METHODOLOGY OF DYNAMIC COMPILER OPTION SELECTION BASED ON STATIC PROGRAM ANALYSIS – IMPLEMENTATION AND EVALUATION

by

Eun Jung Park

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical and Computer Engineering

Summer 2007

© 2007  Eun Jung Park
All Rights Reserved
METHODOLOGY OF DYNAMIC COMPILER OPTION SELECTION BASED ON STATIC PROGRAM ANALYSIS – IMPLEMENTATION AND EVALUATION

by

Eun Jung Park

Approved: _____________________________________________
Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____________________________________________
Gonzalo R. Arce, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____________________________________________
Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: _____________________________________________
Carolyn A. Thoroughgood, Ph.D.
Vice Provost for Research and Graduate Studies
ACKNOWLEDGMENTS

I would like to thank Prof. Gao for giving me the opportunity to study and start my current research in CAPSL. Without his great helps and supports, I was not able to finish this thesis and keep studying.

I thank Dr. Wu, who helped me to start research in CAPSL and gave me a lot of advice as a mentor for two years. I learned a lot from his works and experiences and they are helpful in my future research.

I thank Dr. Li, who gave me a good direction of the current research and helped me to solve many problems I have met in my research. With his advice, I could find what I should do for my future direction in my research.

I thank Prof. Bohacek, who gave me the first opportunity to study in his research group when I came here. Even though I changed the research topics, he gave me a continuous helps and encouraged my research.

I thank Murat, who helped the implementation and experimental works. Especially, without his implementation for OSO searching tool, I could not start any work in this thesis.

I thank Juergen and Joseph, who helped solving any problems in my research. Whenever I have met any problems, their feedback was very important to find any
breakthrough in my research. Especially without Joseph’s proof reading, I could not finish this thesis.

I thank Matthew, who reviewed this thesis carefully for English clarity. He pointed out many problems I have in writing and helped to improve my English writing. Without his help, it was not possible to finish this thesis.

Lastly, I thank everybody in CAPSL. They are very smart and I learned a lot from them. It was great opportunity to have working experience with them.
DEDICATION

To God and parents
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xii</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Challenges</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Methodology</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Major Contribution</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Experimental Result</td>
<td>5</td>
</tr>
<tr>
<td>2 INFRASTRUCTURE</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Background and Terminology</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Overview</td>
<td>7</td>
</tr>
<tr>
<td>3 PATTERN AND TEMPLATE</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Definition</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Creation Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Pattern Creation Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Template Creation Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Matching Algorithm</td>
<td>15</td>
</tr>
</tbody>
</table>
4 OPTION SELECTION ............................................. 17

4.1 Overview .................................................................. 17
4.2 Step 1: Finding Maximal Subsequences of Positively Interacting Options ........................................ 20
4.3 Step 2: Combining Subsequences in C ......................... 21
4.4 Step 3: Selecting the Best Sequences ........................... 21

5 IMPLEMENTATION ....................................................... 23

5.1 Common Datastructure ............................................. 23
5.2 Recording the information into the PSS ......................... 24
5.3 Pattern Creation ..................................................... 26
5.4 Template Creation .................................................. 27
5.5 Dynamic Option Selection ....................................... 27

6 EXPERIMENTAL FRAMEWORKS ................................. 36

6.1 Hardware Platform ................................................ 36
6.2 Benchmark Suite .................................................. 38
6.3 Automatic Test Platform ......................................... 38

6.3.1 OSO Searching Script ........................................ 39
6.3.2 Evaluation Script ............................................... 40

7 PERFORMANCE EVALUATION .................................. 41

7.1 Summary of Result ............................................... 41
7.2 Template Matching Rate .......................................... 43
7.3 Template Matching Rate vs. Performance Improvements .... 44
7.4 Pattern Matching Rate ............................................ 45
7.5 Properness of Decision Region ................................. 46
7.6 Conclusion ......................................................... 46

8 RELATED WORK .................................................. 47

8.1 Program Similarity Detection ................................... 47
8.2 Automated Option Selection .................................... 48
9 CONCLUSIONS AND FUTURE WORK .......................... 49

Appendix

A TEST PLATFORM CONFIGURATION ......................... 52
   A.1 Hardware Configuration .................................. 52
   A.2 Automatic Test Platform Configuration ............... 53

B OSO SEARCHING TOOL MANUAL ............................. 55
C ADDITIONAL TEST RESULT ................................. 57
   C.1 Template and Pattern Matching Rate ................. 57
   C.2 Number of Template found and improvement in Each Pattern ...... 58

BIBLIOGRAPHY ................................................. 59
LIST OF FIGURES

2.1 Overview of Methodology for Supporting Dynamic Option Selection 8
3.1 Algorithm for Pattern Creation ................................. 13
3.2 Algorithm for Template Creation .............................. 15
3.3 Algorithm for Pattern and Template Matching ............... 16
4.1 Flowchart of searching for OSO (courtesy by Murat Bolat) ... 22
5.1 Data Structure used for Representing Program Segments ...... 25
5.2 Flowchart Graph for Recording the Information into a PSS ... 30
5.3 Flowchart Graph for Pattern Creation .......................... 31
5.4 Patterns Created in Our Methodology (1) ....................... 32
5.5 Patterns Created in Our Methodology (2) ....................... 33
5.6 Patterns Created in Our Methodology (3) ....................... 34
5.7 Flowchart Graph to Apply Optimization for Dynamic Option Selection .................................................. 35
6.1 Block Diagram for PXA255 Xscale Architecture ............... 37
6.2 Automatic Test Platform for This Methodology ............... 39
7.1 Template Matching Rate over Pattern Matching Rate in MiBench 42
7.2 Performance Evaluation for MiBench .......................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Template Matching Rate over Pattern Matching Rate in CommBench</td>
<td>44</td>
</tr>
<tr>
<td>7.4</td>
<td>Performance Evaluation for CommBench</td>
<td>45</td>
</tr>
<tr>
<td>TABLE</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic and Global Optimization Options Used in Our Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Local Optimization Options Used in Our Algorithm</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Intel XScale Architecture Characteristics and Quantified Value</td>
<td>28</td>
</tr>
</tbody>
</table>
ABSTRACT

When we develop applications, it is important to write optimized code so that we can achieve good performance. However, the proper use of compiler optimization options is essential because obtaining the maximum possible performance by writing only optimized code (without compiler’s help) is very difficult, if not impossible. Current compilers possess a myriad of options to optimize the application. Nevertheless, they provide a subset of options called the standard optimization options, which will provide safe optimization and give a reasonable optimized code. We can use the standard optimization options; however, they are not always an optimal solution for all applications. Therefore, we should carefully choose the set of options if we need additional performance improvement.

This research area has three main challenges: (1) Due to the number of optimization options in a compiler, finding the optimal set of options for a target program by brute force or any other exhaustive method is not simple. (2) Although we find the best set for the application, the compiler will apply this set of options to the whole program. Thus it is possible to lose some performance improvements because some options may affect negatively specific parts and decrease the overall performance. (3) The absence of an automatic test platform gives more complexity in evaluation process so it is hard to test various benchmarks under different conditions. To resolve these problems, the methodology shown in [1] proposed how we identify performance sensitive code segments automatically. The key challenge in this methodology is how to automatically identify a special code segments to which customized set of optimization options could be applied [1]. To address this
challenge this methodology uses a machine learning method for automated detection of performance sensitive code segments based on static program analysis. This methodology introduces the algorithms to recommend a set of options automatically and support dynamic option selection based on static program analysis. In this methodology, first we create templates trained over a set of random input programs [1] based on program structure and architectural similarity. Secondly, we find special sets of options for each template and create database and these special sets of options will be selected automatically by a compiler. Moreover, dynamic option selection in compilers enables to use different set of options to different sections of the program.

This thesis focuses on the implementation and evaluation works of this methodology. Major contributions of this thesis are as follows: (1) This thesis reports an implementation of algorithms used by the compiler to find program segments and the recommended option sets based on static analysis and change the set of options dynamically. Thus compiler users can obtain more optimized code than when they use any default optimization options using this implementation. (2) This thesis explains algorithm developed to detect candidate loop segment based on a general Intermediate Representation (IR) in a compiler and the implementation in a GCC compiler version 3.3. By using this algorithm, we can extract the information we need for automatic detection of code segments, templates creation, and similar code segment matching. (3) This thesis reports a design and implementation of an automatic testing platform. This platform is used to evaluate the implementation of the methodology presented in this thesis. This test platform enables us to automatically set up a test platform with any kind of compilers, benchmark suites and hardware platforms.

All implementations in a GCC compiler version 3.3 were tested with Mibench
and CommBench benchmarks on Intel Xscale PXA255 Architecture. Evaluation result showed 1.8% overall performance improvement over the -O3 default optimization option. Even though this is slight improvement, this opens research directions in this field.
Chapter 1

INTRODUCTION

This chapter introduces the overview of this thesis. Section 1.1 explains the background of the research introduced in this thesis. Section 1.2 and 1.3 show the challenges of this research and methodologies to solve them. Section 1.4 shows the main contributions of this thesis and section 1.5 shows the experimental works achieved by the implementation in this thesis.

1.1 Background

With well-written code, using compiler options efficiently would be one of the important factors which will affect the overall performance of applications. Even though we have amassed a great deal of experience about compilers and their options, it is hard to predict all their effects in a given application due to the myriad of optimization options. This is exacerbated by the fact that some options affect each other in both positive and negative ways. For example, GCC compiler has more than 60 optimization options. If we want to combine GCC’s options together, the cardinality of the set of possible optimization options is around $10^{18}$. These facts drive most compiler users into using one of the default optimization options like -O2 or -O3. Based on this background, next section explains the main challenges of this research.
1.2 Challenges

Based on the characteristics of current compiler techniques, we can find three main challenges as follows:

(i) Even if we use these default optimization options, we can find that they are not always on optimal solution. Even though we use the highest and safest optimization level, like O3 in GCC, it does not necessarily produce the best performance improvement for all applications [2, 3, 4, 5, 6, 7]. Additional performance improvement could be achieved by carefully choosing optimization options customized to performance sensitive program segments [8].

(ii) Default optimization options turn on many other options that affect the whole program and that cannot be changed dynamically during compilation. This behavior can have an overall negative effect on performance since some options may affect some code segments in negative ways.

(iii) The absence of an automatic test platform increases the complexity of evaluation process. Most benchmark suites have 5-15 cases so we need to set up and compile each test case for evaluating under different conditions.

Next section introduces the methodology to solve these challenges.

1.3 Methodology

To resolve these problems, there exist several research directions to find the optimized set of options automatically. Cavazos and O’Boyle [9] presented a method for an automated search for optimization options. They focus on function level instead of the whole program and the matching strategy is based on the similarity of the features of functions instead of focusing on a fine-grain level program analysis through program segments. Annavaram [10] and Lau et al. [11] suggested the method to predict the performance for applications by using periodic sampling with
SPEC2000 benchmark. Hoste et al. [12, 13] predicted the performance on any architecture by measuring the architecture-independent characteristics of programs.

This thesis used the methodology shown in the paper by Wu et al. [1] This methodology shows how we identify performance sensitive code segments automatically. We focus on fine-grain level program analysis through program segments and finds the set of options for input program automatically even though the compiler users have no idea about optimization options based on machine learning method. First, we recognize code segments according to the structural similarity and then we narrow down the segments according to the architectural behavior prediction. Moreover, we apply the set of options to each code segment in input programs dynamically for achieving additional performance improvement.

This methodology uses three steps: Learning phase, searching phase, and matching phase.

(i) In the learning phase, we find code segments which we need to apply special optimization options according to the static structure analysis and architectural behavior predictions. We define detected code segments with similar structural information as patterns. In each pattern, we extract the architectural behavior and put them into weight vectors and decide templates based on architectural behavior similarity.

(ii) In the searching phase, we find the optimized set of options for each code segment which was detected in the learning phase and create a database. This database is integrated into the compiler and used during matching phase for finding the optimized set of options for each code segment from a input program automatically.

(iii) In the matching phase, we find the similarity between code segments from a input program and any items in the database. Then we find the set of options
that we can use for each code segment. This enables the compiler users to use optimization options efficiently even though they do not have any idea about the compiler options. Moreover, the compiler changes the set of options dynamically for different sections of the program.

1.4 Major Contribution

This thesis shows the implementation of this methodologies introduced in section 1.3.

The learning and matching phases in this methodologies were implemented in the front end of a GCC compiler version 3.3 and the database created from the searching phase was integrated. The searching phase was implemented independently of a compiler and use a compiler to find the set of options for creating the database.

The main contributions of this thesis are:

(i) This thesis reports an implementation of algorithms used by the compiler to find program segments and the recommended option sets based on static analysis and change the set of options dynamically. Thus compiler users can obtain more optimized code than when they use any default optimization options using this implementation.

(ii) This thesis explains algorithm developed to detect candidate loop segment based on a general Intermediate Representation (IR) in a compiler and the implementation in a GCC compiler version 3.3. By using this algorithm, we can extract the information we need for automatic detection of code segments, templates creation, and similar code segment matching.

(iii) This thesis reports a design and implementation of an automatic testing platform. This platform is used to evaluate the implementation of the methodology presented in this thesis. This test platform enables us to automatically set up
a test platform with any kind of compilers, benchmark suites and hardware platforms.

1.5 Experimental Result

The implementation of this methodologies were tested on Intel Xscale PXA255 architecture with MiBench and CommBench. Compared to -O3 optimization option, experimental result showed 1.8% overall performance improvement from test cases which had more than 50% matching rate, 1.4% overall performance improvement from all test cases regardless of matching rate.

The thesis is organized as follows. Chapter 2 introduces the infrastructure of this methodology which includes the three phases –learning, searching and matching. The learning and matching phases are to be explained in Chapter 3, and searching phase is explained in Chapter 4. In Chapter 5, we will describe the implementation in the GCC compiler. This chapter describes the contribution 1 and 2, and specially, section 5.2 describes the contribution 2. We will explain the experimental testbed in Chapter 6 and we can find the contribution 3 in this chapter. The result and analysis of our implementation is explained in Chapter 7. We also introduce some related work in Chapter 8. Finally, in Chapter 9, we will talk about our conclusion and future work.

Appendix A shows the instruction how we can configure the test platform including the hardware platform and automatic test platform. Appendix B shows the instruction how we can use OSO searching tool. Appendix C shows the additional test result which does not appear in Chapter 7.
Chapter 2

INFRASTRUCTURE

This chapter explains the background and the overview of the methodology for dynamic option selection.

The rest of this chapter is organized as follows. Section 2.1 explains the background and important terminologies of the methodology used in this thesis. Section 2.2 describes the overview of three phases in the methodology which will be explained in the following chapters in detail.

2.1 Background and Terminology

To resolve problems mentioned in previous chapter, our methodology finds the set of options for input programs automatically. Moreover, it changes the option set of each section of a program dynamically during compilation for achieving additional performance improvement.

For these purposes, our methodology relies on a static analysis of an input program [1]. More specifically, we use two kinds of information: (1) Structural information and (2) Architectural behaviors. For structural information, we consider loop, condition and switch statements. We check the number of appearances and relations among them. In a similar way, for architectural behavior, we check the kind of instruction i.e., arithmetic, logical, memory operations depending on the kind of architecture. And we assign a weight to each instruction and check the sum of them.
Based on this information, we define several terminologies for the methodology.

- **Pattern**: Program model which has specific structural information about what frequently appears in the application.

- **Template**: Program model which has specific architectural behaviors within patterns.

- **Instance**: Complete C code which has specific structural information and architectural behaviors for representing a template.

- **Optimized Set of Options (OSO)**: The set of options with which the application can achieve better performance over the highest default optimization option provided by the compiler.

- **Program Structure Stream (PSS)**: A linked list type data structure used for representing program structural information and architectural behaviors extracted from given applications.

- **Weight Vector (WV)**: A vector element which shows architectural behavior and it has different dimension according to the pattern.

Based on the background and these terminologies, section 2.2 explains the overview of our methodology.

### 2.2 Overview

Our methodology has two main techniques: (1) Defining the similarity of the program section which we need to apply a customized set of options based on static program structure analysis and architectural behavior prediction and (2) Dynamic option selection in the compiler to apply different set of options to different sections of the program.
These two techniques appeared in three main phases for supporting dynamic options selection and each phase is represented in Figure 2.1.

![Diagram of Methodology for Supporting Dynamic Option Selection]

**Figure 2.1:** Overview of Methodology for Supporting Dynamic Option Selection

The three phases of the methodology are presented below:

- **Learning Phase:** In this phase, the compiler extracts the information from input programs. This phase happens when a compiler generates the intermediate representation from the Abstract Syntax Tree (AST). Before generating the intermediate representation, we find the first piece of information which refers to structural information and put it into the PSS. The compiler collects and finds the most frequent segments in the source code and creates patterns.
Within the created patterns, we decide on fixed points in the WV and create templates, then we go to the next step with the created templates.

- **Searching Phase:** We find the OSO for each template in this phase. We create instance for each template and find the OSO with the OSO searching tool implemented independently of the compiler. This tool works dependent on a compiler and hardware platform. In the result, we generate the template database which contains the templates and corresponding OSOs. At the end, we integrate it into the compiler.

- **Matching Phase:** After both phases have completed, whenever the compiler accepts input programs, it will analyze the program and extract the information for finding the similarity of the code segment and try to find any pattern/template created in template database. If any template is detected, the compiler decides which optimization option flags should be turned on. And it changes the flag value after saving the previous flag value for each optimization. For changing the set of options for each detected templates in a program dynamically, the compiler restore the previous flag value after applying optimizations according to the flag turned on.

These three phases are used in our methodology and each phase is explained in the following chapters. Learning and matching phases are to be explained in Chapter 3 and searching phase is explained in Chapter 4 in detail.
Chapter 3

PATTERN AND TEMPLATE

This chapter explains how we define patterns/templates and how we create them from the structural analysis and architectural behaviors in the learning phase. Moreover, this chapter shows how we match the section of a program to existing patterns and templates in the matching phase.

Since we have myriad of options in a compiler, it is not simple to find the set of options for a given program even though we have amassed experiences of using these options. The algorithms described in this chapter enable the compiler to find the specific code segment of the program and recommend a customized set of options automatically.

This chapter is organized as follows. Section 3.1 describes the definition of pattern and template in our methodology. Based on the definition, section 3.2 explains the algorithm for pattern and template creation and section 3.3 describes the algorithm for matching process.

3.1 Definition

We use the top-down definition for pattern and template which means we define big sets first and then smaller sets in each big set. We derive a big set with the structural information, and then we derive smaller set with architectural behaviors within that structural information. According to this method, our definitions of pattern and template are the following:
**Definition 1:** Pattern is defined as a code segment in a program which has a specific structural information with any architectural behaviors.

**Definition 2:** Template is defined as the code segment in a program which has specific architectural behaviors within each pattern.

This implies that if two templates belong to one specific pattern, they have the same structural information but different architectural behaviors.

This way of definition simplifies any other learning and matching algorithms used in our methodology. For example, if we have two code segments like the following:

**Segment1:**
```
for (i = var1, i < var2; i++) {
    A1;
    A2;
    ........
    An;
}
```

**Segment2:**
```
if (i==var1){
    A1;
    A2;
    ........
    An;
}
```

If we are using revers of our definition, we may define a bigger set according to the architectural behaviors and check the structural information after that. The first problem is that it is hard to find where to start the code segment. Especially, if there are no statements such as a loop, condition or switch, it is hard to decide the beginning of the code segment. A bigger problem is that many optimization options are sensitive to the structural information such as loops and branches. Even
though we found the same architectural behaviors like the above examples, if they have the different structural information, we need to apply the different kind of optimizations in most cases. Also, most compilers are using our approach for compiling or interpretation the source codes. It would be more compatible to implement our methodology into modern compilers if we are using a top-down definition.

3.2 Creation Algorithm

This section describes the pattern and template creation algorithm and they are explained in section 3.2.1 and 3.2.2 respectively.

3.2.1 Pattern Creation Algorithm

We create pattern according to the structural information by the algorithm shown in Figure 3.1. To simplify analysis, we use loop, condition and switch statements for forming the structural information.

First, we extract the information from input programs and put this into a candidate set and we sort all elements in the candidate set whenever we have new elements by the number of appearance. If we collect enough information pieces, we choose elements which appeared frequently and create patterns with the information of them.

3.2.2 Template Creation Algorithm

In this section, we explain how to find architectural similarity and derive templates from this information. To simplify the analysis we only focus on the following architectural information and calculation:

- **Instruction Delay**: This datum shows predicted latencies for arithmetic, logical, shift, move, compare instructions. We assume each operation in the statement is transformed into a related instruction and calculate the total weights for each of them.
Making candidate set for any structural information extracted from source program

 Sorting elements in candidate set according to their number of appearance

 No enough elements in candidate set? Yes

 Creating patterns with the elements in candidate set which appeared more than threshold value.

* This flowchart shows how we create patterns. We collect the structural information from input programs and create patterns from the most frequently appeared code segments.

**Figure 3.1:** Algorithm for Pattern Creation

- **Memory Access Latency:** This shows the predicted latencies for load and store instructions. We assume the number of registers based on a given architecture. For load operations, if one variable appears in the right side and was not loaded into a register, we assume that a load instruction is needed. For store operations, if we find a variable in the left side that is being assigned a value, we assume that a store instruction is needed.

  We put this architectural information into the \( WV \) defined as:

  \[
  \{ \omega_1, \omega_2, \cdots, \omega_n \}
  \]

  The \( n \) in this sequence represents the number of consecutive compound statements that appear in the pattern. Moreover, \( n \) decides the dimensionality of this sequence. Let \( l \) be the number of non-compound statement structures in the pattern, \( n = 2 \times l - 1 \). The \( \omega^i \) represents a pair of values.
Before we start template creation, we need to initialize the Maximum WV (MWV) with the maximum weights for each pattern from. This information comes from the procedure defined in Figure 5.3.

We initialize the WV_Candidate set with the MWV for each pattern and use the MWV as input for the procedure shown in Figure 3.2. A Completed_WP set will contain the output of this procedure which will be determined as templates. To perform the decision in this procedure, we need an assumption about architectural similarity.

When we have given a code segments set \( S_i = \{S_{i1}, S_{i2}, ..., S_{im}\} \), let set \( B_i = \{B_{i1}, B_{i2}, ... B_{in}\} \) represents the architectural information of \( S_i \), and \( O_i \) represent the OSO. The OSO correlation between \( S_i \) and \( B_i \) can be expressed by the function \( \Phi \) in the following way:

\[
\Phi(S_i, B_i) = O_i
\]

Based on the empirical observations, the architectural similarity is defined as:

**Assumption 1:** Let \( B_1 \) and \( B_2 \) be two sets of the architecture-dependent behaviors for program segments \( S_1 \) and \( S_2 \), respectively, \( S_1 \) and \( S_2 \) have similar architecture-dependent behaviors if:

\[
\Phi(S_1, B_1) = \Phi(S_2, B_2)
\]

Based on this, if we find two possible WV points possible that share the same OSO, we decide these points have the same decision region. However if two points need completely different OSOs, we decide that these points need separated decision regions and call them as fixed points. We repeat the procedure in Figure 3.2 until we find all fixed points within the MWV elements and they are defined as templates for each pattern.
Initialize $\text{WV\_Candidate} = \{\text{MWV}\}$
Initialize $\text{Completed\_WV} = \{\emptyset\}$

Sort $\text{WV\_Candidate}$ and $\text{Completed\_WV}$ in ascending order

Choose smallest $x \in \text{WV\_Candidate}$ and set as UB
Choose biggest $y \in \text{Completed\_WV}$ set as LB

Set $\text{LBUBWV}_{\text{New}} = \frac{\text{UB} + \text{LB}}{2}$

1. Create instance of New\_WV
2. Get OSO for New\_WV
3. Get execution time of New\_WV instance with OSO for New\_WV
4. Get execution time of New\_WV instance with OSO for UB

$t_1 < t_2$

1. Put New\_WV into $\text{WV\_Candidate}$
2. Distance between New\_WV and LB is less than 1?
   - Yes: Put LB and New\_WV into Completed\_WV
   - No

$\text{WV\_Candidate}$ is empty?
   - Yes: no
   - No: Set $\text{LBUBWV}_{\text{New}} = \frac{\text{UB} + \text{LB}}{2}$

* This flowchart shows how we create templates. Between 0 and MWV, we find the independent decision region. For finding this, we use OSO for each point of vector and check we can share the same OSO. If we can, we consider two vector points can use same decision region. Otherwise, we consider they are using independent decision region. By repeating this, we create templates by using fixed point of vector element which has independent decision region.

Figure 3.2: Algorithm for Template Creation

### 3.3 Matching Algorithm

This section describes the algorithm used in the matching phase shown in Figure 3.3 after we have all the necessary information from the learning and searching phases.

First we check the structural information and perform pattern matching. If we cannot find any patterns, that means we do not need to perform template matching so we may exit from this algorithm.

Specifically, for template matching, we calculate the minimum Euclidean distance, commonly used in calculating the distance between two vector elements, from all possible template’s WV points in the same pattern and choose the point which shows the minimum distance. However if this minimum distance is larger than the maximum distance between the WVs of all templates, we decide that no template was found. Otherwise, we follow the decision rule for template matching defined as:
* This flowchart shows how we match patterns and templates in matching phase. First we do pattern matching according to the structure information. Within detected pattern, we perform template matching by checking WV and get the closest distance from all existing template’s decision region. If we can find any shortest distance within threshold, we consider we found existing template. Otherwise, we consider we could not find any template in the current database.

**Figure 3.3:** Algorithm for *Pattern* and *Template* Matching

**Definition 3:** If the current code segment $S_i$ and *template* $T_i$ has program structural similarity and the WV of $S_i$ has the minimum distance from the WV of $T_i$, we can say $S_i$ is the closest from the decision region of $T_i$ and we decide that $S_i$ matches to $T_i$.

Using algorithms described in this chapter, we create *patterns* and *templates*. And we find corresponding *OSO* for each *template* and use this information during matching process. In the next chapter, we explain the algorithm used for finding *OSOs* in our methodology.
Chapter 4

OPTION SELECTION

This chapter describes the algorithm we use to search for an optimized sequence of compiler optimization options (OSOs) for the created templates.

The algorithm described in this chapter is used in the searching phase and generates OSOs for each template created from the learning phase. The OSOs are used in the matching phase after we detect any templates in an input program. The implementation of this algorithm is used outside of the compiler as shown in Figure 2.1.

The rest of this chapter is organized as follows. Section 4.1 explains the overview of the algorithm for option selection. And section 4.2, 4.3 and 4.4 describes the three steps in this algorithm respectively.

4.1 Overview

If an OSO is found, the program subjected to analysis can get additional performance improvement if compiled using this set of options [1]. Additional performance means improvements in speed when compared to the baseline version, which uses the highest optimization level of the compiler [1]. If one sequence of options gives the better performance than the baseline, they become OSOs.

Our OSO search algorithm is based on the method proposed in the paper [6]. In contrast to the algorithm described in [6], our algorithm employs a different classification of compiler options and searches for a set of optimized sequences of optimization options [1].
<table>
<thead>
<tr>
<th>Index</th>
<th>Optimization Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function-cse</td>
</tr>
<tr>
<td>2</td>
<td>inline</td>
</tr>
<tr>
<td>3</td>
<td>tracer</td>
</tr>
<tr>
<td>4</td>
<td>optimize-sibling-calls</td>
</tr>
<tr>
<td>5</td>
<td>gcse</td>
</tr>
<tr>
<td>6</td>
<td>gcse-lm</td>
</tr>
<tr>
<td>7</td>
<td>gcse-sm</td>
</tr>
<tr>
<td>8</td>
<td>delete-null-pointer-checks</td>
</tr>
<tr>
<td>9</td>
<td>schedule-instruction</td>
</tr>
<tr>
<td>10</td>
<td>schedule-instruction2</td>
</tr>
<tr>
<td>11</td>
<td>fsched-interblock</td>
</tr>
<tr>
<td>12</td>
<td>caller-saves</td>
</tr>
<tr>
<td>13</td>
<td>reorder-block</td>
</tr>
<tr>
<td>14</td>
<td>reorder-functions</td>
</tr>
<tr>
<td>15</td>
<td>strict-alasing</td>
</tr>
<tr>
<td>16</td>
<td>inline-functions</td>
</tr>
<tr>
<td>17</td>
<td>keep-static-consts</td>
</tr>
<tr>
<td>18</td>
<td>rerun-cse-after-loop</td>
</tr>
</tbody>
</table>

Table 4.1: Basic and Global Optimization Options Used in Our Algorithm

The optimization options used by this algorithm are classified into three sets: basic optimization options $B$, global optimization options $G$, and local optimization options $L$. Global optimization options force the compiler to deal with the whole function instead of just code segments [1]. Considering $G$ optimizations in our methodology would call for a complex analysis of the compiler. Hence, we treat $B \cup G$ showed in Table 4.1 as the default sequence of this searching algorithm. They are always active during the OSO searching process and only the sequence from $L$ set shown in Table 4.2 will be the candidate of OSO.

The OSO search algorithm can be described through the following three steps.

(i) Finding Maximal Subsequences of Positively Interacting Options
<table>
<thead>
<tr>
<th>Index</th>
<th>Optimization Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cse-skip-blocks</td>
</tr>
<tr>
<td>2</td>
<td>cse-follow-jumps</td>
</tr>
<tr>
<td>3</td>
<td>force-addr</td>
</tr>
<tr>
<td>4</td>
<td>force-mem</td>
</tr>
<tr>
<td>5</td>
<td>peephole</td>
</tr>
<tr>
<td>6</td>
<td>unroll-loops</td>
</tr>
<tr>
<td>7</td>
<td>rerun-loop-opt</td>
</tr>
<tr>
<td>8</td>
<td>prefetch-loop-arrays</td>
</tr>
<tr>
<td>9</td>
<td>strength-reduce</td>
</tr>
<tr>
<td>10</td>
<td>move-all-movables</td>
</tr>
<tr>
<td>11</td>
<td>reduce-all-givs</td>
</tr>
<tr>
<td>12</td>
<td>branch-on-count-reg</td>
</tr>
<tr>
<td>13</td>
<td>peephole2</td>
</tr>
<tr>
<td>14</td>
<td>regmove</td>
</tr>
<tr>
<td>15</td>
<td>expensive-optimization</td>
</tr>
<tr>
<td>16</td>
<td>rename-registers</td>
</tr>
<tr>
<td>17</td>
<td>schedule-speculative</td>
</tr>
<tr>
<td>18</td>
<td>schedule-speculative-load</td>
</tr>
<tr>
<td>19</td>
<td>schedule-speculative-load-dangerous</td>
</tr>
<tr>
<td>20</td>
<td>unroll-all-loops</td>
</tr>
</tbody>
</table>

Table 4.2: Local Optimization Options Used in Our Algorithm
(ii) Combining Subsequences in $C$

(iii) Selecting the Best Sequences

The following sections give the detailed description of these steps.

4.2 Step 1: Finding Maximal Subsequences of Positively Interacting Options

In this step, the algorithm finds subsequences of optimization options, which positively interact with each other. The algorithm starts by enabling single options and by extending the subsequences based on their performance and interaction. These subsequences are included into the sequence set $C$.

**Definition 4:** The set $C_k$ represents the sequences of positively interacting options, where $k$ options are switched on in each sequence.

**Definition 5:** The set $L_i$ is the set of individual options that do not appear in the elements of $C_1, ..., C_{i-1}$ $(2 \leq i \leq n)$.

The set $C_1$ is constructed by testing single options of $L$ for their performance. The first $M$ options offering the best performance are selected and placed into the set $C_1$. Therefore the size of $C_1$ is $M$. The elements in the set $C_2$ are the combination of single compiler options from the sets $L$ and $L^2$. In order for a pair of options to be added to the set $C_2$, the options must interact positively, i.e. they need to give better performance than if they were enabled individually. This is a rather strict condition. If two options are combined and performance improves, that improvement should be based on a ”strong-strong” cooperation instead of a ”weak” option being added to a ”strong” one. Therefore, the improvement is tested according to this constraint: The combination brings about a significant improvement over selecting either option without the other.
Every subsequent $C_k$ is constructed by pairing up the options in set $C_{k-1}$ with the individual options in set $L^k$. The combinations that have positive interactions are added to the set $C_k$. This step finishes when $L^i = \emptyset$ and $C_i = \emptyset$. All positively interacting subsequences are stored into $C$. That is, $C = \bigcup_{k=1}^{i} C_k$.

4.3 Step 2: Combining Subsequences in $C$

In this step, the elements of the set $C$ are combined if they interact positively. If the interaction is positive, the elements are combined and included in $C$. The unpaired versions of the combined elements are deleted. A new set $C_{temp}$ is created to include the subsequences which are deleted from $C$ and improve the performance over that of the baseline. At the end of this step the subsequences in $C_{temp}$ are stored back to the subsequences in set $C$.

4.4 Step 3: Selecting the Best Sequences

The performance of each subsequence in set $C$ is tested 10 times and the mean performance is used. The mean performance is compared to the baseline performance. The subsequences performing better than the baseline are kept while others are discarded.

After this step, the algorithm determines a set of subsequences for a given program segment. Each of these subsequences is an OSO. For our purpose, each OSO will be considered a candidate when searching for a common OSO for a given template. Figure 4.1 shows the flowchart diagram of algorithm to search for an optimized sequences.

We create a database with each set of templates and corresponding OSOs from the algorithm described in this chapter and use them for automatic compiler option selection.
Figure 4.1: Flowchart of searching for OSO (courtesy by Murat Bolat)
Chapter 5

IMPLEMENTATION

This chapter shows the implementation of our methodology in a compiler. We implemented all algorithms except searching algorithm described in Chapter 4 in a GCC v3.3.3 compiler for the Xscale architecture. The searching algorithm is implemented independently of a compiler. However it works with a compiler and hardware platform to generate a database for the matching process. The database generated from this algorithm is integrated into the compiler.

Without the implementation of our methodology, we only can use one set of options for the whole program without any change of options dynamically. The implementation described in this chapter enables the compiler to choose a different set of options for different sections of the program dynamically and eliminate the degradation of performance.

This chapter is organized as follows. Section 5.1 explains the common datastructure used in all implementation of our methodology. Section 5.2 demonstrates the implementation of the process to record information into $PSS$ structure. Section 5.3 and 5.4 shows the implementation of pattern and template creation respectively. And in final, section 5.5 shows the implementation of dynamic option selection in a compiler.

5.1 Common Datastructure

First of all, the following information is used for all functions added to the compiler for this methodology.
• **template_num**: This number indicates the number of templates found in an input program. We use this number to check whether we need to use dynamic option selection. If this number is 0, we simply use the original compiling procedure.

• **loop_index**: This number indicates a unique index number given to each loop body in an input program. This will be used to locate the template associated with a loop body in the loop_template.

• **loop_template**: This is implemented as an integer array and indicates the template number corresponding to each loop. This array is indexed by the loop_index value and each of its elements represents the template number associated with the loop_index. If we cannot find any template matching for a certain loop index, we use 0 for that value.

### 5.2 Recording the information into the PSS

This implementation is used in both learning and matching phase for restructuring the information from an input and recording them into the PSS. The PSS is implemented as a double linked list like §5.1. While traversing the AST from top to bottom, we extract the structural and architectural information and we save this information into the PSS by the procedure shown in Figure 5.2. Since most applications include loop bodies and it is important to optimize them in an efficient and safe ways, we only begin to record PSS when we meet a loop body. That means all defined patterns have a loop body as the outermost statement.

Several functions described next are added into c-semantics.c for recording necessary information into the PSS.

• **write_weights_into_PSS()**: This function records the predicted weights of the compound statements up to when this function is called into the PSS.
The type of statement structure
The last node of this statement structure
The weight of operations in this statement itself
Pointer to the previous node
Pointer to the next node

The major statement structure types are
L : Loop statement structure
A : Assignment statement structure
C : Condition statement structure
S : Switch statement structure
G : Compound assignment statement structure

Figure 5.1: Data Structure used for Representing Program Segments

- *template_matching()*: This function checks all recorded information in the PSSs for each loop_index and perform pattern and template matching. We write the result into the loop_template and increase the template_num according to the number of templates found so far.

- *loop_start()*: This function creates a node, put it into the PSS and increases the loop_level and loop_index. Moreover if the loop_level is not 0, we call the write_weights_into_PSS() function to save the predicted weights of previous compound statements. This function is inserted before the compiler calls genrtl_for_stmt(), genrtl_while_stmt(), genrtl_do_stmt() in expand_stmt().

- *loop_end()*: This function decreases the loop_level and calls the write_weights_into_PSS() function. When the loop_level becomes 0, we call the template_matching() function. This function is inserted after the compiler calls genrtl_for_stmt(), genrtl_while_stmt(), genrtl_do_stmt() in expand_stmt().

25
• **cond\_start()**: If the *loop\_level* is not 0, this function creates a node for this type and put into the *PSS*. Also we call the *write\_weights\_into\_PSS()* function to save the predicted weights from previous compound statements. This function is inserted before the compiler calls *genrtl\_if\_stmt()* in *expand\_stmt()*.

• **cond\_end()**: If the *loop\_level* is not 0, we call the *write\_weights\_into\_PSS()* function to save the predicted weights of previous compound statements. This function is inserted after the compiler calls *genrtl\_if\_stmt()* in *expand\_stmt()*.

• **switch\_start()**: If the *loop\_level* is not 0, we create a new node for this type and put into the *PSS*. Then, we call the *write\_weights\_into\_PSS()* function for previous compound statements. This function is inserted before the compiler calls *genrtl\_switch\_stmt()* in *expand\_stmt()*.

• **switch\_end()**: If the *loop\_level* is not 0, we call the *write\_weights\_into\_PSS()* function to save the predicted weights of previous compound statements. This function is inserted after the compiler calls *genrtl\_switch\_stmt()* in *expand\_stmt()*.

• **case\_end()**: This function will accumulate the weights of compound statements instead of recording them into the *PSS* directly. This continues until we meet the end of the switch statement. This function is inserted after the compiler calls *genrtl\_case\_label()* in *expand\_stmt()*.

### 5.3 Pattern Creation

After the compiler completes the procedure shown in Figure 5.2, several *PSSs* are produced. With these *PSSs*, we follow the procedure in Figure 5.3 to find frequently appeared segment of program. For this procedure, we need to make an assumption about the structural similarity.
Assumption 1: $PSS_1$ and $PSS_2$ have similar structural information if they have the same number of nodes, the same relations between nodes, and the same node types in $PSS$.

A Confirmed $PSS$ set is initialized once at the beginning of the methodology. All elements in Confirmed $PSS$ have two attributes. One attribute is named frequency. This one is used to check how many times this set appears in real applications. The other attribute is checks for the maximum weight of operations for each pattern. This attribute will be used for template definition in Section 3.2.2. Every time the compiler reads the input program, it completes the procedure in Figure 5.3 and gathers elements in the Confirmed $PSS$ until enough elements are collected.

After the compiler repeats this procedure with several test cases and amass enough elements in the Confirmed $PSS$ set, we sort the elements by the their frequency and choose the most frequently appeared elements and create patterns. Figure 5.4, 5.5 and 5.6 show patterns and their $PSS$ we created using this algorithm.

5.4 Template Creation

After we create patterns, we need to create templates in each patterns by following algorithms described in Section 3.2.2. We use the Table 5.1 for specific decision of architectural behaviors in our implementation.

5.5 Dynamic Option Selection

To support dynamic optimization option selection according to the template matching result, we need to modify the compiler internal structure and procedures to apply optimizations. In this procedure, the compiler checks the flag for each optimization option and calls the proper optimization functions. If we have any detected templates in an input program, we check which template the current instruction belongs to and decide whether we need to turn on this flag or not. If we
<table>
<thead>
<tr>
<th>Architecture Characteristics</th>
<th>Quantified Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply (short)</td>
<td>3</td>
</tr>
<tr>
<td>Multiply (long)</td>
<td>6</td>
</tr>
<tr>
<td>Multiply-ADD (short)</td>
<td>3</td>
</tr>
<tr>
<td>Multiply-ADD (long)</td>
<td>6</td>
</tr>
<tr>
<td>Compare</td>
<td>1</td>
</tr>
<tr>
<td>Move</td>
<td>1</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>1</td>
</tr>
<tr>
<td>Logical</td>
<td>1</td>
</tr>
<tr>
<td>Shift/rotate</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>6</td>
</tr>
<tr>
<td>Store</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.1: Intel XScale Architecture Characteristics and Quantified Value

need to change the flag value according to our decision, we save the original flag first and recover the original flag value after the compiler finishes to call all necessary optimization functions. If we cannot find any templates, we use the set of options from the command line or we use the highest and safest default optimization option provided by the compiler. Figure 5.7 shows how we process optimization option whose name is optimize1 in dynamic option selection.

Two functions were added to the compiler for dynamic option selection. These two functions become active only when template_num is not 0.

- `template_check_insn()`: This function checks which code segment the current instruction belongs to and check whether we need to apply dynamic option selection or not. It returns the template number of the current instruction.

- `search_flag()`: This function provides the bridge between the template database and the loop_template table. If we find that the current instruction belongs to the code segment matched by any template, this function decides whether the current optimization flag should be turned on or not.
In *expr.c*, we predict the weight of each compound statement. For example, if we meet `PLUS_EXPR`, we decide that the weight of this compound statement is 1. Also we check each operator and decide whether we need load or store operators here.

These two functions will be called by the following functions defined in these option,

- *cse.c*: cse-skip-blocks, cse-follow-jumps
- *explow.c*: force-addr
- *function.c*: force-mem
- *optabs.c*: force-mem
- *final.c*: peephole
- *recog.c*: peephole2
- *loop.c*: unroll-loops, rerun-loop-opt, strength-reduce, move-all-movables, reduce-all-givs, branch-count-on-reg
- *regmove.c*: regmove, expensive-optimization
- *regrename.c*: rename-registers
- *sched-rgn.c*: schedule-speculative, schedule-speculative-load, schedule-speculative-load-dangerous
- *unroll.c*: unroll-all-loops

The following chapter will show the evaluation results for this implementation.
Figure 5.2: Flowchart Graph for Recording the Information into a PSS
Put all PSSs into PSS_Candidate Set

---

Initialize PSS_Candidate =Ø

---

Delete x from PSS_Candidate and (increase the frequency of y by 1)

---

Choose x ∈ PSS_Candidate and y ∈ PSS_Candidate, x and y has the similar structural information?

---

(Put x or y) into Confirmed_PSS and delete x and y from PSS_Candidate

---

Delete x from PSS_Candidate and (increase the frequency of y by 1)

---

Delete x from PSS_Candidate

---

PSS_Candidate is empty?

---

This flowchart shows how we create patterns. From recorded PSSs, we find the most frequently appeared PSSs and create patterns with that.

---

Figure 5.3: Flowchart Graph for Pattern Creation
Figure 5.4: Patterns Created in Our Methodology (1)
for (i = var1; i < var2; i++)
{
    ...
do{

    if(j == var3){
        ....
    }
while(j > var4);

    ...
}

switch{
    case1:
        ....
        if(....)
        ....
    case2:
        ....
}

}
for (i = var1; i < var2; i++)
{
    ....
    for(.....)
    
    ....
    if(.....){
        ....
    }
    ....
}

for (i = var1; i < var2; i++)
{
    ....
    switch{
        case1:
            ....
            for(.....)
            ....
            case2:
                ....
        }
    ....
}

for (i = var1; i < var2; i++)
{
    ....
    switch(.....){
        ....
        for(.....){
            ....
            switch(.....){
                ....
            }
        }
    }
}
This flowchart shows how we apply options dynamically. Whenever the compiler checks whether flag is turned on or not, we check which template the current instruction belongs to and change the flag according to the OSO. We save the original flag and change the flag value. After the compiler applies the optimizations according to the flag value, we restore the original flag value and process next instruction.

Figure 5.7: Flowchart Graph to Apply Optimization for Dynamic Option Selection
Chapter 6

EXPERIMENTAL FRAMEWORKS

This section presents the experimental frameworks to evaluate this methodology. We used the Intel Xscale PXA255 architecture as our hardware testbed and tested with several benchmarks described in the following sections. For the whole evaluation process, we developed an automatic test platform to apply this methodology to any kind of compilers, benchmarks, and hardware platforms.

The rest of this chapter is organized as follows. Section 6.1 shows the detailed hardware platform we used for the evaluation of our implementation and section 6.2 shows the benchmark suites we used during evaluation process. Section 6.3 explains an automatic test platform developed for applying and evaluating our methodology.

6.1 Hardware Platform

Intel Xscale architecture is used for many mobile devices such as PDA and smart phone. The block diagram for this architecture is shown in Figure 6.1 and the main characteristics of this architecture is like following:

- 32bit Xscale RISC core
- 4 Memory banks
- 64MB Memory capacity in each memory bank
- 32KB instruction cache
- 32KB data cache
- 2KB cache for stream multimedia data
- 15 General purpose I/O ports
- 16 DMA channels

We only used PXA255 board for getting an actual performance improvement and all other works were done in the host machine connected to the board. With Xscale GCC compiler, we generated all executable files in the host machine and mount them to the board. For serial connection between the host machine and PXA255 board, we used minicom provided by the operating system for serial communication.

Figure 6.1: Block Diagram for PXA255 Xscale Architecture
6.2 Benchmark Suite

We have chosen 2 benchmark suites for this experimental works described in following:

- **Mibench**: Benchmark suites for embedded systems including applications for automotive and industrial control, network, security, consumer devices, office automation and telecommunications [14]

- **CommBench**: Telecommunication benchmark for evaluating network processors including 8 programs for packet header processing and payload processing.

6.3 Automatic Test Platform

We developed an automatic test platform shown in Figure 6.2. This platform includes the the OSO searching script, evaluation script, and the configure database. The configure database includes the following items:

- List of benchmark suites
- List of hardware platforms
- List of compilers matching to each hardware platform

It has the possible list of benchmark suites and hardware platforms. It also has the mapping between each hardware platform and compiler. This database is used by the OSO searching and evaluation scripts and makes possible to apply our methodology to any kind of compilers, hardware platforms and benchmark suites. Section 6.3.1 and 6.3.2 explains the details of these two scripts in automatic test platform.
Figure 6.2: Automatic Test Platform for This Methodology

6.3.1 OSO Searching Script

Whenever we use this methodology on different architecture, we need to get a new *OSO* for each template. This script helps to set up and begin OSO searching tool automatically.

We need to give inputs to this script as follows:

- **Option list**: We need to give the location of the option list file. This option list should include all options listed in the Table 4.2.

- **Repeat**: We can set up how many times we want to repeat for testing each instance file for a template. If we increase this number, we can find more
accurate execution time of each instance file, moreover, we can find more accurate OSO for each template. Default number is 10.

Once we set up the above items, the script first checks this configuration is correct. If there is no configuration errors, it begins to find OSO without aborting until it finds all OSOs for all templates.

6.3.2 Evaluation Script

This script is used for evaluating our methodology and we need to give inputs as follows:

- **Type of architecture:** Script will find the compiler to be used automatically according to the type of architecture.

- **Type of compiler:** Instead of specifying the architecture, we can also give the compiler name. If we set up the type of architecture in the above, we do not need to set up this item.

- **List of Benchmark:** We can choose the list of benchmark we want to test.

- **Test Mode:** We can choose to test our methodology or normal compiler option. If we want to test benchmarks with any option instead of using our methodology, we can specify here

- **Repeat:** We can set the number of repeat times for better accuracy of evaluating. Default number is 5.

Once we start the script, it checks the above configuration is correct first. Then it compiles all the benchmarks we want to use and runs it as many as times we specify.

Based on this experimental frameworks, the following chapter shows the analysis of the evaluation results.
Chapter 7

PERFORMANCE EVALUATION

This chapter shows the performance evaluation result and analysis of our methodology. The evaluation results support our methodology by showing the positive performance improvement over any default optimization option with given benchmark suites.

The rest of this chapter is organized as follows. Section 7.1 shows the summary of performance evaluation result. Section 7.2, 7.3, 7.4 and 7.5 show the observations and analysis of the experimental results. Section 7.6 shows the conclusion of this evaluation works.

7.1 Summary of Result

We have chosen 13 cases from the MiBench benchmark and 5 cases from CommBench benchmark and corresponding results.

Figures 7.1 and 7.3 show template matching rate over pattern matching in MiBench and CommBench benchmarks, respectively. Figures 7.2 and 7.4 show the performance improvement over -O3 option in two benchmark suites.

In MiBench, we achieved an average of 1.1% improvement from all 13 cases and an average of 1.8% improvement from the 9 cases which have good template matching rate. In CommBench, we achieved an average of 1.8% improvement for all cases. We have 4 main observations about this evaluation works and each observation is explained in the following sections

41
Figure 7.1: Template Matching Rate over Pattern Matching Rate in MiBench

Observation 1 (See section 7.2 in detail): Good template matching rate over pattern matching rate means that we can apply dynamic option selection for most code segment in a given program and have additional performance improvement.

Observation 2 (See section 7.3 in detail): Test cases which have good template matching rate tend to have positive performance improvement over any default optimization options.

Observation 3 (See section 7.4 in detail): Pattern matching rate over the whole program is important to decide template matching rate and performance of improvement.

Observation 4 (See section 7.5 in detail): Even though we have good template matching rate, if we choose improper decision region, it is hard to expect additional
Figure 7.2: Performance Evaluation for MiBench

performance improvement.

7.2 Template Matching Rate

The importance of the results shown in Figure 7.1 and 7.3 is that we can check the efficiency of the template matching algorithm we are using. Some test cases show good template matching rate, especially Qsort and Dijkstra with more than 50% in MiBench and all cases in CommBench. However other test cases still have less than 50% matching rate which means that even though we find many patterns, we could not find the proper templates and failed to apply OSO for additional optimizations.
7.3 Template Matching Rate vs. Performance Improvements

All 4 cases which show poor template matching rate in Figure 7.1 have negative performance in comparison to the baseline. On the other hand, all 9 cases in Figure 7.1 which showed good template matching rate have positive performance improvement. And all cases in Figure 7.3 showed positive performance improvement. However such improvements are still relatively small.

However, we also can find some cases that have good template matching rate but show poor improvements even though they are positive. For example, Qsort in MiBench has a 100% template matching rate but shows only 1% of a performance improvement and most cases in CommBench shows small improvement even though they have a good template matching rate.
7.4 Pattern Matching Rate

Our dynamic option selection methodology was only used in a small portion of the whole program. If we only found a very small number of patterns and it forced us to use OSOs for a small portion of the code segment, it means that we still lose the opportunity for additional improvements for most of the program. This observation is supported by Basicmath and Susan in MiBench which have 9.6% and 1.7% of pattern matching rate over the whole program, respectively. Also Rtr in CommBench has 4.6% of pattern matching rate over the whole program and supports this observation.
7.5 Properness of Decision Region

Our template matching algorithm cannot find the proper decision region for each code segment. In this case, even though we found many portions of the program that matched to templates and used special OSOs for each, it still cannot have additional improvement because we did not choose the proper OSO. This observation is supported by Qsort, Bitcount and Basicmath in MiBench which have 40.0%, 45.0% and 28.9% of pattern matching rate over the whole program, respectively. Also Drr in CommBench showed 24.6% of pattern matching rate over the whole program but showed poor improvement.

7.6 Conclusion

In our experimental results, most test cases which showed good template matching rate over pattern matching showed the positive performance improvement. However, few test cases showed negative performance improvement even though they have good template matching rate. It means our template matching algorithm could not find the proper template and it leaded the compiler to choose wrong OSO. Also pattern matching rate over the program is low for most test cases which means we do not have enough number of patterns/templates in our experimental works. Our experimental results do not show great improvements, but it opens many possible future research directions discussed in Chapter 9.
Chapter 8

RELATED WORK

This chapter introduces some related works to our research and discusses about them. Section 8.1 introduces related research of program similarity detection and section 8.2 shows related research of automated option selection.

8.1 Program Similarity Detection

Kontogiannis et al. [15] developed code-to-code matching techniques for detecting code clones and for measuring the similarity distance between two program segments. They used the abstract syntax tree (AST) as the program representation scheme. Baxteret et al. [16] used standard parse analysis techniques to detect exact and near miss clones over arbitrary program segments in program source code by transforming source code into an AST. Ducassw et al. [17] used simple line-based string matching to detect duplicated code. Lee and Hall [18] developed a tool (Code Isolator) to extract ”hot spot” program segments from large scientific applications and targets tuning specially on the isolated segments.

In contrast to these studies, our program detection method operates in terms of program structure. The source code is transformed into a program structure stream representation and each program structure is associated to a weight vector. We define similarity between two program segments based on their structure similarity and through the distance between their weight vectors.
8.2 Automated Option Selection

Wu et al. [8] proposed a technique for an automated search for optimization options. It creates a database for each application domain by turning a set of kernel programs from this domain. The optimized sets of optimization options are used to compile the other domain-specific matching programs. Cavazos and O’Boyle [9] presented another method for an automated search for optimization options. It focuses on method (function) level instead of the whole program. The matching strategy is based on the similarity of pure method features. What makes our method different is that we focus on a fine-grain level program analysis through program segments. We narrow down the similarity of program segments to the syntax structure and further more we correlate the segment features to the underlying architecture behaviors.

Annavaram [10] and Lau et al. [11] discuss the correlation between program source code and performance. They examined the use of code signatures obtained through periodic sampling to predict performance for database applications and SPEC2000. Hoste et al. [12, 13] propose a methodology to predict program performance on any architecture. They measure the architecture-independent characteristics of programs and then relate measured information to pre-profiled benchmarks. In contrast to these studies, our method employs a different criteria to determine program similarity. Furthermore, our method can be used not only to identify similar program segments, but also to direct compiler to generate a custom highly optimized sequence of optimization options, which best fits each detected program segment.
Chapter 9

CONCLUSIONS AND FUTURE WORK

Our methodology for dynamic option selection has three phases based on the static analysis of a program: (1) In the learning phase, we learn from the input program and create a template database based on structural and architectural information. (2) In the searching phase, we find the OSOs for each template and maps to template database. (3) In matching phase, we extract the information from the input program and perform the template matching. Finally, we apply the different OSOs dynamically for each template found in input program.

While searching phase is implemented outside of the compiler and work independently, the learning and matching phases are implemented in GCC compiler v3.3 by modifying the front-end of the compiler. Our implementations are tested on the Intel Xscale PXA255 architecture with MiBench and CommBench. When compared to -O3 optimization options, it showed an average of a 1.8% performance improvement from test cases which have more than a 50% template matching rate. Moreover, it showed an average of 1.4% of improvement from all test cases with additional compilation overhead being less than 1.0%.

This evaluation result shows our methodology gives additional performance improvement with a smaller set of options than using -O3. All of this by paying only small additional compilation time. However, this research opens the door to many future works and they will be discussed in following part.
• We need to create more patterns and templates. Right now we only have 11 patterns and each pattern has 2-4 templates. Because of the small size of the template database, we still have a large portion of code segments which cannot match to any templates.

• We need to consider about loop boundaries. Even though we have two loops with different loop boundaries, their predicted weights of operation must be different which it is not the case in our current methodology. For this work, we can define more detailed templates that have a polynomial to represent loop boundaries since most loop boundaries are defined as variables, not constant.

• With the classification of basic, global and local optimization in the current searching tool, we can classify them into the option’s goal. In this case, we can reduce some redundant usage of options which perform the same kind of optimization. It will simplify the searching algorithm by choosing one option from each goal.

• In dynamic option selection, we can extend the current template definition. We can consider functions or files instead of code segments. In this case, we may need to find the OSO for each function and file, so it can be more efficient in implementation and overall performance.

• This methodology need to be applied to the latest version of GCC compiler for the x86 architecture. The most challenging issue is that the latest version of GCC uses SSA form instead of RTL. Thus we need to resolve this issue first before porting to the newer version of GCC compiler. Also this methodology can be ported to other kind of compilers, like Open64.

• This methodology can be applied to the automatic tuning program. In the tuning program, we can find the important sections for achieving an additional
improvement. However, we still can meet a problem if we do not know which optimization we can apply to those sections. With applying this methodology to the automatic tuning program, we can find the proper optimization for each important sections in a program automatically.
Appendix A

TEST PLATFORM CONFIGURATION

A.1 Hardware Configuration

(i) Create your account on the xscalehost machine.

(ii) In the xscalehost machine, boot the PXA255 board by using minicom.

```
[ejpark@xscalehost /]$ minicom
Welcome to minicom 1.83.1
OPTIONS: History Buffer, F-key Macros, Search History Buffer, I18n
Press CTRL-A Z for help on special keys

RedBoot> load -r -b 0xa0200000 vmlinuz-intel-dbpxa250
RedBoot> go 0xa0200000
```

(iii) Create your account on the PXA255 board.

(iv) Make a directory what you want to use on the xscalehost machine and give
    the rwx permission.

```
[ejpark@xscalehost /]$ mkdir /home/ejpark/test
[ejpark@xscalehost /]$ chmod -R 777 /home/ejpark/test
```

(v) Open /etc/exports and add the following line,
/home/ejpark/test *(rw,no_root_squash,no_all_squash)

(vi) Restart NFS by using following command.

```
[ejpark@xscalehost /]$ /etc/rc.d/init.d/nfs restart
```

(vii) Log in to the PXA255 board (192.168.7.10) and mount this directory into the pxa255 board file system. When 192.168.2.41 is the IP address of the xscalehost machine,

```
mount -t nfs 192.168.2.41:/home/ejpark/test /home/ejpark/test
```

(viii) Modify Makefile on the xscalehost machine to use xscale-elf’s include and uclibc’s c library.

```
-I/usr/local/gnupro-3.1/H-i686-pc-linux-gnulibc2.1/xscale-elf/include
-L/opt/uclibc/le/lib -lc
```

(ix) Compile source code with modified Makefile on xscalehost to get executable file and execute the executable file on PXA255 board in the mounted directory.

Note: above 1,2,3,5 and 6 require super user mode

### A.2 Automatic Test Platform Configuration

(i) First give the three environment values by modifying one Makefile existing in the top level of the whole benchmarks:

```
[ejpark@xscalehost /]$ cd /home/common/AutoTestPlatforms
[ejpark@xscalehost /]$ cp Makefile Makeall.sh runall.sh
your_working_directory

# Put the architecture name
# Choose one of i386, i686, or xscale
ARCHI =
```
# If you do not choose the architecture name, 
# put this line manually.
# Choose one of gcc or xscale-elf-gcc
# If you already chose the architecture name, leave this blank.
CC = xscale-elf-gcc

# Choose the name of benchmark suites.
# If you need to choose the specific cases in a benchmark suite,
# go to each Makefile and edit them.
# Choose one of CommBench, MiBench, Mediabench and DSP
BENCH1 = CommBench
BENCH2 = MiBench
BENCH3 =
BENCH4 =

(ii) run Makeall.sh on the xscalehost machine

(iii) Go to the PXA255 board and execute runall.sh. In runall.sh file, you can simply set the repeat time like the following:

# Put the number as you want to repeat
repeat=10;

for((j=1; j<=repeat; j+=1)); do
...
done

In a for loop, we execute the scripts file provided by each benchmark suite.
Appendix B

OSO SEARCHING TOOL MANUAL

(i) First copy three files under your the directory what you want to get OSO,

```bash
[ejpark@xscalehost /]$ cd /home/common/Algorithms
[ejpark@xscalehost /]$ cp Algorithm format.txt options.txt your_working_directory
```

(ii) Modify Makefile like the following example:

```bash
CC=gcc
...
$(CC) $(OPTIONS) -o test test.c
```

By using OPTIONS tag, algorithm tool can apply the combination of options automatically.

(iii) Modify format.txt to put command line for executing. For example we use executable file named test and 1000 as an argument,

```bash
# ./test 1000
```

(iv) Modify options.txt if you need to add or delete option set by the following format:

```bash
# 000000000001 force-addr
% 000000000002 fmove-all-movables
```
For the option set what you want to use, put a pound sign in the beginning of the line. If you do not include some options for testing, put a percent sign in the beginning of the line. Identification number should be the power of 2 for making the combination of options in the OSO searching tool. And put the exact option name used in a command line.

(v) Run ./Algorithm and it will generate performance.txt for the result. In performance.txt, you can find the several sets of following lines,

c2 Execution_time
OPTIONS=... 

First number means the combination of identification number of options. C2 means that we used three options whose identification numbers are 40, 80 and 2. Execution_time shows the achieved execution time with this combination. We can check how much improvement we got by this combination of options for the current template.
Appendix C

ADDITIONAL TEST RESULT

C.1 Template and Pattern Matching Rate

Pattern matching rate means the rate over the whole program and template matching rate means the rate over the pattern matching rate.

<table>
<thead>
<tr>
<th>Case name</th>
<th>Pattern Matching Rate</th>
<th>Template Matching Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drr</td>
<td>24.5%</td>
<td>92 %</td>
</tr>
<tr>
<td>Frag</td>
<td>26.2%</td>
<td>86 %</td>
</tr>
<tr>
<td>Reed</td>
<td>9.6%</td>
<td>91 %</td>
</tr>
<tr>
<td>Rtr</td>
<td>4.7%</td>
<td>83 %</td>
</tr>
<tr>
<td>Qsort</td>
<td>40.0%</td>
<td>100 %</td>
</tr>
<tr>
<td>Basicmath</td>
<td>29.9%</td>
<td>83 %</td>
</tr>
<tr>
<td>Bitcount</td>
<td>45.0%</td>
<td>50 %</td>
</tr>
<tr>
<td>Susan</td>
<td>1.7%</td>
<td>55 %</td>
</tr>
<tr>
<td>Jpeg</td>
<td>9.6%</td>
<td>71 %</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>7.9%</td>
<td>100 %</td>
</tr>
<tr>
<td>Patricia</td>
<td>21.8%</td>
<td>44 %</td>
</tr>
<tr>
<td>CRC32</td>
<td>13.0%</td>
<td>33 %</td>
</tr>
<tr>
<td>FFT</td>
<td>34.4%</td>
<td>44 %</td>
</tr>
</tbody>
</table>
### C.2 Number of Template found and improvement in Each Pattern

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Number of Template Found</th>
<th>Min. Improvement</th>
<th>Max. Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>25%</td>
<td>75%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2%</td>
<td>75%</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4%</td>
<td>74%</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>20%</td>
<td>74%</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3%</td>
<td>75%</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>29%</td>
<td>72%</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>7%</td>
<td>73%</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3%</td>
<td>75%</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>27%</td>
<td>75%</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1%</td>
<td>75%</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


