0	1	0	0	0	1	1	1	15	4
0	1	1	0	1	0	0	0	1	1	17	5
1	0	1	1	0	1	0	0	0	1	18	6
1	1	0	1	1	0	1	0	0	0	14	2
1	1	1	0	1	1	0	1	0	0	23	11
0	1	1	1	0	1	1	0	1	0	13	1
0	0	1	1	1	0	1	1	0	1	19	7
0	0	0	1	1	1	0	1	1	0	22	10
1	0	0	0	1	1	1	0	1	1	21	9
0	1	0	0	0	1	1	1	0	1	16	3

Table 2.1: Example of Experimental Settings for 10 Compiler Options

There are two kinds of analysis in inferential statistics, which are called *parametric statistics* and *non-parametric statistics* [19]. They differ in what kind of data they can handle. Parametric statistics requires some assumptions on the data distribution, and non-parametric statistics allows us to analyze data without an underlying distribution. Parametric statistics uses raw scores from experimental data and non-parametric statistics uses ranked experimental data. That is, we first order or rank the raw data and apply statistics on the resulting ranking numbers [35].

Our approach uses orthogonal arrays as experimental designs. An example of an experimental design is shown in Table 2.1. We divide the experiment into two groups according to the value of one option. When we want to know the effect of option O_1 in the experiment, we select the rows which contain $O_1 = 1$ for the experimental group, and we select the rows which contain $O_1 = 0$ for the control group. It is clear that the rows which contain $O_1 = 1$ are all different from the other options. The same holds for the rows which contain $O_1 = 0$. This means that we cannot say that the distribution of values in the two groups have a normal distribution. Therefore, we better employ non-parametric statistics in our method.

2.3.1 The Mann-Whitney test

The Mann-Whitney test is a well known test in inferential non-parametric statistics [35]. We apply the Mann-Whitney test to analyze our experimental data. The Mann-Whitney test uses ranked experimental data to handle the data of non-normal distribution.

We explain the Mann-Whitney test using an example. Table 2.1 shows an experimental design and the resulting execution times for a compiler with ten options. The first ten columns correspond to the ten compiler options and each row expresses a total compiler setting. The last column shows the ranks in ascending order.

We want to decide whether compiler option O_2 affects the optimization process signif-

	Group 1 ($O_2 = 1$)	Group 2 ($O_2 = 0$)
	4	8
	5	12
	2	6
	11	7
	1	10
	3	9
Total	$T_1 = 26$	$T_2 = 52$

Table 2.2: The Effect of Compiler Option O_2

icantly. The experimental data is separated into two groups, one group uses option O_2 and the other does not use option O_2. We call them group 1 and group 2, respectively. Table 2.2 shows that each group consists of six data points. The last row of the table is the sum of the ranks for each group. We denote the total sum of each group by T_1 and T_2, respectively.

The test statistic in the Mann-Whitney test is based on the value of T_1. In order to discuss how the test works, assume that the two groups respectively contain m members and n members and that the option to be analyzed is the only difference between the groups. Suppose that the option is not effective, that is, that the null hypothesis is true. Then the assignment of ranks is basically random, resulting from experimental errors in the measurements. Looking at which values T_1 can have, T_1 is at least $1 + 2 + \cdots + m$ and at most $(m + 1) + \cdots + (m + n)$. The first value occurs when all measurements in group 1 happen to be slightly smaller than the measurements from group 2. The second value occurs in the opposite case. For an intermediate value, there are more possibilities to rank the measurements. Hence, the chance that T_1 has such an intermediate value is larger. It has been shown [19, 35] that T_1 has a *normal distribution* with mean

$$\mu = \frac{m(m+n+1)}{2} \tag{2.5}$$

and standard deviation

$$\sigma = \sqrt{\frac{m \cdot n \cdot (m+n+1)}{12}} \tag{2.6}$$

The Mann-Whitney test considers the test statistic z which is given by

$$z = \frac{T_1 - \mu}{\sigma}, \tag{2.7}$$

That is, z measures how far T_1 lies from the mean expressed in units of standard deviation. Then z is normally distributed also and this distribution is given by

$$Y(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}z^2} \tag{2.8}$$

A graphical representation of this normal distribution is shown in Figure 2.2. The normal distribution expresses the chance to measure a certain value for z. Hence,

$$\int_{-\infty}^{\infty} Y(z)dz = 1$$

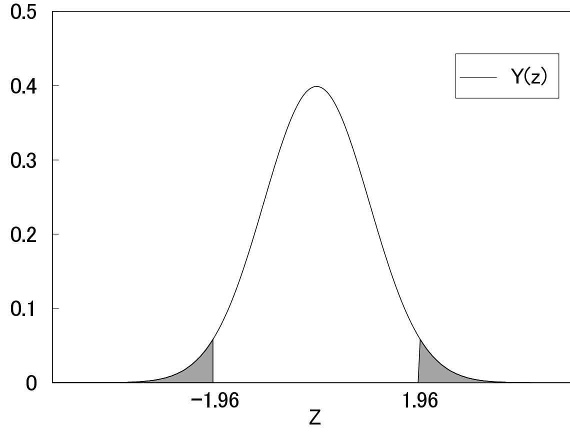


Figure 2.2: Normal Distribution of z

If the measured value of T_1 is significantly different from the mean μ, then we may conclude that the null hypothesis is false because it is highly unlikely that we measure such a value by chance. A standard criterion for "significant difference" is when z is larger than 1.96 or smaller than -1.96, as indicated in the figure. This means that the chance of measuring such a value for z when the null hypothesis is true, is less than 5%.

For a value t with $t > 0$, $P(t)$ denotes the chance that we measure a value for z such that $|z| \geq t$. That is, either $z \geq t$ or $z \leq -t$. $P(t)$ is given by

$$P(t) = \left(1 - 2 \cdot \int_0^t Y(z)dz\right) \cdot 100\% \tag{2.9}$$

In the Mann-Whitney test, a z value which leads to $P(|z|) < 5\%$ is often used to conclude that the observed data is considered not to belong to the data which defines the normal distribution [45]. The probability to reject the null hypothesis is called the *critical value*. This enables us to decide whether or not to reject the null hypothesis since the null hypothesis assumes that the two groups in an experiment are the same. We employ a critical value of 5% in this thesis.

In the current example, $m = 6$ and $T_1 = 26$. Hence, the z value for O_2 is computed as follows:

$$\sigma = \sqrt{\tfrac{6 \cdot 6 \cdot (12+1)}{12}} = \sqrt{39}$$
$$\mu = \tfrac{6 \cdot (1+12)}{2} = 39$$
$$z = \tfrac{(26-39)}{\sqrt{39}} = -2.08$$

By using Equation 2.9, we determine whether the observed data satisfies the criterion $P(|z|) < 5\%$. Since $z = -2.08$, this yields $P(|-2.08|) = 3.75\%$. Hence O_2 satisfies $P(|z|) < 5\%$, and we can conclude that compiler option O_2 has a significant effect on the optimization. The obtained z value is negative so that the observed data in group 1 implies shorter execution times than group 2. This means that the effect of O_2 is positive. This procedure can be applied

for each optimization in Table 2.1. Therefore, we can determine the important options among the 10 options with 12 experimental runs.

The example in this section uses very small dataset to simplify the problem. In general, it is better to use big dataset for the reliability of analysis. Of course, we want to keep the data size as small as possible because of the resource cost. The next section discusses the appropriate size of experiment.

2.3.2 Determining the sample size

This section discusses how to determine the sample size, the size of the control and experimental group. This is then used to determine the size of the Orthogonal Array. The size is based on the so-called *power analysis* which aims to asses the reliability of the Mann-Whitney test. Since the test estimates the effectiveness of compiler optimizations by executing several compiler settings, there always exists the possibility that the outcome of the test is erroneous. There are two types of errors which may occur when we apply the Mann-Whitney test. One is that we reject the null hypothesis while it is true, and another one is that we do not reject the null hypothesis while it is false. These two types of errors are called respectively a type *I* error and a type *II* error.

We have specified the critical value in Equation 2.9 to decide whether the null hypothesis is to be rejected or not. This value indicates the probability of a type *I* error since this probability shows how hard the observed data happens based on the null hypothesis. We have set this probability to 5%.

The probability that a type *II* error does not occur is called the *power* of the hypothesis test. Hence, the power of the test expresses the probability to reject the null hypothesis, given that the null hypothesis is false. Obviously, we want to have this probability as large as possible, subjected to our conflicting wish to have the sample sizes as small as possible to reduce profiling time. The power of the test is determined by the sample size and the critical value for the Mann-Whitney test [35]. Since the critical value is set to 5% for the sake of reliability of the test, we only can change the sample size to influence the power. To give more insight in the concept of power, consider the distribution in Figure 2.2. A power of $X\%$ is determined by the area that lies symmetrically around the mean and that takes up $X\%$ of the total area. Hence, each power gives rise to a value for z.

Power analysis is a methodology which aims to choose an adequate sample size. The analysis formulates the trade off due to the choice of the critical value for the Mann-Whitney test and the sample size. A power of 80% is sufficient to have high confidence in the outcome of the Mann-Whitney test [35]. We denote the critical value for the Mann-Whitney test by α and the power by $1-\beta$. That is, if we want to have a power of 80%, then $\beta = 20\%$. Corresponding z-values are expressed by z_α and z_β, respectively. The sample size N is given by the following formula [35].

$$N = \left\lceil 5 \cdot (z_\alpha + z_\beta)^2 \right\rceil \tag{2.10}$$

Chapter 3

Using main effect to optimize the execution time of a single application

The Main effect is our first methodology employed to achieve our goal of finding optimal compiler settings. This approach has first been described in [53] using the Simplescalar framework, which is a microarchitecture simulator, and *gcc 2.6*. [53] concludes that the compiler settings which are determined using main effects of compiler optimizations outperform the standard setting *-O3*, which is provided by *gcc*. This chapter shows the results of the same algorithm as described in [53] using the Pentium 4 architecture and *gcc 3.3.1*. The major difference from [53] is that the results shown in this chapter use measured execution times of applications, and the number of compiler optimizations, which is 14 in *gcc 2.6* and more than 60 in *gcc 3.3.1*.

This chapter is structured as follows. In Section 3.1, we describe our algorithm using the main effect. The experimental setting is described in Section 3.2. We interpret our algorithm with one of the results from our experiment in Section 3.3. All results are shown in Section 3.4, and we discuss the impact of a parameter in our algorithm in Section 3.5. Finally, Section 3.6 summarizes this chapter.

3.1 The iterative search algorithm

In this section, we present our iterative algorithm, which employs the main effect. The algorithm, given in Figure 3.1, aims to find an optimal compiler setting for a given application. The iterative algorithm, first, identifies options with a large effect and switches them on or off. Then another experiment is run to look at the remaining options to see what improvement they can produce with the partial setting already constructed. Thus, the algorithm in Figure 3.1 starts with a high dimensional optimization space and subsequently cuts down this space by fixing some dimensions and zooming in on the remaining options that have a smaller

- Repeat:
  - Compile the application with each row from OA A as compiler setting and execute the optimized application.
  - Compute the relative effect of option using Equation 2.2 and 2.3.
  - If the effect of an option is larger than a threshold of 10%,
    - if the option has a positive improvement according to Equation 2.4, switch the option on, or else
    - if it has a negative improvement, switch the option off.
  - Construct a new OA A by dropping the columns corresponding to the options selected in the previous step.
- until
  - All options are set, or
  - The variance in the experimental data becomes less than 0.5%, or
  - No more options have a significant effect.

Figure 3.1: Iterative search algorithm

effect.

Note that we do not select each option that has a positive effect but only those that have an effect larger than a threshold of 10%. The value of this threshold has been determined empirically. We use it since, due to the inaccuracy of small OAs, small measured effects can be distorted. This threshold value filters this phenomenon.

It is obvious that this algorithm finds options that have a large impact on the execution of an application in the primary stage of this iterative search. The options with small effect are selected in later iterations, as well as the options which perform well when they are turned on together. Suppose that there are two options O and O' that have a small effect when turned on separately but a significant effect when turned on together. Their main effects are small during the early iterations. However, their relative effect becomes larger in later iterations. So eventually, they will be selected. The algorithm stops when all options are set. However, we also added an additional stop criterion for practical reasons.

3.2 The experimental environment

To show the effectiveness of our algorithm, a case study is performed using *gcc 3.3.1* which implements more than 60 optimizations [13]. Of these optimizations, 45 optimizations are selected, and arranged into 42 factors as shown in Table 3.1(a). These optimizations are also arranged into 27 factors, as shown in Table 3.1(b). Certain options are grouped together because they are most likely to influence each other. This is done because of the way the main effect is computed. Interacting options are not dealt with directly. In order to investigate how

1	defer-pop
2	force-mem
3	force-addr
4	inline-functions
5	optimize-sibling-calls
6	merge-constants
7	strength-reduce
8	thread-jumps
9	cse-follow-jumps
10	cse-skip-blocks
11	rerun-cse-after-loop
12	rerun-loop-opt
13	gcse gcse-lm gcse-sm
14	loop-optimize
15	crossjumping
16	if-conversion
17	if-conversion2
18	delete-null-pointer-checks
19	expensive-optimizations
20	optimize-register-move
21	schedule-insns
22	sched-interblock
23	sched-spec
24	schedule-insns2
25	sched-spec-load
26	sched-spec-load-dangerous
27	caller-saves
28	move-all-movables
29	reduce-all-givs
30	peephole peephole2
31	reorder-blocks
32	reorder-functions
33	strict-aliasing
34	align-functions
35	align-labels
36	align-loops
37	align-jumps
38	rename-registers
39	cprop-registers
40	function-sections
41	data-sections
42	unroll-loops

(a)Total Option List

1	defer-pop
2	force-mem
3	force-addr
4	inline-functions
5	optimize-sibling-calls
6	merge-constants
7	strength-reduce
8	thread-jumps
9	gcse gcse-lm gcse-sm cse-follow-jumps cse-skip-blocks rerun-cse-after-loop
10	loop-optimize rerun-loop-opt
11	crossjumping
12	if-conversion if-conversion2
13	delete-null-pointer-checks
14	expensive-optimizations
15	optimize-register-move
16	schedule-insns sched-interblock sched-spec schedule-insns2 sched-spec-load sched-spec-load-dangerous
17	caller-saves
18	move-all-movables
19	reduce-all-givs
20	peephole peephole2
21	reorder-blocks reorder-functions
22	strict-aliasing
23	align-functions align-labels align-loops align-jumps
24	rename-registers
25	cprop-registers
26	function-sections data-sections
27	unroll-loops

(b)Option List
(arranged into 27 factors)

Table 3.1: List of Option Set

	Name (#Lines)	Description
1	164.gzip (4333)	gzip (GNU zip) is a data compression program. gzip uses Lempel-Ziv coding (LZ77) as its compression algorithm.
2	175.vpr (8899)	VPR is a placement and routing program for technology-mapped circuit.
3	181.mcf (1120)	The program is designed for the solution of single-depot vehicle scheduling problems occurring in the planning process of public transportation companies.
4	197.parser (6839)	The Link Grammar Parser is a syntactic parser of English.
5	256.bzip2 (2955)	256.bzip2 is a data compression program.
6	254.gap (27523)	It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
7	255.vortex (31128)	Vortex is a single-user object-oriented database transaction benchmark.

Table 3.2: List of Benchmark Programs

well the main effect is handling interaction we also conducted experiments with the option list in Table 3.1(a), which separates the options as much as possible.

In this chapter, we do not take into account *omit-frame-pointer*, since standard *-Ox* switches also do not turn on this option (it cancels out debugging). *inline* is not used since this option disables the inline directive in the target applications and we do not intend to change the contexts of the applications. *keep-static-consts* and *function-cse* are not included since these options aim to keep the assembler code readable. *branch-count-reg* is left out since this option disables the use of instructions regarding the branch count registers when it is negated. *prefetch-loop-arrays* is not used since this option disables the use of prefetch instructions. *float-store* and *single-precision-constant* are also left out to avoid generating invalid executables. The option *delayed-branch* is also not present since our target architecture Pentium 4 does not support delayed branching. As we mentioned in Section 2.1, we use Orthogonal Arrays derived from Hadamard matrices. Therefore, the number of rows of an orthogonal array which we can choose is a multiple of 4.

The option list in Table 3.1(b) contains 27 options, therefore we employ an 28×27 orthogonal array in our experiments. It is the smallest size of an orthogonal array which can be used in our experiments. Similarly, we choose a 44×42 orthogonal array for the option list in Table 3.1(a).

We use a *Pentium 4* at 2.8GHz as our target architecture and the *SPECint 2000* benchmark suite with the train input dataset as the target application. We use 7 out of 12 programs from the benchmark suite due to some technical problems (i.e., compile errors). The benchmark programs are listed in Table 3.2. The *UNIX* time command is used to measure the execution time. Execution time can be different when we run the same executable several times. To smooth out this experimental fluctuation, we run the same executable code 10 times, and take the average value of these runs. However, if a run causes the standard deviation of all

10 runs to change by more than 0.5% of the average value, then this run is deleted from the experiments, and the overall average is taken from the remaining runs.

The *gcc* compiler provides users a number of optimizations, which can be configured explicitly. All improvements discussed in this chapter are with respect to the optimization level *-O* with the options in Table 3.1 explicitly turned off. We denote this setting as O_{base}. Besides these, *gcc* provides several optimizations which cannot be disabled explicitly, namely common subexpression elimination, dead code elimination, and so on. By definition O_{base} includes these optimizations. The improvement of O_{new} is computed by using the following equation. We denote the execution time of O_{base} by $T(O_{base})$, and the execution time of O_{new} by $T(O_{new})$.

$$\frac{T(O_{base}) - T(O_{new})}{T(O_{base})} \tag{3.1}$$

Throughout this chapter, we refer to this definition when we discuss the improvement of a compiler setting

3.3 A case study

First, we show how our algorithm works on *parser* which is one of the applications we used using the 27 options in Table 3.1(b). Figure 3.2(a) shows the experimental data from the first iteration of the algorithm. The X-axis represents 28 compiler settings, which are determined by the 28×27 orthogonal array, and the Y-axis represents the improvement in execution time. At the first iteration, no partial setting has yet been determined, so that the 28 compiler settings are fully configured by the orthogonal array. The data values vary from −6% to 12%. The average improvement of all these setting is expressed by the dotted line. Figure 3.2(b) shows the measured main effect of the 27 options. The X-axis represents each of the 27 options and the Y-axis represents the effect, which is defined in Equation 2.3. A light color bar shows that the option has a positive effect and a dark color bar means that the option has a negative effect. There are several blank columns, for example for option 1, 3, and so on. This is caused by the fact that the measured value is too small. In Figure 3.2, we can observe that the 4th option, which corresponds to *inline-functions*, has a large positive effect. Since this value exceeds the 10% threshold value, the algorithm sets this option *on*, and constructs the next experiment to determine the reminder options. The setting is shown in Table 3.3.

Figure 3.3(a) shows the experimental data from the second iteration. The average improvement in the settings is significantly improved, from 3.7% to 8.6%. Also, the variation in the experimental data is significantly smaller. We can say that the search space is much smaller as an outcome of the first iteration. In this second iteration, the main effect of all the options with the exception of option 4, is shown. In Figure 3.3(b), we can see that option 9, which corresponds to *gcse*, has a positive effect, and options 15 and 16, which correspond to *optimize-register-move* and *schedule-insns*, respectively, have negative effects, hence 9 is turned on and 15, 16 are turned off. In the same way, the algorithm generates the third experiment, and the outcomes are shown in Figure 3.4. Again, we can observe significant increase on the average improvement of the compiler settings, from 8.6% to 10.7%, and the experi-

parser	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				1																							
2				1					1						0	0											
3			0	1					1		0				0	0				0	1						

Table 3.3: Settings per Iteration for Parser (until 3rd iteration)

mental data has also less variance compared to previous iterations. Now, we turn on option 21 and turn off option 3, 11, and 20.

3.4 Results

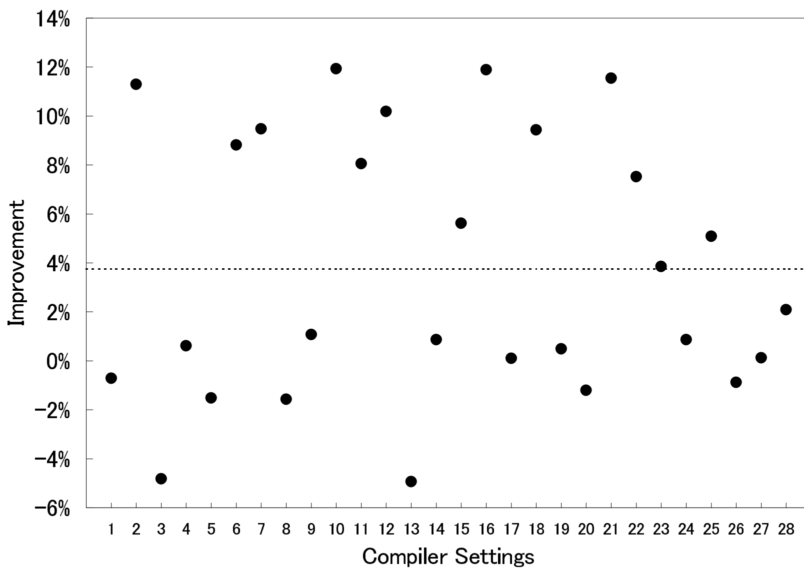
In this section, we present the results of our algorithm which is applied to 7 applications in the *SPEC 2000* benchmark suite. We used both option lists shown in Table 3.1. Therefore, we have two results, which are denoted in Figure 3.5 by O_{new27} and O_{new42} respectively. We also put the improvement of the *-O3* setting as a reference. The numbers in brackets indicate the number of iterations needed to reach the final setting. The first number represents the number of iterations using 27 options, and the second number represents the number of iterations using 42 options. On average, it costs 9 iterations to identify a compiler setting. Table 3.5 shows the selected options for each iteration of the experiment using 27 options. Table 3.6 shows the selected options for each iteration of the experiment using 42 options. The setting in the last iteration in Table 3.5 is denoted by O_{new27}, and the setting in the last iteration in Table 3.6 is denoted by O_{new42} in Figure 3.5. O_{max} in the tables is the setting, which has the maximum improvement in the experiment. We put O_{max} in the tables to compare the O_{max} setting with the selected options in the experiment. We also put the O_{new27} setting, which is expanded into 42 options, in Table 3.6.

Figures 3.8 and 3.9 show the results of the selected setting, which is the intermediate setting with options not selected yet turned off when it is before the final iteration. Each graph consists of 4 lines, denoted by *partial-improvement*, *improvement-max*, *ave* and *Coefficient of Variance*. The dotted horizontal line represents improvement of the *O3* setting. *partial-improvement* represents the improvement of a compiler setting which only applies selected optimizations during the experiment. *improvement-max* shows the maximum improvement in the experimental data per iteration. *ave* represents the average improvement of the experimental data per iteration. *Coefficient of Variance* expresses the convergence of the experimental data, which is the value of the normalized standard deviation using the mean of the data [9]. A small value of the coefficient of variance indicates the collected data has low variance.

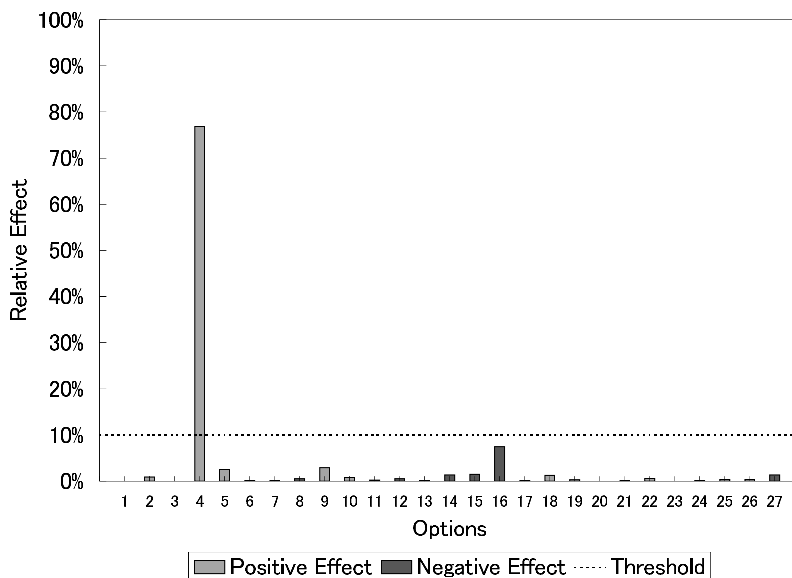
From the results, we can see that our algorithm identifies better compiler settings than the *-O3* setting except for *mcf* and *parser*. Mostly, the compiler settings, which are identified from the list of 27 options are better than the ones from the list of 42 options. Especially, we observe degradation of performance for the O_{new42} setting when configured for *vpr*.

The O_{new} settings and the O_{max} settings differ on the value of several options. The O_{max} setting of *bzip2* is found in the last iteration so that there is no difference between O_{max} and O_{new27}. For the other benchmarks, O_{max} and O_{new27} are different at several options.

In Figure 3.8(g), we observe that there is a large difference between the performance of O_{new27} and O_{max}, which yields the conclusion that our algorithm does not find optimal

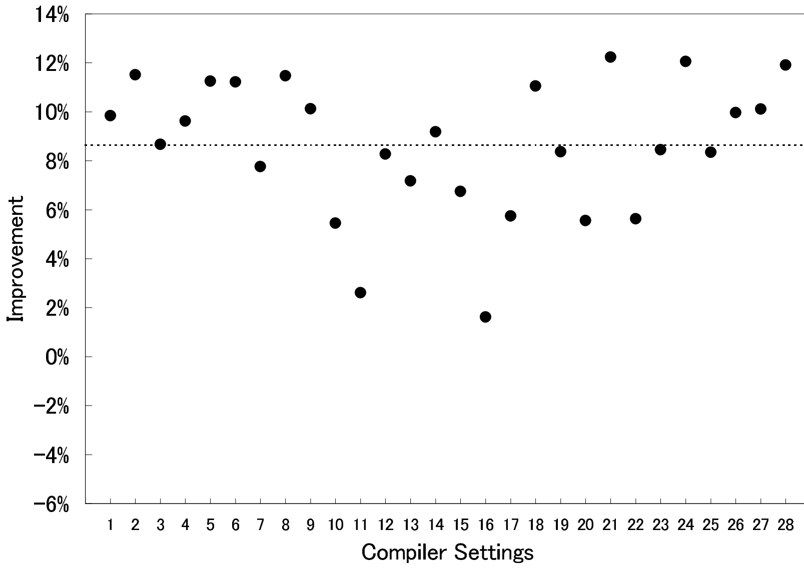


(a) Improvements for 28 Settings

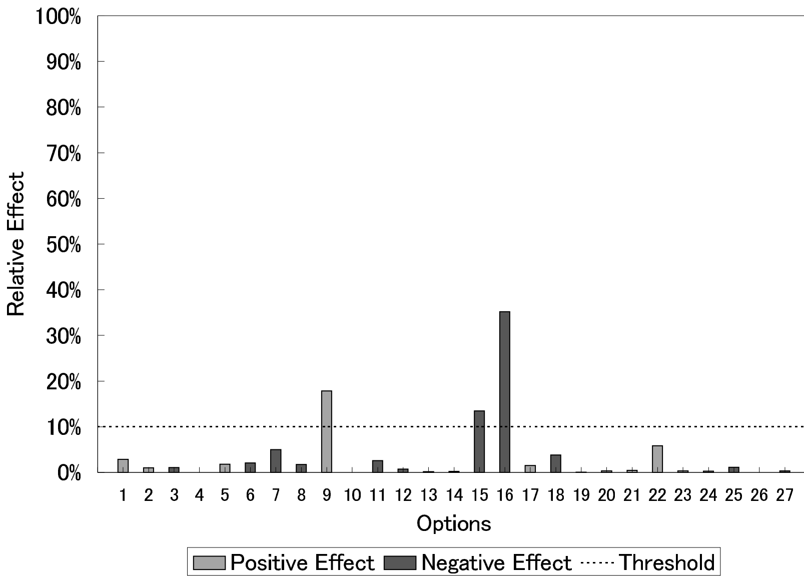


(b) Main Effects for 27 Factors

Figure 3.2: 1st Iteration of parser (Main Effect OA28)

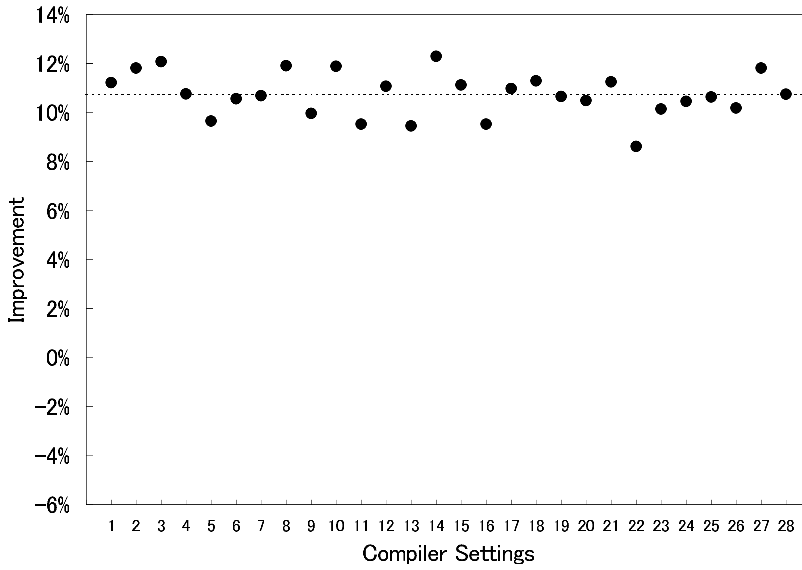


(a) Improvements for 28 Settings

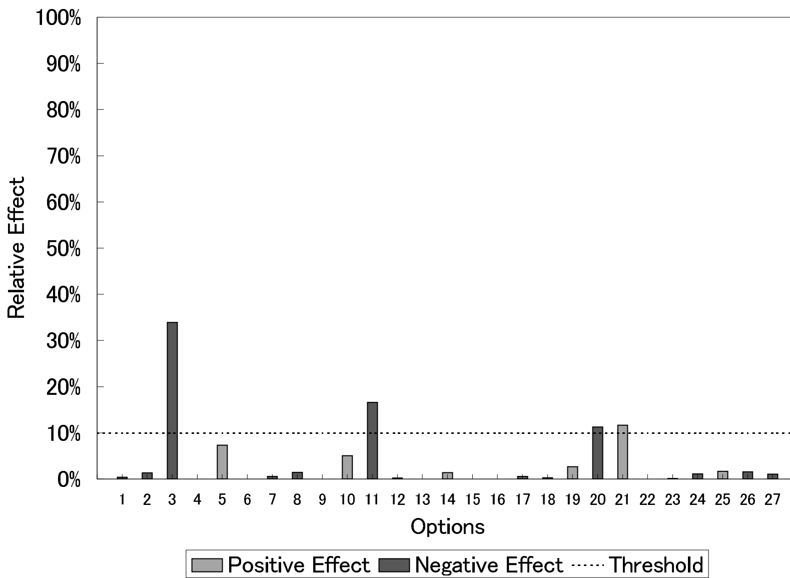


(b) Main Effects for the Remaining Factors

Figure 3.3: 2nd Iteration of parser (Main Effect OA28)



(a) Improvements for 28 Settings



(b) Main Effects for the Remaining Factors

Figure 3.4: 3rd Iterations of parser (Main Effect OA28)

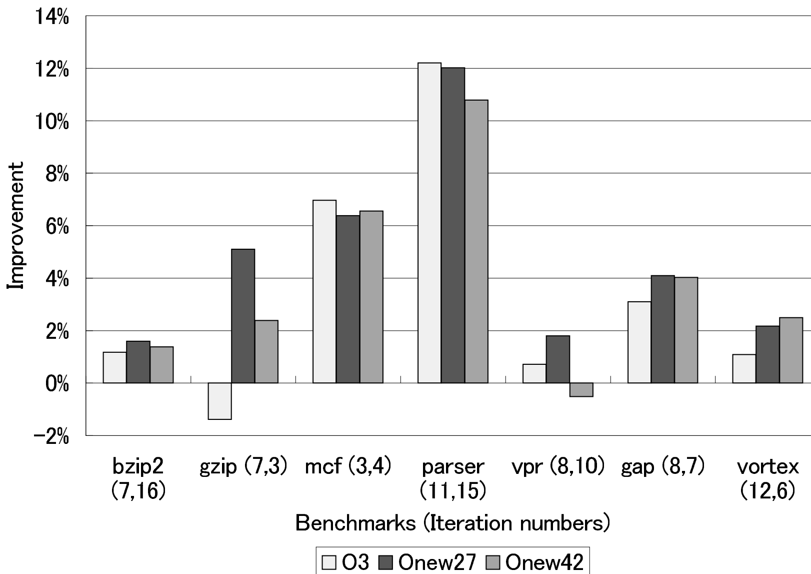


Figure 3.5: Improvement of the -O3 Setting, O_{new27}, and O_{new42} (Main Effect)

settings. These mis-selections occurred because the main effect, which determines the setting of options, becomes too small after several iterations. Since the main effect of an option is computed without taking into account the setting of the other options, the main effect is not computed correctly because of the side effect of the other options. This means that a small main effect should not be used to determine the setting of options.

3.5 Sensitivity of the algorithm to the threshold value

The threshold value 10% was determined by our observation in [53] which uses *gcc 2.6*. Therefore, it should be reconsidered for this experiment using *gcc 3.3.1* since the number of factors is increased from 15 to 27 or 42. However, it may be that the threshold value of 10% is still reasonable. We look at this possibility by focusing on *vortex* since for this benchmark the performance difference between the O_{new27} setting and the O_{max} setting is large, as shown in Figure 3.8.

Figure 3.6 shows the relative effect after the first iteration. From this figure, we can conclude that 8% also acts as a good threshold value. After we apply a threshold value of 8%, the results change as shown in Figure 3.7. The figure shows the improvement of the selected options at each iteration for each of the two threshold values. This observation implies that a threshold of 8% is a better choice to identify a compiler setting of *vortex*. The resulting setting is shown in Table 3.4.

This example shows that our approach using the main effect can be more beneficial when we can find a good threshold value to apply. However, it is difficult to formulate how to

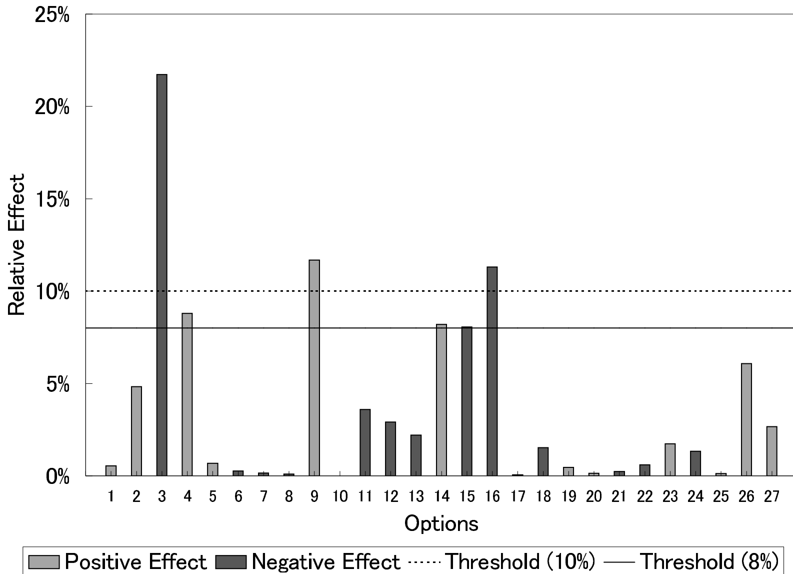


Figure 3.6: Relative Effect from the Experimental Data of the First Iteration

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
8%	0	1	0	1	0	0	0	1	1	0	1	0	0	1	0	0		1	0	1	0	0	0	1	0	1	0
10%	1	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	1

Table 3.4: Settings for vortex Using Threshold 8% and 10%

determine such a threshold value.

3.6 Conclusion

In this chapter, we described our algorithm using the main effect of compiler option to determine compiler settings. We have introduced the framework of Design of Experiments to configure compiler optimization settings. An experiment which is planned by an orthogonal array enables us to compute main effects of compiler options without collecting profile data of all possible compiler settings. The experimental results are better than the standard setting *-O3* provided by *gcc*, from which we conclude that our algorithm is adequate to solve the problem. This methodology is completely separated from the implementation of compiler so that we can apply this methodology to any compiler with any target architecture and any application.

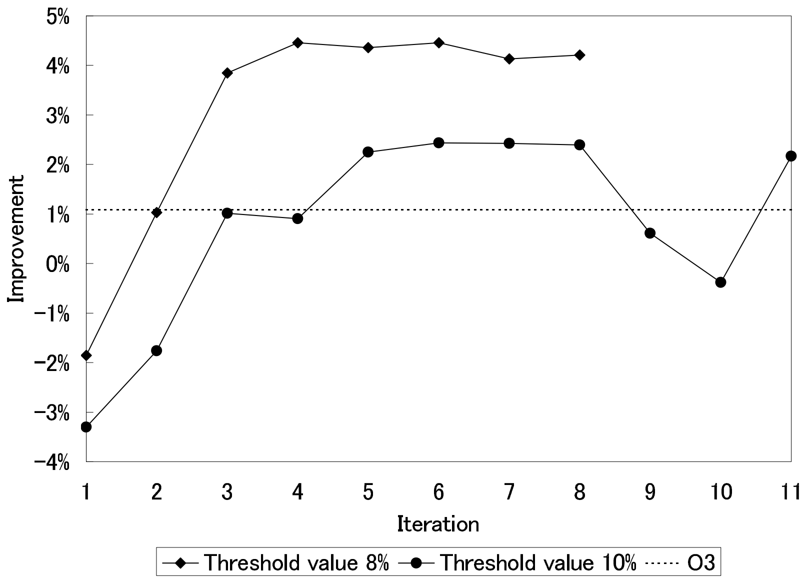
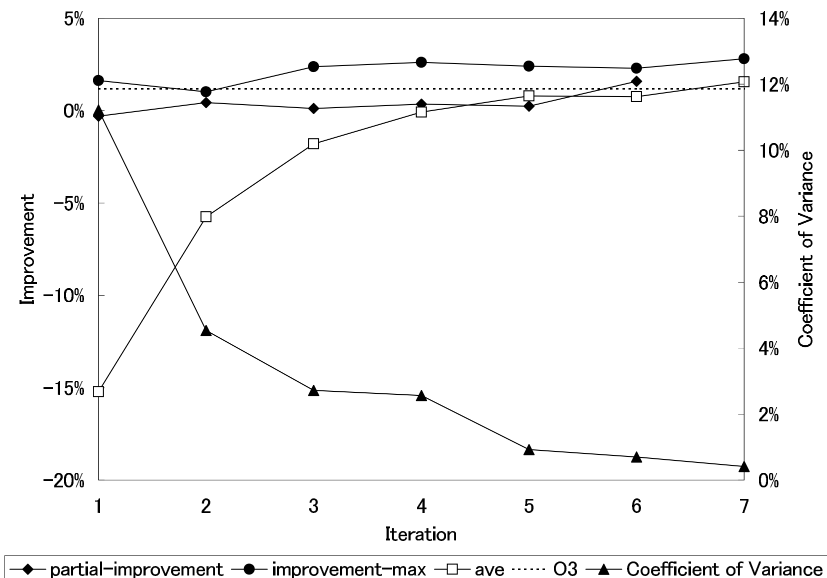
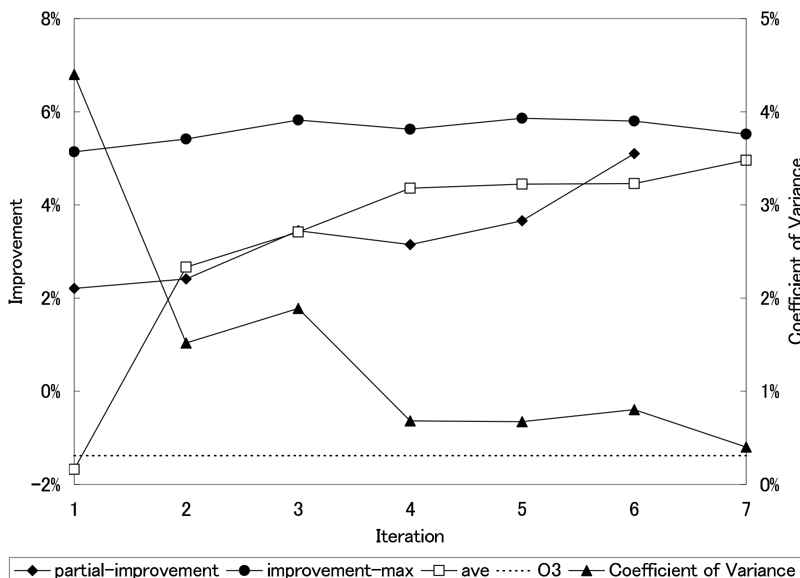


Figure 3.7: Improvement of the Resulted Settings Using Threshold 8% and 10%

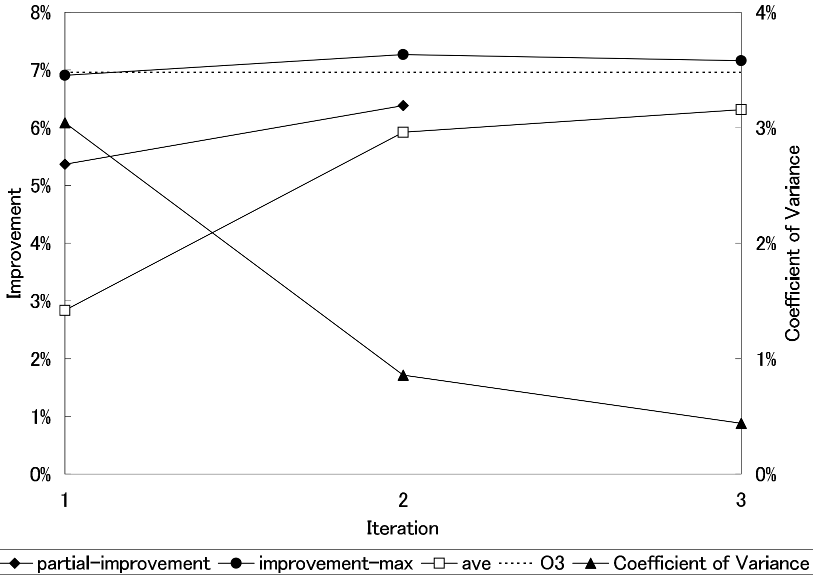


(a)bzip2

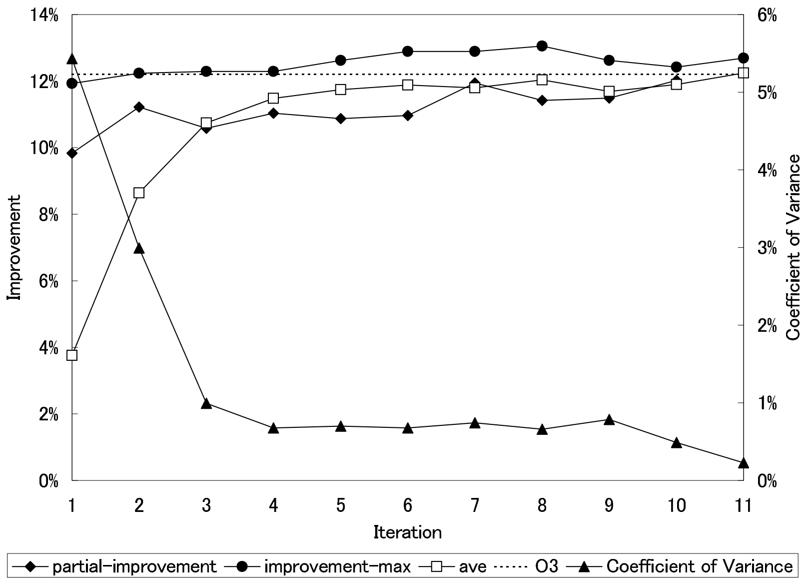


(b)gzip

Figure 3.8: Improvement per Iteration for 27 Options (Main Effect)

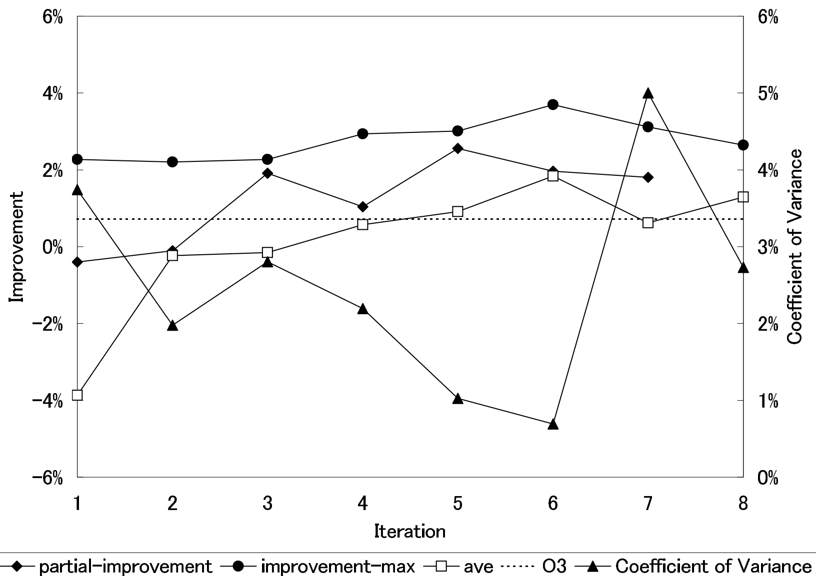


(c)mcf

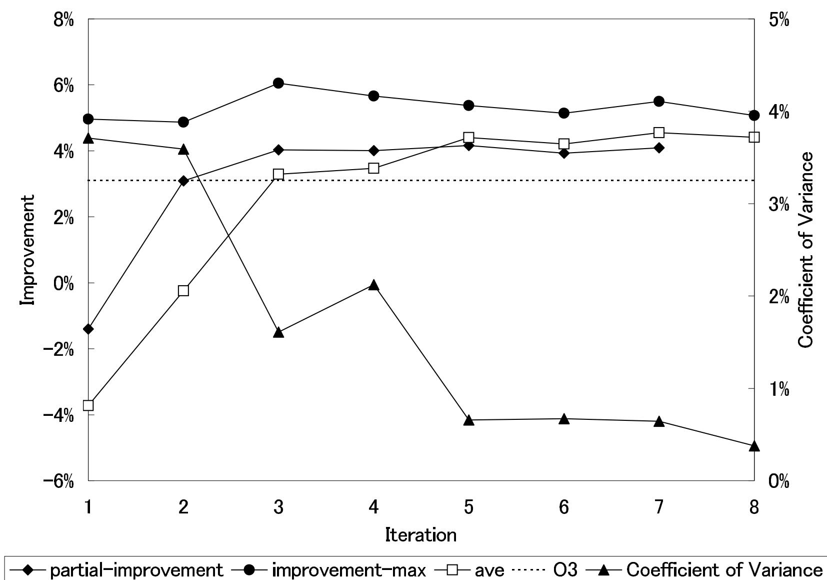


(d)parser

Figure 3.8: Improvement per Iteration for 27 Options (Main Effect) (cont'd)

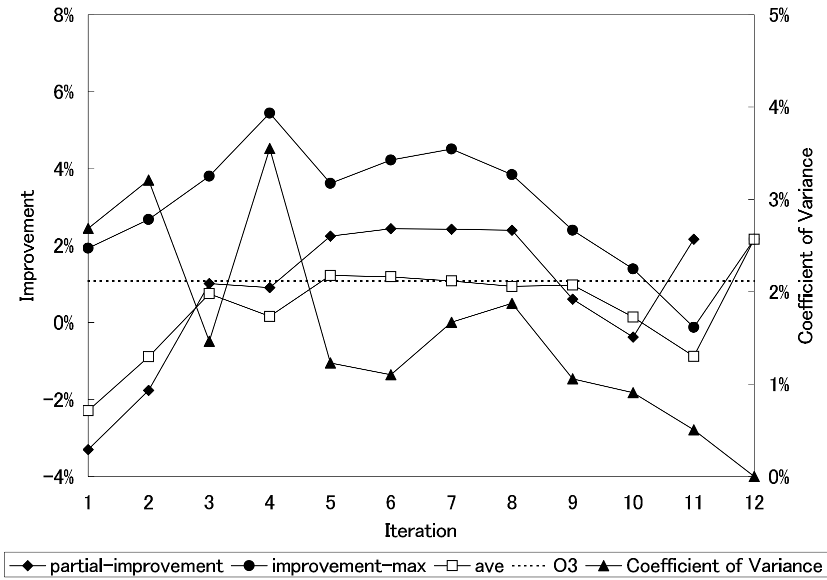


(e)vpr



(f)gap

Figure 3.8: Improvement per Iteration for 27 Options (Main Effect) (cont'd)



(g)vortex

Figure 3.8: Improvement per Iteration for 27 Options (Main Effect) (cont'd)

bzip2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			0											0													
2			0											0		0											
3			0								0			0		0											
4			0							0	0			0		0			0								
5			0					0		0	0	1		0		0			0							0	
6		0	0	1				0	1	0	0	1		0		0			0							0	
O_{max}	0	0	0	1	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	1	1	0	0

gzip	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			0													0										1	
2			0						0	0						0										1	
3		1	0						0	0						0		0								1	
4		1	0	1			0	1	0	0			0			0		0								1	
5		1	0	1		1	0	1	0	0			0			0		0						1		1	
6		1	0	1	1	1	0	1	0	0	1		0	0		0		0		1				1		1	
O_{max}	1	1	0	1	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0	0	1	0

mcf	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				1																							
2				1					1										0	1							
O_{max}	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0

parser	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				1																							
2				1					1						0	0											
3			0	1					1		0				0	0				0	1						
4		1	0	1				1	1		0				0	0		1		0	1						
5		1	0	1			1	1	1		0				0	0		1		0	1						
6		1	0	1		1	1	1	1		0				0	0		1		0	1						
7		1	0	1	1	1	1	1	1		0			1	0	0		1		0	1						
8		1	0	1	1	1	1	1	1		0	1		1	0	0		1		0	1	0					0
9		1	0	1	1	1	1	1	1		0	1	0	1	0	0		1		0	1	0			0	0	0
10	0	1	0	1	1	1	1	1	1	1	0	1	0	1	0	0		1		0	1	0			0	0	0
Omax	1	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1	1	1	0	1	1	1	0	1	0	1

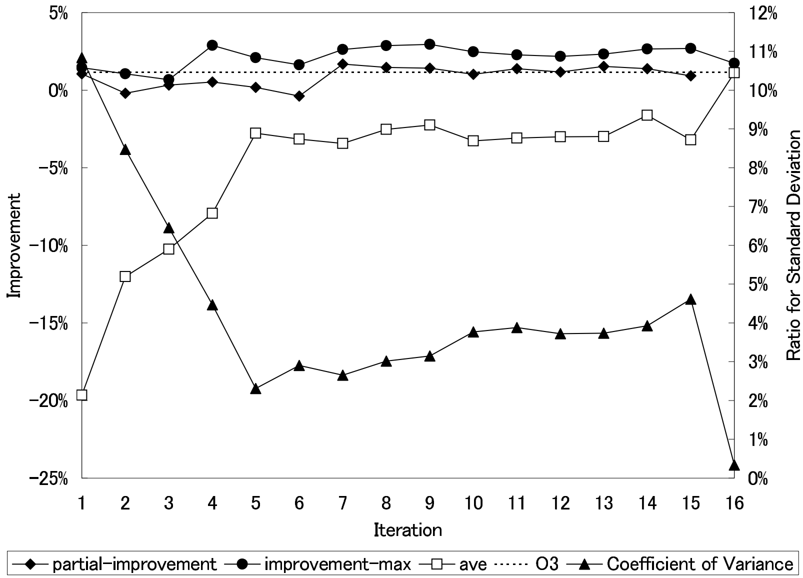
vpr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			0						1																		
2			0						1							0	1								1		
3			0						1		0			1		0	1								1		
4			0						1		0			1		0	1								1	1	0
5		1	0		0	0			1		0	0		1		0	1								1	1	0
6		1	0		0	0			1		0	0	1	1		0	1				0			0	1	1	0
7	1	1	0		0	0			1		0	0	1	1	1	0	1	1	0		0			0	1	1	0
O_{max}	0	1	0	1	0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	0

gap	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			0																			1					
2			0						1							0						1					
3			0						1							0	1		0			1	0				
4			0	1		0			1		0					0	1		0	0		1	0				
5	0		0	1		0	1		1		0					0	1		0	0	1	1	0			1	
6	0	1	0	1		0	1		1		0	0	1			0	1		0	0	1	1	0	0	1	1	
7	0	1	0	1		0	1		1	0	0	0	1	1	1	0	1	1	0	0	1	1	0	0	1	1	
O_{max}	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	0	1	0	1	1	1	0

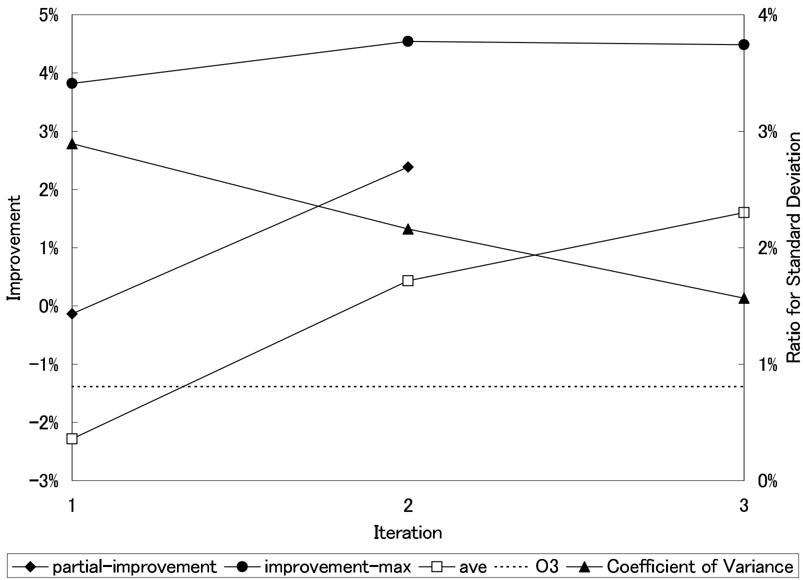
vortex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			0						1							0											
2			0	1					1							0											
3		1	0	1			1		1		0					0											
4		1	0	1			1		1		0					0			1								
5		1	0	1			1		1		0			1		0			1	0							
6		1	0	1	0		1		1	0	0			1		0		0	1	0							
7		1	0	1	0		1		1	0	0		0	1	0	0	0	0	1	0							
8		1	0	1	0	0	1	0	1	0	0		0	1	0	0	0	0	1	0	0						
9		1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0		1	0	0		
10	1	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	1	0	0		1
11	1	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	1
O_{max}	1	1	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0	1	1	0

O3	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0

Table 3.5: Settings per Iteration for 27 Options(Main Effect)

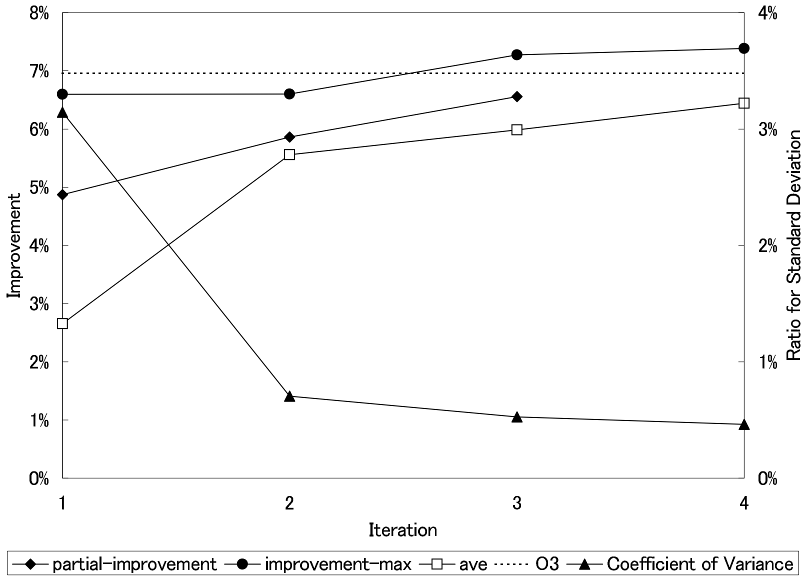


(a)bzip2

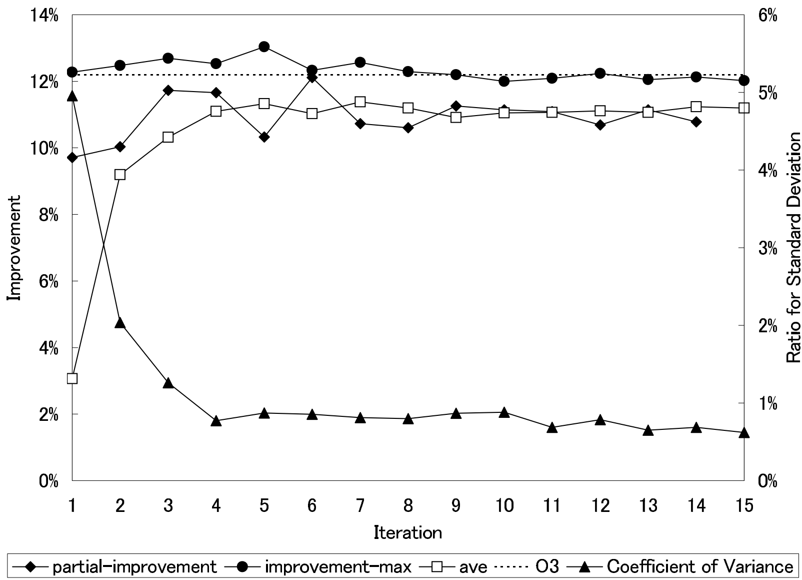


(b)gzip

Figure 3.9: Improvement per Iteration for 42 Options (Main Effect)

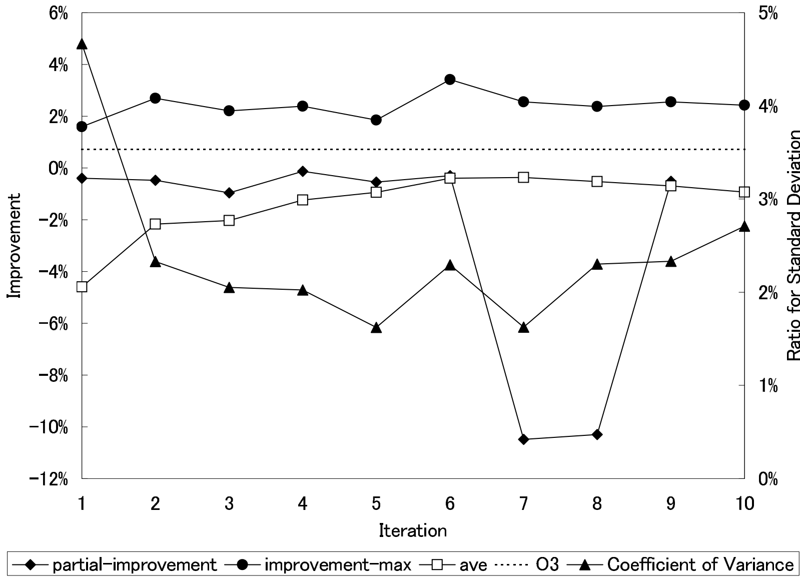


(c)mcf

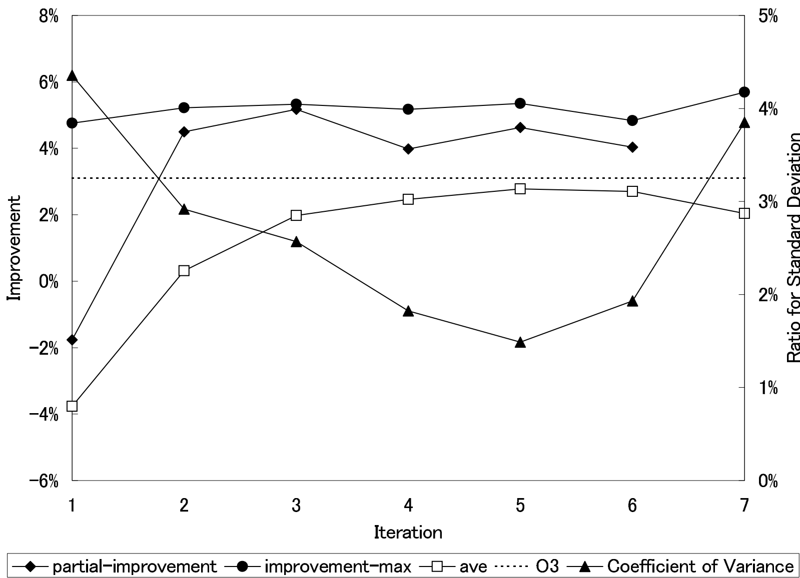


(d)parser

Figure 3.9: Improvement per Iteration for 42 Options (Main Effect) (cont'd)

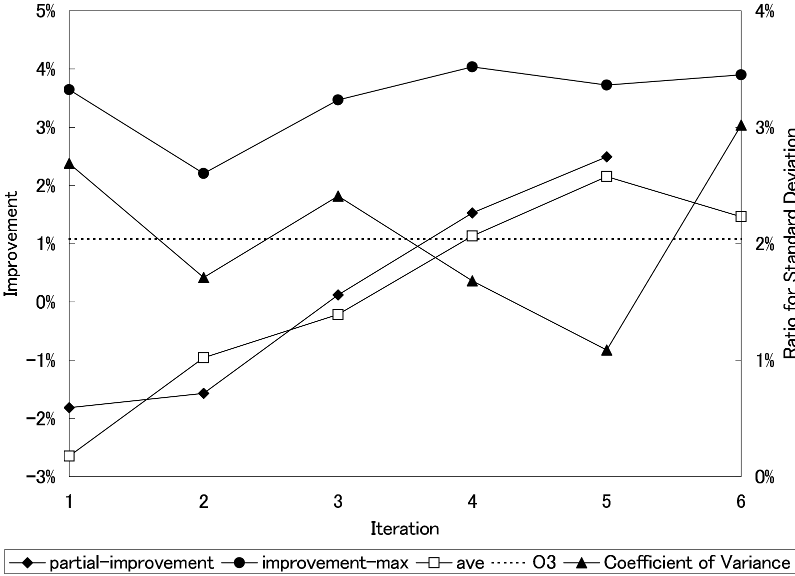


(e)vpr



(f)gap

Figure 3.9: Improvement per Iteration for 42 Options (Main Effect)(cont'd)



(g)vortex

Figure 3.9: Improvement per Iteration for 42 Options (Main Effect) (cont'd)

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
bzip2	1			0											1							
	2			0											1		0					
	3			0											1		0			0		
	4			0		1									1	0	0			0		
	5			0		1					0				1	0	0			0	0	
	6			0		1					0				1	0	0		0	0	0	
	7			0		1					0	1			1	0	0		0	0	0	0
	8			0	0	1					0	1			1	0	0	0	0	0	0	0
	9			0	0	1					0	1	0	0	1	0	0	0	0	0	0	0
	10	0		0	0	1				0	0	1	0	0	1	0	0	0	0	0	0	0
	11	0		0	0	1			0	0	0	1	0	0	1	0	0	0	0	0	0	0
	12	0		0	0	1		0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
	13	0		0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
	14	0		0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
	15	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
	O_{new27}		0	0	1				0	1	1	1	0	1	0	0	1	1		0		0
	O_{max}	0	0	0	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0
gzip	1					1																
	2					1					1			1								
	O_{new27}		1	0	1	1	1	0	1	0	0	0	0	0	0	1			0	0		0
	O_{max}	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0
mcf	1					1																
	2					1					1			1								
	3					1					1	1		1								
	O_{new27}				1					1	1	1		1								
	O_{max}	1	0	1	0	1	1	0	0	1	1	1	0	1	0	0	1	1	1	0	0	1

		22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
bzip2	1																					
	2			1							1											
	3			1							1											
	4			1							1		1									
	5		0	1			1				1		1									
	6	0	0	1			1				1	0	1						1			
	7	0	0	1			1				1	0	1					0	1			
	8	0	0	1			1		0		1	0	1					0	1			
	9	0	0	1			1		0		1	0	1					0	1			
	10	0	0	1			1		0		1	0	1					0	1			
	11	0	0	1			1		0		1	0	1					0	1			
	12	0	0	1			1		0		1	0	1					0	1	1		
	13	0	0	1		0	1		0		1	0	1				0	0	1	1		
	14	0	0	1		0	1	0	0		1	0	1			0	0	0	1	1		
	15	0	0	1	0	0	1	0	0	0	1	0	1		0	0	0	0	1	1		1
	O_{new27}	0	0	0	0	0			0											0	0	
	O_{max}	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	1	0	1	0	1	1
gzip	1																				1	
	2																				1	
	O_{new27}	0	0	0	0	0		0		1								1		1	1	
	O_{max}	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0	1	0	0	1	1
mcf	1																					
	2																					
	3							0														
	O_{new27}								0	1												
	O_{max}	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1	1	0	0	0	1	1

Table 3.6: Settings per Iteration for 42 Options (Main Effect)

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
parser	1					1															0	
	2					1										0				1	0	
	3					1								1		0				1	0	
	4			0		1								1		0				1	0	
	5			0		1								1		0				1	0	
	6		1	0		1								1	1	0				1	0	
	7		1	0		1	0							1	1	0				1	0	
	8		1	0	0	1	0							1	1	0				1	0	
	9		1	0	0	1	0							1	1	0				1	0	
	10		1	0	0	1	0							1	1	0				1	0	1
	11		1	0	0	1	0			0				1	1	0				1	0	1
	12		1	0	0	1	0			0				1	1	0			1	1	0	1
	13		1	0	0	1	0			0				1	1	0		1	1	1	0	1
	14		1	0	0	1	0			0				1	1	0	1	1	1	1	0	1
	O_{new27}	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0
	O_{max}	1	1	0	0	1	1	1	1	1	1	0	0	1	0	0	0	1	0	1	0	0
vpr	1			0																		
	2			0															0			
	3			0						1									0			
	4			0						1									0			
	5			0	0					1						0			0			
	6			0	0					1				0		0			0			
	7		1	0	0					1				0		0			0		0	
	8		1	0	0					1				0		0			0		0	
	9		1	0	0		1		1	1				0		0			0		0	1
	O_{new27}	1	1	0		0	0			1	1	1		1		0	0	0	1	1	1	0
	O_{max}	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	1	0	1

		22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
parser	1																					
	2		0																			
	3		0									1										
	4		0									1										
	5		0							0	1	1										
	6		0						1	0	1	1										
	7		0			1			1	0	1	1										
	8		0	1		1			1	0	1	1										
	9	1	0	1		1			1	0	1	1										
	10	1	0	1		1			1	0	1	1										
	11	1	0	1		1		0	1	0	1	1										
	12	1	0	1		1		0	1	0	1	1										
	13	1	0	1		1		0	1	0	1	1		0								
	14	1	0	1		1		0	1	0	1	1		0							1	
	O_{new27}	0	0	0	0	0		1		0	1	1	0						0	0	0	0
	O_{max}	0	0	1	0	0	1	0	1	0	0	1	1	0	0	0	1	0	1	1	1	0
vpr	1												1									
	2				0								1									
	3				0								1									
	4				0		1						1									
	5				0		1						1		1							
	6				0		1						1		1							
	7				0		1						1		1			1				
	8		1		0		1						1		1			1			1	
	9	1	1		0		1						1		1			1			1	
	O_{new27}	0	0	0	0	0	1	1	0		0							0	1	1	1	0
	O_{max}	0	0	1	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1	1	0

Table 3.6: Settings per Iteration for 42 Options (Main Effect) (cont'd)

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
gap	1			0																		
	2			0						1	1			1								
	3			0		1				1	1			1								
	4			0		1		0		1	1			1								
	5			0	1	1		0		1	1			1		0						
	6			0	1	1		0		1	1			1		0			0			
	O_{new27}	0	1	0	1		0	1		1	1	1	0	1	0	0	0	0	1	1	1	0
	O_{max}	0	1	0	1	1	0	0	0	1	1	0	1	1	1	0	1	1	0	1	0	0
vortex	1			0																		
	2			0										0								
	3			0		1					1			0								
	4		1	0		1					1			0						1		
	5		1	0		1					1			0		1				1		
	O_{new27}	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	0	0	0	1	0	0
	O_{max}	1	0	0	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	1	0	0
O3		1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

		22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
gap	1												1									
	2												1									
	3												1									
	4									0			1									
	5			0						0			1									
	6			0						0			1									
	O_{new27}	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	
	O_{max}	1	0	0	0	0	0	0	1	0	1	1	1	1	1	0	1	1	0	1	0	1
vortex	1																					
	2				0																	
	3				0			0														
	4	1			0			0														
	5	1		0	0			0														
	O_{new27}	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	1
	O_{max}	1	0	1	0	0	0	0	1	1	0	1	1	0	1	0	1	1	0	0	0	1
O3		1	1	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0

Table 3.6: Settings per Iteration for 42 Options (Main Effect) (cont'd)

Chapter 4

Using the Mann-Whitney test to optimize the execution time of a single application

This chapter proposes another methodology to determine a compiler setting automatically. We again use profile information to determine a compiler setting, since only real execution times can accurately describe the interaction of compiler options, architectures, and applications. In Chapter 3, we used the so-called *main effect* of options to determine their setting. However, subsequent experimentation showed that this approach has a number of shortcomings. In this chapter, we employ a more sophisticated statistical technique, namely, *inferential non-parametric statistics* and, in particular, the *Mann-Whitney test*, to predict the significance of the effect of factors in an experiment [35]. We apply our methodology again to the GNU Compiler Collection (gcc) version *3.3.1* and 10 benchmarks from the *SPEC2000* benchmark suite. For each benchmark, the setting found by our methodology performs better than the standard *-O1*, *-O2*, and *-O3* settings of *gcc*.

This chapter is structured as follows. Our iterative algorithm is given in Section 4.1. Section 4.2 describes our experimental environment and we discuss our results in Section 4.3. Section 4.4 investigates the effectiveness of compiler settings across different kinds of input datasets. We summarize this chapter in Section 4.5.

4.1 Our methodology

Figure 4.1 describes our algorithm to determine a compiler setting for an application based on the statistical theory discussed in the previous section.

The algorithm tries to detect compiler options with a significant effect. It starts with an factor list which includes all compiler options, and produces a compiler setting using an appropriate Orthogonal Array. We use execution times to obtain the test statistic for each compiler option, and this tells us which options have a significant effect, and whether they

- Choose an orthogonal array A with as many columns as there are options.
- Repeat
  - Compile the application with each row from A as compiler setting and execute the optimized application.
  - Compute test statistic z for each compiler option with equation (2.7).
  - If the test statistic meets $P(|z|) < 5\%$,
    - If z is negative then the compiler option has a positive effect, and the option is turned on.
    - If z is positive then the compiler option has a negative effect, and the option is turned off.
  - Remove the compiler options that have been selected from the factor list and drop the same number of columns from A.
- Until
  - All options are set, or
  - No option with a significant effect is detected anymore, or
  - The experimental data has not enough variation (low standard deviation) to apply the Mann-Whitney test meaningfully.
- Choose the compiler setting which has the best execution time in the last experiment.

Figure 4.1: Iterative Search Algorithm Using the Mann-Whitney Test

should be turned on or off. The compiler options whose settings are determined are removed from the factor list, and the reduced factor list is used to design the next experiment in which a smaller Orthogonal Array can be used. The algorithm starts to explore a large search space in which there is much variation, and it cuts down the search space every iteration, obtaining new search spaces with less variation.

4.2 The experimental environment

We use the gcc compiler version 3.3.1, and we use 45 options for our factor list [13]. *inline* is not used since this option disables the inline directive in the applications and we do not intend to change the contexts of the applications. *keep-static-consts* and *function-cse* are not included since these options are for the sake of keeping the assembler code readable. *branch-count-reg* is left out since this option disables the use of instructions regarding the branch count registers when it is negated. *prefetch-loop-arrays* is not used since this option disables the use of prefetch instructions. *float-store* and *single-precision-constant* are also left out to

avoid generating invalid executables. We do not use *rename-registers* because, according to the manual, it contains bugs in the present version of the compiler. The option *delayed-branch* is also not present since our target architecture Pentium 4 does not support delayed branching. As discussed in Section 2.3.2, we want to use 23 factors in order to achieve a minimum power of 60%. Hence, we arranged these 45 options into the 23 factors shown in Table 4.1(a). In several cases, we have heuristically assigned different but similar options to one factor, based on the description in the manual of the compiler [13]. For instance, we group all 6 options that enable some form of common subexpression elimination together in factor O_7. We found that some options do produce (almost) no changes to the compiled code and we grouped them together in O_{21}. Note, however, that in some cases interfering options are assigned to different factors. For example, option *unroll-loops* automatically turns on options *strength-reduce* and *rerun-cse-after-loop* [13]. Nevertheless, we use separate factors for these options. As the results below show, our method can handle this situation.

We use 10 benchmarks from the *SPEC2000* benchmark suite. The list of programs is shown in Table 4.1(b). We used the train data set as the input for the benchmarks in order to reduce profiling time.

We use a Pentium 4 at 2.8GHz as the target architecture for our experiments. We use VTune, a tool to analyze performance of applications, to measure execution times [37]. Clock ticks obtained from hardware counters are used as the execution time.

When we measure the execution time of an application several times with the same compiler configuration, we may obtain different results. In order to cope with this error in measurement, we measure the execution times for an executable ten times, and compute the mean and standard deviation. For about one in 20 cases, measured execution time is an order of magnitude larger or smaller than average execution time. We could not discover what causes this phenomenon but it is necessary to omit these values. We remove data points to obtain a standard deviation that is less than 0.5% of the average of the data. We have defined this threshold value heuristically. This procedure has been automated.

If we have K settings given by the rows of an Orthogonal Array, we also compute the standard deviation and mean for these K settings. If this standard deviation is less than a certain percentage of their mean value, there is not enough variation in the data to try to apply the Mann-Whitney test. This threshold is called the *variability threshold*. We have experimented with threshold values of 0.5%, 0.7%, 1%, and 2%. This is the last stop criterion in our algorithm in Section 4.1.

4.3 Results

In this section, we show the results obtained by our method. We show the final improvement and compiler setting for all benchmarks, for three sizes of Orthogonal Arrays, comparing them to the standard settings *-O1*, *-O2* and *-O3*. All improvements are relative to *-O0*. We discuss which options are selected per iteration. Finally, we discuss the influence of the variability threshold defined in Section 4.2 on the number of iterations executed.

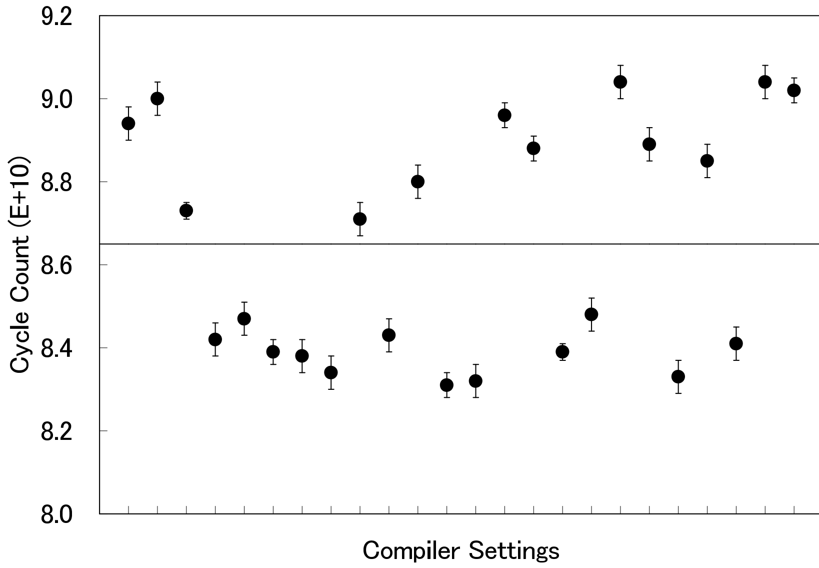
Factor	Option Names	Factor	Option Names
O_1	force-mem	O_{14}	schedule-insns
	force-addr		schedule-insns2
O_2	omit-frame-pointer		sched-interblock
O_3	optimize-sibling-calls		sched-spec
	inline-functions		sched-spec-load
O_4	merge-constants		sched-spec-load-dangerous
O_5	strength-reduce	O_{15}	move-all-movables
O_6	thread-jumps	O_{16}	reduce-all-givs
O_7	cse-follow-jumps	O_{17}	peephole
	cse-skip-blocks		peephole2
	rerun-cse-after-loop	O_{18}	reorder-blocks
	gcse		reorder-functions
	gcse-lm	O_{19}	strict-aliasing
	gcse-sm	O_{20}	align-functions
O_8	loop-optimize		align-labels
	rerun-loop-opt		align-loops
O_9	crossjumping		align-jumps
O_{10}	if-conversion	O_{21}	cprop-registers
	if-conversion2		caller-saves
O_{11}	delete-null-pointer-checks		defer-pop
O_{12}	expensive-optimizations	O_{22}	function-sections
O_{13}	optimize-register-move		data-sections
		O_{23}	unroll-loop

(a) Compiler options

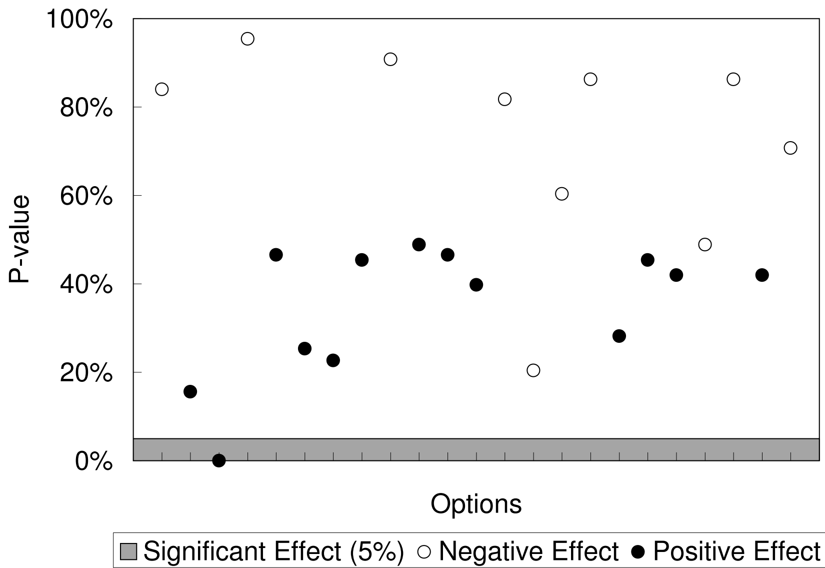
Name (#Lines)	Description
Integer	
164.gzip (4333)	gzip (GNU zip) is a data compression program. gzip uses Lempel-Ziv coding (LZ77) as its compression algorithm.
175.vpr (8899)	VPR is a placement and routing program for technology-mapped circuit.
181.mcf (1120)	The program is designed for the solution of single-depot vehicle scheduling problems occurring in the planning process of public transportation companies.
197.parser (6839)	The Link Grammar Parser is a syntactic parser of English.
254.gap (27523)	It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
255.vortex (31128)	Vortex is a single-user object-oriented database transaction benchmark.
256.bzip2 (2955)	256.bzip2 is a data compression program.
Floating point	
168.wupwise (1382)	”wupwise” is an acronym for ”Wuppertal Wilson Fermion Solver”, a program in the area of lattice gauge theory.
171.swim (326)	Benchmark weather prediction program for comparing the performance of current supercomputers.
179.art (776)	The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image.

(b) Benchmarks

Table 4.1: Experimental Framework: Option List and Benchmark Programs

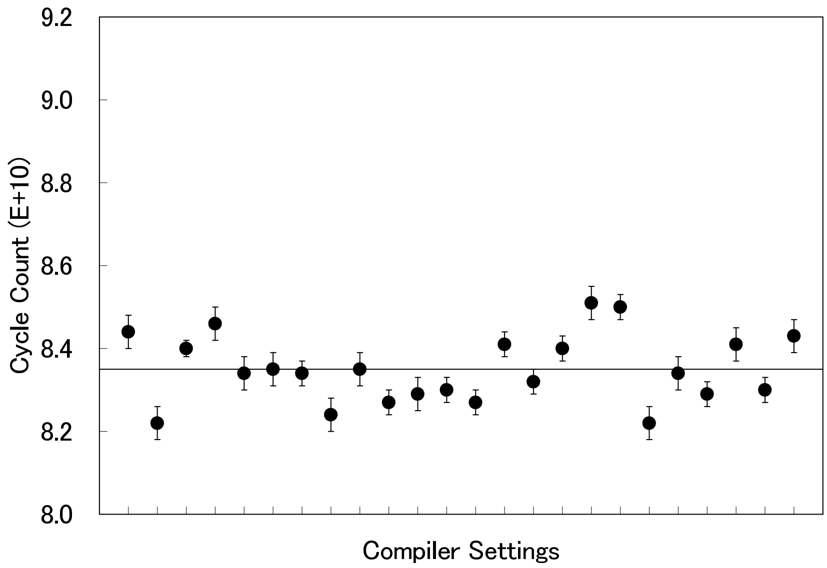


(a) Improvements for 24 Settings

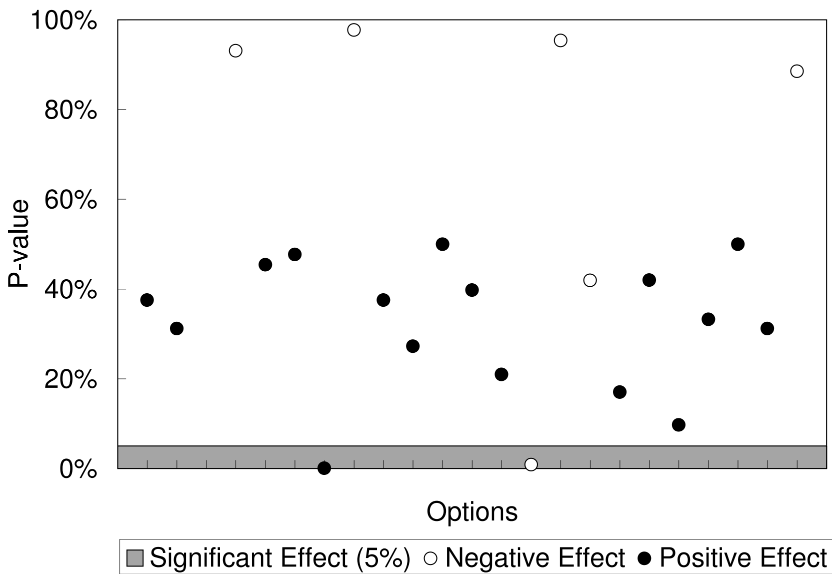


(b) P values for 23 Factors

Figure 4.2: 1st Iteration of mcf



(a) Improvements for 24 Settings



(b) P values for the Remaining Factors

Figure 4.3: 2nd Iterations of mcf

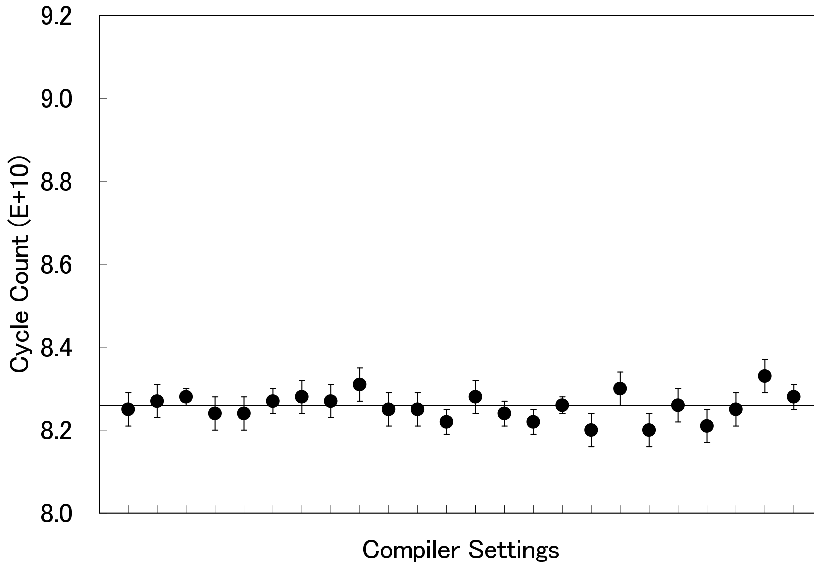


Figure 4.4: Improvements for 24 settings at 3rd Iteration of mcf

	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}
1st			1									
2nd			1				1					
3rd	0	0	1	1	1	0	1	0	1	1	0	1

	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}	
1st												
2nd		0										
3rd	0	0	1	1	1	1	1	0	0	0	0	

Table 4.2: Resulting setting

4.3.1 Example: the iterations for mcf

We illustrate the performance of our procedure with the results per iteration for *mcf*. We use the factor list given in Table 4.1(a), and an orthogonal array which consists of 24 rows with 23 columns is used for the experimental design. We compile the application according to the rows of the orthogonal array, and measure the execution time.

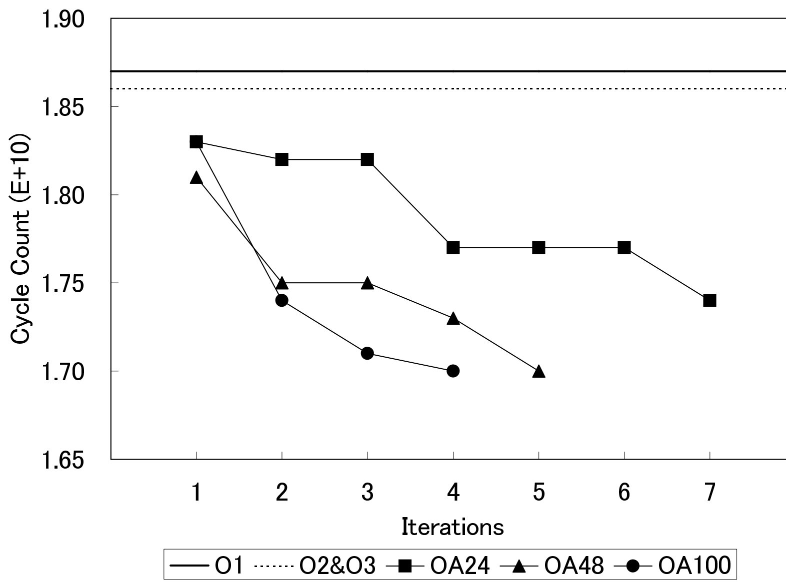
Constructing a compiler setting for *mcf* took three iterations. Figures 4.2(a), 4.3(a), and 4.4 show the resulting execution times together with the uncertainty in the measurement. The solid line in these figures shows the overall mean of the 23 settings. Figures 4.2(b) and 4.3(b) show the *P* values from Equation (2.9) for each factor for the first two iterations. Black points are used when the effect is positive and white points are used to show a negative effect. In Figure 4.2(b) we can see that O_3 has a *P* value which is below 5%. The effect is positive and the algorithm selects O_3 in this stage. From Figure 4.3(b) it follows that O_7 must be selected since it has positive effect, and O_{14} is explicitly turned off since it has negative effect. The third iteration is the last iteration, since the variation in the experimental data is too small to apply the Mann-Whitney test: all values converge around $8.26e+10$. According to the algorithm shown in Section 4.1, the compiler setting with the best result in this iteration is chosen. Table 4.2 shows the compiler options which are determined in each iteration where 1 and 0 represent ‘on’ and ‘off’, respectively.

4.3.2 Settings for SPEC2000

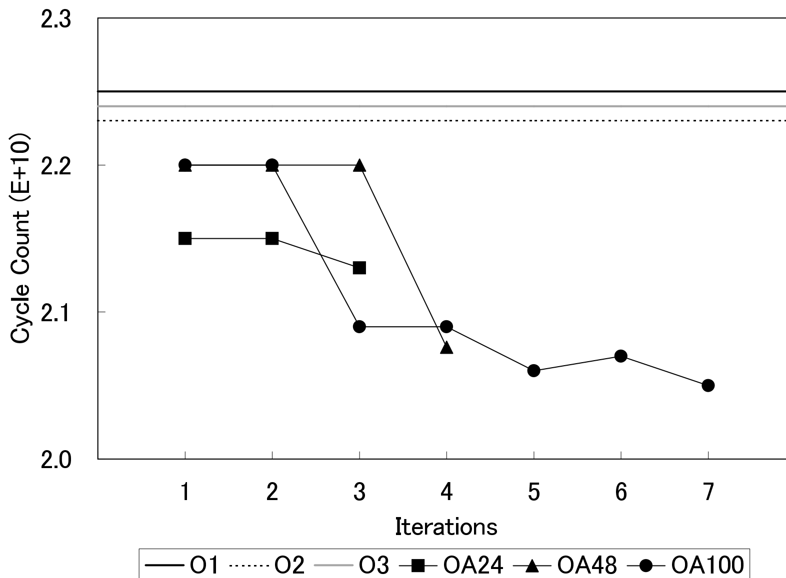
In Figure 4.5, we show how our methodology performs after each iteration for the benchmarks *vortex* and *vpr*. We observe that larger Orthogonal Arrays require fewer iterations for *vortex*, but the reverse holds for *vpr*. Therefore, we cannot in general say that larger arrays require fewer iterations although for most benchmarks this is the case. Note however that less iterations for a large array does not necessarily imply less program runs. We measure performance after an iteration by setting the options that are already determined accordingly, and switching off the other options. Observe that the performance we report after some iterations is the same as earlier. In this case, an option is explicitly turned off in this iteration. This does not mean that we did not do relevant work in this iteration: the variability of the resulting reduced space is reduced.

In Figure 4.6(a) and (b) we show the results for the SPEC2000 benchmarks using *gcc 3.3.1* on the P4 platform. We see that in all cases, we outperform standard *-Ox* settings, for all three sizes of the Orthogonal Array used to profile the different compiler settings. In some cases (*gzip, gap, vortex*) we are significantly better than standard switches. Note in particular *art* where *-O2* and *-O3* give degradations of more than 80%. This clearly shows that relying on standard switches is not always profitable.

In Table 4.3 we show the options that are set by the Mann-Whitney test for each iteration. Note that the row OA 48 for *art* is empty, indicating that already in the first iteration so little variation is seen that the Mann-Whitney test is not used at all. In this table, the options that are set by *-Ox* switches are shown for comparison. Note that *-O2* and *-O3* seem to set the same options. This, however, is caused by our grouping of options into factors. We now give some observations that we have made. The first is that many options are not yet decided at the last iteration but one, as can be seen from Table 4.3. This means that these options have a very small impact on program performance. Options that are not switched on for most

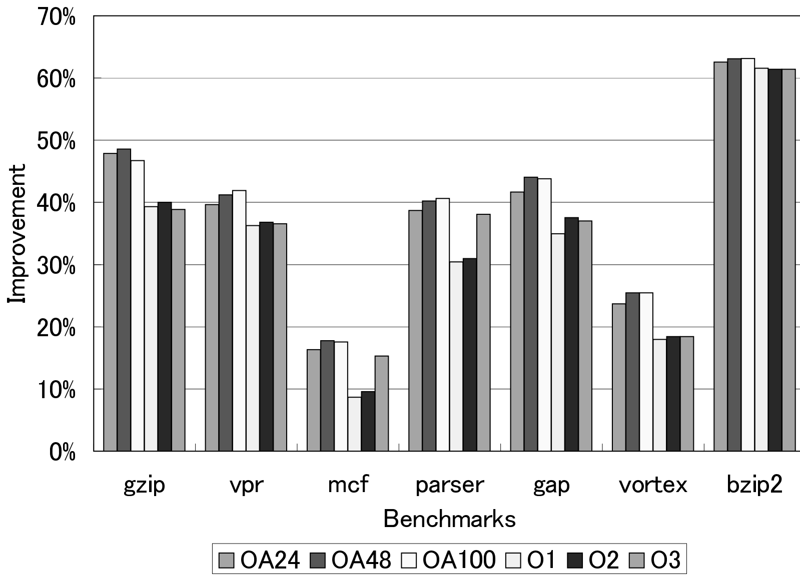


(a) vortex

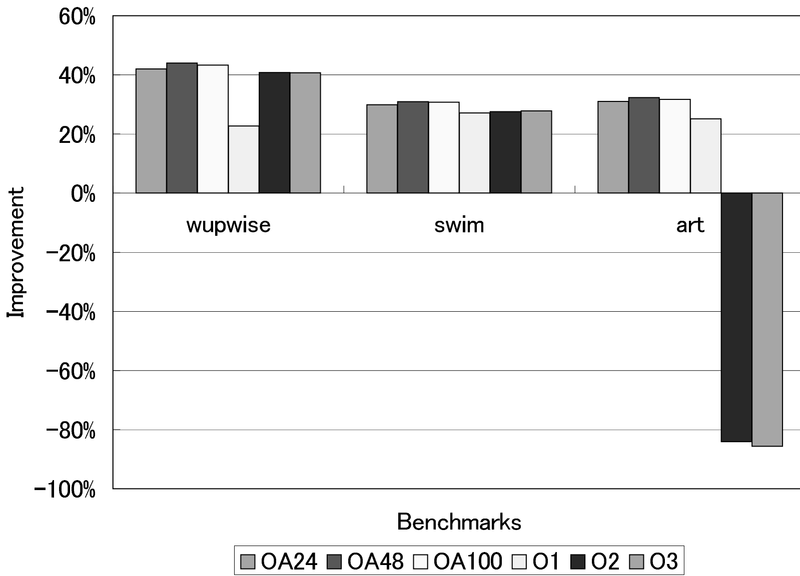


(b) vpr

Figure 4.5: Execution times after each iteration



(a) SPECint



(b) SPECfp

Figure 4.6: Results for (a) SPECint2000 and (b) SPECfp2000

		O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}
art	OA 24																			0				
	OA 48																							
	OA 100																			0				
bzip2	OA 24	0																						
		0	1														0							
		0	1						1								0						1	
		0	1						1						0		0						1	
		0	1					0	1						0		0						1	
		0	1	1				0	1						0		0						1	0
	OA 48	0	1																					
		0	1					1									0							
		0	1					1									0						1	
		0	1					1									0			1			1	
		0	1					1									0			1			1	0
		0	1					1	1								0			1			1	0
	OA 100	0	1					1															1	
		0	1					1									0						1	0
		0	1					1	1				1				0			1			1	0
gap	OA 24	0	1																					
		0	1					1																
		0	1					1												1				
	OA 48	0	1					1												1				
		0	1					1											0	1	0	0		
		0	1					1							0		0		0	1	0	0		
		0	1	1				1							0		0	1	0	1	0	0		
		0	1	1				1	1				0		0		0	1	0	1	0	0		
		0	1	1				1	1		1		0		0		0	1	0	1	0	0	1	
	OA 100	0	1					1												1				
		0	1					1	0						0			1	0	1	0	0		
		0	1	1				1	0	0					0			1	0	1	0	0		
		0	1	1				1	0	0	0		0		0			1	0	1	0	0		

Table 4.3: Final selection of options by Mann-Whitney test

		O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}
gzip	OA 24		1																					
			1						0														1	0
		0	1						0														1	0
		0	1	1					0														1	0
	OA 48	0	1																					
		0	1						0														1	0
		0	1	1				1	0					1	0								1	0
	OA 100		1																					
		0	1												0								1	0
mcf	OA 24			1																				
				1				1							0									
	OA 48			1																				
				1				1							0									
	OA 100		1	1				1							0									
			1	1				1							0		0				0	0		
parser	OA 24			1																				
			1	1																				
		0	1	1				1									0							
	OA 48		1	1																				
		0	1	1				1																
		0	1	1				1											0					
		0	1	1			1	1									0		0	1				
		0	1	1			1	1		1							0		0	1				
		0	1	1			1	1		1							0		0	1	0	0		0
	OA 100	0	1	1																				
		0	1	1			1	1									0			1				
		0	1	1			1	1									0	1	0	1				
		0	1	1			1	1		1	1				0		0	1	0	1				0

Table 4.4: Final selection of options by Mann-Whitney test(cont'd)

		O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}
swim	OA 24								1						0									
									1						0									1
							1		1						0			0						1
							1		1						0			0	1					1
		1					1		1						0			0	1					1
		1					1		1						0		1	0	1					1
		1					1		1	0					0		1	0	1					1
	OA 48								1						0									
									1						0									1
								0	1						0									1
								0	1						0		1							1
								0	1						0	1	1							1
								0	1				1		0	1	1							1
	OA 100								1						0									
		1				1		0	1			0			0									1
		1				1		0	1			0			0		1							1
vortex	OA 24		1																					
			1	1																				
			1	1				1											0					
			1	1				1										1	0					
			1	1				1					1					1	0					
			1	1		1		1					1					1	0					
	OA 48	0	1																					
		0	1	1														1						
		0	1	1			1								0			1		0				
		0	1	1			1	1						0	0			1	0	0				
	OA 100	0	1					1																
		0	1	1			1	1										1						
		0	1	1			1	1					1		0			1	0					
vpr	OA 24	0	1																					
		0	1				0																	
	OA 48	0																						
		0	1																	0				
		0	1	0										0						0				
	OA 100	0	1																					
		0	1	0										0						0				0
		0	1	0				1		1				0						0				0
		0	1	0		1		1	1	1				0						0			0	0
		0	1	0		1		1	1	1				0	1					0	0	0	0	0
		0	1	0		1		1	1	1				0	1				0	0	0	0	0	0

Table 4.5: Final selection of options by Mann-Whitney test (cont'd)

		O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}
wupwise	OA 24								1															
						1			1															
						1			1						0									0
			1			1		1	1						0									0
	OA 48								1															
						1			1															
			1			1			1						0	1								0
			1			1		1	1				1		0	1								0
		1	1			1		1	1				1		0	1	1							0
		1	1			1		1	1				1		0	1	1		1					0
		1	1			1		1	1				1	0	0	1	1		1					0
		1	1			1		1	1	0		0	1	0	0	1	1		1					0
		1	1			1		1	1	0		0	1	0	0	1	1	1	1		0	0		0
		1	1			1		1	1	0		0	1	0	0	1	1	1	1		0	0		0
	OA 100								1															
					0	1			1							1								
			1		0	1		1	1						0	1								0
			1		0	1		1	1				1	1	0	1								0
		1	1		0	1		1	1				1	1	0	1	1							0
		1	1	1	0	1		1	1				1	1	0	1	1							0
		1	1	1	0	1		1	1	0			1	1	0	1	1	1	0					0
		1	1	1	0	1		1	1	0			1	1	0	1	1	1	0		1	1		0

Table 4.6: Final selection of options by Mann-Whitney test (cont'd)

benchmarks are O_4, O_{10}, O_{11}, O_{13}, O_{15}, O_{20}, O_{21}, and O_{22}. This fact clearly shows that it can pay off to be critical about which optimizations to implement in a compiler. Second, for almost all benchmarks, the most important options that are switched on or off using a small OA are the same as the options switched on or off using a large OA. This means that our method is quite robust. Next, several optimizations are explicitly turned off already early in the search procedure. This means that they have a statistically significant negative impact on performance. For instance, O_{16} is turned off for many benchmarks. Also, option O_1 is explicitly turned off for several benchmarks and only turned on for *swim* and *wupwise*. This observation may be more serious since this option is turned on in *-O2* and *-O3*. Next, it is remarkable that the options O_{14} (scheduling) and O_{23} (loop unrolling) are mostly turned off explicitly. These options are generally considered to be instrumental for superscalar processors. The probable reason to turn them off in our experiments is that in our target architecture, very few registers are available, implying that loop unrolling can increase register pressure to unacceptable levels and that there are too few registers to perform meaningful scheduling. This probably means that the performance that could be obtained using the actual pipeline in the P4 is not realized due to limitations of the ISA supported. Next, the improvement found for an OA with 48 rows (a power of 85%) is equally good as the improvement found for an OA with 100 rows (a power of 99.5%). Finally, we see that for all benchmarks, except for the fp programs *art* and *swim*, the option *omit-frame-pointer* is turned on. This option is not present in the standard *-Ox* switches, presumably because it prevents symbolic debugging of the resulting code. Since our approach clearly is intended for debugged production codes, we have included this option. Experiments with *-Ox* with *omit-frame-pointer* have shown that this option increases the performance of the standard switches by a few percent and *-O3* then comes close to our results in several cases. However, also in these cases we still obtain higher improvements.

4.3.3 Number of iterations

In this section, we discuss the number of iterations required. Obviously, the variability threshold defined in Section 4.2 has a strong influence on this number. In Table 4.7 we show the number of iterations for different values of this threshold for each benchmark and size of the Orthogonal Array. The resulting improvements of the benchmarks decrease slightly with higher threshold values and for values larger than 2% this decrease becomes quite noticeable. However, in several cases, the resulting improvement is the same as for a threshold of 0.5%. These results suggest that the variability threshold provides a natural trade-off between optimization time and improvement. If the variability in improvement between the different settings is low, we can simply generate 48 settings using an Orthogonal Array and pick the best one from this set. If, on the other hand, this variability is high, we use our method to set a number of important switches which can cause the remaining space to have low variability. If the variability of this remaining space is low enough, we again pick the best setting from it. This approach is consistent with results reported in [40] where it has been shown that a random search strategy produces good results using few profiles.

	OA 24				OA 48				OA 100			
	0.5%	0.7%	1%	2%	0.5%	0.7%	1%	2%	0.5%	0.7%	1%	2%
art	2	2	2	2	1	1	1	1	2	2	2	2
bzip2	7	5	3	1	7	4	4	2	4	3	3	1
gap	4	4	4	4	7	6	6	6	5	5	5	5
gzip	5	5	5	4	4	4	4	2	3	3	3	3
mcf	3	3	2	2	3	3	2	1	3	2	2	2
parser	4	4	4	2	7	5	3	2	5	4	2	2
swim	8	8	3	2	7	7	4	2	4	4	4	2
vortex	7	7	7	4	5	5	5	5	4	4	4	4
vpr	3	3	3	3	4	4	4	4	7	7	6	3
wupwise	5	5	5	4	11	11	7	4	9	9	6	4
Average	4.9	4.7	3.9	2.8	5.6	5	3.6	2.8	4.6	4.3	3.7	2.8

Table 4.7: Influence of variability threshold on number of iterations

4.4 The robustness of the methodology

A possible drawback of the profile guided search for an appropriate compiler setting is that applications are optimized using a single or a limited set of data inputs. It is well known that programs can exhibit vastly differing behaviors for different inputs. Therefore, it is not clear whether the performance numbers reported are still valid for other input than the input used to optimize the program. In this section, we address this problem for a specific statistical compiler tuning method. We use three different platforms and several SPECint2000 benchmarks. We show that when we tune the compiler using train data, we obtain a compiler setting that still performs well for reference data. These results suggest that profile guided optimization may be more stable than is sometimes believed and that a limited number of train data sets is sufficient to obtain a well optimized program for all inputs.

In this section, we use the same experimental environment in Section 3.2. We employ the set of 42 optimizations for this experiment. Additionally, we use two more target architectures for our experiments: a SPARC Sun Fire V25 server dual 1.28 GHz processor and a IA64 dual Itanium2 1.296 GHz processor. Firstly, we show the results we obtained using our methodology on three different platforms when we select compiler options using train data. Next, we show the improvements we obtain when we run the resulting optimized program on reference data.

In Figures 4.7 through 4.9 we show the improvements for `-O3` and the setting obtained from the Mann-Whitney test, denoted by O_{new42}, for three different platforms. We have used 7 SPEC2000 benchmarks for the P4 and 5 for the IA64 and SPARC since the two other benchmarks did not compile correctly on these platforms. We have used the train data to run our methodology and we show improvements when using train data again. The numbers in brackets after the benchmark name denote the number of iterations required to fully run the selection method.

We immediately observe that for the P4 and the SPARC, significantly better improvements are obtained than `-O3`. The situation is different for the IA64 where our method actually reduces the improvements for `bzip2`. For the other benchmarks, the improvements that

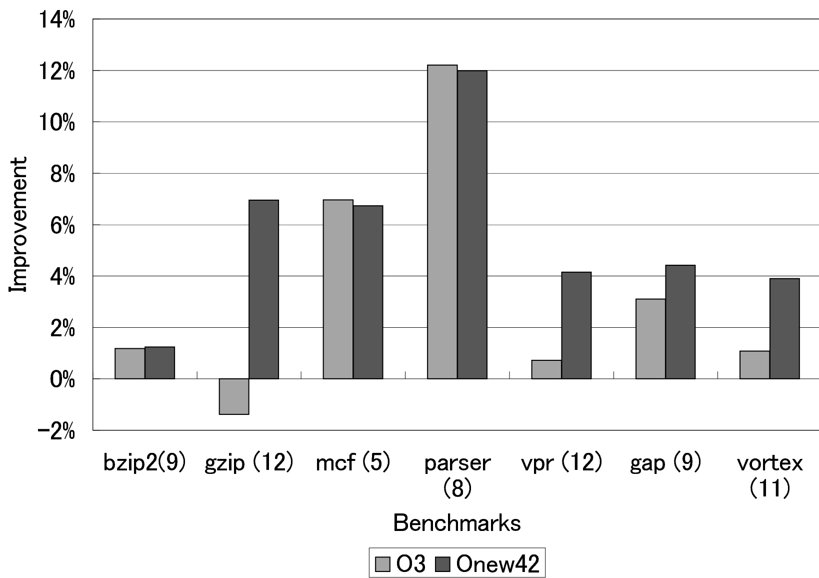


Figure 4.7: Improvements of -O3 and O_{new42} for P4 using train data

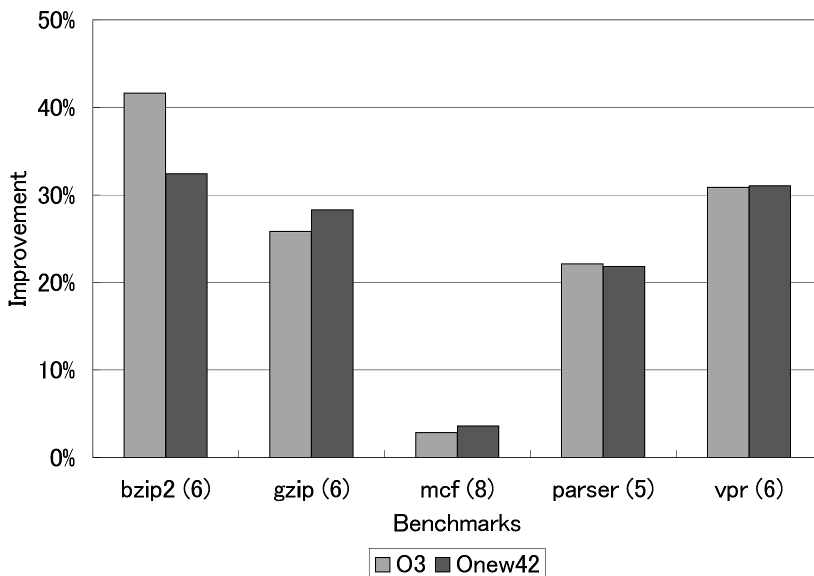


Figure 4.8: Improvements of -O3 and O_{new42} for IA64 using train data

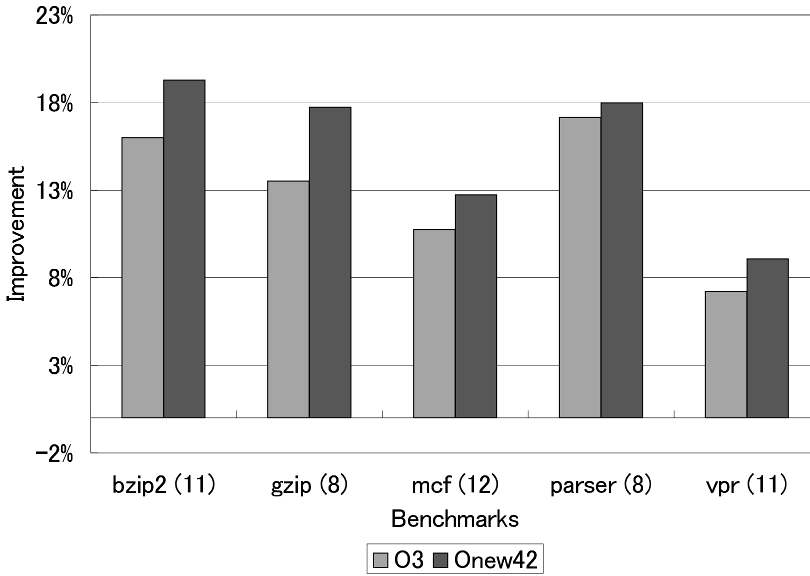


Figure 4.9: Improvements of -O3 and O_{new42} for SPARC using train data

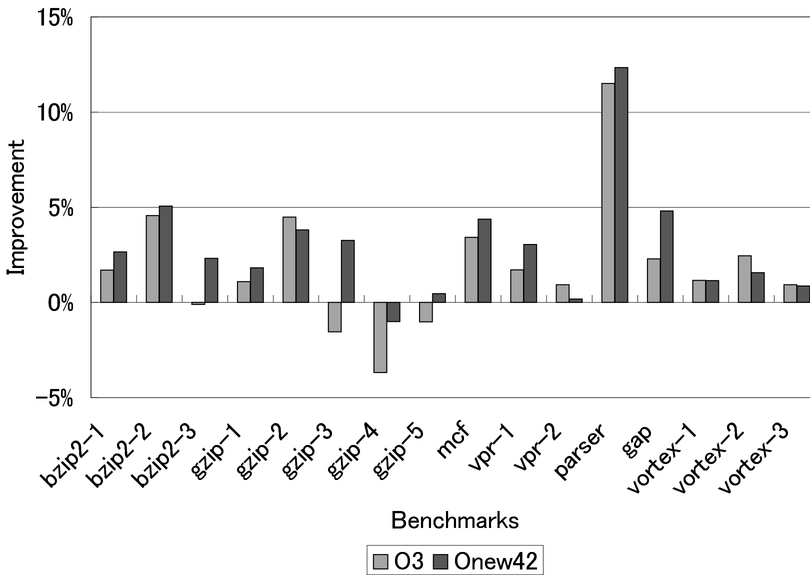


Figure 4.10: Improvements of -O3 and O_{new42} for P4 using reference data

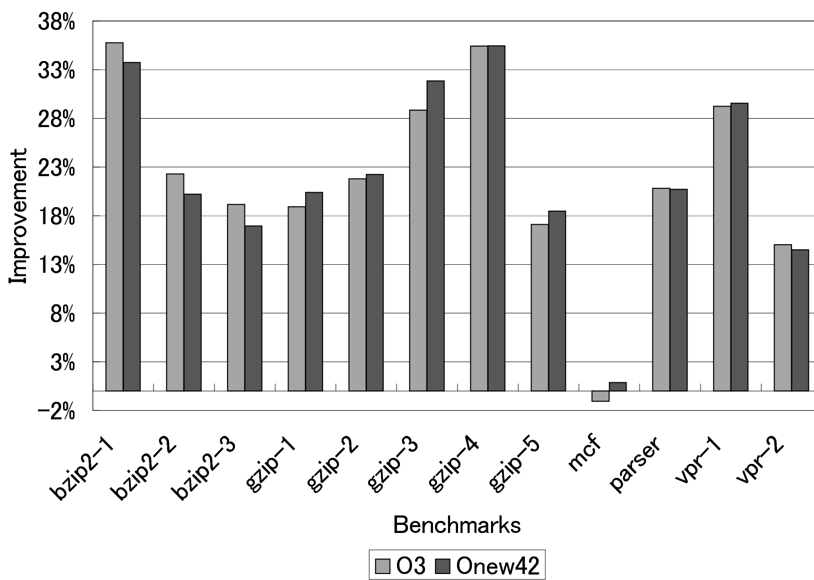


Figure 4.11: Improvements of -O3 and O_{new42} for IA64 using reference data

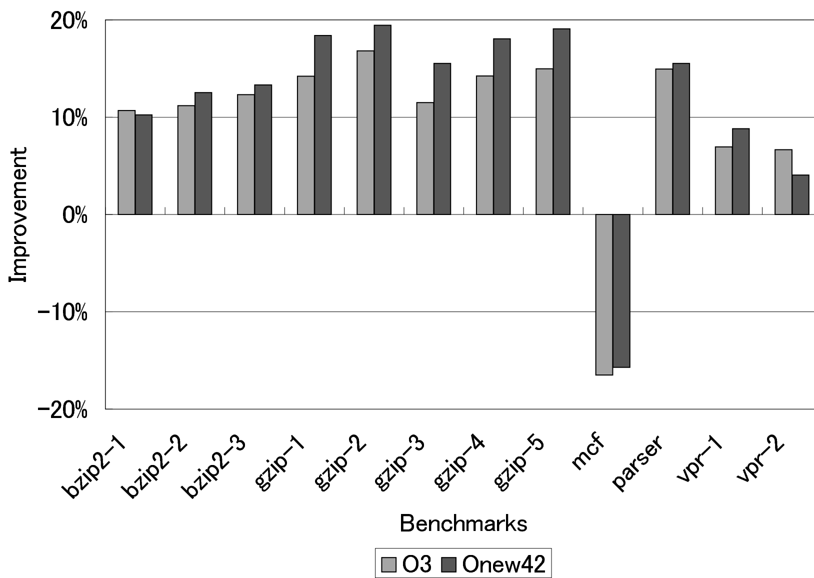


Figure 4.12: Improvements of -O3 and O_{new42} for SPARC using reference data

we found are almost equal to the improvements obtained by `-O3`. A possible explanation for this last observation is that almost all improvement comes from correct instruction scheduling for this EPIC architecture and that little extra improvement can be found using other options. These results show that our methodology can be effective for other platforms.

In Figures 4.10 through 4.12 we show the improvements when we run our optimized program on the reference data. Since the benchmarks have several distinct reference data sets, that should be used together, we show the results when running the optimized programs on all reference data sets. We observe that for the P4 and the SPARC, generally better performance is obtained using our new setting. For the IA64, the improvements we obtain are more or less equal to the improvements of `-O3`, much in line with the previous results. Comparing these figures with the previous ones, we immediately observe that the performance improvements are much in par. In fact, for the P4, the improvements for `mcf` and `parser` are actually better than for the train data. However, for `vortex`, the situation is reverse. For the IA64, the improvements for `bzip2` for the train data are also less than for the reference data. Finally, on the SPARC, `bzip2` improvements are slightly better for train data than for reference data. However, for the vast majority of our benchmarks, performance improvements for the train data (which is used to select the compiler options) is more or less the same as for the reference data.

4.5 Conclusion

In this chapter, we have introduced an approach to the problem of selecting a compiler optimization setting using inferential non-parametric statistics, in particular, the Mann-Whitney test. The statistical analysis reveals which compiler options have a significant effect. We use the options with significant positive effect and we explicitly turn off the options with significant negative effect. Our algorithm uses an iterative approach to the detection of compiler options with large positive effect. It can be used for any compiler to determine a setting. Our results suggest that a power of the Mann-Whitney test of 85%, corresponding to a size of the search space of 48, is sufficient to obtain the best results. Using a threshold on the variability of the search space of 2% entails that we need less than 3 iterations on average.

We have shown effectiveness of our approach in other plat forms, SPARC and IA64. Besides one benchmark for the IA64, the resulting settings have better or equal performance compared to standard switches which in some cases may cause a degradation in performance. We can also conclude that the dependence of the SPECint2000 benchmarks on data input is not as strong as is sometimes believed. Profile guided compiler tuning using one (representative) data input set produces compiler settings that work well for other data input sets. This result is of importance to other approaches that use profile guided search to optimize programs or library routines. However, a more thorough investigation of this dependence is in place, in particular for high level loop transformations that may exhibit a stronger dependence. Nevertheless, the results in this section show that we may be optimistic and that profile guided compiler searches may need only a limited set of representative inputs to produce good results for many or even all inputs.

Chapter 5

Using the Mann-Whitney test to optimize the code size of a single application

Memory is a main cost factor for many high volume electronic devices and constitutes an increasing portion of the total product cost. Code size reduction therefore may reduce the direct cost of a product by reducing the size of required memory. On the other hand, a reduction in code size can also be used to fit more features into the same ROM which may enhance the value of a product. Many approaches have been proposed to reduce the code size of an application, ranging from code compression by means of, e.g., Huffman coding, to specific compiler based techniques like code factoring [7].

In this chapter, we approach the problem from a different perspective. Instead of proposing yet another technique that may reduce code size, we want to explore the possibilities standard compiler optimizations can offer to decrease the number of generated assembly instructions. This approach is orthogonal to the approaches mentioned above and can be used in conjunction with them, possibly leading to smaller compressed codes. Modern compilers implement many optimizations that often can explicitly be turned on or off using compiler flags or switches. For example, *gcc* 3.4.3 has over 50 switches. Obviously, some optimizations, like loop unrolling or procedure inlining, can increase code size. Others, like dead code removal or strength reduction, can decrease code size. While these statements seem obvious, some care needs to be taken since it has been shown [14] that procedure inlining can actually decrease code size in some cases. This trivially holds when functions with only one call site are inlined or when the body of a function is smaller than the code needed to call and return from that function. For example, in many cases, registers need to be saved before and restored after a function call which may require two instructions per register. Moreover, function inlining may enable other optimizations that reduce the number of instructions, like common subexpression elimination, that can now be applied across the old function boundaries. Therefore, it is clear that sweeping remarks on the effect of optimizations need not be generally true.

1	defer-pop
2	force-mem
3	force-addr
4	omit-frame-pointer
5	optimize-sibling-calls
6	inline-functions
7	merge-constants
8	strength-reduce
9	thread-jumps
10	cse-follow-jumps
11	cse-skip-blocks
12	rerun-cse-after-loop
13	rerun-loop-opt
14	gcse gcse-lm gcse-sm gcse-las
15	loop-optimize
16	crossjumping

17	if-conversion
18	if-conversion2
19	delete-null-pointer-checks
20	expensive-optimizations
21	optimize-register-move
22	schedule-insns
23	sched-interblock
24	sched-spec
25	schedule-insns2
26	sched-spec-load
27	sched-spec-load-dangerous
28	caller-saves
29	move-all-movables
30	reduce-all-givs
31	peephole peephole2
32	reorder-blocks
33	reorder-functions
34	strict-aliasing

35	align-functions
36	align-labels
37	align-loops
38	align-jumps
39	rename-registers
40	web
41	cprop-registers
42	tracer
43	unit-at-a-time
44	function-sections
45	data-sections
46	unroll-loops
47	peel-loops
48	unswitch-loops
49	old-unroll-loops
50	branch-target-load-optimize
51	branch-target-load-optimize2
52	delayed-branch
53	prefetch-loop-arrays

Table 5.1: Options from gcc 3.4.3 used

However, it is clear that all options in a compiler may change the generated assembly code and thus may have an effect on code size. Whether they increase or decrease code size is largely unknown as is their effect on code size if we take into consideration the interaction of options. To the best of our knowledge, besides our work, there exists no approach, which systematically investigates how an existing production compiler that has 53 options can be tuned in order to reduce code size. We show that we can obtain a reduction in the number of assembly instructions in the generated code that can be as high as 30% over the standard *-Os* option of *gcc* in which the optimizations are coordinated to achieve small code size.

Our method is based on statistical analysis of many versions of the code obtained by using different compiler settings. In Chapter 4, we propose an algorithm using the Mann-Whitney test to optimize for execution time. We apply the same algorithm to optimize for code size in this chapter.

In this chapter, we only optimize for code size. We do not take into consideration the speed of the resulting code. Therefore, we may end up with a code that is short but too slow to be useful. In future work, we plan to integrate our approaches to code size and speed optimization. A possible solution is to optimize for size under speed constraints. In this case, possible candidate settings need to be profiled in order to check that they do not run too slow. Conversely, we can optimize for speed under code size constraints. Finally, a third possibility is to optimize for both at the same time by using a suitable function of both speed and size improvement.

This chapter is structured as follows. In Section 5.1, we discuss our experimental environment. In Section 5.2, we present our results. Section 5.3 summarize this chapter.

5.1 The experimental environment

We have used *gcc* version 3.4.3 as our compiler. It contains more than 60 options and we chose a subset of 57 options that are not described as experimental in the manual nor are

options that may violate IEEE floating point standards, like *fast-math*. The resulting list of options is given in Table 5.1. *inline* is not used since this option disables the inline directive in the application and we do not want to change the contexts of original applications. *keep-static-consts* and *function-cse* are not included since these options are for the sake of keeping the assembler code readable. *branch-count-reg* is left out since this option disables the use of instructions regarding the branch count registers when it is negated. *float-store* and *single-precision-constant* are also left out to avoid generating invalid executables. Note that in some cases, we have grouped a few options into one factor since the gcc manual explicitly states that these options should be turned on together [13]. For example, in factor 14, we have grouped all global common subexpression elimination options since they are enabled by default when *gcse* is enabled. Note also, that in factors 26 and 27, we turn on instruction scheduling together with a speculative scheduling option because the gcc manual states that this speculative scheduling option needs instruction scheduling, but we also have instruction scheduling as a separate option (22).

We have configured *gcc* as a cross-compiler for the following platforms. We have used *mips* and *arm* which are two well known embedded RISC processors. We use the Motorola *m68k* which is a CISC processor that 25 years after its introduction still is a popular embedded processor. Current implementations are known as *68HC000*. We have also used two processors that are more specifically geared toward the embedded domain: the Vitesse *IQ2000* network processor and the Renesas *M32R* processor.

We use the MediaBench benchmark suite for our test programs. However, the benchmarks *pgp* and *gostscript* only compiled on the *mips* and *arm* platforms. Hence, we do not show results for these benchmarks for the other platforms.

5.2 Results

In this section, we show the results obtained from our iterative selection algorithm using the *gcc* 3.4.3 compiler and the MediaBench suite for five different platforms.

5.2.1 The optimization time requirements

We let the algorithm run until completion when no more options are selected. The number of iterations required is between 5 and 12, with an average of 8. This means that on average we require 432 program compilations. Concerning the time it took to complete our iterative method, we performed our experiments on a P4 at 2.8 GHz platform. Depending on the size of the source code for the benchmark, it took between 30 minutes and 2 hours to complete the iterative procedure. Hence, when developing embedded applications, this time is certainly affordable. Please, note also that our approach is essentially 'for free': all that is required to implement it is a small driver on top of the compiler that generates different settings, compiles the source code using these settings, and counts the number of instructions in the assembly code. No complex new transformations or other adaptations of the compiler are needed.

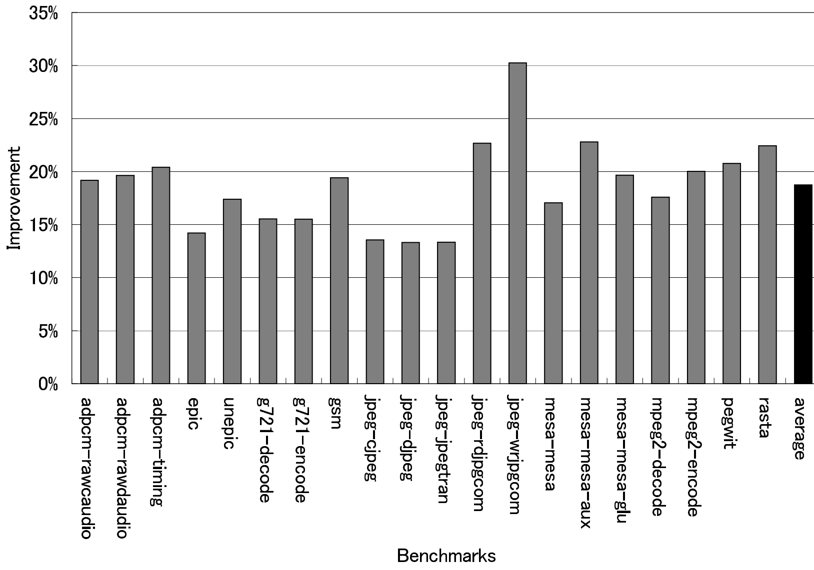


Figure 5.1: Code size reduction with respect to –Os for mips

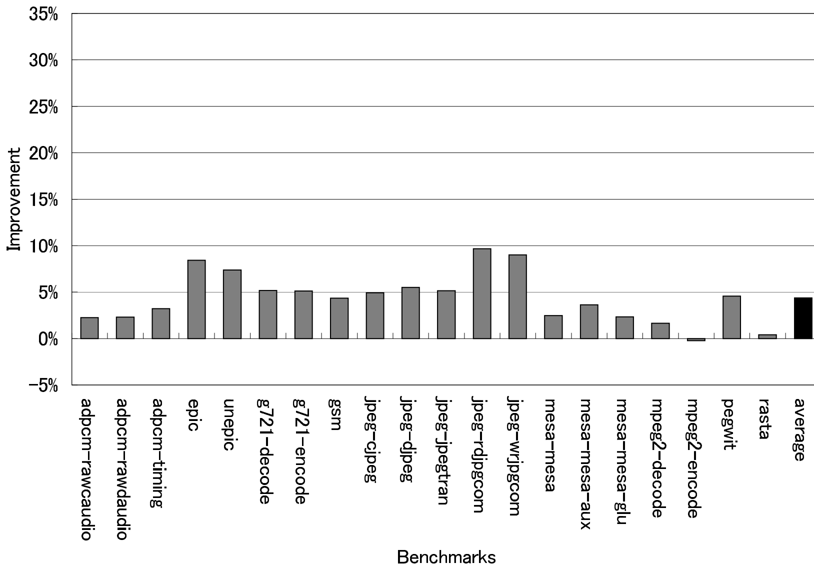


Figure 5.2: Code size reduction with respect to –Os for m68k

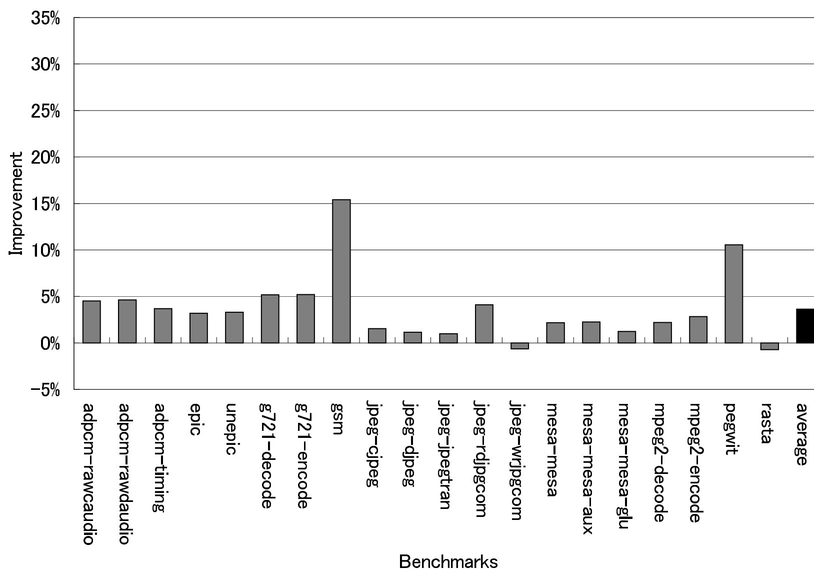


Figure 5.3: Code size reduction with respect to –Os for M32R

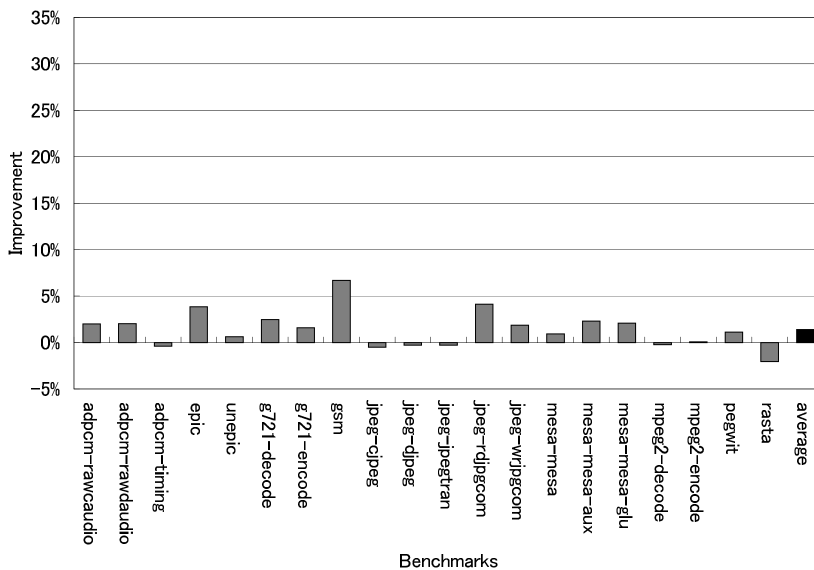


Figure 5.4: Code size reduction with respect to –Os for IQ2000

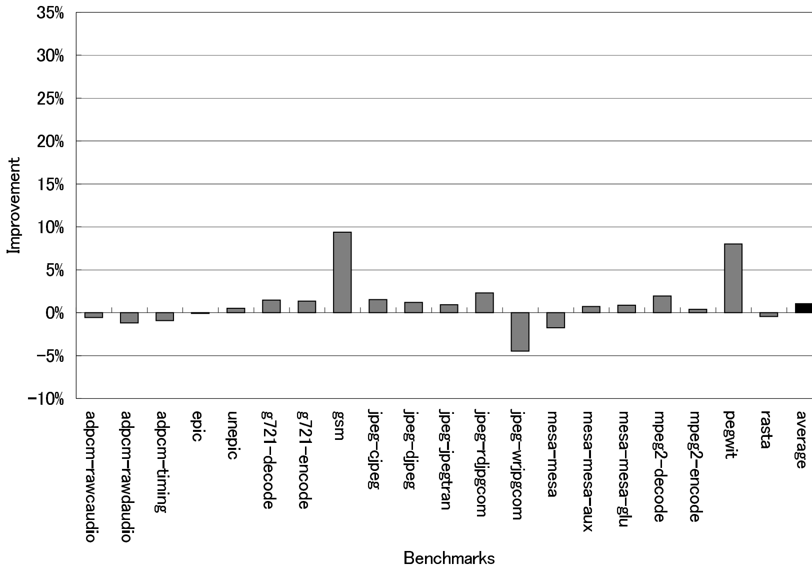


Figure 5.5: Code size reduction with respect to –Os for arm

5.2.2 Code size reduction

In Figures 5.1 through 5.5 we show the code size reduction with respect to the standard option *-Os* which is specifically geared toward code size reduction [13]. This reduction is computed as follows. For an application A, let $S_s(A)$ be the size obtained by using *-Os* given by the number of instructions in the resulting assembly code, and let $S_n(A)$ be the size obtained from our new method. Then the code size reduction $R(A)$ is computed as

$$R(A) = \frac{S_s(A) - S_n(A)}{S_s(A)} \cdot 100\%$$

This definition implies that when the code size obtained from our new method is larger than the size obtained from *-Os*, the reduction has a negative value. We immediately observe that in almost all cases, on all platforms, our method produced code sizes that are shorter than the code sizes produced by *-Os*, up to 30% shorter. The amount of reduction is highly dependent on the platform used. For the *mips*, Figure 5.1 shows that high reductions are obtained for all benchmarks, with an average of 18%. For one benchmark, a reduction of 30% is obtained. For the *m68k* and *M32R*, almost every benchmark is reduced in size with respect to *-Os* and in some cases reductions of 10 to 15% are achieved. In the three cases we produce code that is larger than *-Os*, this degradation is very small. However, for the *IQ2000*, the improvements are modest and in many cases the difference between the code size we produce and that produced by *-Os* is 2 to 3%. In some cases we produce a code that is slightly larger than *-Os*. We perform worst on the *arm*. The reason for this is that most options in *gcc* have little effect on code size for this architecture. We have observed that after a few iterations, the variance (standard deviation) in the 54 different settings tested becomes less than 0.5%. This means

that there exist several hundreds of different settings that give rise to almost the same code size. In fact, there are more than one hundred settings that give rise to exactly the shortest code size found. We have also observed that *-O1*, *-O2*, *-Os* and our method give rise to almost the same code sizes. Only *-O3* produces code sizes that are significantly larger, mainly due to inlining. Nevertheless, also for the *arm* there exist two applications that are significantly reduced in size.

For each platform, there exist at least a few benchmarks that obtain a significant reduction in code size. On the other hand, there are no benchmarks that suffer a significant degradation in size, except *jpeg-wrjpgcom* on the *arm*. As mentioned before, these code size reductions are obtained by carefully exploiting the existing code generator in *gcc* and are essentially 'for free'. This means that our method can be applied, the resulting code size can be compared to *-Os*, and the shortest code can be selected.

5.2.3 The compiler settings

In Figures 5.6 through 5.10, we have shown the final selection of compiler options in the last but one iteration of our iterative method. In these table, '1' denotes that the option has been turned on, '0' that it has been turned off, and a blank space that it has not yet been decided. In the setting that is finally produced, these blanks are filled with values that give rise to the shortest code in the final iteration. However, the variance in this last iteration is very low, sometimes as low as 0.002%. This means that the effect of these options on code size is very low and it is not important which value they receive. For comparison purposes, we have also shown the setting *-Os*.

From these tables, we observe that many options do not have much effect on code size for any benchmarks or platform. Also, we observe that there are a few options (14, 15, and 43) that in our method are explicitly turned off whereas they are turned on in *-Os*. This means that we measure a degradation in size. From these tables, we see that *inline-functions* (6) is turned off in almost all cases, as is *loop-optimize* (15) and *tracer* (42). This last option performs tail duplication to enlarge superblock sizes. In many cases, instruction scheduling (22-27) and *reorder-blocks* (32) are turned off also. The option *omit-frame-pointer* (4) is turned on in almost all cases since it drops the instruction required to create this frame pointer. Remarkably, the loop unrolling option (46) is turned on in several cases. These observations are valid across benchmarks and platforms.

Several options are switched on or off depending on the application and platform, showing that compiler tuning for a particular application and platform can be worthwhile. Finally, we observe that many options have very little effect on code size and are neither switched on nor off by our procedure.

5.3 Conclusion

In this chapter, we have proposed an iterative approach to setting compiler options in order to generate as few instructions in the assembly code as possible. We use a technique that is based on non-parametric inferential statistics, in particular, the Mann-Whitney test, to decide which options should be switched on or off. We have shown that our technique performs

better in almost all cases considered than the standard *-Os* switch that is designed to optimize for size. However, this improvement is highly dependent on the target platform. For the *mips* platform, we obtain high reductions in code size of 18% on average over *-Os*. In some cases, we produce code that is 30% smaller than *-Os*. For the *m68k* and *M32R* we reduce code size by 4 to 5% on average, and 10 to 15% in some cases. Finally, for the *arm* gains are less and in one case we are even 5% larger than *-Os*. However, our technique is easy to implement and requires no adaptation of the compiler. Therefore, it can be worthwhile to try to optimize for size using our method and switching to *-Os* in the few cases it should fail.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-Os	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
adpcm-rawcaudio			0	1							0			1	0		1	1
adpcm-rawdaudio	0			1										1	0		1	1
adpcm-timing			0	1				0			0			1	0			1
epic-epic		1	0	1		0		0		1	1	1		0	0	1	1	1
epic-unepic		0	0	1		0				1	1			1	0	1	1	1
g721-decode		0	0	1		0				1	1			0	0	1	1	1
g721-encode		0	0	1		0				1	1	1		0	0	1	1	1
gsm	1	0	0	1	1	0		1		1	0		1	1	1	1	1	1
jpeg-cjpeg		0	0	1		0				1	1			0	0	1	0	1
jpeg-djpeg		0	0	1		0				1	1	1		0	0	1	0	1
jpeg-jpegtran		0	0	1		0				1	1	1		0	0	1	0	1
jpeg-rdjpgcom				1		1		0		1		1		0	0	1	1	
jpeg-wrjpgcom				1		0		0	1	1	1			0	0	1	1	1
mesa-mesa		0	0	1		0			1	1	1	1		1	0	1	1	1
mesa-mesa-aux	1	1	1	1	1	0	1		1		1			0	0	1	1	1
mesa-mesa-glu		1	0	1	1	0			0	1	1	1		0	0	1	0	1
mpeg2-mpeg2decode		1	0	1	1	0		1	1	1	1			0	0	1	0	1
mpeg2-mpeg2encode		1	0	1		0			1	1	1	0		1	0	1	1	1
pegwit-pegwit	1	0	0	1		0		0	1	1	1	1		0	0	1	0	1
rasta-rasta		0	0	1		0			1	1	1	1		1	0	1	0	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
-Os	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	0	0
adpcm-rawcaudio																		
adpcm-rawdaudio														1	0			
adpcm-timing										1				0				
epic-epic		1		0			0	0	0					0		0		
epic-unepic		1		0			0	0	0	1				0		1		
g721-decode		1		1	0		0							1				
g721-encode		1		1	1	1	0							1				
gsm		1		0	0	1	1	0	0		1	0				1		
jpeg-cjpeg		1		0			0	0	0	1				0		1		
jpeg-djpeg		1		0			0	0	0	1				0		1		
jpeg-jpegtran		1		0			0	0	0	1				0		1		
jpeg-rdjpgcom		0		0	1	1	0	1	1					0				
jpeg-wrjpgcom				0			0	0	0							0		
mesa-mesa		0	1	0			0	0	0	1				1		1		
mesa-mesa-aux		0	1	1	1	1	0	1		1	1	1	1	0	1			1
mesa-mesa-glu	1	0		0			0	0	0	1				0		1		
mpeg2-mpeg2decode	1	0	1	0	1	1	0	0	0			1		1	1	1	1	
mpeg2-mpeg2encode		1		0				0	0	1				0		1		
pegwit-pegwit		1		0			0	0	0					1		1		
rasta-rasta				0			0	0	0					0	1	1		0

	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
-Os	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
adpcm-rawcaudio				0		0	0									0	
adpcm-rawdaudio				0		0	0									0	
adpcm-timing				0		0	0						0			0	
epic-epic			0	1		0	0						0			0	
epic-unepic				1	1	0	0			1						0	
g721-decode			0	0	1	0	0			1			0			0	
g721-encode			0	0	1	0	0									0	
gsm			1	1	1	0	0	0		1			1	0		0	
jpeg-cjpeg			0	1	1	0	0			1						0	
jpeg-djpeg			0	1	1	0	0			1	1					0	
jpeg-jpegtran			0	1	1	0	0			1	1					0	
jpeg-rdjpgcom			1	1	1	0	0									0	
jpeg-wrjpgcom			1	0	1	1	0						0			0	
mesa-mesa			0	1	1	0	0									0	
mesa-mesa-aux		1	1	0	1	0	0	1	1	1	1			1		0	
mesa-mesa-glu			1	1	1	0	0									0	
mpeg2-mpeg2decode	1		1	1	1	0	0		1	1	1	1			1	0	
mpeg2-mpeg2encode			0	1	1	0	0			1	1		1			0	
pegwit-pegwit			1	1	1	0	0						0			0	
rasta-rasta		1	1		1	0	0	1	1			0	0		0	0	

Figure 5.6: Generated settings for mips

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-Os	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
adpcm-rawcaudio	1	1	0	1		0								1	0		1	
adpcm-rawdaudio	1		0	1		0								1	0		1	
adpcm-timing	1		0	1		0		0						0	1	1	0	
epic-epic	1	0	0	1		0				1	0	1		0	0	1	1	
epic-unepic	1	0	0	1		0				0	0			1	0	1	1	
g721-decode	1	0	0	1		0				1	0	1		1	0	1	1	
g721-encode	1	0	0	1		0				1	0	1		1	0	1	1	
gsm	1	0	0	1		0				0	0			1	0	1	0	1
jpeg-cjpeg	1	0	0	1		0		0		0	0	0		0	0	1	0	1
jpeg-djpeg	1	0	0	1		0				1				0	0	1	0	1
jpeg-jpegtran	1	0	0	1		0				0	0			0	0	1	0	1
jpeg-rdjpgcom	1	0	0	1		0				1				1	0	1	1	
jpeg-wrjpgcom	1	0	0	1		0		0		1				0	0	1	0	1
mesa-mesa	1	0	0	1	0	0	1			1	1		1	0	0	1	1	1
mesa-mesa-aux	1	1	0	1	1	0	1	1		0	0	1	1	0	0	1	0	1
mesa-mesa-glu	1	0	0	1	1	0				1	1	1		0	0	1	0	1
mpeg2-mpeg2decode	1	0	0	1		0				1	1			0	0	1	0	1
mpeg2-mpeg2encode	1	0	0	1		0				1	1	0		0	0	1		1
pegwit-pegwit	1	0	0	1		0		0		1	1	0		0	0	1	0	1
rasta-rasta	1	0	0	1	0	0		0		0	0	1		0	0	1	0	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
-Os	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	0	0
adpcm-rawcaudio		0	1											0				
adpcm-rawdaudio			1											0				
adpcm-timing		1	1								0			0				
epic-epic		0	1			1								0		1		
epic-unepic		1	1											0		1		
g721-decode		1	1										1	0				
g721-encode		1	1										1	0				
gsm		1	1										1	0		1		
jpeg-cjpeg		0	1							0			1	0		1		
jpeg-djpeg		0	1							0			1	0		1		
jpeg-jpegtran		0	1							0			1	0		1		
jpeg-rdjpgcom		1	1											0		1		
jpeg-wrjpgcom			1	0										0		1		
mesa-mesa	1	0	1	1	1	1	1	1		1			1	1	1	1	1	
mesa-mesa-aux	1	1	0	1		1		1		1		1		0				
mesa-mesa-glu		0	1	1										0				
mpeg2-mpeg2decode		0	1										1	0		1		
mpeg2-mpeg2encode		0	1							0				0		1		
pegwit-pegwit		1	1	1										0		1		
rasta-rasta		0	1											0		1		

	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
-Os	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
adpcm-rawcaudio				0		0	0						0				
adpcm-rawdaudio				0		0	0						0				
adpcm-timing				0	1	0	0						0				
epic-epic			0	0	1	0	0			1			0				
epic-unepic			0	0	1	0	0			1			0				
g721-decode			1	0	1	1							0				
g721-encode			1	0	1	1											
gsm			0	1		0	1			1							
jpeg-cjpeg			1	0	1	0	1			1	1						
jpeg-djpeg			1	0	1	0	1			1			0				
jpeg-jpegtran			1	0	1	0	1			1	1						
jpeg-rdjpgcom				0		0	1						0				
jpeg-wrjpgcom			1	0	1	0	1						0				
mesa-mesa			0	0	1	0	1	1			1		1	1			1
mesa-mesa-aux			0	1	1	0	1		1		1		1		1		
mesa-mesa-glu			0	0	1	0	1										
mpeg2-mpeg2decode			1	0	1	0	1			1				1			
mpeg2-mpeg2encode			1	1	1	0	0										
pegwit-pegwit			1	1	1	0	1			0	0		0				
rasta-rasta			0	0	1	0	1										

Figure 5.7: Generated settings for m68k

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-Os	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
adpcm-rawcaudio			0	1		0		0						1	0		1	1
adpcm-rawdaudio			0	1		0								1	0	1	1	
adpcm-timing				1		0								1	0	1		1
epic-epic	1	0	0	1		0				1	1			1	0	1	1	1
epic-unepic	1	0	0	1		0				1	1	1		1	0	1	1	1
g721-decode		0	0	1		0		1	1	1	1		1	0	1	1	1	1
g721-encode		0	0	1		0		0		1			1	0	1	1	1	1
gsm		0	0	1		0		1		1	1		1	1	1	1	1	1
jpeg-cjpeg		0	0	1		0				1	1	1		0	0	1	0	1
jpeg-djpeg		0	1	1	1	0				1	1	1		0	0	1	0	1
jpeg-jpegtran		0	0	1		0		0	0	1	1	1		1	0	1	0	1
jpeg-rdjpgcom				1		0				1		1		1	0	1		
jpeg-wrjpgcom				1		0			1	1	1			0	0	1	1	
mesa-mesa		0	0	1		0		1	1	1	1	1		0	0	1	1	1
mesa-mesa-aux		0	0	1		0		1			1		1	1	1	1	1	
mesa-mesa-glu		0	1	1	0	0		1	1	1	1	0		1	0	1	1	1
mpeg2-mpeg2decode	1	0	0	1		0			1	1	1		1	0	0	1	0	1
mpeg2-mpeg2encode		0	0	1		0		1	0	1	1	1		1	0	1	0	1
pegwit-pegwit		0	0	1		0			0	1	1	1		0	0	1	0	1
rasta-rasta		0	0	1		0			1	0	1	1		0	0	1	0	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
-Os	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	0	0
adpcm-rawcaudio				0				0	0					0				
adpcm-rawdaudio				0				0	0					0				
adpcm-timing			1	0				0	0					0				
epic-epic		1	1	0			0	0	0					0		1		
epic-unepic		0	1	0			0	0	0					0		1		
g721-decode		1	0	1	0	0	1					0		1				
g721-encode		1	0	1	0		1					0		1				
gsm		1	1	0			1	0	0		1	0		0		1		
jpeg-cjpeg		1	1	0	0		0	0	0	1				0		1		
jpeg-djpeg		0	1	0			0	0	0	1				0		1		
jpeg-jpegtran		0	1	0			0	0	0	1				0		1		
jpeg-rdjpgcom			1	1												1		
jpeg-wrjpgcom			1	1	0	0	0							0		1		
mesa-mesa		1	0	0		0	0	0	0	1				1		1	1	
mesa-mesa-aux		1	1	0	0		0	0	0			0						
mesa-mesa-glu	1	1	1	0			0	0	0	1				0		0		1
mpeg2-mpeg2decode		1	1	0			0	0	0	1				0		1		
mpeg2-mpeg2encode		1	1	0			0	0	0	1				0		1		
pegwit-pegwit		1	1	0			0	0	0	1				0		1		
rasta-rasta		1	0	0			0	0	0					0		1		

	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
-Os	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
adpcm-rawcaudio				0		0	0						0				
adpcm-rawdaudio				0		0	0										
adpcm-timing				0		0	0										
epic-epic			1	1	1	0	0			1	1		0				
epic-unepic				0	1	0	0										
g721-decode				0	1	0				1				1			
g721-encode				0	1	0				1			1	0			
gsm				0	1	1	1			1					0		
jpeg-cjpeg			0	1	1	0	1			1	1						
jpeg-djpeg			0	1	1	0	1				1						
jpeg-jpegtran			0	1	1	0	1			1	1		0				
jpeg-rdjpgcom				1		0	1						0				
jpeg-wrjpgcom				1	1	0	1										
mesa-mesa			0	1	1	0	0										
mesa-mesa-aux			0	0	1	0	1			1			0				
mesa-mesa-glu			0	1	1	0	1				1		1				
mpeg2-mpeg2decode				1	1	0	1										
mpeg2-mpeg2encode			1	1	1	0	1										
pegwit-pegwit			0	1	1	0	1										
rasta-rasta	0		0	1	1	0	1				1						

Figure 5.8: Generated settings for M32R

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-Os	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
adpcm-rawcaudio			1	1		0				0	0	0		1	0	0	1	1
adpcm-rawdaudio	0			1		0				0	0			1	0	0	1	1
adpcm-timing				1		0					1			0	0	0	0	1
epic-epic		1	0	1		0		0		1	1	1		0	0	1	1	1
epic-unepic		0		1	1	0			1	1	1	1	1	0	0	1	1	1
g721-decode			0	1		0		1		1			1	0	1	1	1	1
g721-encode			0	1		0				1	0			0	0	1	1	1
gsm		1	0	1		0		1	1	0	1		1	1	1	1	1	
jpeg-cjpeg		0	0	1		0			0	1	1	1		1	0	1	0	1
jpeg-djpeg		0	0	1		0			0	1	1	1	0	1	0	1	0	1
jpeg-jpegtran		0	0	1		0				1	1			0	0	1	0	1
jpeg-rdjpgcom				1		0		1		1	1	0		1	0	1		
jpeg-wrjpgcom				1		0				1	1	1		1	0	1	1	
mesa-mesa		0	0	1		0	1		1	1	1	1		0	0	1	1	1
mesa-mesa-aux		0	0	1		0					1	1		1	0	1	1	1
mesa-mesa-glu		0	0	1	1	0	0		0	1	1	1	1	0	0	1	1	1
mpeg2-mpeg2decode		1	0	1	1	0				1	1	0		0	0	1	0	1
mpeg2-mpeg2encode	0	0	1	1		0		0	1	1	1			1	0	1	1	1
pegwit-pegwit		1		1	0	0			0	1	1			0	0	1	0	1
rasta-rasta	0	0	0	1		0				1	0	1		0	0	1	1	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
-Os	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	0	0
adpcm-rawcaudio					0	0	0							0			0	
adpcm-rawdaudio					0	0	0	1			0			0				
adpcm-timing							0							0				
epic-epic		1		1	1	0	1							0		0		
epic-unepic	1	0		1	0		1	1		1	1		1	0		1		1
g721-decode			1	1	0		1	1			1	0		1				
g721-encode		1	1	1	0		1	1						1				
gsm		1		0			1	0	0		1	0		1		1		
jpeg-cjpeg		1		1	0	0	1			1				0		1		
jpeg-djpeg		1		0	0	0	1	0	0	1				0		1		
jpeg-jpegtran		0		0	0		1	0	0	1				0		1		
jpeg-rdjpgcom				0			1	0	0					1		1		
jpeg-wrjpgcom				0	0		1	0	0					1		1		
mesa-mesa	1	0	1	0	1	1	1	0	0	1	1	1		1		1	1	
mesa-mesa-aux		0	1	0			1	0	0	0				1		0		
mesa-mesa-glu	1	0	1	1	0	0	1							1		1		
mpeg2-mpeg2decode		0		1	0		1			1				1		1		
mpeg2-mpeg2encode		0	1	1	1	1	1	0		1		0		0		1		
pegwit-pegwit		1		0			1	0	0					0		1		
rasta-rasta		0		0			1	0	0					0		1		

	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
-Os	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
adpcm-rawcaudio				0		0	0						0			1	
adpcm-rawdaudio						0	0			1				0		1	
adpcm-timing		1		1		0	0			1						1	
epic-epic			1	0	1	0	0			1			0			1	
epic-unepic	1	1	0	1	1	0	0	1			1					1	1
g721-decode			1	1	1	0				1			0			1	
g721-encode				1	1	0				1						1	
gsm			1	1	1	0	1			1			1			1	
jpeg-cjpeg			1	1	1	0	1				1					1	
jpeg-djpeg			1	1	1	0	1				1					1	
jpeg-jpegtran			1	1	1	0	1			1						1	
jpeg-rdjpgcom			1	1	1	1	1			1	1		0			1	
jpeg-wrjpgcom			1	0	1	0	1									1	
mesa-mesa	1	1	1	1	1	0	1		1			1	1	1	1	1	
mesa-mesa-aux			1	0	1	0	0						0			1	
mesa-mesa-glu			1	1	1	0	1			1			0			1	
mpeg2-mpeg2decode			1	0	1	0	1			1						1	
mpeg2-mpeg2encode			1	1	1	0	1				1		0			1	
pegwit-pegwit			1	0	1	0	1			1						1	
rasta-rasta			1	1	1	0	0			1						1	

Figure 5.9: Generated settings for IQ2000

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-Os	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
adpcm-rawcaudio		1	1	1	1	0	1	0	1	1		0		1	0	1	0	1
adpcm-rawdaudio		1		1		0				1	0	0		1	0		0	1
adpcm-timing		1	0	1										0	0		0	1
epic-epic	1	1	0	1	1	0				1	1	1	1	1	0	1	0	1
epic-unepic		1	0	1		0			0	1	1	1		0	0	1		1
g721-decode		1	0	1		0				1		1		1	0	1		1
g721-encode		1	0	1		0				0	1			1	0	1	1	1
gsm		1	0	1		0			0	1	1	0		1	0	1	0	1
jpeg-cjpeg		1	0	1	0	0			0	1	0			0	0	1	0	1
jpeg-djpeg		1		1		0							0		1	1	1	
jpeg-jpegtran				1		0										1		
jpeg-rdjpgcom			0	1		0				1				1	0	1		1
jpeg-wrjpgcom			0	1		0				1	1			0	0	1	0	1
mesa-mesa	1	0	0	1	0	0	1		1	1	1		1	0	0	1	1	1
mesa-mesa-aux			0	1	0	0				1	1			0	0	1	1	1
mesa-mesa-glu	1	1		1	0	0			0	0	1	1		0	0	1	1	1
mpeg2-mpeg2decode		0	0	1	0	0			1	1	1	0		0	0	1	0	1
mpeg2-mpeg2encode		0	0	1	0	0		0	1	0	1	0		1	0	1	1	1
pegwit-pegwit	1	1	0	1		0		0	1	1	1	1		1	0	1	0	1
rasta-rasta		0	0	1		0	1		1	0	0	1		1	0	1	1	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
-Os	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1	0	0
adpcm-rawcaudio				1	1		1		1	1	1	1			1		1	
adpcm-rawdaudio		1		1				1										
adpcm-timing				1			1						1	0				
epic-epic	1	1	1	0	0	1	0	0	0			1	1	0	1	1		1
epic-unepic		1		0			0	0	0			0		0		1		
g721-decode		1		1	0		0						1	1				
g721-encode		1		1	0		0						1	1				
gsm	1	1		0			0	0	0				1	0		1		
jpeg-cjpeg		1		0			0	0	0	0			1	0		1		
jpeg-djpeg		0		0				0	0							1		
jpeg-jpegtran		0		0					0							1		
jpeg-rdjpgcom		1		0				0	0					0				
jpeg-wrjpgcom				0				0	0					0				
mesa-mesa	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1	1		
mesa-mesa-aux		1		0			0	0	0				1	1				
mesa-mesa-glu	1	1		0			0	0	0				1	0		1		
mpeg2-mpeg2decode		0		0			0	0	0	0			1	1		1		
mpeg2-mpeg2encode		1		0			0	0	0	1			1	0		1		
pegwit-pegwit	0	1	1	0	1	1	1	0	0	1	1	1	1	1		1		
rasta-rasta	1	1	1	0	1	1	0	0	0			0	1	1	1	1		1

	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
-Os	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
adpcm-rawcaudio	1			0		0	0	1	1	0	0	1	0	1			1
adpcm-rawdaudio				0		0	0			0	0		0				
adpcm-timing			0	0		0	0										
epic-epic				1	1	0	0				1		0	1	1		
epic-unepic				0	1	0	0						0				
g721-decode			1	0	1	0											
g721-encode			1	0	1	0				1			0				
gsm			0	0	1	0	1										
jpeg-cjpeg			0	0	1	0	1			1	1						
jpeg-djpeg				0			1			0		0	1				
jpeg-jpegtran				0						0			1				
jpeg-rdjpgcom					1	0	1										
jpeg-wrjpgcom				0	1	0	1										
mesa-mesa	1		0	0	1	0	1		1	1		1	1			1	1
mesa-mesa-aux			0	0	1	0	1			1							
mesa-mesa-glu			0	1	1	0	1										
mpeg2-mpeg2decode			0	1	1	0	1					1					
mpeg2-mpeg2encode			0	1	1	0	0			0	0	1	0				
pegwit-pegwit	1	1	1	1	1	0	1	1				1	0			1	
rasta-rasta	1		0	1	1	0	1	1			1	1	0	1	1		

Figure 5.10: Generated settings for arm

Chapter 6

The determination of compiler settings for multiple applications taking into account interaction between optimizations

Modern compilers implement a host of back end optimizations, however, very little is known about what all these options exactly do, how they interact, whether or not they enable or disable one another, etc. In order to deal with all these options, compiler writers usually define standard settings that can be used by programmers as *-Ox* switches. These switches contain those options that the compiler writer thinks are beneficial for many programs based on his experience. It is well known that for a single application better compiler settings can be found than those standard settings as we have shown in Chapter 3 and 4 of this thesis. However, it is not clear whether better settings can be found for a collection of applications. In this chapter, we show that this indeed is the case. We construct one single compiler setting that produces better optimized programs for a collection of widely differing applications than standard *-Ox* settings do, with up to 20% improvement. The collection of programs we use in this study is the SPECint95 benchmark suite which is meant to be representative for a large collection of integer programs and hence has widely differing properties.

In this chapter, we focus on an extension of the problem of optimizing one single program, namely, how to find a compiler setting that optimizes many programs simultaneously. This setting should perform better than standard *-Ox* settings for most programs in the collection and equally well for the remaining programs. By this we mean that each program is optimized as well as the best performing *-Ox* switch and most programs even significantly better. This problem is relevant for example for supercomputing centers where a fixed collection of applications is used extensively and a single compiler optimization setting must be used for maintainability purposes. There are two main obstacles in finding these settings. First, since the number of different combinations of optimizations is exponentially large, the space, which has to be searched in order to find the optimal one, is huge. Second, the best

sequence depends both on the application being compiled and on the target architecture.

The aim of our research is to develop a methodology which yields a feasible and near optimal solution for this problem. Our methodology is based on the simple idea that some compiler optimizations can positively influence each other and, therefore, should be turned on together. The fact that optimizations can interact is well known and expert compiler writers choose settings based on empirical experience of positive interaction. In this chapter, we introduce a formal definition of the interaction between optimizations and present a quantitative model which allows us to measure these interactions.

The heuristic we propose in this chapter is based on an experimental analysis of the effect of compiler options. We proceed by a three step approach. In the first step, we determine the set of options that have a large effect of their own and the set of pairs of options that positively interact. We then iteratively add single optimizations to the sets already obtained to get maximal sets of positively interacting options. In the second step, we try to combine the sets found in the previous step under the condition that these sets do not negatively influence each other. This weaker condition allows us to find compiler settings which have as many optimizations turned on as possible. Finally, in the third step, we test a small number of settings found in step two, selecting the setting with the best average improvement. All analysis has been performed using a small but representative subset of the entire optimization space. This entails that the effects we measure are only approximations but it allows us to construct a small fragment of the entire search space beforehand and use the measured execution times freely during the subsequent analysis phases. Since our heuristic is intended to be used by compiler tuners in order to define a general setting usable for many programs in a given set, we can afford some profiling time in order to set up this initial search space. The ultimate goal of this research that we hope to address in future work is to find a strategy to find a compiler setting that is applicable to a specific application domain and that produces highly optimized code for that domain by incorporating domain specific knowledge.

The rest of this chapter is organized as follows. Section 6.1 discusses our methodology to find a setting that is optimized for a collection of programs. Section 6.2 discusses the strategy to obtain such an optimized setting. Section 6.3 gives the results of our method on *gcc 3.3.1* and Section 6.4 gives an analysis of our methodology. In section 6.5, we summarize this chapter and discuss directions for future work.

6.1 A methodology to define a compiler setting

In this section, we introduce a systematic method to generate a compiler setting that is optimized for a collection of programs. We focus on the interaction between optimizations since this is an important issue in the determination of an optimal compiler setting. We believe that a nearly optimal compiler setting can be derived by combining optimizations which positively interact. Since there is no clear definition of the interaction between optimizations, we give a definition of interaction based on the performance improvement of a collection of different optimization settings on a set of applications. We describe how execution times can be used to define the interaction between optimizations.

6.1.1 The detection of interaction between optimizations

In this section, we give the definition of interaction between compiler optimizations and we describe the procedure to construct subsets of optimizations that positively interact. We define the effect of compiler optimizations as the improvement expressed in terms of the real execution time of applications. The effect is dependent on the compiler setting used and therefore the effect is used to define interaction between these different compiler settings. In order to define the notion of interaction properly, we first need some notation.

We use a sequence of binary digits to describe a compiler setting s. The set of all possible sequences is the complete search space.

Definition 6.1.1 *Let $n \geq 1$. Ω_n is the set of all sequences of n binary digits.*

$$\Omega_n = \{(a_1, \ldots, a_n) \ : \ a_i \in \{0,1\}, 1 \leq i \leq n\}$$

The sequences in Ω_n express whether compiler options are on or off. If we denote the set of all available compiler options by $W = \{w_i \ : \ 1 \leq i \leq n\}$, then $a_i = 1$ in a sequence s means that optimization w_i is turned on and $a_i = 0$ means that optimization w_i is turned off.

Let s_{base} be the compiler setting without any optimization:

$$s_{base} = (0, \ldots, 0) \tag{6.1}$$

Let P be a set of benchmark programs and let $T(s,p)$ be the execution time for compiler setting $s \in \Omega_n$ and program $p \in P$. The effect of compiler setting $s \in \Omega_n$ on a program $p \in P$ is defined as a normalized function $e(s,p)$.

Definition 6.1.2 *Let $s \in \Omega_n$ and let $p \in P$. We define the effect of compiler setting s on program p, denoted by $e(s,p)$, as follows.*

$$e(s,p) = \frac{T(s_{base},p) - T(s,p)}{T(s_{base},p)}$$

The value $e(s,p)$ describes how much the optimizations in s change the execution time of a non-optimized program p. The effect of a compiler setting s, denoted by $E(s)$, is defined as the average effect over all programs in P. We can simply take the average because $e(s,p)$ is normalized. In fact, $E(s)$ is the average of the improvements in performance of all the benchmark programs. Note that it can very well happen that a compiler setting generates substantial improvement on certain benchmark programs which are nullified by slow downs on other benchmark programs. In this case, the overall improvement is considered minimal. Also, $E(s)$ can have a negative value.

Definition 6.1.3 *We define the effect of a compiler setting $s \in \Omega_n$ on a set of programs P, denoted by $E(s)$, as follows.*

$$E(s) = \frac{\sum_{p \in P} e(s,p)}{|P|}$$

Using these definitions, we define the interaction between two optimizations w_i and w_j as follows.

Definition 6.1.4 *For each $1 \leq i, j \leq n$, $i \neq j$, we define*

$$s_i = (a_1, \ldots, a_n) \quad \text{where } a_k = \begin{cases} 1 & : \ k = i \\ 0 & : \ \text{otherwise} \end{cases}$$

$$s_{ij} = (a_1, \ldots, a_n) \quad \text{where } a_k = \begin{cases} 1 & : \ k = i \ \text{or} \ k = j \\ 0 & : \ \text{otherwise} \end{cases}$$

Given this notation, we define for each $w_i, w_j \in W$ the following interaction predicate I_1.

$$I_1(w_i, w_j) = \begin{cases} \text{true} & : \ E(s_{ij}) - E(s_i) > \text{threshold, and} \\ & \ E(s_{ij}) - E(s_j) > \text{threshold} \\ \text{false} & : \ \text{otherwise} \end{cases}$$

In other words, s_i is the compiler setting which turns on optimization w_i, s_j is the compiler setting which turns on optimization w_j, and s_{ij} is the compiler setting which turns on both w_i and w_j. The first inequality guarantees enough performance improvement if optimization w_j is added to w_i, and the second inequality checks if there is enough performance improvement if w_i is added to w_j. If $I_1(w_i, w_j) = \textit{true}$, then we conclude that w_i and w_j positively interact.

The problem with the above definition of interaction is that in practice the effect of one single optimization turned on can be minimal and the true effect of an optimization can only be estimated in the presence of other optimizations that are turned on. Therefore, we need a definition of interaction which allows several optimizations to be turned on simultaneously.

Definition 6.1.5 *For $K \subseteq W$, $w_k \in W$, and $w_k \notin K$, we define*

$$s_K = (a_1, \ldots, a_n) \quad \text{where } a_i = \begin{cases} 1 & : \ w_i \in K \\ 0 & : \ \text{otherwise} \end{cases}$$

$$s_{Kk} = (a_1, \ldots, a_n) \quad \text{where } a_i = \begin{cases} 1 & : \ w_i \in K \ \text{or} \ i = k \\ 0 & : \ \text{otherwise} \end{cases}$$

We define the following interaction predicate I_2

$$I_2(K, w_k) = \begin{cases} \text{true} & : \ E(s_{Kk}) - E(s_K) > \text{threshold} \\ \text{false} & : \ \text{otherwise} \end{cases}$$

As we can see from this definition, the threshold in the inequality does not need to hold for $E(s_{Kk}) - E(s_k)$, making this definition of interaction different from the previous one. The reason for this is that this definition is used in an iterative algorithm which looks for successful grouping of optimizations by incrementally extending already successful groupings. Hence, the definition assumes that the effect of s_K is already significant and only for the addition of one optimization s_k it is investigated whether this leads to improved performance.

Although this definition of interaction is useful when searching for an optimal setting of compiler optimizations, the sheer number of possible optimization settings makes it impossible to apply this definition arbitrarily. For instance, in our experiments below we want to investigate the optimal setting for the GNU C-compiler *gcc 3.3.1* in which more than

60 compiler optimizations can be turned on or off. This yields more than $2^{60} \approx 10^{18}$ different possible settings. Hence using this definition arbitrarily would mean that 10^{18} different experiments would have to be conducted to find an optimal setting. Therefore, we want to trim down the possible number of different compiler settings to be investigated considerably. This is achieved in two ways: first, the proposed iterative search algorithm tries to use already successful settings which are incrementally extended. The second measure consists of trimming down the search space of possible settings. This is achieved by constructing a representative subset of these different possible settings using an orthogonal array, as explained in Section 2.1.

6.1.2 Defining interaction using a representative subset of the search space

Having defined a representative subset S of the complete search space by means of an Orthogonal Array, our algorithm is based on using the measured performance results of the compiler settings which are represented by S. This means that initially all the compiler settings represented by S are executed on a specific platform for all the benchmarks considered. The execution times are stored in a database to be used by our methodology and no other execution times are used. This means that the experimental results needed to verify previously given definitions of interaction might not be available. For instance, in Definition 6.1.5, the compiler settings s_K, or s_k, or s_{Kk} might not be part of the subset S. Therefore, we need to adopt the previously given definitions so that only experimental results derived from S are necessary to establish interaction or not.

Definition 6.1.6 *Let $S \subseteq \Omega_n$ and let $A, B \subseteq W$ with $A \cap B = \varnothing$. We define*

$$S^{(A=1)(B=0)} = \{s : \begin{array}{l} s \in S \\ \text{with } a_i = 1 \text{ for } w_i \in A, \text{ and} \\ \text{with } a_i = 0 \text{ for } w_i \in B\} \end{array}$$

$S^{(A=1)(B=0)}$ is a (possibly empty) subset of S, in which the optimizations in A are turned on and the optimizations in B are turned off. Below we write $S^{(A=1)}$ in case $B = \varnothing$. Note that $A \cup B$ is not necessarily equal to the complete set W. Optimizations that do not appear in either A or B may be turned on or off arbitrarily. The approximate effect of $S^{(A=1)(B=0)}$ is described as follows.

Definition 6.1.7 *Let $S \subseteq \Omega_n$ and let $A, B \subseteq W$ with $A \cap B = \varnothing$. The* approximate effect *of $S^{(A=1)(B=0)}$ is defined as*

$$E(S^{(A=1)(B=0)}) = \begin{cases} \frac{\sum_{s \in S^{(A=1)(B=0)}} E(s)}{|S^{(A=1)(B=0)}|} & : \ S^{(A=1)(B=0)} \neq \varnothing \\ 0 & : \ \textit{otherwise} \end{cases}$$

The approximate effect is applied to Definition 6.1.5 to obtain a new definition of interaction.

Definition 6.1.8 *Let be $K \subseteq W$ and let $w_k \in W$ such that $w_k \notin K$. We define the interaction between K and w_k as follows.*

$$I(K, w_k) = \begin{cases} true & : \quad E\left(S^{(K \cup \{w_k\}=1)}\right) \\ & \quad - E\left(S^{(K=1)(\{w_k\}=0)}\right) > threshold \\ false & : \quad otherwise \end{cases}$$

The set $S^{(K=1)}$ is split into two parts, $S^{(K \cup \{w_k\}=1)}$ and $S^{(K=1)(\{w_k\}=0)}$ depending on the state of w_k. The first part $S^{(K \cup \{w_k\}=1)}$ consists of the compiler settings that turn on the optimizations in K as well as w_k. The second part $S^{(K=1)(\{w_k\}=0)}$ consists of compiler settings that turn on the optimizations in K and turn off optimization w_k.

6.2 The algorithm to find a compiler setting

In this section we discuss the three steps of our algorithm to find a compiler setting that is highly optimized for a set of applications simultaneously.

6.2.1 Step 1: Finding maximal subsets of positively interacting options

Using Definition 6.1.8, we construct an algorithm to produce subsets of optimizations that positively interact in two phases. This is done iteratively by taking an initial choice of single optimizations which have a relatively large effect in the first phase. Then, in each iteration, the subsets of positively interacting optimizations are being extended with optimizations which enforce the previous subsets in the second phase. At each step of the algorithm, the set C consists of subsets of optimizations that positively interact.

In the first phase, the initial set C_1 is selected as the collection of single optimizations that have a relatively large effect. We calculate the effect of each individual optimizations with Definition 6.1.7 as follows. For each i, $A = \{w_i\}$ and Definition 6.1.7 is applied with $B = \emptyset$. So we compute the average improvement of all optimization settings in S in which w_i is turned on. Then we rank all optimizations w_i according to their largest average effect. C_1 is then constructed by taking the top M optimizations of this list. In our implementation we have heuristically chosen M to be 10, so the algorithm starts with a subset C_1 which consists of 10 individual optimizations.

After this first phase in which a collection of individual optimizations are selected which have a significant effect, in the second phase of Step 1 we select from the remaining optimization those optimizations which are successful if they are combined with another optimization. The reason for this is that certain optimizations are only successful if they are enabled by another optimization. If the start set of successful optimizations would only consist of individual optimizations, the optimizations which are successful only when combined with an (enabling) optimization would be left out. For defining the second phase of Step 1 of the algorithm we need the notion of 'left out' optimizations W^k.

Definition 6.2.1 *W^k is the set of optimizations that do not appear in the elements of $C_1, \ldots, C_{k-1}$.*

$$W^k = \{w \in W \ : \ \forall 1 \le i \le k-1 \ . \ \forall c \in C_i \ . \ w \notin c\}$$

Therefore, W^2 is the set of those optimizations that are left over from the construction of C_1. After the construction of C_1, each optimization in W is combined with all the optimizations in W^2. Each combination is examined with Definition 6.1.8 to form potential subsets of C_2. The combination of two optimizations in C_1 is not investigated because we want to find two optimizations that enable each other while they have little effect when used on their own. If one of the optimizations which is used in the combinations in C_2 appears in C_1, the optimization is removed from C_1 since the optimization is considered to work better when combined with another combination.

After C_2 has been found, C_i is inductively constructed by adding one optimization from the set of optimizations W^i to one of the elements in C_{i-1} when it satisfies the predicate I given in Definition 6.1.8. The formal algorithm is given in Algorithm 1. For each iteration to construct C_i, Algorithm 1 removes elements from C_{i-1}. This procedure is based on the concept that the extended combination may have better improvement than the original one according to Definition 6.1.8. The algorithm produces C which consists of all C_i, that is, $C = \bigcup_i C_i$. Each C_i consists of sets of i positively interacting optimizations. The algorithm stops if no set can be extended anymore. Hence we construct sets of maximal sequences of options.

6.2.2 Step 2: Combining subsets

Up till now we described how successful subsets of optimizations can be identified. Because the goal of this study is to define a compiler setting which works well for a set of different applications, in this phase of the algorithm we identify combinations of these subsets which do not negatively interfere with each other. So instead of looking for optimizations which enforce each other, we now look at sets of optimizations which do not have a negative impact on each other.

The algorithm for constructing these compiler settings uses an evaluation predicate again. This predicate decides whether two subsets must be combined.

Definition 6.2.2 *Let $K_1, K_2 \subset W$ and $K_1 \neq K_2$. We define the approximate interaction between K_1 and K_2 as follows.*

$$I'(K_1,K_2) = \begin{cases} true & : \; E\left(S^{(K_1 \cup K_2=1)}\right) - E\left(S^{(K_1=1)}\right) \geq 0 \\ & \quad and \\ & \quad E\left(S^{(K_1 \cup K_2=1)}\right) - E\left(S^{(K_2=1)}\right) \geq 0 \\ false & : \; otherwise \end{cases}$$

The first equation compares the effect of the optimizations in K_1 and K_2 with the effect of the optimizations in K_1. The second equation compares the effect of the optimizations in K_1 and K_2 with the effect of the optimizations in K_2. As can be seen from this definition, no threshold value is used in the inequality indicating that turning on the optimization represented by K_2 in addition to the optimizations represented by K_1 does not lead to a performance degradation and visa versa.

The algorithm now investigates which combinations of subsets in C can be combined. It starts with all the elements in C and tries to combine them. K_2 in the algorithm is always

Algorithm 1

$C \leftarrow \varnothing$
Create C_1
$C \leftarrow C \cup C_1$
Create W^2
$C_2 \leftarrow \varnothing$
for all $k \in W$ **do**
  for all $w \in W^2$ **do**
    if $I(\{k\}, w)$ is true **then**
      $C_2 \leftarrow C_2 \cup \{\{w, k\}\}$
      if $k \in C_1$ **then**
        $C_1 \leftarrow C_1 - \{k\}$
      end if
    end if
  end for
end for
$C \leftarrow C \cup C_2$
$i \leftarrow 2$
repeat
  $i \leftarrow i + 1$
  $C_i \leftarrow \varnothing$
  Create W^i
  $V \leftarrow C_{i-1}$
  for all $K \in V$ **do**
    for all $w \in W^i$ **do**
      if $I(K, \{w\})$ is true **then**
        $C_i \leftarrow C_i \cup \{K \cup \{w\}\}$
      end if
    end for
    if there exists $c \in C_i$ with $K \subset c$ **then**
      $C_{i-1} \leftarrow C_{i-1} - \{K\}$
    end if
  end for
  $C \leftarrow C \cup C_i$
until $C_i = \varnothing$ or $W^i = \varnothing$

Algorithm 2

$C'_1 \leftarrow C$
$i \leftarrow 1$
repeat
  $i \leftarrow i+1$
  $V \leftarrow C'_{i-1}$
  $C'_i \leftarrow \varnothing$
  for all $K_1 \in V$ **do**
    for all $K_2 \in C$ **do**
      if $K_1 \neq K_2$ **then**
        if $I'(K_1, K_2)$ is true **then**
          $C'_i \leftarrow C'_i \cup \{K_1 \cup K_2\}$
          $C'_{i-1} \leftarrow C'_{i-1} - \{K_1\}$
        end if
      end if
    end for
  end for
until $C'_i = \varnothing$

one of the elements of C, and the algorithm stops when a set C'_i is empty, that is, when there are no sets left that can be combined and yield a higher improvement. This algorithm produces $C'_1, C'_2, \ldots$ that contain subsets of optimizations having various numbers of optimizations turned on. For convenience sake, we classify them according to the number of optimizations turned on and store them in S_i, where i denotes the number of optimizations. The algorithm is given in Algorithm 2. Again, the algorithm stops when sets cannot be extended anymore.

6.2.3 Step 3: Selecting the best setting

We have generated candidate compiler settings in $\mathcal{S} = \bigcup_i S_i$ in the previous two steps generated based on an analysis of the reduced search space. Since there is no exact information about the compiler settings we generated, we examine them again in this step and identify one setting from them as our compiler setting. Since it is not feasible to execute all settings in S to select the overall best one, we restrict our search space again. Because we are looking for a most versatile compiler setting which performs well on a number of different applications, a setting is selected from those settings which have most optimizations turned on. Starting from the set S_i with the largest settings (i largest in the collection of settings found in the Step 2), only a limited number of sets S_k with $k < i$ are considered. We use 50 optimizations out of more than 60 optimizations. We did not employ a number of options that optimize floating point operations, like *fast-math* or *unsafe-math-optimizations* since these options violate IEEE and ISO specifications for mathematical functions. Therefore, in our case study, at most 50 of the largest compiler settings are considered. After execution, we calculate $E(s)$ for each setting s chosen. We choose the compiler setting s as our setting when $E(s)$ is maximal.

1	defer-pop
2	force-mem
3	force-addr
4	omit-frame-pointer
5	optimize-sibling-calls
6	inline
7	inline-functions
8	keep-static-consts
9	merge-constants
10	branch-count-reg
11	function-cse
12	strength-reduce
13	thread-jumps
14	cse-follow-jumps
15	cse-skip-blocks
16	rerun-cse-after-loop
17	rerun-loop-opt
18	gcse
19	gcse-lm
20	gcse-sm
21	loop-optimize
22	crossjumping
23	if-conversion
24	if-conversion2
25	delete-null-pointer-checks

26	expensive-optimizations
27	optimize-register-move
28	schedule-insns
29	schedule-insns2
30	sched-interblock
31	sched-spec
32	sched-spec-load
33	sched-spec-load-dangerous
34	caller-saves
35	move-all-movables
36	reduce-all-givs
37	peephole peephole2
38	reorder-blocks
39	reorder-functions
40	strict-aliasing
41	align-functions
42	align-labels
43	align-loops
44	align-jumps
45	rename-registers
46	cprop-registers
47	float-store
48	single-precision-constant

Table 6.1: Options of gcc 3.3.1 used.

6.3 Results

We have evaluated our methodology on the *gcc 3.3.1* compiler that has more than 60 compiler optimizations implemented, and we use 49 optimizations arranged into 48 factors. These options are given in Table 6.1. In this chapter, *prefetch-loop-arrays* is not used since this option disables the use of prefetch instructions. The option *delayed-branch* is also not present since our target architecture Pentium 4 does not support delayed branching.

An orthogonal array with 48 columns and 400 rows is used as the search space S. The compiler settings in S are executed with the seven benchmark programs from the *SPECint95* suite. We did not use *m88ksim* because it did not compile correctly for all settings on our platform. We have used a Pentium 4 at 2.8GHz as the architecture in this experiment.

We use 10 optimizations out of 48 optimizations for the initial set C_1. The 10 optimizations are selected as those optimizations that have a large effect according to Definition 6.1.7. The optimizations are shown in Table 6.2.

Table 6.3 shows the number of elements of the set C_i generated by Algorithm 1. We have

force-mem
omit-frame-pointer
optimize-sibling-calls
inline
inline-functions
keep-static-consts
function-cse
thread-jumps
gcse
align-labels

Table 6.2: Initial Optimizations in C_1

Threshold	C_1	C_2	C_3	C_4	C_5	C_6
0.004	3	72	-	-	-	-
0.005	6	14	12	3	2	-
0.006	7	5	2	1	12	6
0.007	10	3	-	-	-	-

Table 6.3: Number of partial sets in C

applied several threshold values to see which value is most adequate: the values 0.004, 0.005, 0.006, and 0.007 have been used for this experiment. A threshold value of 0.006 tends to construct a large number of subsets. This is because a small threshold leads to many combinations in the early steps in the algorithm. For instance, a threshold value of 0.004 has 72 elements in C_2. When there are many elements in C_k, there are few candidates in W^{k+1} since W^{k+1} consists of the optimizations that do not appear in C_k This prevents us to construct many subsets of optimizations. Hence, we choose 0.006 as the threshold value in this case study.

Next, we apply Algorithm 2 to the results for a threshold value of 0.006 in Table 6.3. The number of compiler settings generated by this algorithm is shown in Table 6.4.

There are sets S_i for $6 \leq i \leq 19$ in $\mathcal{S}$ and we choose the settings in S_{18} and S_{19} to execute in step 3 of our algorithm. Since the total number of elements in S_{18} and S_{19} is 45, it is feasible

S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
1	6	25	62	142	233	320

S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}	S_{19}
342	259	162	141	100	41	4

Table 6.4: Number of Compiler Settings in $\mathcal{S}$

omit-frame-pointer
inline
keep-static-consts
merge-constants
branch-count-reg
strength-reduce
cse-follow-jumps
rerun-cse-after-loop
gcse-lm
if-conversion2
sched-interblock
sched-spec-load-dangerous
reduce-all-givs
peephole
peephole2
reorder-blocks
reorder-functions
align-labels

Table 6.5: Setting found by our methodology

to execute all compiler settings in these sets. The setting which has the best improvement is chosen as our setting, denoted by s_{new}. In this case study, a setting in S_{18} has the best improvement. The setting turns on the 18 optimizations given in Table 6.5 and turns off the others. For comparison, option *-O1* defined by *gcc* uses 10 optimizations, option *-O2* uses 36 optimizations, and option *-O3* uses 38 optimizations.

Figure 6.1 shows the performance improvement of the setting generated by using our methodology, and the performance improvement of the option *-O1*, *-O2*, and *-O3*. The improvement of the generated optimization setting s_{new} is calculated as $E(s_{new})$. A first observation we make is that in several cases, *-O1* performs better than *-O2*. For *ijpeg*, it is even the case that *-O1* performs better than both *-O2* and *-O3*.

The generated compiler setting, denoted by *new* in Figure 6.1, performs better than the *-O1*, *-O2* and *-O3* switches for all benchmark programs we used, except for *li* when optimized with *-O3* which performs slightly better than our setting s_{new} (39.2% vs. 38.4%). In particular, *new* delivers the best performance for *perl*, giving almost twice as much improvement as *-O2* (18.4% vs. 10.5%). This shows that our methodology is capable of finding a compiler setting for a collection of programs that behave quite differently and would require different compiler settings when tuned individually.

6.4 Observations

In section 6.1, we have proposed our methodology to define a compiler setting. In that section, Algorithm 1 and Algorithm 2 have been given. Algorithm 1 uses a threshold value in the

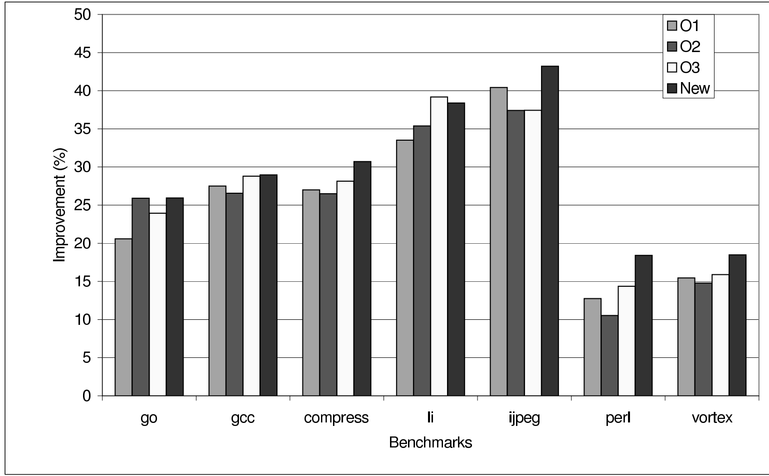


Figure 6.1: Improvements of New Setting and -*Ox* Options

evaluation predicate given in Definition 6.1.8. Several threshold values, 0.004, 0.005, 0.006, and 0.007, have been studied in the case study in Section 6.3. We have chosen a threshold value which produces the most subsets. If there are several threshold values that produce subsets that have the same maximum number of optimizations turned on, the threshold value that produces the largest number of subsets must been chosen.

In Table 6.3, we can also see that the results are different for each threshold value. A value of 0.006 has the most subsets that positively interact, but there are few subsets produced with a value of 0.007. This means that the algorithm is very sensitive to the threshold value. Therefore, it is necessary to choose this value carefully. Since the search space is already fixed beforehand, it is possible to have some trials of the algorithm to choose the most suitable threshold value which produces the most subsets.

The algorithm incrementally adds one optimization to an already derived set and checks whether this addition is significant. Note that it could be possible to find another set of, say, 2 (or more) optimizations which, combined with the previous set, makes all optimizations significant. This means that it can be the case that, for instance,

$$I(\{w_1, w_4, w_5\}, w_2) = 0 \quad \text{and}$$
$$I(\{w_1, w_4, w_5\}, w_6) = 0$$

which means that neither $\{w_1, w_2, w_4, w_5\}$ nor $\{w_1, w_4, w_5, w_6\}$ is in S_4. However, it could be the case that

$$I(\{w_1, w_4, w_5, w_6\}, w_2) = 1 \quad \text{or}$$
$$I(\{w_1, w_2, w_4, w_5\}, w_6) = 1 \quad \text{or}$$
$$I(\{w_2, w_4, w_5, w_6\}, w_1) = 1 \quad \text{or}$$
$$I(\{w_1, w_2, w_5, w_6\}, w_4) = 1 \quad \text{or}$$
$$I(\{w_1, w_2, w_4, w_6\}, w_5) = 1$$

By construction, however, $\{w_1, w_2, w_4, w_5, w_6\}$ will not be in S_5 while it is a significant set. However, the likelihood of the previously sketched situation to occur is low and therefore we do not consider these cases in our methodology. In a certain sense, the fact that the combination w_2 and w_6 interacts with $\{w_1, w_4, w_5\}$ is a higher order effect that we do not consider in this chapter. The other reason is that the proposed algorithm would grow exponentially in compute time.

In Algorithm 1 and Algorithm 2, the approximate effect of compiler optimizations defined in Definition 6.1.7 is used. The approximate effect is calculated by using execution times using an Orthogonal Array. From the definition of the effect, at least one execution result satisfying Definition 6.1.7 is necessary to calculate the effect. It is possible that there is no execution result which satisfies Definition 6.1.7 due to the reduced search space. When the algorithms encounter this situation, they regard the optimizations as non-interacting optimizations, and do not use them.

6.5 Conclusion

This chapter introduced a systematic method to generate a compiler setting that is optimized for a collection of applications. First, we detect several subsets of compiler optimizations that positively interact. Next, we combine these sets to yield complete compiler settings. Since the method can generate several compiler settings, we choose one of them by profiling them and selecting the one with best average improvement.

We have applied our method to *gcc 3.3.1*, which has a large number of compiler options, to examine the efficiency of our method. By using 400 compiler settings generated by an orthogonal array as a reduced search space, we have obtained real execution times for the SPECint95 benchmark suite which is a collection of programs with widely differing behavior. This reduced search space has been used to make all decisions in the algorithm. Our methodology produced a compiler setting that gives better performance than the standard *-O1*, *-O2*, and *-O3* options provided by *gcc*.

Our methodology enables us to define an optimized compiler setting without any knowledge of the available compiler options. Therefore, our methodology can be useful to determine a compiler setting for an arbitrary architecture, or for an arbitrary set of applications.

Chapter 7

Using random search to determine a compiler setting

In this chapter, we tackle the problem of the increasing number of potential optimizations which causes an exponentially growing complexity of interaction between these optimizations, see previous chapters. We try to achieve this by investigating other potential general optimization settings which perform equally well but utilize far fewer optimizations than the default settings. In order to find these general settings, we rely on iterative compilation techniques as described in the previous chapters.

This chapter consists of four sections. Section 7.1 discusses our experimental setup. Section 7.2 describes the effectiveness of the proposed methodology on single target applications. After this, we target multiple applications in Section 7.3. Finally, in Section 7.4, the results are summarized and a case is made for future research in this area.

7.1 The experimental environment

We use the *gcc 3.3.1* compiler which implements more than 60 optimizations [13]. Of these optimizations, we only potentially enable 47 optimizations, as shown in Table 7.1. The options which are left out are may violate IEEE floating point standards, like *fast-math*. We do not consider *omit-frame-pointer* that disables debugging on x86 architectures. Standard *-Ox* switches also do not turn on this optimization, although we have observed that it is one of the most important flags of *gcc* for this architecture. *inline* is not used since this option disables the inline directive in the application and we do not want to change the contexts of original applications. *prefetch-loop-arrays* is not also used since this option disables the use of prefetch instructions. *float-store* and *single-precision-constant* are left out to avoid generating invalid executables. The option *delayed-branch* is also not present since our target architecture Pentium 4 does not support delayed branching.

Therefore, $2^{44} \approx 10^{13}$ different settings are possible. In *gcc 3.3.1* for x86 that we used, option *-O0* already turns on certain loop optimizations. All improvements discussed below are with respect to this optimization level.

1	defer-pop	23	delete-null-pointer-checks
2	force-mem	24	expensive-optimizations
3	force-addr	25	optimize-register-move
4	optimize-sibling-calls	26	schedule-insns
5	inline-functions	27	schedule-insns2
6	keep-static-consts	28	sched-interblock
7	merge-constants	29	sched-spec
8	branch-count-reg	30	sched-spec-load
9	function-cse	31	sched-spec-load-dangerous
10	strength-reduce	32	caller-saves
11	thread-jumps	33	move-all-movables
12	cse-follow-jumps	34	reduce-all-givs
13	cse-skip-blocks	35	peephole peephole2
14	rerun-cse-after-loop	36	reorder-blocks
15	rerun-loop-opt	37	reorder-functions
16	gcse	38	strict-aliasing
17	gcse-lm	39	align-functions
18	gcse-sm	40	align-labels
19	loop-optimize	41	align-loops
20	crossjumping	42	align-jumps
21	if-conversion	43	rename-registers
22	if-conversion2	44	cprop-register

Table 7.1: Optimizations in *gcc 3.3.1*

We use the SPECint95 benchmark suite with the reference input data. We did not use *m88ksim* since this application did not compile correctly for every compiler setting. We measured the execution time using the Unix time command. We ran each application 6 times, removed the slowest and the fastest execution time and took the average of the remaining 4 times. A Pentium 4 at 2.8GHz is chosen as our target architecture.

7.2 The random generation of compiler settings for a single program

In this section, we investigate how the random generation of settings works in the presence of a large number of possible settings. Figure 7.1 shows the variation of improvement for a single program, namely, *147.vortex*, when we applied 5000 different compiler optimization settings. The vertical axis expresses the improvement in execution time, and the horizontal axis corresponds to the evaluated settings. We define the improvement of compiler setting s on application p as follows.

$$I_s(p) = \left(\frac{t_{s_0}(p) - t_s(p)}{t_{s_0}(p)} \right) \cdot 100\%, \tag{7.1}$$

where $t_{s_0}(p)$ stands for the execution time of compiler setting s_0 which is the setting without any optimization turned on, and $t_s(p)$ is the execution time using compiler setting s. The

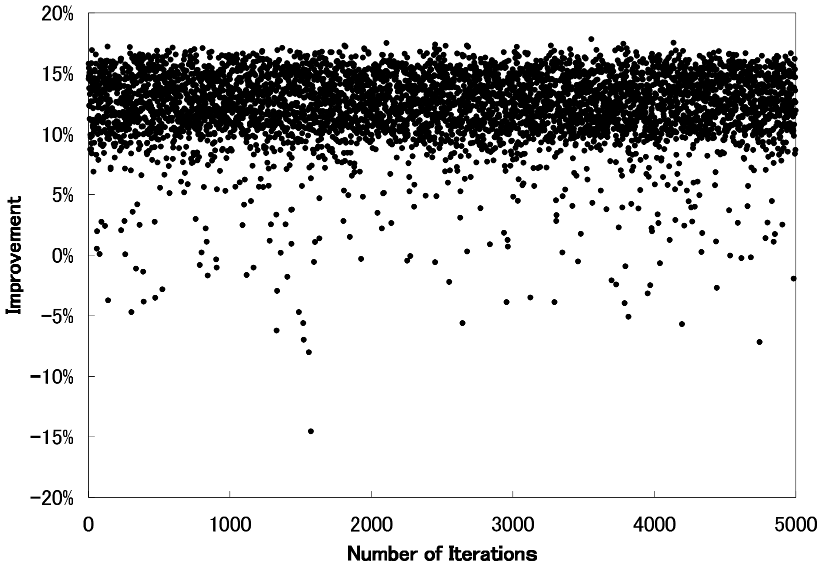


Figure 7.1: Improvement of 5000 Different Optimization Settings for 147.vortex

improvement in execution time varies between −15% and 17.5%. This shows the importance of choosing an appropriate optimization setting of the compiler.

According to Figure 7.1, the maximum improvement which can be obtained for *147.vortex*, is approximately 17.5%. During the time that the iterative compilation system investigates the search space, the function $f_0(n,p)$ keeps the current maximum value:

$$f_0(n,p) = max\{I_{s_t}(p) \ : \ 1 \leq t \leq n\}, \tag{7.2}$$

where n is the iteration number and s_t represents the compiler setting of the t-th iteration. Figure 7.2 plots Equation 7.2 and this graph represents how fast the randomly generated settings get close to the maximum improvement.

Figure 7.2 also shows that the value of f_0 increases by less than 1% after the 26th iteration. It seems appropriate to say that the random generation of settings identified one of the nearly optimal settings at the 26th iteration. Of course, that this number is so low does not guarantee that we will always succeed with so few iterations. Even worse, in general, the maximally achievable performance is not known in advance, therefore the stopping criterion should be carefully handled.

The criterion used in [40] stops the iterative compilation system in 400 iterations. Using the same criterion to the search space of two factors and the search space of far more factors, for instance, 46 factors, may not adequate. In order to maximize the opportunity to gain more improvement, we use a more strict stop criterion in this chapter. Our criterion stops the system when the current maximum $f_0(n,p)$ has not been changed by more than 1% after 100 iterations. This criterion is expressed as the function F_0, defined as

$$F_0(n,p) = f_0(n,p) - f_0(n-100,\ p) \tag{7.3}$$

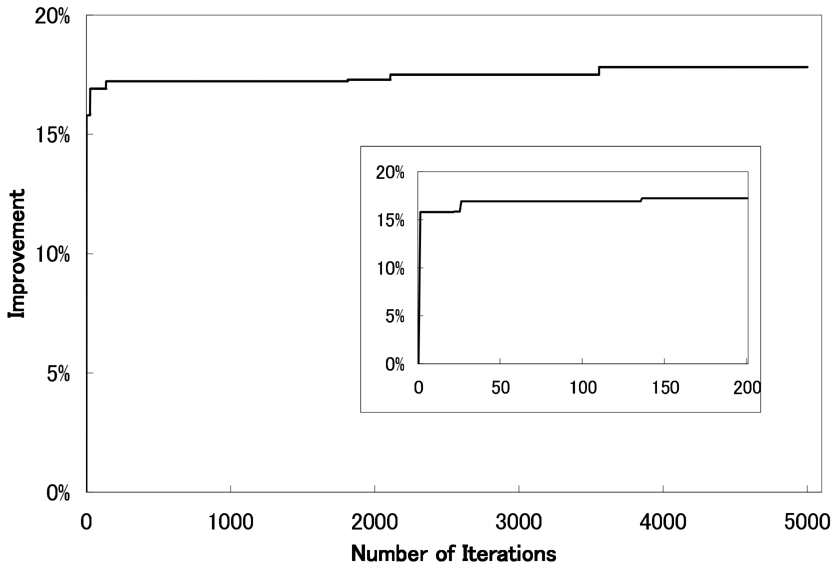


Figure 7.2: Improvement of 147.vortex

for $n > 100$. Using this stop criterion, the iterative compilation system for finding a nearly optimal setting for a single application is described in Figure 7.3.

We apply this iterative compilation system to seven integer programs in *SPEC95*. Figure 7.4 plots the value of f_0 in Equation 7.2 for each program. The graph shows that the highest number of iterations which is necessary to determine a compiler setting is 127 for these seven programs. We can indeed see that for each program at least 100 iterations are required.

Table 7.2 shows the setting found for each program in case that we do not include *omit-frame-pointer*. Options that are switched on are denoted by the symbol '1'. In Figure 7.5, *New1* shows the improvement obtained. In the same figure, *New2* shows the improvement obtained if we admit the option *omit-frame-pointer*. Clearly, better performance improvement is obtained in this case. This is to be expected, since this option leaves out instructions to set up the frame pointer and moreover can free a register that would be used to hold this pointer. Since the x86 architecture has few visible registers, instruction scheduling can be greatly improved and less spill code needs to be inserted. The improvements obtained by *-O1*, *-O2*, and *-O3* are shown for comparison.

As can be seen, the setting derived by our technique performs better than *-O1*, *-O2*, and *-O3* for each benchmark program. Of course, *-O1*, *-O2*, and *-O3* are general compiler settings and as such not difficult to defeat if *gcc* is specifically tuned for one program. Therefore, in the next section, we will investigate whether our method can also be used to derive just one setting for multiple programs.

Let p be an application.
n is the iteration number.

- Repeat:
  - Measure the execution time of p compiled with a randomly generated optimization setting s_n.
  - If $n > 100$
    - * Compute $F_0(n, p)$.
- until $F_0(n, p) \leq 1\%$
- $I_{s_{max}}(p) = I_{s_n}(p)$

Figure 7.3: The Iterative Compilation System

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
099.go	1				1	1	1			1		1	1	1	1	1			
126.gcc		1	1	1	1	1			1	1	1	1			1		1		1
129.compress	1		1	1	1	1	1	1		1		1					1		
130.li		1	1	1	1	1	1	1		1	1		1		1		1		1
132.ijpeg	1	1		1		1		1	1	1	1	1					1		
134.perl	1		1				1	1		1		1	1	1		1	1	1	
147.vortex		1				1					1	1				1	1		

	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
099.go			1			1		1	1	1	1	1	1						
126.gcc	1	1	1		1	1		1	1		1			1		1	1		
129.compress	1	1		1		1	1	1		1		1	1	1	1	1	1	1	
130.li	1					1	1		1		1		1	1	1			1	
132.ijpeg		1	1				1	1				1		1			1	1	
134.perl			1	1	1		1		1		1		1			1	1	1	1
147.vortex						1	1	1	1	1	1		1						1

	39	40	41	42	43	44													
099.go	1	1																	
126.gcc		1			1	1													
129.compress	1	1	1		1														
130.li			1	1	1														
132.ijpeg	1	1	1		1	1													
134.perl	1	1	1	1															
147.vortex	1	1	1	1	1	1													

Table 7.2: Optimization Settings Per Program

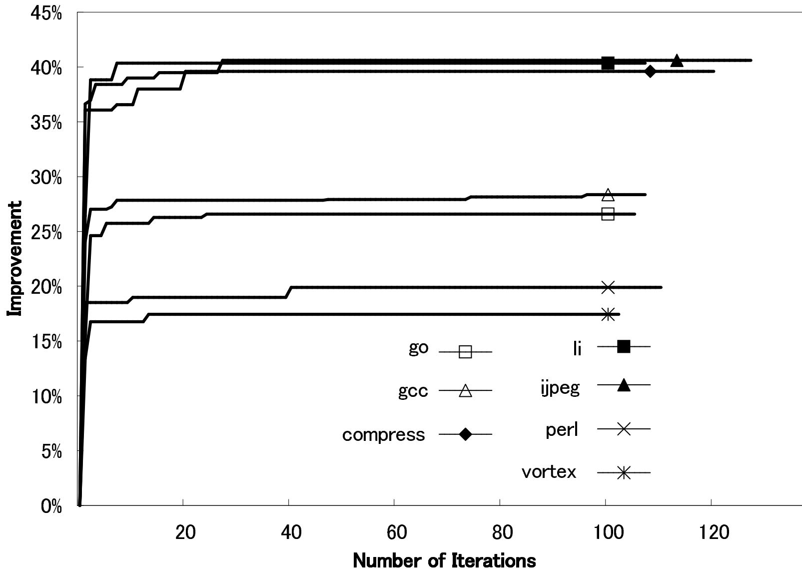


Figure 7.4: Improvement of Programs

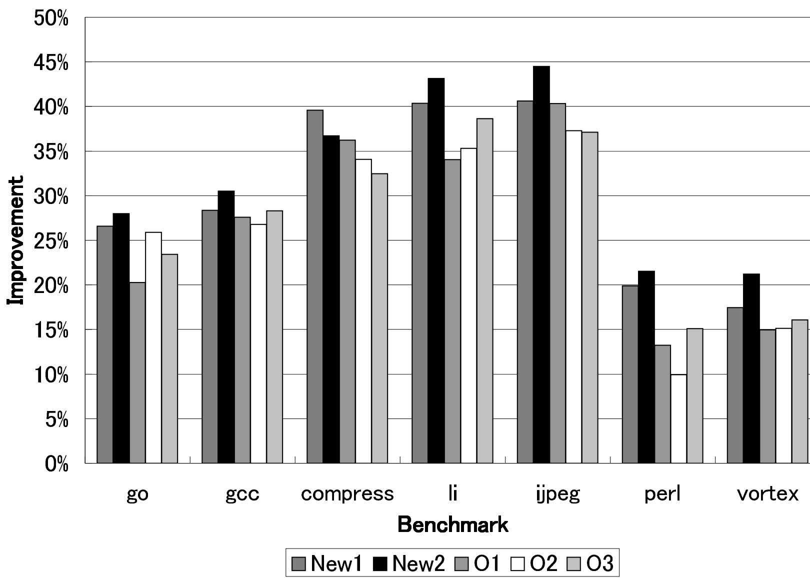


Figure 7.5: Improvement of the Generated Optimization Settings

Let P be a set of target programs.
n is the iteration number.

- Estimate $I_{max}(p)$ with the procedure in Figure 7.3 for each program in P.
- Repeat:
  - Randomly generate optimization setting s.
  - Measure the execution time of each $p \in P$ compiled with this setting s.
  - If $n > 100$
    * Compute $F(n,P)$.
- Until $F(n,P) \leq 1\%$

Figure 7.6: The Iterative Compilation System for Multiple Programs

7.3 Multiple program optimization

In this section, we extend our algorithm to find a compiler setting for many programs at the same time. Randomly generated settings are evaluated by using execution times for several programs. Since it is impossible to compare the improvement in execution time between different programs, we normalize the effect of the compiler settings. The improvement in execution time is normalized as the accomplished rate of the maximum improvement for each application. For instance, when the improvement of setting A is 3% and the maximum improvement, which can be gained for the application, is 30%, the normalized value of setting A becomes $\frac{3}{30} \cdot 100\% = 10\%$. We denote the normalized effect of a setting s on a program p as $e_s(p)$, and it is defined as follows.

$$e_s(p) = \frac{I_s(p)}{I_{s_{max}}(p)} \cdot 100\% \tag{7.4}$$

where $I_{s_{max}}(p)$ represents the maximum improvement of application p.

Since we do not know the maximum improvement of an application in advance, we need to estimate it. In this chapter, we use the nearly maximum improvement identified in Section 7.2. After the normalization, it is possible to evaluate an optimization setting over multiple programs. We take the minimum improvement over all applications for each setting as its effect. For a set P of programs, the effect of a setting s over the programs in P is expressed by $E_s(P)$.

$$E_s(P) = \min\{e_s(p) \ : \ p \in P\} \tag{7.5}$$

The function $f(n,P)$ is defined to record the current maximum improvement thus far obtained, while the iterative compilation system carries on.

$$f(n,P) = \max\{E_{s_t}(P) \ : \ 1 \leq t \leq n\} \tag{7.6}$$

Number of Programs	Program Name
P_2	*099.go 126.gcc*
P_3	*099.go 126.gcc 129.compress*
P_4	*099.go 126.gcc 129.compress 130.li*
P_5	*099.go 126.gcc 129.compress 130.li 132.ijpeg*
P_6	*099.go 126.gcc 129.compress 130.li 132.ijpeg 134.perl*
P_7	*099.go 126.gcc 129.compress 130.li 132.ijpeg 134.perl 147.vortex*

Table 7.3: Combinations of Programs

where n is the iteration number and s_t expresses the compiler setting used in the t-th iteration. We define the stop criterion of the iterative compilation system to find the estimated nearly optimal setting for multiple target programs, as follows.

$$F(n,P) = f(n,P) - f(n-100,P) \tag{7.7}$$

for $n > 100$.

In Figure 7.6, the iterative compilation system to search for an optimal compiler setting for multiple programs is shown using this stop criterion. We applied this procedure to the *gcc 3.3.1* compiler and the seven integer programs in *Spec95*. To see the effect of the number of programs used on the magnitude of the value $E_{s_{max}}(P_k)$ and on the number of iterations required to reach the stop criterion, we grouped the benchmark programs into six sets P_2, $\ldots$, P_7, where each P_k contains k programs. The programs included in each P_k are shown in Table 7.3. In Figure 7.7, $f(n,P_k)$ is plotted for each k. We observe that the performance of the final setting becomes lower as the number of programs in P increases. The graph also shows that the iteration always stops in at most 114 iterations for the different P_k. We can see that P_2 requires the most iterations to reach the stop criterion. It seems that the number of programs does not affect the required number of iterations. Figure 7.8 shows $E_{s_{max}}(P_k)$ for each k. We also show $E_{O_x}(P_k)$ for *-O1*, *-O2*, and *-O3* for comparison. The graph shows that the nearly optimal settings obtained with the procedure in Figure 7.6 reach better values than the *-Ox* options.

Figure 7.9 shows the improvement of the generated setting for P_7 (shown as *New1*) on each program in P_7 together with the improvements from the *-Ox* options. The generated setting does not have low performance for any program, although it cannot optimize *ijpeg* as much as *-O1* does. Nevertheless, it reaches the same performance as *-O2* and *-O3*. For completeness sake, we have also included *omit-frame-pointer* in the iterative compilation process and the resulting performance improvements of this setting, denoted by *New2*, are shown in the same figure. Not surprisingly, we can see in this figure that the obtained setting performs better than the default *-Ox* settings. These results show that iterative compilation also finds a nearly optimal setting for multiple programs.

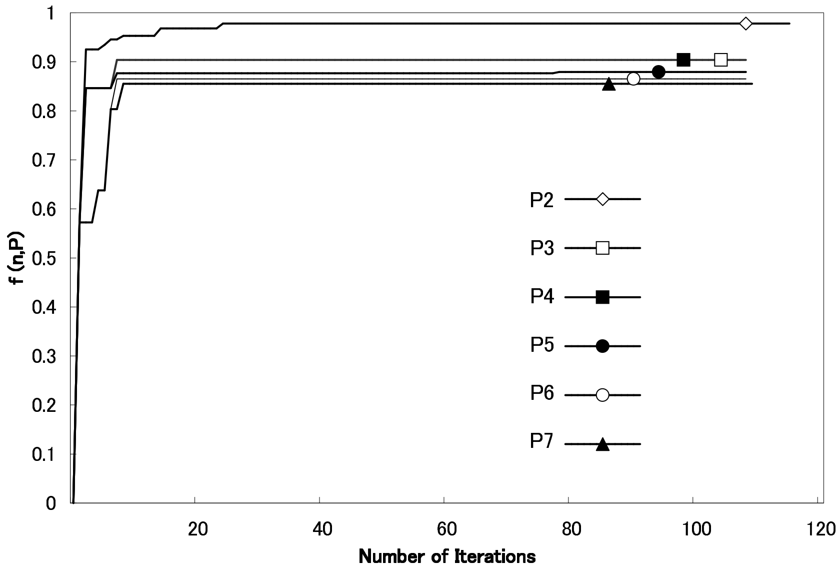


Figure 7.7: Improvement of the Programs in P_k

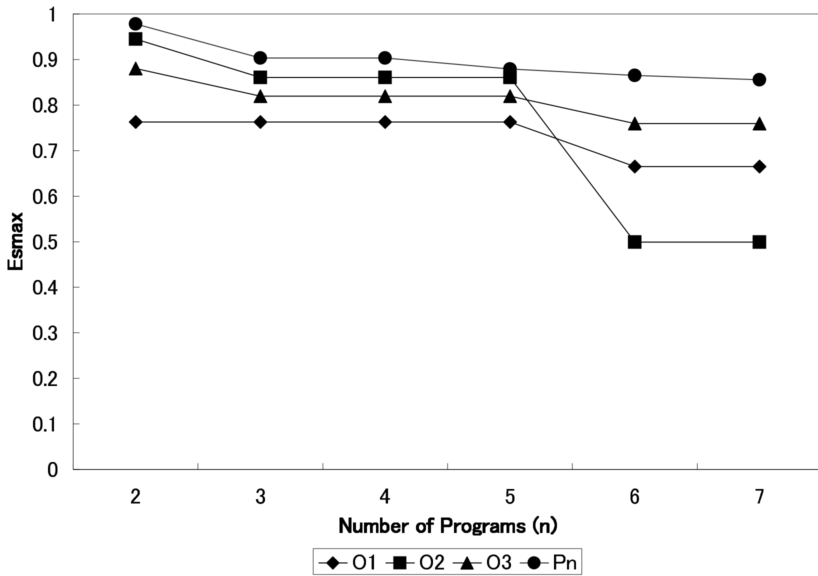


Figure 7.8: Transition of $E_{s_{max}}$

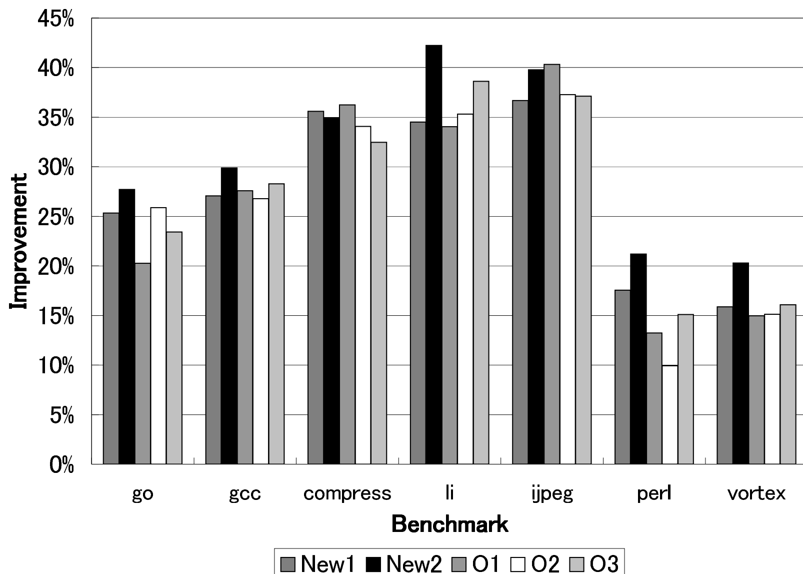


Figure 7.9: Improvement of Identified Setting for P_7

7.3.1 The effectiveness of the new setting

In order to show the effectiveness of the new setting, we apply this setting for other programs than the seven programs in *SPEC95* also. The results are shown in Figure 7.10. The list of programs used in the graph and a short description of them is shown in Table 7.4.

As can be seen from Figure 7.10, the newly generated compiler setting (without *omit-frame-pointer*) is definitely comparable to the standard *-Ox* settings, even if we compile programs which were not part of the training set used to derive this setting.

More importantly, in Table 7.5 the optimizations can be found which are enabled by the new settings and by *-Ox*. From this table, we can see that although *-O1* utilizes just 10 optimizations, *-O2* and *-O3* require 36 and 38 optimizations, respectively, to be turned on. The general compiler setting derived from the iterative process only uses 25 optimizations. Hence, we can clearly see that about 40% of the optimizations used by *-O2* and *-O3* do not contribute to better overall performance, leading to the question whether the common practice of turning on more and more optimizations in modern compilers really contributes to better performing compilers.

7.4 Conclusion

From the results in this chapter, two major conclusions can be drawn. First we have shown that iterative compilation can also be employed to obtain general optimization settings for multiple programs. For this, we applied this process to a set of integer programs form the

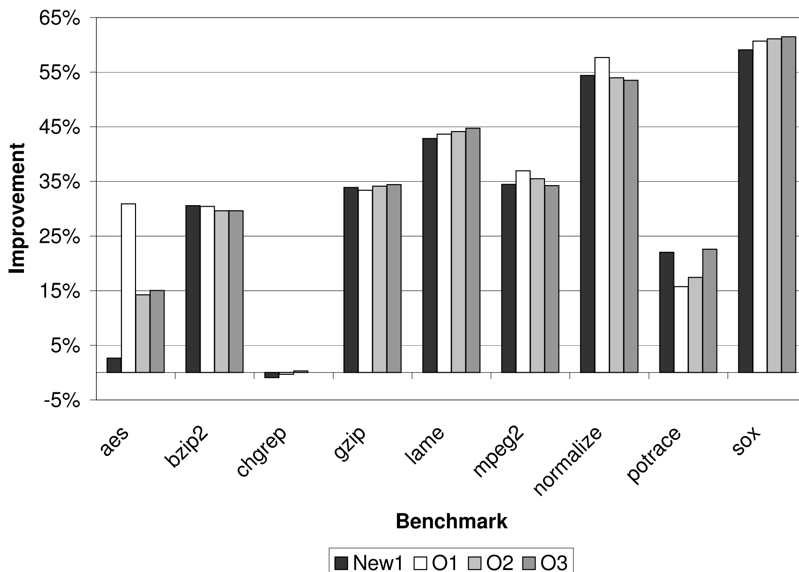


Figure 7.10: Improvement of the Generated Optimization Setting for Other Programs

bzip2-1.0.2	Compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding.
chgrep-1.2.4	Search the input files for old string and changes them to new string.
gzip-1.2.4	Reduce the size of the named files using Lempel-Ziv coding (LZ77).
lame-3.89	A program which can be used to create compressed audio files.
mpeg2decode	Takes one or more ISO/IEC DIS 13818-2 [1] MPEG-2 video bit streams and converts them to uncompressed video.
normalize-0.7.6	Adjust the volume of wav audio files to a standard volume level.
potrace-1.4	A utility for tracing a bitmap, which means, transforming a bitmap into a smooth, scalable image.
sox-12.17.4	Convert audio files to other audio file formats.

Table 7.4: List of Programs

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
New1		1			1	1		1	1	1	1	1		1	1	1			
New2		1		1	1	1		1	1		1	1				1			
O1	1				1		1		1		1								1
O2	1	1		1	1		1		1	1	1	1	1	1	1	1	1	1	1
O3	1	1		1	1		1		1	1	1	1	1	1	1	1	1	1	1
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
New1	1					1				1	1	1		1	1	1	1		1
New2				1	1						1	1		1		1			
O1	1	1	1																
O2	1	1	1	1	1	1	1	1	1	1			1			1	1	1	1
O3	1	1	1	1	1	1	1	1	1	1			1			1	1	1	1
	39	40	41	42	43	44													
New1	1		1	1	1														
New2		1																	
O1						1													
O2	1	1	1	1		1													
O3	1	1	1	1	1	1													

Table 7.5: Optimization Settings

SPECint95 benchmark suite. Our results show that the new compiler setting thus derived shows comparable performance to the default *-Ox* settings and even better performance if *omit-frame-pointer* is included as a possible optimization choice. We verified these results by running on other programs found on the internet. These experiments also confirmed our conclusion that a generic compiler setting can be obtained using a rather simple (black box) approach.

As a second and major conclusion, we found that the number of optimizations required to obtain a good performing general compiler setting is far less than the number of optimizations used in the standard *-O2* and *-O3* settings. Moreover, preliminary experimentation suggests that turning off many switches in these default settings do not affect their performance. This raises the question whether the common practice of adding more and more optimization options to modern compilers is the right way forward. This practice makes development and maintenance of compilers more difficult. Future research will have to be focused more and more on the interaction of optimization settings rather than on constructing new and more advanced optimization choices.

Chapter 8

Conclusion

In this thesis, we have first introduced an approach to the problem of selecting a compiler optimization setting for a single application or sets of applications. Our approach is based on generating many compiler settings using an Orthogonal Array, and an analysis of the resulting execution times. Firstly, we applied statistical analysis, namely, the main effect and the Mann-Whitney test, to collected execution times to determine effective compiler options. The main effect and the Mann-Whitney test only identify few optimizations which have a sufficiently large effect. To detect the optimizations which have a large effect among all optimizations, we proposed a methodology which applies statistical analysis iteratively to find a complete compiler setting. The methodology using the Mann-Whitney test is also applied to reduce code size, which is an important optimization target for embedded systems.

Secondly, we studied another approach which takes into account interaction between compiler options. It is well known that compiler optimizations positively or negatively interact with each other. However, there exists no clear definition of interaction so that this study introduced a formal definition of this interaction, used to select compiler optimizations.

Finally, we applied iterative compilation using a random search algorithm to identify the optimal compiler setting for a single application. This study shows that we can find a nearly optimal compiler setting with a reasonable number of executions. This approach enables us to know the potential effect of the compiler optimizations we are investigating. We also showed that this methodology can identify a compiler setting which is effective for several/many applications. The results can be useful to evaluate the standard compiler settings (e.g., *-Ox*).

These three approaches generate compiler optimization settings which outperform the standard setting *-O3* from the *gcc* compiler on compiling applications in *SPECint2000*. These results show the adequacy of our approaches for the problem of finding a good compiler settings. Our approaches are completely separated from the implementation of the compiler as well optimization target, which can be execution time, code size, energy consumption, and so on. Therefore, our proposed methodologies can be applied on the top of any compiler for any target architecture for any optimization purpose.

Currently, we are only taking into account the sequence of optimizations although the problem of optimization strategy can be affected by other factors, application order, parameter setting for certain optimizations like loop unrolling. Future work will address the extension

of our approach to include these other factors.

Appendix A

Compiler Options

Table A.1 shows the list of all optimization options implemented in *gcc 3.3.1*. '1' shows which option is turned on explicitly for each of the *Ox* settings. '*' shows the options which are implicitly turned on when *gcc* does the *Ox* optimization. Table A.2 shows the *gcc 3.4.3* options which are not in Table A.1. We employed part of them for our experiments. The list of optimizations which we used for our experiments are different between chapters. In each chapter, we describe which of the options are used and for what reasons.

The options after the 55th in Table A.1 are not used throughout this thesis. The reasons are as follows. We did not use the 55th option *no-default-inline* since it is the option for applications written in C++ which we did not use in our experiments. The 56th option is not included since this option intends to keep the inlined functions in the executable code, and it does not do any optimization. The 57th option, *guess-branch-probability* uses a randomized model to guess branch probability which means the produced code may be different for each compile time. Since we need static results for our experiments, we did not use this option. We did not use options 58 through 63 since they are experimental options in this version of *gcc*. We also did not use the 64th option, *fast-math*, since it may not produce correct results all the time. Also, the options 65 through 69 were not used since they are used when *fast-math* is enabled.

	Option	O1	O2	O3	Os
1	defer-pop	1	1	1	1
2	force-mem		1	1	1
3	force-addr				
4	omit-frame-pointer				
5	optimize-sibling-calls		1	1	1
6	inline	*	*	*	*
7	inline-functions			1	
8	merge-constants	1	1	1	1
9	strength-reduce		1	1	1
10	thread-jumps	1	1	1	1
11	cse-follow-jumps		1	1	1
12	cse-skip-blocks		1	1	1
13	rerun-cse-after-loop		1	1	1
14	rerun-loop-opt		1	1	1
15	gcse		1	1	1
16	gcse-lm		1	1	1
17	gcse-sm		1	1	1
18	loop-optimize	1	1	1	1
19	crossjumping	1	1	1	1
20	if-conversion	1	1	1	1
21	if-conversion2	1	1	1	1
22	delete-null-pointer-checks		1	1	1
23	expensive-optimizations		1	1	1
24	optimize-register-move		1	1	1
25	schedule-insns		1	1	1
26	schedule-insns2		1	1	1
27	sched-interblock		1	1	1
28	sched-spec		1	1	1
29	sched-spec-load				
30	sched-spec-load-dangerous				
31	caller-saves		1	1	1
32	move-all-movables				
33	reduce-all-givs				
34	peephole	1	1	1	1
35	peephole2	1	1	1	1
36	reorder-blocks		1	1	
37	reorder-functions		1	1	1
38	strict-aliasing		1	1	1
39	align-functions		1	1	
40	align-labels		1	1	
41	align-loops		1	1	
42	align-jumps		1	1	
43	rename-registers			1	
44	cprop-registers	1	1	1	1
45	unroll-loops				
46	prefetch-loop-arrays				
47	function-sections				
48	data-sections				
49	float-store				
50	single-precision-constant				
51	delayed-branch	1	1	1	1
52	keep-static-consts				
53	function-cse	*	*	*	*
54	branch-count-reg	*	*	*	*
55	no-default-inline				
56	keep-inline-functions				
57	guess-branch-probability	1	1	1	1
58	branch-probabilities				
59	new-ra				
60	tracer				
61	ssa				
62	ssa-ccp				
63	ssa-dce				
64	fast-math				
65	math-errno				
66	unsafe-math-optimizations				
67	finite-math-only				
68	trapping-math				
69	signaling-nans				

Table A.1: All Options in gcc 3.3.1

	Option	O1	O2	O3	Os
1	web				
2	unit-at-a-time				
3	peel-loops				
4	unswitch-loops				
5	old-unroll-loops				
6	branch-target-load-optimize				
7	branch-target-load-optimize2				
8	gcse-las				

Table A.2: Additional Options in gcc 3.4.3

Appendix B

Benchmark Suites

In this thesis, three benchmark suites are used: Mediabench, SPEC95 and SPEC2000.

Chapter 5 employs Mediabench. We used all applications in this benchmark suite, however two of them are dropped at some platforms due to the compile errors.

Chapter 6 and Chapter 7 use SPEC95. From this benchmark suite, we used specially the integer programs. All these programs in SPEC CINT95 are written in C. We used them except for *124.m88ksim*, since this application had some problems to compile.

SPEC CINT2000 is used in the remainder of the chapters. This benchmark suite contains 12 benchmarks, and we used 7 of them in our experiments. We have chosen again the applications written in C and dropped 4 applications, 176.gcc, 186.crafty, 252.eon, and 253.perlbmk, due to compile errors. Three applications, 168.wupwise, 171.swim, and 179.art, in the SPEC CFP2000 benchmark suite are used in Chapter 4. The rest of applications are left out due to compile errors.

Mediabench

- adpcm (C)
  ADPCM stands for Adaptive Differential Pulse Code Modulation. It is a family of speech compression and decompression algorithms. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The ADPCM code used is the Intel/DVI ADPCM code which is being recommended by the IMA Digital Audio Technical Working Group.

- epic (C)
  EPIC (Efficient Pyramid Image Coder) is an experimental image data compression utility written in the C programming language. The compression algorithms are based on a biorthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding on conventional (i.e., non-floating point) hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition).

- g721 (C)
  The files in this package comprise ANSI-C language reference implementations of the

CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions. This source code is released by Sun Microsystems, Inc. to the public domain.

- ghostscript (C)
  Ghostscript is the name of a set of software that provides: (1) An interpreter for the PostScript (TM) language, and (2) A set of C procedures (the Ghostscript library) that implement the graphics capabilities that appear as primitive operations in the PostScript language, and (3) An interpreter for Portable Document Format (PDF) files.

- gsm (C)
  GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits; for compatibility with typical UNIX applications, our implementation turns frames of 160 16-bit linear samples into 33-byte frames (1650 Bytes/s). The quality of the algorithm is good enough for reliable speaker recognition; even music often survives transcoding in recognizable form (given the bandwidth limitations of 8 kHz sampling rate).

- jpeg (C)
  This package contains C software to implement JPEG image compression and decompression. JPEG is a standardized compression method for full-color and gray-scale images. JPEG is intended for compressing "real-world" scenes; line drawings, cartoons and other non-realistic images are not its strong suite. JPEG is lossy, meaning that the output image is not exactly identical to the input image.

- mesa (C)
  Mesa is a 3-D graphics library with an API which is very similar to that of OpenGL.

- mpeg2 (C)
  mpeg2play is a player for MPEG-1 and MPEG-2 video bitstreams. It is based on mpeg2decode by the MPEG Software Simulation Group. In mpeg2decode the emphasis is on correct implementation of the MPEG standard and comprehensive code structure. The latter is not always easy to combine with high execution speed. Therefore a version has been derived which is optimized for higher decoding and display speed at the cost of a less straightforward implementation and slightly non-compliant decoding. In addition all conformance checks and some fault recovery procedures have been omitted from mpeg2play.

- pegwit (C)
  Pegwit is a program for performing public key encryption and authentication. It uses an elliptic curve over GF(2^{255}), SHA1 for hashing, and the symmetric block cipher square.

- pgp (C)
  PGP uses "message digests" to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message (MD5). To encrypt data,

it uses a block-cipher IDEA, RSA for key management and digital signatures. A session key is generated for an individual message and the message is encrypted by IDEA using the session key and the session key is encrypted using RSA.

- rasta (C)
  RASTA is a program for the rasta-plp processing and it supports the following front-end techniques: PLP, RASTA, and Jah-RASTA with fixed Jah-value. The Jah-Rasta technique handles two different types of harmful effects for speech recognition systems, namely additive noise and spectral distortion, simultaneously, by bandpass filtering the temporal trajectories of a non-linearly transformed critical band spectrum.

SPEC CINT95

- 099.go (C)
  Go plays the game of go against itself. The benchmark is stripped down version of a successful go-playing computer program. The benchmark is implemented in ANSI C (with function prototypes). There is a great deal of pattern matching and look-ahead logic. As is common in this type of program, up to a third of the run-time can be spent in the data-management routines.

- 124.m88ksim (C)
  M88ksim is a simulator written in C. It can measure the number of clocks which an 88100 microprocessor would take to execute a program. It is essentially an integer program, although the exact instruction mix of the simulator depends on the program being simulated. The simulator can pass system call requests from the simulated program through to the host system running the simulator.

- 126.gcc (C)
  gcc is based on the GNU C compiler version 2.5.3 distributed by the Free Software Foundation. The benchmark measures the time it takes for the GNU C compiler to convert a number of its pre-processed source files into optimized Sparc assembly language (.s files) output.

- 129.compress (C)
  Compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each files is replaced by one with the extension .Z. If no files are specified, standard input is compressed to standard output. Compressed files can be restored to their original form using Un- compress. The amount of compression obtained depends upon the size of the input, the number of bits per character, and the distri- bution of common substrings. Typically, text such as source code or English is reduced by 50-60%. Compression is gen- erally better than pack (Huffman coding) or compact (adaptive Huffman) and takes less time to compute.

- 130.li (C)
  Li is a Lisp interpreter written in C. The workload used is a translation of the Gabriel benchmarks by John Shakshober of DEC. These are boyer, browse, ctak, dderiv, deriv, destru-mod, destru, div2, fft, puzzle, tak, takl, takr, triang. These are from the public

domain and are described in the "Performance Evaluation of Lisp Systems" by Richard Gabriel.

- 132.ijpeg (C)
  Image compression and decompression on in-memory JPEG images. The benchmark application performs a series of compressions at differing quality levels over a variety of images. The workload is taken from the behavior of someone seeking the best tradeoff between space and time for a variety of images.

- 134.perl (C)
  This is an interpreter for the Perl language. The inputs are the scripts that performing some basic math calculations and word lookups in associative arrays. As much as 10% of the time can be spent in routines commonly found in libc.a: malloc, free, memcpy, etc.

- 147.vortex (C)
  The benchmark 147.vortex is a subset of a full object oriented database program called VORTEx. VORTEx stands for "Virtual Object Runtime EXpository". Transactions to and from the database are translated through a schema. A schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application. The schema as provided with the benchmark is pre-configured to manipulate three different databases: mailing list, parts list, and geometric data. Both litle-endian and big-endian binaries for the schema are provided. The benchmark builds and manipulates three separate, but inter-related databases based on the schema. The size of the database is scalable, and for CINT95 guidelines has been restricted to about 40 Mbytes. VORTEx been modified to not commit transactions to memory in order to remove input-output activity from this CINT95 (component) benchmark. The workload of VORTEx has been modeled after common object-oriented database benchmarks with modifications to vary the mix of transactions.

SPEC CINT2000

- 164.gzip (C)
  gzip (GNU zip) is a popular data compression program written by Jean-Loup Gailly for the GNU project. 'gzip' uses Lempel-Ziv coding (LZ77) as its compression algorithm.

  SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to just the CPU and the memory subsystem.

- 175.vpr (C)
  VPR is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs.

- 176.gcc (C)
  176.gcc is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. 176.gcc has had its inlining heuristics altered slightly, so as to inline more code than would be typical on a Unix system in 1997. It is expected that this effect will be more typical of compiler usage in 2002. This was done so that 176.gcc would spend more time analyzing its source code inputs, and use more memory. Without this effect, 176.gcc would have done less analysis, and needed more input workloads to achieve the run times required for SPECint2000.

- 181.mcf (C)
  A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic.

  The program is designed for the solution of single-depot vehicle scheduling (sub-)problems occurring in the planning process of public transportation companies. It considers one single depot and a homogeneous vehicle fleet. Based on a line plan and service frequenciesd, so-called timetabled trips with fixed departure/arrival locations and times are derived. Each of this timetabled trip has to be serviced by exactly one vehicle. The links between these trips are so-called dead-head trips. In addition, there are pull-out and pull-in trips for leaving and entering the depot.

- 186.crafty (C)
  Crafty is a high-performance Computer Chess program that is designed around a 64-bit word. It runs on 32 bit machines using the “long long” (or similar, as _int64 in Microsoft C) data type. It is primarily an integer code, with a significant number of logical operations such as and, or, exclusive or and shift. It can be configured to run a reproducible set of searches to compare the integer/branch prediction/pipe-lining facilities of a processor.

- 197.parser (C)
  The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words.

  The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. The parser is robust; it is able to skip over portions of the sentence that it cannot understand, and assign some structure to the rest of the sentence. It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words.

- 252.eon (C++)
  Eon is a probabilistic ray tracer. It sends a number of 3D lines (rays) into a 3D polygonal model. Intersections between the lines and the polygons are computed, and new lines are generated to compute light incident at these intersection points. The final result of the computation is an image as seen by camera. The computational demands of

the program are much like a traditional deterministic ray tracer as described in basic computer graphics texts, but it has less memory coherence because many of the random rays generated in the same part of the code traverse very different parts of 3D space.

- 253.perlbmk (C)
  253.perlbmk is a cut-down version of Perl v5.005-03, the popular scripting language. SPEC's version of Perl has had most of OS-specific features removed. In addition to the core Perl interpreter, several third-party modules are used: MD5 v1.7, MHonArc v2.3.3, IO-stringy v1.205, MailTools v1.11, TimeDate v1.08.

- 254.gap (C)
  This program implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).

- 255.vortex (C)
  VORTEx is a single-user object-oriented database transaction benchmark which which exercises a system kernel coded in integer C. The VORTEx benchmark is a derivative of a full OODBMS that has been customized to conform to SPEC CINT2000 (component measurement) guidelines. The benchmark 255.vortex is a subset of a full object oriented database program called VORTEx. Transactions to and from the database are translated through a schema. A schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application.

- 256.bzip2 (C)
  256.bzip2 is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

- 300.twolf (C)
  The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells) which constitute the microchip. The placement problem is a permutation. Therefore, a simple or brute force exploration of the state space would take an execution time proportional to the factorial of the input size. For problems as small as 70 cells, a brute force algorithm would take longer than the age of the universe on the world's fastest computer. Instead, the TimberWolfSC program uses simulated annealing as a heuristic to find very good solutions for the row-based standard cell design style. In this design style, transistors are grouped together to form standard cells. These standard cells are placed in rows so that all cells of a row may share power and ground connections by abutment. The simulated annealing algorithm has found the best known solutions to a large group of placement problems. The global router which follows the placement step interconnects the microchip design. It utilizes a constructive algorithm followed by iterative improvement.

SPEC FP2000

- 168.wupwise (Fortran 77)
  "wupwise" is an acronym for "Wuppertal Wilson Fermion Solver", a program in the area of lattice gauge theory (quantum chromodynamics).

  Lattice gauge theory is a discretization of quantum chromodynamics which is generally accepted to be the fundamental physical theory of strong interactions among the quarks as constituents of matter. The most time-consuming part of a numerical simulation in lattice gauge theory with Wilson fermions on the lattice is the computation of quark propagators within a chromodynamic background gauge field. These computations use up a major part of the world's high performance computing power.

- 171.swim (Fortran 77)
  Benchmark weather prediction program for comparing the performance of current supercomputers. The model is based on the paper, "The Dynamics of Finite-Difference Models of the Shallow-Water Equations", by Robert Sadourny, J. ATM. SCIENCES, VOL 32, NO 4, APRIL 1975.

  Adapted by SPEC for use in the SPEC CPU suites as an example of a compute intensive floating point program that was once relegated only to "supercomputers" but can now be done on current computer systems.

- 172.mgrid (Fortran 77)
  172.mgrid demonstrates the capabilities of a very simple multigrid solver in computing a three dimensional potential field.

  Adapted by SPEC from the NAS Parallel Benchmarks with modifications for portability and a different workload.

- 173.applu (Fortran 77)
  Solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit psuedo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. This scheme is functionally equivalent to a nonlinear block SSOR iterative scheme with lexicographic ordering. Spatial discretization of the differential operators are based on second-order accurate finite volume scheme. As a result, the degree of exploitable parallelism during this phase is limited to $O(N^2)$ as opposed to $O(N^3)$ in other phases and its spatial distribution is non-homogenous.

- 177.mesa (C)
  Mesa is a free OpenGL work-alike library, since it supports a generic frame buffer. It can be configured to have no OS or window system dependencies. Any number of client programs can be written to stress FP, scalar or memory performance (or a mix). Output can be written to image files for verification.

- 178.galgel (Fortran 90)
  This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids.

- 179.art (C)
  The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image. A window corresponding to the size of the learned objects is scanned across the scanfield image and serves as input for the neural network. The neural network attempts to match the windowed image with one of the images it has learned.

- 183.equake (C)
  The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.

- 187.facerec (Fortran 90)
  This is an implementation of the face recognition system described in [41].

- 188.ammp (C)
  The benchmark runs molecular dynamics (i.e. solves the ODE defined by Newton's equations for the motions of the atoms in the system) on a protein-inhibitor complex which is embedded in water. The energy is approximated by a classical potential or "force field". The protein is HIV protease complexed with the inhibitor indinavir. There are 9582 atoms in the water and protein making this representative of a typical large simulation. This benchmark is derived from published work on understanding drug resistance in HIV.

- 189.lucas (Fortran 90)
  Performs the Lucas-Lehmer test to check primality of Mersenne numbers 2^{p-1}, using arbitrary-precision (array-integer) arithmetic. This is accomplished by the Mersenne-mod squaring via the discrete weighted transform technique of Crandall and Fagin. A data-local, cache-friendly FFT is used to efficiently perform the large-integer squaring of the Lucas-Lehmer iterations.

- 191.fma3d (Fortran 90)
  FMA-3D is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. As an explicit code, the program is appropriate for problems where high rate dynamics or stress wave propagation effects are important. In contrast to programs using implicit time integration algorithms, the program uses a large number of relatively small time steps, with the solution for the next configuration of the body being explicit (and inexpensive) at each step. To further reduce

the computational effort, the program has a complete implementation of Courant sub-cycling in which each element is integrated with the maximum time step permitted by local stability criteria. For simulations that have large differences in element critical time steps over the mesh, very significant savings in execution time are achieved. There are no inherent limits on the size of an analysis model, and storage allocation is dynamic within the code.

- 200.sixtrack (Fortran 77)
  The function of this program is to track a variable number of particles for a variable number of turns round a model of a particle accelerator such as the Large Hadron Collider (LHC) to check the Dynamic Aperture (DA), i.e. the long term stability of the beam.

- 301.apsi (Fortran 77)
  This program solves for the mesoscale and synoptic variations of potential temperature, U AND V wind components, and the mesoscale vertical velocity W pressure and distribution of pollutants C having sources Q. The synoptic scale components are in quassi-steady state balance, while the mesoscale pressure and velocity W are found diagnostically.

Bibliography

[1] EmbeddedC++. http://www.caravan.net/ec2plus/.

[2] EmbeddedJAVA. http://java.sun.com/j2se/embedded/.

[3] Open research compiler for itanium processor family. http://ipf-orc.sourceforge.net.

[4] Standard C. http://www.open-std.org/jtc1/sc22/wg14/.

[5] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, 1988.

[7] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[8] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[9] G.E.P. Box, W.G. Hunter, and J.S. Hunter. *Statistics for Eperimenters. An Introduction to Design, Data Analysis, and Model Building*. Wiley and Sons, 1978.

[10] M.J. Breternitz and R. Smith. Enhanced compression techniques to simplify program decompression and execution. In *Proc. International Conference on Computer Design (ICCD)*, page 170, 1997.

[11] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of arm binaries. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 211–220, 2004.

[12] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[13] GNU Consortium. GCC online documentation. http://gcc.gnu.org/onlinedocs/.

[14] K.D. Cooper, M.W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[15] K.D. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 139–149, 1999.

[16] K.D. Cooper, P.J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[17] 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 Trans. Programming Languages and Systems*, 13(4):451–490, 1991.

[18] S.K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Programming Languages and Systems*, 22(2):378–415, 2000.

[19] W.J. Dixon and F.J. Massey. *Introduction to Statistical Analysis*. McGraw-Hill, 1957.

[20] J. Ernst, W. Evans, C.W. Fraser, T.A. Proebsting, and S. Lucco. Code compression. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 358–365, 1997.

[21] Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan-Kaufmann, 2005.

[22] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[23] C.W. Fraser, E.W. Myers, and A.L. Wendt. Analyzing and compressing assembly code. In *Proc. SIGPLAN symposium on Compiler Construction*, pages 117–121, 1984.

[24] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[25] S.V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, 1(4):509–520, 2005.

[26] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[27] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proc. Parallel Architectures and Compilation Techniques (PACT)*, pages 123–132, 2005.

[28] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating new general compiler optimization settings. In *Proc. International Conference on Supercomputing (ICS)*, pages 161–168, 2005.

[29] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proc. Computing Frontiers*, pages 180–188, 2005.

[30] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Code size reduction by compiler tuning. In *Proc. Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 186–195, 2006.

[31] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proc. Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2006.

[32] A.S Hedayat, N.J.A Sloane, and John Stufken. *Orthogonal Arrays: Theory and Applications*. Springer Series in Statistics, 1999.

[33] J.L. Hennessy and D.A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[34] K. Heydeman, F. Bodin, P.M.W. Knijnenburg, and L. Morin. A global trade-off strategy for loop unrolling for VLIW architectures. In *Proc. Workshop on Compilers for Parallel Computers (CPC)*, pages 59–70, 2003.

[35] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics, 1999.

[36] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software: Practice and Experience*, 29(11):1005–1023, 1999.

[37] Intel. Vtune performance analyzers. http://www.intel.com/software/products/vtune/.

[38] Mark S. Johnson and Terrence C. Miller. Effectiveness of a machine-level, global optimizer. In *Proc. SIGPLAN symposium on Compiler construction*, pages 99–108. ACM Press, 1986.

[39] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002.

[40] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. Parallel Architectures and Compilation Techniques (PACT)*, pages 237–246, 2000.

[41] Martin Lades, Jan C. Vorbrüggen, Joachim Buhmann, J. Lange, Christoph von der Malsburg, Rolf P. Würtz, and Wolfgang Konen. Distortion invariant object recognition in the dynamic link architecture. *IEEE Transactions on Computers*, 42:300–311, 1993.

[42] Monica S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

[43] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. International Symposium on Microarchitecture (MICRO 30)*, pages 194–203, 1997.

[44] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. Design Automation Conference*, pages 516–521, 1998.

[45] Robert L Mason, Richard F. Gunst, and James L. Hess. *Statistical Design and Analysis of Experiments*. Willey Interscience, 2003.

[46] K.E. Mathias, L.J. Eshelman, J.D. Schaffer, L. Augusteijn, P.F. Hoogendijk, and R. van de Wiel. Code compaction using genetic algorithms. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pages 710–717, 2000.

[47] The MathWorks. MATLAB and Simulink. `http://www.mathworks.com/`.

[48] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proc. GCC Developers Summit*, pages 171–180, 2003.

[49] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, LNCS 2443, pages 41–50, 2002.

[50] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[51] M. Naik and J. Palsberg. Compiling with code-size constraints. *Trans. on Embedded Computing Sys.*, 3(1):163–181, 2004.

[52] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[53] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *Proc. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 494–501, 2004.

[54] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[55] N.J.A. Sloane. A library of orthogonal arrays. `http://www.research.att.com/~njas/`.

[56] M. Stephenson, M. Martin, and U.M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.

[57] B. De Sutter, H. Vandierendonck, B. De Bus, and K. De Bosschere. On the side-effects of code abstraction. In *Proc. Language, Compiler, and Tool for Emebedded Systems (LCTES)*, pages 244–253, 2003.

[58] J.P. Tremblay and P.G Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill, 1985.

[59] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *Proc. International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[60] W.M. Waite and G. Goos. *Compiler Construction*. Springer Verlag, 1984.

[61] D.L. Whitfield and M.L. Soffa. An approach for exploring code improving transformations. *ACM Trans. on Programming Languages and Systems*, 19(6):1053–1084, 1997.

[62] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. In *Proc. International Symposium on Microarchitecture (MICRO 25)*, pages 81–91, 1992.

[63] M. Wolfe and C.-W. Tseng. The Powertest for data dependence. Technical Report CS/E 90-015, Oregon Graduate Institute of Science and Technology, 1990.

[64] M. Zhao, B. Childers, and M.L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, 2003.

[65] W. Zhao, B. Cai, D. Whalley, M.W. Bailey, R. van Engelen, X. Yuan, J.D. Hiser, J.W. Davidson, and K. Gallivan. VISTA: A system for interactive code improvement. In *Proc. Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES-SCOPES)*, pages 155–164, 2002.

Samenvatting

Sinds de uitvinding van de eerste compiler zijn vele technieken ontwikkeld om de code op het moment van compileren te optimaliseren. Het is bekend dat zulke compiler optimalisaties elkaar kunnen beïnvloeden: sommige combinaties van optimalisaties kunnen elkaars effect versterken, terwijl andere combinaties elkaars effect kunnen verkleinen. Daarom is het cruciaal om, bij een toename van het aantal mogelijke compiler optimalisaties, uit te zoeken hoe je ze zo goed mogelijk kunt benutten. Hier is al veel onderzoek naar verricht, maar er is tot nu toe geen doorbraak bereikt. In dit proefschrift stellen wij methodes voor om op automatische wijze een instelling van compiler optimalisaties te bepalen, en wij analyseren de effectiviteit van deze methodes.

De eerste benadering valt binnen het kader van *Design of Experiments* (DoE) [9]. DoE wordt gebruikt om op effectieve wijze gegevens te verzamelen en te analyseren. Bij DoE worden de verzamelde gegevens met statistische methodes geanalyseerd. In dit proefschrift passen wij twee soorten statistische analyse toe, namelijk het *main effect* [32] en de *Mann-Whitney test* [35]. We maken gebruik van orthogonale arrays (OAs) [32] om onze experimenten te ontwerpen. OAs staan bekend als geschikte hulpmiddelen bij het plannen van experimenten waarmee je het effect van verschillende factoren op een bepaalde uitkomst wilt bepalen. In onze benadering correspondeert een kolom van een OA met een compiler optie. Derhalve legt een rij van een OA een complete instelling van compiler optimalisaties vast. Het main effect en de Mann-Whitney test worden toegepast op de resulterende *profiling data*, en zij identificeren compiler opties met een groot effect. De analyses met het main effect en de Mann-Whitney test zijn echter conservatief, zodat ze maar een klein aantal optimalisaties met een groot effect vinden. Daarom stellen wij een iteratief algorithme voor. Hierin wordt in eerste instantie een gedeeltelijke instelling van compiler opties bepaald. Vervolgens wordt het experiment herhaald met deze gedeeltelijke instelling. Dit levert weer nieuwe compiler opties op die vastgesteld kunnen worden. Op deze manier krijgen we stap voor stap een complete instelling van compiler opties.

Bij de eerste benadering meten we het effect van iedere compiler optie apart. Bij de tweede benadering, daarentegen, kijken we meer naar de interactie tussen verschillende compiler opties, de invloed die de compiler opties op elkaar kunnen hebben. Bij deze aanpak meten we het effect van combinaties van compiler opties. We gebruiken een vast aantal compiler instellingen, die zijn ontworpen met een orthogonaal array. Nadat we het begrip interactie hebben gedefinieerd, berekenen we met de gekozen compiler instellingen het effect van de interactie van bepaalde combinaties van compiler opties. Deze benadering identificeert een optimale compiler instelling, die geschikt is voor meerdere toepassingen.

Onze derde benadering maakt gebruik van iteratieve compilatie [8, 40] met *random search* om een optimale instelling van de compiler te vinden. Deze aanpak laat zien dat we al na een beperkt aantal iteraties een optimale compiler instelling kunnen identificeren. Het belangrijkste verschil tussen deze benadering en de twee eerdere is dat de twee eerdere benaderingen alleen optimalisaties selecteren die significant effectief zijn. Bij iteratieve compilatie, daarentegen, bevat de gekozen compiler instelling, als gevolg van de willekeurige generatie van testinstellingen, meerdere optimalisaties zonder effect. Deze benadering maakt het mogelijk om te schatten hoeveel verbetering we maximaal nog kunnen bereiken met de compiler optimalisaties. Daarom kunnen de resultaten van deze benadering gebruikt worden voor de evaluatie van compiler instellingen. Bijvoorbeeld de evaluatie van de standaard instellingen van de compiler (zoals *-Ox*) of de instelling die we met de vorige benadering hadden geïdentificeerd. We hebben de iteratieve compilatie ook toegepast om één compiler instelling te vinden die zo goed mogelijk is voor meerdere toepassingen. We vergelijken de resulterende instelling vervolgens met de geschatte optimale compiler instelling voor iedere toepassing afzonderlijk.

De compiler instellingen die we met onze drie benaderingen vinden, werken beter dan de *-O3* instelling van de *gcc*-compiler. Uit de resultaten in dit proefschrift concluderen we dat het inderdaad mogelijk en zinvol is om de compiler optimalisaties met statistische methodes in te stellen.

Met onze methodes kunnen gebruikers hun eigen optimalisatiedoel kiezen. Ze kunnen zich bijvoorbeeld richten op de *execution time*, de lengte van de code, of het energieverbruik. Onze methodes zijn derhalve breed toe te passen.

Bovendien zijn alle benaderingen in dit proefschrift onafhankelijk van de feitelijke implementatie van compilers en toepassingsprogramma's. Daarom is het gemakkelijk om onze methodes toe te passen op een willekeurige combinatie van een compiler en een toepassing. Deze veelzijdigheid is uniek. Hiermee onderscheiden onze resultaten zich van die van ander onderzoek in dit vakgebied.

Acknowledgement

I would like to thank my parents and my sisters for their support during my study in Leiden. I am also grateful to my uncle and his family living in the Netherlands for their concern about me. I felt reassured because of them. I would like to thank my roommate Rudy for making tea at end of the day, it was refreshing after a tiring study day. My friends helped me all the time, and it was impossible to finish writing my thesis without them. I hope that I can help them in some way in the future.

Curriculum Vitae

Masayo Haneda was born in Hyogo, Japan on January 21st in 1978. She received her BS and MS degrees of Science from Nara Women's University in 2000 and 2002, respectively. Since May 2002, she has been a Ph.D. student at Leiden Institute of Advanced Computer Science under the supervision of Professor Harry A.G. Wijshoff and Dr. Peter M.W. Knijnenburg. She has worked on the FAME project from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO). The results of her research are presented in this thesis.