MODEL-DRIVEN CODE OPTIMIZATION

by

Min Zhao

B.E. Computer Science and Engineering, Xi'an Jiaotong University, P.R. China, 1996

M.S. Computer Science, University of Pittsburgh, 2001

Submitted to the Graduate Faculty of

Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Min Zhao

It was defended on

August 4, 2006

and approved by

Bruce R. Childers, PhD, Assistant Professor, University of Pittsburgh

Mary Lou Soffa, PhD, Professor, University of Virginia

Youtao Zhang, PhD, Assistant Professor, University of Pittsburgh

Peter Lee, PhD, Professor, Carnige Mellon University

Dissertation Co-adviors: Dr. Bruce R. Childers, Assistant Professor, University of Pittsburgh

and Dr. Mary Lou Soffa, Professor, University of Virginia

Copyright © by Min Zhao

2006

MODEL-DRIVEN CODE OPTIMIZATION

Min Zhao, Ph.D.

University of Pittsburgh, 2006

Although code optimizations have been applied by compilers for over 40 years, much of the research has been devoted to the development of particular optimizations. Certain problems with the application of optimizations have yet to be addressed, including when, where and in what order to apply optimizations to get the most benefit. A number of occurring events demand these problems to be considered. For example, cost-sensitive embedded systems are widely used, where any performance improvement from applying optimizations can help reduce cost. Although several approaches have been proposed for handling some of these issues, there is no systematic way to address the problems.

This dissertation presents a novel model-based framework for effectively applying optimizations. The goal of the framework is to determine optimization properties and use these properties to drive the application of optimizations. This dissertation describes three framework instances: FPSO for predicting the profitability of scalar optimizations; FPLO for predicting the profitability of loop optimizations; and FIO for determining the interaction property. Based on profitability and the interaction properties, compilers will selectively apply only beneficial optimizations and determine code-specific optimization sequences to get the most benefit. We implemented the framework instances and performed the experiments to demonstrate their effectiveness and efficiency. On average, FPSO and FPLO can accurately predict profitability 90% of the time. Compared with a heuristic approach for selectively applying optimizations, our model-driven approach can achieve similar or better performance improvement without tuning the parameters necessary in the heuristic approach. Compared with an empirical approach that experimentally chooses a good order to apply optimizations, our model-driven approach can find similarly good sequences with up to 43 times compile-time savings.

This dissertation demonstrates that analytic models can be used to address the effective application of optimizations. Our model-driven approach is practical and scalable. With model-driven optimizations, compilers can produce higher quality code in less time than what is possible with current approaches.

TABLE OF CONTENTS

LIST OF TABLES VIII
LIST OF FIGURES IX
ACKNOWLEDGEMENTS XI
1.0 INTRODUCTION 1
1.1 MOTIVATION 1
1.2 OVERVIEW OF THIS RESEARCH 4
1.3 ORGANIZATION OF THIS DISSERTATION 5
2.0 BACKGROUND AND RELATED WORK 7
2.1 PRIOR WORK 7
2.1.1 Empirical 9
2.1.2 OSE (Analytic Resource Model-based) 10
2.1.3 Jalapeño (Experimental Resource Model-based) 10
2.1.4 Unimodular (Analytic Model-based) 11
2.1.5 Analytic Interaction 12
2.2 PRIOR WORK AND THIS RESEARCH 13
3.0 OVERALL DESIGN OF THE MODEL-BASED FRAMEWORK 15
3.1 COMPONENTS OF THE FRAMEWORK 16
3.1.1 Code Models 16
3.1.2 Optimization Models 17
3.1.3 Resource Models 18
3.1.4 Engine 19
3.2 USES OF THE FRAMEWORK 20
4.0 FPSO: PREDICTING PROFITABILITY OF SCALAR OPTIMIZATIONS 21
4.1 CODE MODELS FOR REGISTERS AND COMPUTATION 23
4.2 OPTIMIZATION MODELS 23
4.2.1 PRE Optimization Model 24

4.2.2 LICM Optimization Model 25
4.2.3 VN Optimization Model 26
4.2.4 Register Allocation Optimization Model 29
4.2.5 Other Scalar Optimizations 30
4.3 RESOURCE MODELS FOR REGISTERS AND COMPUTATION 31
4.4 PROFITABILITY ENGINE 31
4.5 AN EXAMPLE OF PROFIT-DRIVEN VN 36
4.6 EXPERIMENTAL RESULTS 39
4.6.1 Selectively Applying Optimizations 39
4.6.1.1 A heuristic approach 40
4.6.1.2 Comparing prediction accuracy 42
4.6.1.3 Comparing performance improvement 45
4.6.1.4 Comparing compile-time overhead 49
4.6.2 Searching for Code-specific Optimization Sequences 53
4.6.2.1 Comparing compile-time overhead 54
4.6.2.2 Comparing performance improvement 55
5.0 FPLO: PREDICTING PROFITABILITY OF LOOP OPTIMIZATIONS 57
5.1 CODE MODEL FOR CACHE 57
5.2 OPTIMIZATION MODELS 59
5.2.1 Loop Interchange 59
5.2.2 Loop Unrolling 60
5.2.3 Loop Tiling 61
5.2.4 Other Loop Optimizations 63
5.3 CACHE MODEL 63
5.4 PROFITABILITY ENGINE 65
5.5 EXPERIMENTAL RESULTS 66
5.5.1 Model Accuracy 66
5.5.2 Comparing with Always-applying Approach 69
5.5.3 Choosing the Best Optimization 73
5.5.4 Compile-time Overhead for Prediction 74
6.0 FIO: DETERMINING THE INTERACTION PROPERTY 76

6.1 CODE MODEL FOR INTERACTION 77
6.2 A SPECIFICATION LANGUAGE 78
6.2.1 SpeLO PRECONDITION Section 79
6.2.2 SpeLO ACTION Section 80
6.3 OPTIMIZATION MODELS 80
6.3.1 Dead Code Elimination 81
6.3.2 Partial Redundancy Elimination 81
6.3.3 Value Numbering 82
6.3.4 Other Optimizations 84
6.4 INTERACTION ENGINE 84
6.4.1 Generating Specific Conditions 86
6.4.2 Matching Conditions 90
6.5 AN EXAMPLE OF DETERMINING THE INTERACTION 92
6.6 USING INTERACTION TO ORDER OPTIMIZATIONS 94
6.7 EXPERIMENTAL RESULTS 95
6.7.1 Evaluation Function: the Number of Optimizations 96
6.7.1.1 Compile-time overhead 96
6.7.1.2 Performance improvement 98
6.7.1.3 Memory requirement 99
6.7.2 Evaluation Function: Profitability 100
6.7.2.1 Compile-time overhead 100
6.7.2.2 Performance improvement 101
6.7.2.3 Memory requirement 103
7.0 CONCLUSIONS 104
7.1 SUMMARY OF CONTRIBUTIONS 104
7.2 LIMITATIONS 106
7.3 FUTURE WORK 107
APPENDIX A OPTIMIZATION MODELS 110
APPENDIX B RESOURCE MODEL FOR COMPUTATION 118
APPENDIX C EXPERIMENTAL RESULTS FOR ATHLON MACHINE 120
BIBLIOGRAPHY 126

LIST OF TABLES

Table 2.1: Approaches to explore the effective application of optimizations 8
Table 4.1: Incremental computation of the new register code model.. 32
Table 4.2: Updates of the computation code model 33
Table 4.3: Computing profit on registers (R_{total}) and computation (C_{total})................................... 35
Table 4.4: Prediction accuracy of H-PRE and P-PRE 42
Table 4.5: Prediction accuracy of H-LICM and P-LICM.. 43
Table 4.6: Prediction accuracy of P-VN 44
Table 4.7: Compile-time for PRE 50
Table 4.8: Compile-time for LICM 51
Table 4.9: Compile-time for VN... 52
Table 5.1: Terms used in cache model.. 64
Table 5.2: Prediction accuracy for single-loop nest benchmarks 68
Table 5.3: Prediction accuracy for multi-loop nest benchmarks 69
Table 5.4: Compile-time overhead for prediction (millisecond) 74
Table 6.1: Semitics of primitive operations.. 80
Table 6.2: Generating enabling and disabling conditions for check_code_pattern..................... 86
Table 6.3: Generating enabling and disabling conditions for check_depend.............................. 88
Table 6.4: Generating post conditions for primitive operations 90
Table 6.5: Compile-time overhead of three approaches (minutes)... 97
Table 6.6: Comparing the number of optimization applied.. 98
Table 6.7: Memory requirement of our approach (KB).. 99
Table 6.8: Compile-time overhead of three approaches (minutes)... 101
Table 6.9: Memory requirement of our approach (KB).. 103

LIST OF FIGURES

Figure 3.1: Overall design of model-based framework 15
Figure 4.1: Structure of FPSO 21
Figure 4.2: An example of PRE impacting registers 22
Figure 4.3: PRE optimization model 25
Figure 4.4: An example of LICM 26
Figure 4.5: LICM optimization model 26
Figure 4.6: An example of VN 27
Figure 4.7: VN optimization model 28
Figure 4.8: Register allocation optimization model 30
Figure 4.9: Impact of PRE on computation code model 34
Figure 4.10: An example of model-driven VN 37
Figure 4.11: Improvement of heuristic-driven PRE with different limits 41
Figure 4.12: Improvement of heuristic-driven LICM with different limits 41
Figure 4.13: Memory access improvement for PRE 46
Figure 4.14: Run-time performance improvement for PRE 46
Figure 4.15: Memory access improvement for LICM 47
Figure 4.16: Run-time performance improvement for LICM 47
Figure 4.17: Memory access improvement for VN 48
Figure 4.18: Run-time performance improvement for VN 48
Figure 4.19: Compile-time of the experimental and model-based approaches 54
Figure 4.20: Performance of three approaches 55
Figure 5.1: A loop nest and its code model 58
Figure 5.2: Loop interchange optimization model 60
Figure 5.3: Loop unrolling optimization model 60

Figure 5.4: Loop tiling optimization model ... 62
Figure 5.5: Loop interchange on *irkernel* with different cache models ... 67
Figure 5.6: Performance impact of always-applying approach ... 72
Figure 5.7: Improvement of profit-driven approach vs. always-applying ... 72
Figure 5.8: Performance impact of profit-driven approach ... 72
Figure 5.9: Accuracy and distribution of the most beneficial optimizations ... 73
Figure 6.1: Overview of FIO ... 76
Figure 6.2: The format of SpeLO specification ... 78
Figure 6.3: DCE optimization model ... 81
Figure 6.4: PRE optimization model ... 82
Figure 6.5: VN optimization model ... 83
Figure 6.6: The overview algorithm for the interaction engine ... 85
Figure 6.7: Matching O_i's E/D conditions with post conditions ... 91
Figure 6.8: An example of determining the interaction ... 93
Figure 6.9: Determining a good optimization sequence using interaction ... 95
Figure 6.10: Comparing performance improvement ... 99
Figure 6.11: Comparing performance improvement ... 102

ACKNOWLEDGEMENTS

During the phase of this research I have been aided by a number of people. I take this opportunity to thank them for the various contributions that they have made to help me reach my goals.

I would like to express my deepest appreciation and thanks to my co-advisors, Dr. Bruce R. Childers and Dr. Mary Lou Soffa for their support, encouragement, friendship, and guidance. Without them, this dissertation would never have happened. I am also grateful to the members of my thesis committee, Dr. Youtao Zhang and Dr. Peter Lee, for their review and suggestions concerning this research.

I would also like to thank my fellow graduate students. I am happy to be officemate with Yuqiang, Naveen, Jonathan and Jason. It is fun and helpful to discuss various research questions with them.

I would especially like to thank my families. Thanks to my husband, Shuyi, for his constant love, emotional support and understanding through all the stages of this degree. I give my deepest appreciation to my son, Raymond, for his unconditional love and being my motivation to finish. I would like to thank my parents, parents-in-law, and my sister for their love and overwhelmingly faith in my ability to achieve my goals.

Last, I would like to thank the University of Pittsburgh and the cast of thousands that make the University of Pittsburgh such a wonderful place to learn. This research has been supported in part by the National Science Foundation, Next Generation Software, grants CNS-0305198 and CNS-0203945, and an Andrew Mellon Graduate Fellowship.

1.0 INTRODUCTION

This dissertation addresses the effective application of code optimizations. Although compilers have applied optimizations for over 40 years, certain properties of optimizations are still not well understood. This dissertation describes a model-based framework for determining optimization properties, which are then used to drive the application of optimizations to get the most benefit.

1.1 MOTIVATION

The field of code optimization has been extremely successful over the past 40 years. As new languages and new architectures have been introduced, new optimizations have been developed to target and exploit both the software and hardware innovations. Various reports from research and commercial projects have indicated that the performance of software can be improved by 20 to 40% with aggressive optimization [38].

Most of the success in the field has come from the development of particular optimizations, such as Partial Redundancy Elimination [6] and Path Sensitive Optimization [5]. However, it has long been known that there are various problems with the application of optimizations. First, optimizations may degrade performance in certain circumstances. For example, Briggs and Cooper reported improvements ranging from +49% to –12% for their algebraic reassociation optimization [6]. However, so far there is no efficient way to determine the profitability of optimizations to avoid the performance degradation. Second, optimizations enable and disable other optimizations so the order of applying optimizations can have an impact on performance [15], [46], [51]. Also, the optimization configuration can impact the effectiveness of optimizations (e.g., how many times to unroll a loop or tile size) [36], [14], [26]. However, typically, compilers apply optimizations in a predetermined order and assume a fixed

optimization configuration. The choice of the order and optimization configuration is guided by a compiler writer's expertise and used for all programs. In some compilers, especially high-performance compilers for parallel computing systems, the choice can also be directed by users' specifications [19]. It is unrealistic to expect that a single choice of order and optimization configuration can achieve the best performance for every program. Because of these problems, optimizing compilers are not achieving the potential benefits of applying optimizations.

Instead of trying to understand and solve the problems, the compiler community has ignored them for the most part because there were performance improvements. However, a number of events are forcing these problems to be considered. First, because of the continued growth of embedded systems and the use of high level languages in writing software for embedded systems, there is a need for high quality optimizing compilers that can handle the challenges offered by embedded systems. For example, resource constraints are more severe than in desktop computers and thus optimizing compilers must be able to intelligently apply optimizations to better satisfy these constraints. Furthermore, embedded systems have irregular resources (i.e., irregular register file), and thus optimizing compilers must be able to consider these resources and apply optimizations to exploit them. Also, embedded systems are very cost-sensitive. Any performance improvement can help reduce cost. Another event that has brought these problems to focus is the trend toward dynamic optimization. Dynamic optimization requires that we understand optimizations in order for optimizations to be effective. It is unclear when and where to apply optimizations dynamically and how aggressive optimizations can be and still be profitable after factoring in the cost of applying optimizations. Lastly, as new optimizations continue to be developed, the incremental performance improvement is shrinking. The question is whether the field has reached its limit or the problems that have been ignored simply limit the progress. We believe the latter is true.

To systematically tackle these problems, we need to identify and study the properties of optimizations, especially those that target the application of optimizations. For example, to selectively apply only beneficial optimizations, we need to determine the impact of applying an optimization at a particular code point given the resources of the targeted platform (i.e., profitability property). To efficiently determine a code-specific optimization sequence, we also need to detect the disabling and enabling interferences among optimizations (i.e., the interaction property).

There are a number of challenges in determining the properties of optimizations. First, optimization properties depend on many factors, including 1) the specific code being compiled, 2) the semantics and implementation of optimizations, and 3) the target machine resources. For example, the profitability of a given optimization varies widely as a function of the input program and machine resources. These factors have different characteristics. To determine optimization properties with accuracy, all factors need be characterized and formalized. Also, for different optimizations, the factors that dominantly impact optimization properties are different. For example, loop behavior dominates data cache performance [36]. Thus, for loop optimizations, the cache is the dominant resource impact and can be used as an indicator for overall profitability. The knowledge of the dominant factors is important in determining optimization properties.

Previous research on optimization properties has been very limited [15]. However, more recently, there has been a flurry of research activity focusing on studying optimization properties. There are generally two approaches. One approach is through formal techniques. These include developing formal specifications of optimizations, analytic models, and proofs through model checking and theorem proving. This approach has been used to prove the soundness and correctness of optimizations [33], [35], [39]. Work has also been done to automatically generate the implementation of optimizations and detect interactions among optimizations through formal specifications [50], [51], [28], [32]. Another approach uses experimental techniques. That is, after actually performing optimizations, the properties are experimentally determined (e.g., executing the code to evaluate performance for determining profitability). The empirical approach has been used to determine the correctness of an optimizer through comparing the unoptimized and optimized code [23]. It has also been used to determine the profitability and interactions of optimizations to find a good order and configuration to apply optimizations [1], [15], [26], [46], [48].

Although the empirical approach can be effective in addressing some of the problems with the application of optimizations, a major disadvantage is its high cost and scalability [15]. For example, to search for good optimization sequences, the empirical approach may involve dynamic measures (e.g., dynamic instruction count or cycle count). And thus, the execution of the program is required. It may take hours, or even days, to find a good optimization sequence

for a program [30]. Ideally, we need a systematic approach for addressing the effective application of optimizations, which is practical, efficient and scalable [55].

1.2 OVERVIEW OF THIS RESEARCH

This research presents an effective model-driven approach to address the problems with the application of optimizations. It develops a general framework that uses analytic models to study optimization properties. With the framework, it presents instances that determine profitability and the interaction properties. Given code context, machine resources and the optimization, profitability can be accurately predicated and the interactions among a set of optimizations can be automatically detected.

This research develops different techniques for effectively applying optimizations based on optimization properties. Profitability is used to apply only beneficial optimizations to avoid performance degradation. Profitability can also be used to evaluate candidate sequences in searching for a good order to apply optimizations. The interaction property is used to determine a code-specific optimization sequence. The search space is greatly reduced by using the disabling and enabling interactions among optimizations.

Using the general framework, this research presents two framework instances, FPSO and FPLO, for predicting the profitability of scalar and loop optimizations. To predict profitability, models of machine resources, optimizations and code are developed. We analyzed the machine resources and found that the registers, computation (i.e., functional units) and cache are the most important factors that impact performance. We developed a model for each machine resource to describe the configuration and the cost to use the resource. For scalar and loop optimizations, the resources that dominantly impact profitability are different. Thus, FPSO considers registers and computation, and FPLO considers cache. We studied the semantics of optimizations and developed models for a set of scalar optimizations and loop optimizations. These optimization models specify how the optimizations change the code and thus impact the use of machine resources. We also automatically extract code models from the program to express those code characteristics that are changed by optimizations and impact the use of machine resources. In

FPSO and FPLO, there is a profitability engine that uses the models to predict the profit of applying an optimization at any code point where the optimization is applicable.

This research also presents a framework instance, FIO for detecting the interaction property. To automatically detect interactions among a set of optimizations, the models of optimizations and code are developed. An optimization specification language, SpeLO, is designed to express (1) the conditions under which an optimization can be safely applied and (2) the actions of applying the optimization in the code. An optimization model using SpeLO is developed for each optimization. As part of FIO, there is an interaction engine that uses these models to generate the specific enabling, disabling and post conditions for each optimization at a program point. These enabling and disabling conditions are then matched with the post conditions of other optimizations to determine the interaction property.

Our framework instances have been developed, implemented and experimentally evaluated. We compare our model-driven approach with a heuristic approach for selectively applying optimizations. Our model-driven approach can perform as good as, or even better than, the heuristic approach without having to tune parameters necessary in the heuristic approach. We compare our model-driven approach with an empirical approach that searches for code-specific optimization sequences. Our model-driven approach can find similarly good sequences as the empirical approach with much less compile-time.

This research demonstrates that the analytic models can be used to effectively address the problems with the application of optimizations. Our model-driven approach is practical and scalable. With model-driven optimizations, compilers can produce higher quality code in less compile-time than what is possible with current approaches.

1.3 ORGANIZATION OF THIS DISSERTATION

The remainder of this dissertation is organized as follows. Chapter 2.0 presents prior work on tackling the problems with the application of optimizations. We categorize the previous research efforts by the models used in their approaches. The relationship of this research and prior work is also discussed. Chapter 3.0 discusses the overall design of our framework. It describes the components and uses of the framework. Chapter 4.0 presents the framework instance, FPSO, for

predicting the profitability of scalar optimizations. In this chapter, models for code, scalar optimizations and machine resources are presented. The profitability engine is also described to show how to use models to make a prediction. The experimental results shown in this chapter demonstrate the prediction accuracy and the usefulness of FPSO. Similarly, Chapter 5.0 presents the framework instance, FPLO, for predicting the profitability of loop optimizations. Chapter 6.0 presents the framework instance, FIO, for automatically detecting the enabling and disabling interactions among optimizations, considering code context. It describes the code model, optimization models and the interaction engine. It also shows how to use the interaction property to determine a code-specific optimization sequence. We compare our model-driven approach with an empirical approach for searching for code-specific optimization sequences. The experimental results demonstrate that our model-driven approach is practical and scalable. Conclusions, limitations and directions for future research are discussed in Chapter 7.0 .

2.0 BACKGROUND AND RELATED WORK

In the past 40 years, much of the research in the compiler community has been devoted to the development of particular optimizations (e.g., path sensitive optimization [5]) and program analysis techniques (e.g., demand-driven data flow analysis [16]). Since this dissertation is focused on using models to study optimization properties and effectively apply optimizations, this related work section describes techniques that explore the effective application of optimizations. In this section, we discuss these related approaches. We categorize them according to the models used in the approaches. We also discuss the optimization properties and the problems that have been addressed in each approach. Lastly, the relationship of this research and prior work is discussed.

2.1 PRIOR WORK

As stated in Chapter 1.0 , due to the problems with the application of optimizations and the use of fixed strategies for handling these problems (e.g., always applying applicable optimizations, using a fixed order to apply optimizations), traditional compilers do not achieve the potential benefits from optimizations. There have been several approaches to address some of these problems. Table 2.1 categorizes these approaches. In the table, each row represents a class of approaches that uses similar models to explore the effective application of optimizations. For each row, there are four columns. The first column gives a name for these approaches. The second column indicates optimization properties that were studied in these approaches. The third column shows models that were used. The last column gives the uses of these approaches. To facilitate the discussion, we first describe the terminology used in the table.

- *A code model* represents code characteristics that are changed by optimizations and affect the use of machine resource and the code conditions needed before applying optimizations to maintain program semantics.
- *An optimization model* expresses the changes made by an optimization on code characteristics and the pre-and post conditions of the optimization.
- *An analytic resource model* estimates the cost to use a resource by analyzing the code characteristics.
- *An experimental resource model* estimates the cost to use a resource by experimentally executing the program.

Table 2.1: Approaches to explore the effective application of optimizations

<table>
<tr><th>Approach</th><th>Properties</th><th>Models</th><th>Uses</th></tr>
<tr><td>Empirical</td><td>Profitability
Interaction
Code size</td><td>No</td><td>Order optimizations
Configure optimizations</td></tr>
<tr><td>OSE</td><td>Profitability</td><td>Analytic resource model</td><td>Configure optimizations</td></tr>
<tr><td>Jalapeño</td><td>Profitability</td><td>Experimental resource model</td><td>Select optimization levels</td></tr>
<tr><td rowspan="2">Unimodular</td><td>Profitability</td><td>Code model
Loop optimization models
Analytic resource model</td><td rowspan="2">Order optimizations
Configure optimizations
Combine optimizations</td></tr>
<tr><td>Interaction</td><td>Heuristic & experimental</td></tr>
<tr><td>Analytic interaction</td><td>Interaction</td><td>Optimization models</td><td>Detect interaction</td></tr>
<tr><td rowspan="2">This research</td><td>Profitability</td><td>Code model
Loop & scalar optimization models
Analytic resource models</td><td rowspan="2">Profit-driven optimization
Order optimizations</td></tr>
<tr><td>Interaction</td><td>Code model
Optimization models</td></tr>
</table>

2.1.1 Empirical

The representation of an empirical approach is shown in the second row of Table 2.1. In this approach, optimizers search the optimization space, apply optimizations, and then evaluate performance by executing the optimized code. The properties of optimizations are determined by performing optimizations and experimentally evaluating their performance. For example, the interactions among optimizations are detected by applying an optimization on the code and recomputing the data flow needed for the analysis of other optimizations. This approach has been used to discover a code-specific optimization sequence [1], [15], [29], [30], [31] and to select an optimization configuration [17], [26].

Cooper et al. [1], [15] proposed a compiler framework, called an adaptive optimizing compiler, which explores different orders to apply optimizations at compile time. In their system, the traditional fixed-order optimizer is replaced with a pool of optimizations, a steering algorithm and an explicit objective function. An objective function is the criteria to optimize the code; for example, improving performance, reducing the code size or reducing energy consumption. The steering mechanism uses a search algorithm (e.g., a genetic algorithm) to select an optimization sequence to transform the code. The compiler evaluates the performance of the optimization sequence by executing the optimized code. The results serve as an input to the steering algorithm to refine future choices. Through repeated experimentation, the steering algorithm discovers a good optimization sequence, given the source code, the available optimizations, and the target machine. They performed a large experimental study using a prototype adaptive compiler. Their findings indicate that for the cost of 200 to 4550 compilations and executions, they can find sequences that are 15 to 25% better than a fixed-order sequence.

In a similar approach, the select-best-from function in VISTA [29], [30], [31] selects an optimization sequence that maximizes the objective function (i.e., reducing the code size or improving the performance). In this approach, an algorithm is designed to carefully and aggressively prune the search space and thus make exhaustive enumeration feasible for 98% of the functions in their benchmark suite. However, most of their benchmarks are from MiBench [34] and have relatively small functions.

Knijnenburg et al. [17], [26] propose an iterative compilation approach to explore optimization configurations. They implement a compiler that traverses the optimization space for

different configurations of loop unrolling, loop tiling and padding. They apply optimizations with different configurations and execute the transformed code to choose the best optimization configuration.

Compared with other approaches, the empirical approach evaluates the properties of optimizations via execution, which is its major disadvantage (i.e., high overhead) [15], [46]. As Triantafyllis et al. [46] point out, the adaptive optimizing compiler's proof-of-concept experiment, which involved a small kernel of code, took about a year to complete. Moreover, an optimizer that uses search techniques must be able to remove optimizations when the candidate of sequence or configuration is not desirable. This removal may also have high time or space overhead.

2.1.2 OSE (Analytic Resource Model-based)

Triantafyllis et al. [46], [47] propose an approach to discover a best optimization configuration based on an analytic resource model (shown in the third row of Table 2.1). In this approach, the profitability of optimizations is determined by using this analytic resource model. They present an Optimization-Space Exploration (OSE) compiler. To search the optimization space, OSE prunes the search space in advance and searches within a small number of promising optimization configurations. After applying optimizations with a candidate configuration, OSE uses an analytic resource model (i.e., static estimator) to evaluate the performance of the optimized code. In their approach, the code and optimizations are not modeled. Thus, OSE still needs to apply optimizations to get the optimized code and to remove optimizations when not desirable. Because of the high compile-time overhead, they apply their techniques only to hot code segments.

2.1.3 Jalapeño (Experimental Resource Model-based)

Arnold et al. [2] propose an approach to select an optimization level based on an experimental resource model, shown in the fourth row of Table 2.1. In this approach, the profitability of optimizations is determined by integrating parameters achieved from offline experiments. They present the adaptive optimization in the Jalapeño JVM. Optimizations are grouped into several

levels. When deciding at which optimization level a method should be recompiled, they use a simple benefit-cost analysis: they estimate the profitability of each optimization level as a constant based on offline measurements and they use a function of method size to estimate the cost of recompilation. This approach is simple and can be used to select the optimization level at run time. However, it neglects many aspects of optimization behavior. For example, the benefits of an optimization level should be varied according to code context. Also, for an optimization level, the order of applying optimizations impacts the effectiveness of this optimization level.

2.1.4 Unimodular (Analytic Model-based)

As shown in the fifth row of Table 2.1, other researchers have explored the use of code, optimization and analytic resource models to determine the profitability of optimizations. In this approach, the interactions among optimizations are determined by heuristics and experiments. This approach has been used for discovering a best sequence of optimizations [52], [42], [53], optimization configuration [14], [11], [55], [25], [43] and combining optimizations [12], [13].

Sarkar [42] describes the IBM ASTI optimizer in the IBM XL FORTRAN compilers for RS/6000 and PowerPC uniprocessors and symmetric multiprocessors. ASTI automatically selects a sequence of loop optimizations for a given input program and a target processor to improve utilization of the memory hierarchy and instruction–level parallelism. The selection is based on an analytic memory cost model and optimization models of loop optimizations. Wolf and Lam [52] propose an algorithm that finds a sequence of loop optimizations to improve the locality of a loop nest. The algorithm is based on two components: a mathematical formulation of reuse and locality (i.e., analytic cache model) and a loop optimization theory that unifies the various transformations as unimodular matrix transformations (i.e., optimization models for loop optimizations). Wolf et al. [53] present a compiler algorithm that intelligently searches the various optimizations, using analytic models of resource and optimizations to select the sequence of optimizations leading to the best performance. The analytic resource model they use estimates total machine cycles taking into account cache misses, software pipelining, register pressure and loop overhead. All of these approaches use heuristics to decide which optimizations should be considered first, according to the potential enabling interactions. They check the applicability of further loop optimizations to explore the interactions experimentally.

Another model-based approach derives a best optimization configuration. Coleman et al. [14] and Sarkar et al., [43] present algorithms for choosing the best tile size based on the optimization model for loop tiling and the resource model for cache. Chandramouli et al. [11] and Kandemir et al. [25] choose the configuration for other optimizations, including data reconstructing optimizations. Yotov et al. [55] use an analytic model to choose an optimization configuration and compare with the empirical approach in ATLAS (a system for generating a dense numerical linear algebra library, called the BLAS).

Click et al. [12], [13] propose a model-based approach to combine optimizations. They formalize the optimizations as monotone analysis frameworks. When applying a monotone analysis framework to a specific program, a set of equations (i.e., code model) can be derived directly from the program. The equations have a maximal solution, called the Greatest Fixed Point. To combine optimizations, monotone analysis frameworks are combined and a new framework is produced. Also, the new code model can be derived by applying the resulting framework. If the new code model is still monotonic and its maximal solution is better than the combined maximal solution of individual code models, the combination yields better results.

Although these model-based approaches can be very efficient, they have some problems. First, they do not always achieve good performance. Yotov et al. [55] showed that their analytic model-based approach has an average of 7% performance decrease compared to the empirical approach. Second, these approaches are not integrated into a general framework that is applicable to other optimizations (e.g., scalar optimizations) and machine resources (e.g., registers).

2.1.5 Analytic Interaction

As shown in sixth row of Table 2.1, researchers have explored the use of models (i.e., specification language) to specify optimizations and to analytically study the interaction property of optimizations [28], [50], [51], [32].

Knuth and Bendix [28] proposed an approach to detect the interaction property of optimizations. They express optimizations as a set of rewrite rules. Their algorithm detects potential conflicts and resolves them by introducing new rewriting rules, derived from the existing set. Unfortunately their procedure is difficult to generalize.

Whitfield and Soffa [50], [51] describe a framework that enables the exploration, both analytically and experimentally, of the interaction property of optimizations. They proposed a specification language, Gospel, to express the pre- and post conditions of optimizations. They detect the existence of interactions by examples and prove the non-existence of interactions among optimizations. However, they can not automatically detect the interactions among optimizations based on code context.

Lacey [32] introduces a specification language, TRANS, for automatically generating the implementation of optimizations and formally analyzing optimizations. TRANS combines elements of rewriting, temporal logic and logic programming. TRANS is used to prove the soundness of optimizations and detect disabling interaction for a certain class of optimizations. However, their algorithm for detecting disabling interaction is limited since it can not handle all the optimizations described by TRANS.

2.2 PRIOR WORK AND THIS RESEARCH

In comparison with empirical approaches, the model-based approaches are very efficient in addressing the effective application of optimizations. Yet, none of the previous model-based work has been integrated into a general framework that can be used for studying different properties of optimizations and is applicable to a wide range of optimizations and machine resources.

This research (shown in the seventh row of Table 2.1), presents a general framework that uses models to determine the properties of optimizations, including profitability and the interaction property. The framework can also be extended to study other optimization properties, such as the impact of optimizations on code size and power consumption. The framework includes a variety of models, including 1) code models, 2) optimization models for scalar and loop optimization, and 3) resource models for cache, registers and computation. Thus, it is applicable to both scalar and loop optimizations. It also considers several machine resources, which can be combined to determine overall profit.

Another difference between this research and previous model-based approaches is that the interaction property of optimizations is determined by models instead of heuristics and experiments. This research presents an automatic technique that considers code context to determine the enabling and disabling interactions of a set of optimizations without actually applying optimizations on the code.

Based on optimization properties, the framework can handle the problems with the application of optimizations. For example, the framework can be used to perform profit-driven optimization, which selectively applies only profitable optimizations to avoid performance degradation. The framework can also be used to determine a code-specific order of applying optimizations to get the most benefit.

3.0 OVERALL DESIGN OF THE MODEL-BASED FRAMEWORK

As described in Section 1.0 , properties of optimizations are difficult to determine because they depend on a number of factors, including code, optimizations and resources. Furthermore, several resources may impact overall performance. Thus, our approach is to develop models that can express the characteristics of these factors. For example, to determine the profitability of an optimization, we require models that are useful for predicting the impact of the optimization on performance. Performance is generally affected by registers, computation and cache. So, we need resource models for each of them, as well as the models for code and optimizations.

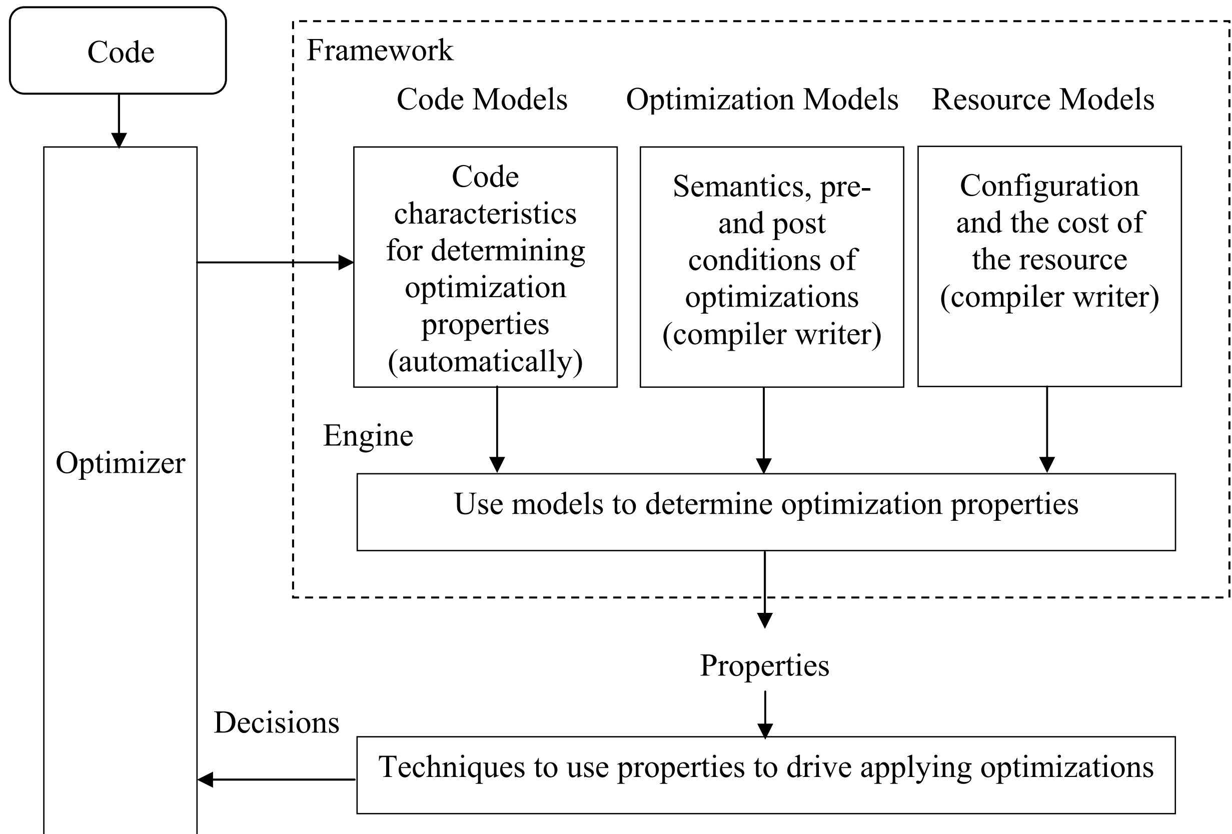


Figure 3.1: Overall design of model-based framework

Figure 3.1 shows the overall design of our model-based framework. In the framework, there are three types of analytic models (code, optimization and resource models). The code model is generated automatically by the optimizer. The models of optimizations and resources are developed by a compiler writer. As part of the framework, there is an engine that processes the models and determines optimization properties. Based on these properties, techniques are designed to make decisions for optimizers on when, where and in what order to apply optimizations to get the most benefit. The models in the framework are plug-and-play components. When new models for code, optimizations or machine resources are needed, they can be developed and easily added into the framework. Note our framework uses optimization properties to decide how to effectively apply optimizations. Thus, to determine the properties of optimizations, we do not require exact numbers but numbers "accurate enough" that the right decisions as to when and what optimizations to apply can be made.

In this chapter, we first describe the components of the framework, including code models, optimization models, resource models and the engine. Also, we discuss the framework's uses in effectively applying optimizations.

3.1 COMPONENTS OF THE FRAMEWORK

3.1.1 Code Models

The code model expresses characteristics of a code segment needed to determine optimization properties. For example, to predict profitability, the code model needs to represent the code characteristics that are changed by an optimization and impact the use of a machine resource. Because several resources impact overall performance, there is a code model for each machine resource. For example, there is a register code model to express live range information because live ranges can be changed by an optimization and impact register uses. There is a computation code model to specify the frequency of the occurrence for operations. There is also a code model for cache to specify the iteration space and array reference sequence. For determining the interaction property, we require the code model to represent the code characteristics that are

needed for verifying the pre-conditions of an optimization and are changed by the actions of an optimization.

The code models are extracted from an intermediate representation of the program. The code models are automatically generated by the optimizer for an optimization or a complete function. When safe conditions for applying an optimization are detected, the code models for the optimization are automatically generated by the optimizer to determine profitability. For profitability, we assume that data flow information is available to determine if an optimization is legal. If legal, we then predict the profit of applying the optimization. However, we could also do the reverse: we could determine the hot regions of the code and the profitability of an optimization in a region and if the transformation is profitable, use data flow analysis (in particular, demand-driven data flow analysis [16]) to determine if the optimization is legal. The code model can also be generated for a complete function before determining optimization properties.

3.1.2 Optimization Models

Optimization models are written by the compiler engineer when developing a new optimization. For predicting profitability, an optimization model expresses the semantics (i.e., effect) of an optimization, from which the impact of the optimization on each resource can be determined. For detecting the interaction property, an optimization model represents the conditions under which an optimization can be safely applied and the code modifications that implement the optimization.

The effect of an optimization is determined from the code changes that the optimization introduces. Optimizations can cause non-structural and structural code changes, which can be expressed by editing changes on a control flow graph. The edits are insert/delete a statement (including its operation and operators), insert/delete a block and insert/delete an edge. All optimization code changes can be expressed with these edits [4]. Thus, the code changes of an optimization can be described as a series of basic edits. For example, constant propagation can be represented as "delete variable *v* at statement *s*; insert constant *c* at statement *s*".

To determine the profit of an optimization on a resource, we may need a model that represents the impact of other optimizations on the resource. For example, to determine the

register profit, an optimization model for the register allocator must be developed. The characteristics of the register allocator that need to be modeled are whether the allocator is local or global and how it spills the live ranges (i.e., how the additional loads and stores are inserted into the code). A model for the register allocator can be developed that approximates a particular register allocation scheme; for example, graph coloring [10] or linear scan [40]. In this dissertation, we are interested in the profit of optimizations on registers rather than the impact of different register allocation schemes. Hence, we only need a representative optimization model for register allocation, such as one for graph coloring.

In this research, optimizations models are developed for a number of scalar and loop optimizations. They are copy propagation (CPP), constant propagation (CTP), dead code elimination (DCE), loop invariant code motion (LICM), partial redundancy elimination (PRE), global value numbering (VN), branch chaining (BRC), branch elimination (BRE), register allocation, loop tiling (LPT), loop interchange (LPI), loop unrolling (LPU), loop reversal (LPR), loop fusion (LPF), and loop distribution (LPD).

3.1.3 Resource Models

The profitability of optimizations depends on several machine resources, including registers, functional units and cache. Our framework has a model for each resource, which describes the resource configuration and benefit/cost information in using the resource. A resource model is developed based on a particular platform. For example, to determine the register profit, we need to know the number of available hardware registers and the cost of memory accesses (i.e., loads and stores). When considering functional units, the computational operations available in the architecture and their execution latencies are needed. The enabling and disabling interactions exist because an optimization may create or destroy the conditions of applying another optimization. Thus, no resource models are needed for detecting the interaction.

3.1.4 Engine

The models in the framework are descriptive and provide the information needed to determine optimization properties. The other important component of our framework is the engine, which uses the models to determine optimization properties.

To predict profitability, the engine inputs the code, optimization and resource models after an optimization is detected to be safe. The engine uses information in the models to compute the profit. The profit can be computed for one resource or for combined resources. From an optimization model, the engine determines the code model changes caused by the application of the optimization. It then applies these changes to the code model and generates a new code model that represents the effect of the optimization. Finally, it uses the resource model to determine the impact of the changes on the resource. The engine can also use profile information (e.g., the basic block frequencies) in computing the profits.

For example, assume the register profit of an optimization is desired. The engine inputs the register code model, an optimization model, a register allocation model and a register resource model. Then it determines the changes on the live ranges (i.e., the register code model) based on the optimization model. Since an optimization model expresses the semantics of the optimizations as basic edits, the engine takes the edits and computes the changes on the live ranges using an incremental dataflow algorithm [41]. The engine then uses a register allocation model to determine how the spills (i.e., loads and stores) are changed according to these live range changes. Finally, the engine computes the profit associated with the change in the spills.

To detect the interactions among a set of optimizations, the engine inputs the optimization models and the code model. It then generates the specific enabling, disabling and post conditions for each optimization opportunity at a program point. It then matches the enabling and disabling conditions with the post conditions among all the optimization opportunities and determines the interaction property.

3.2 USES OF THE FRAMEWORK

As previously discussed, there are several problems with the application of optimizations. Our model-based framework can be used to address these problems based on optimization properties.

First, using our framework, the optimizer can perform profit-driven optimization. Once the optimizer finds that an optimization is applicable, it generates the code models involved in the optimization and triggers the engine to predict the profit of the optimization. When the engine is triggered, it takes the code models, optimization models and resources models, updates the code models and determines the profit on resources under consideration. Based on whether there is a benefit or not, the optimizer applies the optimization accordingly. In this way, performance can be improved by avoiding applying optimizations when they are not profitable.

Secondly, using our framework, the optimizer can determine code-specific optimization sequences. There are several ways that the optimization properties can be used to decide the order to apply optimizations for the most benefit. One is to use profitability. Previous work showed the effectiveness of using genetic algorithm to discover code-specific optimization sequences [15], [29], [30]. However, they experimentally evaluated the candidate sequences: They performed optimizations and executed the optimized code to evaluate. Thus, the search time is large. Using our framework, we can predict the profitability of optimizations in a sequence without applying optimizations and executing the code. The search time can be reduced by avoiding effort and time to perform the optimizations and execute the optimized code. To determine the order of applying optimizations, the interaction property can be used. The interaction can help prune the search space by knowing what optimizations are legal after applying an optimization (through enabling and disabling relationships). According to an evaluation function (e.g., the number of optimizations), we can select one optimization from the legal optimizations. We can construct a good optimization sequence to maximize the evaluation function. We also can combine profitability with the interaction property and use profitability as the evaluation function to determine the optimization sequences.

Our framework also has other uses. For example, an optimizer can use the framework to find a good configuration of an optimization. Instead of different optimization sequences, different optimization configurations are evaluated, and the one with the best performance is determined and used.

4.0 FPSO: PREDICTING PROFITABILITY OF SCALAR OPTIMIZATIONS

In this chapter, we describe a framework instance, called FPSO, for predicting the profitability of scalar optimizations, including Partial Redundancy Elimination (PRE), Loop Invariant Code Motion (LICM) and Value Numbering (VN). Because scalar optimizations have negligible effect on cache (i.e., loop behavior dominates data cache performance [36]), we consider only two machine resources: registers and computation (i.e., functional units). Figure 4.1 shows the overall structure of FPSO. There are three kinds of analytic models in FPSO. Code models include models for representing live ranges and operations in the code. An optimizations model is developed for each scalar optimization, such as PRE and Register Allocation (RA). Machine resource models include models for expressing the machine configuration and the cost of using registers and functional units. The profitability engine in FPSO uses the model to predict the impact of optimizations on registers and computation.

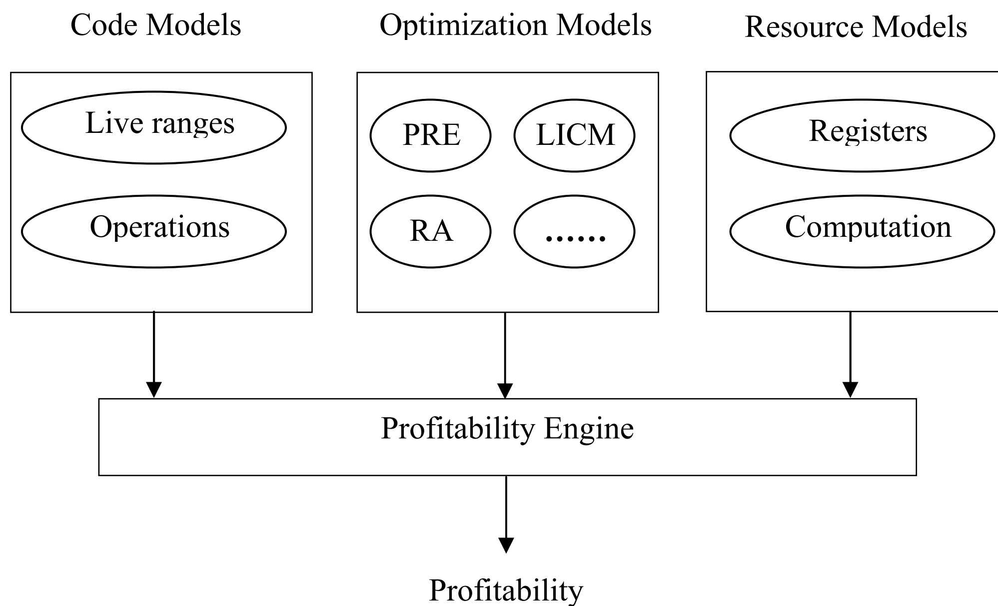
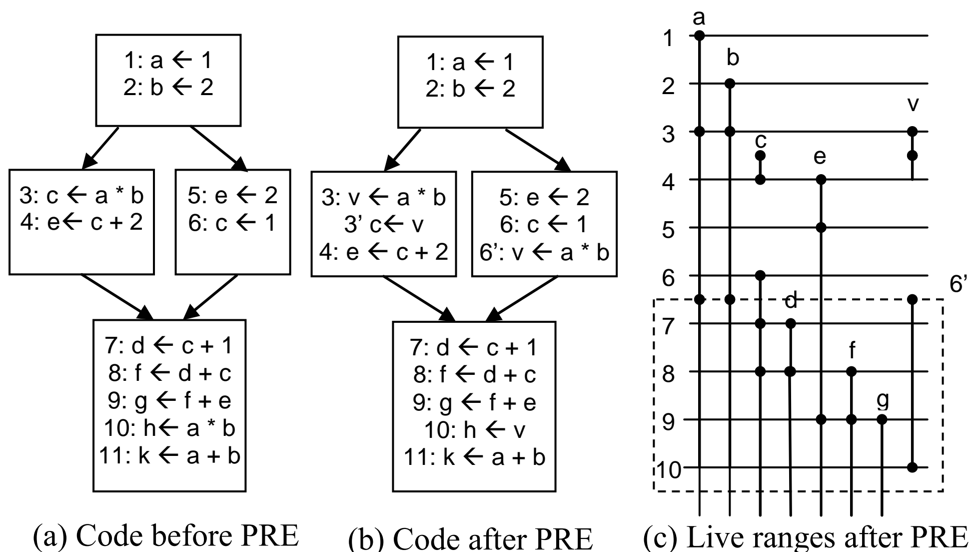


Figure 4.1: Structure of FPSO

The impact of PRE, LICM and VN on computation is clear: They insert or delete operations at some program points. Their impact on registers is more complicated and depends on code context. Sometimes they may introduce register spills, while in other cases they may decrease the number of spills.



(a) Code before PRE (b) Code after PRE (c) Live ranges after PRE

PRE increases the number of register spills by one,
if there are 7 hardware registers.

Figure 4.2: An example of PRE impacting registers

Figure 4.2 shows an example where PRE increases register pressure by introducing one more spill. The PRE algorithm is lazy code motion, which inserts the computation as late as possible [27]. In the figure, the code before and after applying PRE are given in (a) and (b). Figure (c) shows the live ranges after applying PRE. In (b), PRE moves the use of *a* and *b* at statement 10 up in the code. Because *a* and *b* are used after statement 10, their live ranges remain the same (shown in (c)). PRE also introduces a new live range for the temporary variable, *v*. Thus, if there are seven hardware registers, the inserted live range will cause a spill to memory. However, if *a* and *b* were not used after statement 10, their live ranges would be shortened. In that case, the total number of live ranges would be decreased by one, leading to fewer spills.

In the following sections, we present the components of FPSO. We describe (1) code models for registers and computation, (2) optimization models for PRE, LICM, VN and register allocation, (3) resource models, and (4) the profitability engine. We use an example to show how FPSO works in determining profitability of VN. Experimental results are given in Section 4.6.

4.1 CODE MODELS FOR REGISTERS AND COMPUTATION

The register code model represents the code as live ranges of global and local variables (including temporaries and parameters). We express a live range by $LR^{x}_{[n,\ldots,m]}$, where x is a variable name and $[n,\ldots,m]$ is the range of statements over which x is defined and referenced. The live range of a variable is not necessarily contiguous. For example, in Figure 4.2 (c), after PRE, the live range of v consists of two parts and can be expressed as $LR^{v}_{[3..4,\,6'..10]}$, where [6'..10] is a shorthand notation to represent a contiguous range. When a variable v is defined outside a loop at n and used inside the loop at m, we use $[n,\ldots,m]$ to represent its live range for simplicity. However, v's live range includes the whole loop.

The computation code model expresses the frequency of occurrence for each operation in the code. For an operation, *op*, its frequency is represented as a sequence $\langle fB1, fB2, \ldots, fBn \rangle^{op}$, where f_{Bi} is the number of *op* in block B_i and *op* appears in blocks B_1, B_2, … B_n.

4.2 OPTIMIZATION MODELS

All optimization code changes can be expressed with basic edits. For example, a code movement can be expressed as a deletion of the statement at the source location and an insertion of a statement at the destination location. Thus, an optimization model expresses the semantics (i.e., effect) of the optimization as a series of basic edits. We represent a basic edit by its action and code location. For example, we express "insert a statement $x \leftarrow a + b$ at code location S" by "$\text{Insert} < \text{DEF}\ x\ \ \text{USE}\ a, b\ \ \text{OP}\ add > @\ S$". In some cases, only a part of a statement is involved in a basic edit. For example, to replace a statement "$x \leftarrow a + b$" at location S with a statement "$x \leftarrow v$", only the use variables and the operations are involved in the replacement. We represent the replacement by:

"$\text{Delete} < \text{USE}\ a, b\ \ \text{OP}\ add > @\ S$"

"$\text{Insert} < \text{USE}\ v\ \ \text{OP}\ copy > @\ S$".

For clarity, we use S_s to represent the source location and S_d for the destination location in a code movement. Also, we express the new code location as *S'*. We next describe the optimization models for partial redundancy elimination, loop invariant code motion, value numbering and register allocation.

4.2.1 PRE Optimization Model

PRE inserts and deletes computations using a flow graph representation of a program. After PRE, each path contains no more occurrences of the computation than what is in the original code. The PRE algorithm that we model is lazy code motion (LCM), which takes register pressure into account by hoisting an expression no earlier than necessary [27]. Although LCM considers register pressure, there are still cases where PRE introduces more register spills (as shown in Figure 4.2).

PRE has three semantic actions that create code changes:

- Insert a statement: insert the redundant expression *EXP* and introduce a temporary *v* to hold the result of *EXP* at a destination code location;
- Replace the computation: replace *EXP* with a copy from the temporary *v* at the source code position; and
- Update each same expression *T* (that has the same operation and operands as *EXP*): replace *T*'s destination with the temporary and insert a copy statement after it.

The PRE optimization model expresses these code changes (given in Figure 4.3). In the figure, lines 2 and 3 show that an assignment from the expression *EXP* to a temporary *v* is inserted at a new code location S_d'. The variables of *EXP* are inserted as uses and the temporary *v* is inserted as the definition with the operation *op* at S_d', where S_d' is a new code location immediately after S_d. Lines 5 and 6 show that at the source code location S_s, the expression *EXP* is deleted and a copy from the temporary *v* is inserted. The definition variable is unchanged. Finally, lines 8 to 12 express the code changes of updating the same expressions. For each expression *T* that has the same computation as *EXP* at the code location S_w, the destination *w* is replaced by the temporary *v* and a copy from *v* to *w* is inserted at the new location S_w' immediately after S_w.

Eliminate the partial redundant expression *EXP* (y op z) at S_s

1: Insert a statement:
2: $S_d' = S_d + 1$
3: Insert $<$ DEF v USE y, z OP $op >$ @ S_d'

4: Replace the computation:
5: Delete $<$ USE y, z OP $op >$ @ S_s
6: Insert $<$ USE v OP $copy >$ @ S_s

7: Update the same expressions:
8: $\forall T \mid T = w \leftarrow EXP(y\ op\ z)$ at S_w
9: Delete $<$ DEF w $>$ @ S_w
10: Insert $<$ DEF v $>$ @ S_w
11: $S_w' = S_w + 1$
12: Insert $<$ DEF w USE v OP $copy >$ @ S_w'

Figure 4.3: PRE optimization model

After PRE, the temporary v can be propagated and copy statements can be deleted by applying copy propagation, which is modeled separately (see Appendix A).

4.2.2 LICM Optimization Model

LICM moves a statement from a loop body to the outside of the loop. There are certain conditions that must be met to safely apply LICM. An example of LICM is shown in Figure 4.4, where the invariant statement "$a \leftarrow b + 1$" is moved out of the loop body because each of its operands is either defined outside of the loop or a constant.

The semantic action of LICM is simply a code movement. The optimization model for LICM is shown in Figure 4.5. At a new code location S_d', which is immediately after the destination code location S_d (i.e., the loop preheader), an invariant statement is inserted (as given in lines 2 and 3). Line 5 shows that at the source location S_s (i.e., inside the loop), the invariant statement is deleted.

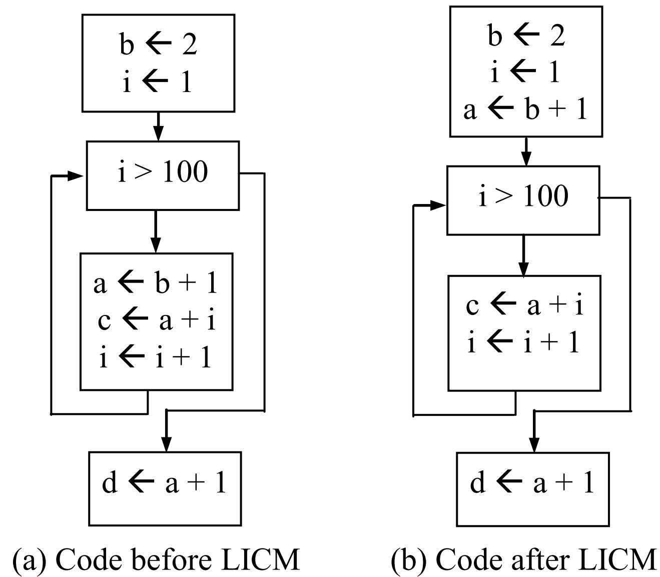


Figure 4.4: An example of LICM

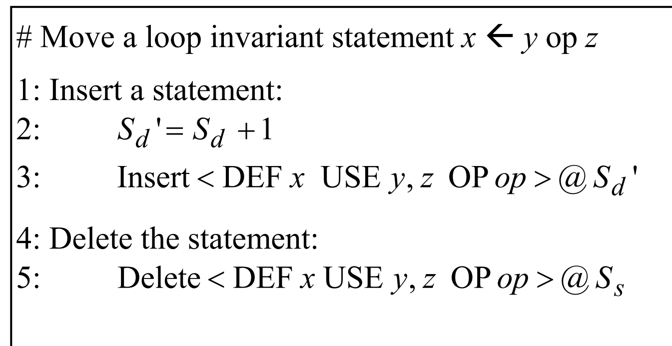


Figure 4.5: LICM optimization model

4.2.3 VN Optimization Model

The goal of VN is to find and remove redundant expressions that are equivalent based on their values (unlike PRE which considers lexically equivalent expressions). It assigns an identifying number to each expression in a particular way and then uses the number to find and remove redundant computations.

We model dominator-based VN, which is a global technique that uses hashing to discover redundant computations and to fold constants [8]. It works on Static Single Assignment (SSA) intermediate code. An example of VN is shown in Figure 4.6. Because the expression "$d_0 + c_0$"

at statement 4 has the same value number as "$a_0 + c_0$" at statement 2, it is redundant and can be replaced by the destination of "$a_0 + c_0$". Thus, statement 4 is replaced by a copy from b_0 to e_0.

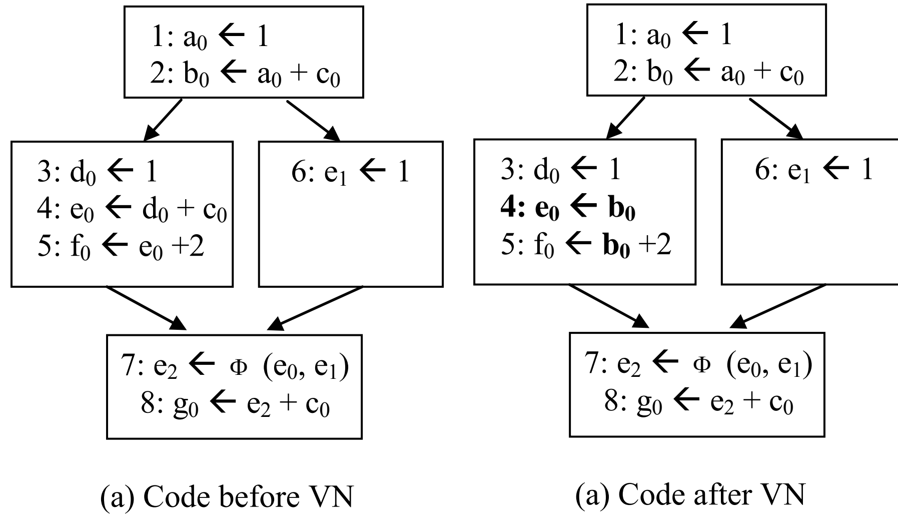
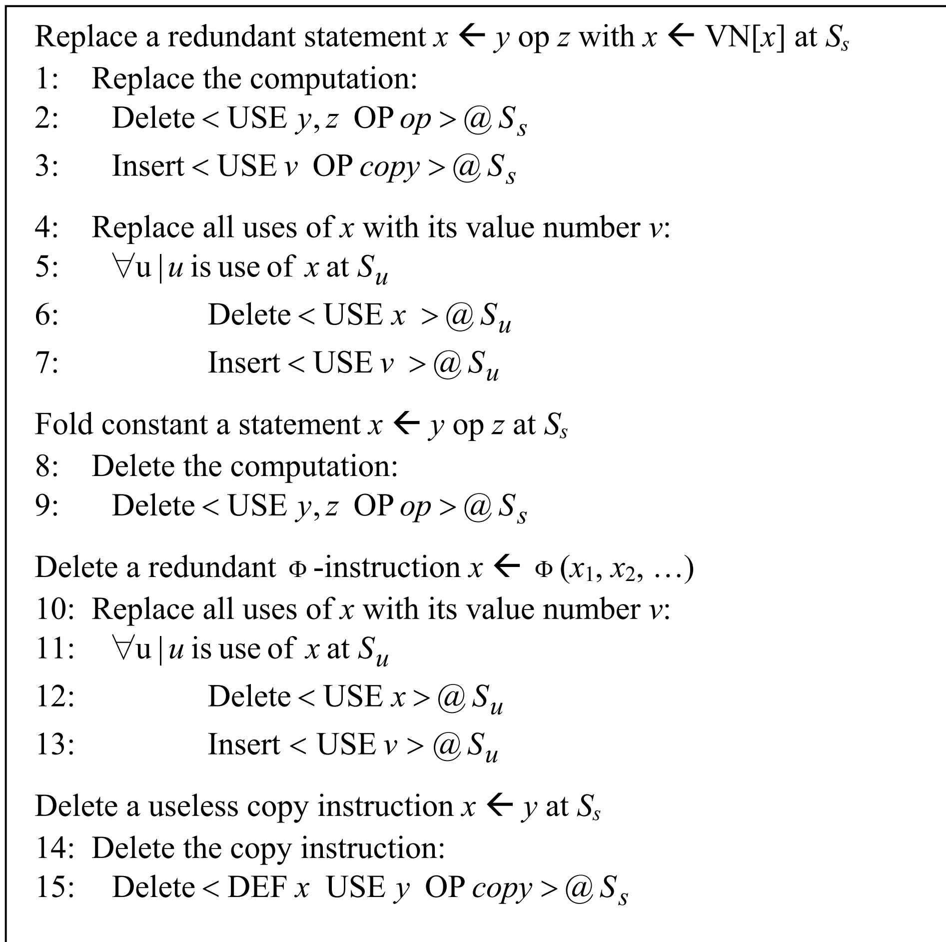


Figure 4.6: An example of VN

VN has three actions for a basic block: 1) remove redundant or meaningless Φ-instructions (Φ-instruction is a pseudo-assignment that introduces a new definition point at the merge point in the control-flow graph [7]); 2) simplify computation (constant folding) or remove the redundant computation; and 3) adjust the inputs of Φ-instructions in successor blocks. When converting SSA to non-SSA intermediate code, some Φ-instructions should be replaced by copy instructions in predecessor blocks. Because the inputs of the Φ-instructions have been adjusted, they do not show where they were originally defined (i.e., where the copy should be inserted). A general algorithm can be used to replace the Φ-instructions with copy instructions [7]. To accurately predict the impact of VN, the replacement algorithm should be modeled.

A simplification is to incrementally add the copy statements as VN progresses. In our VN, we replace the redundant computations with copy statements (instead of removing them) and retain the inputs of Φ-instructions when processing each basic block. We then use Φ-instructions to keep the useful copy statements and remove the useless ones. In this way, no copy statements will be inserted when converting SSA to non-SSA code.



Replace a redundant statement $x \leftarrow y$ op z with $x \leftarrow$ VN[x] at S_s
1: Replace the computation:
2: Delete $< \text{USE } y, z \ \text{ OP } op > @ S_s$
3: Insert $< \text{USE } v \ \text{ OP } copy > @ S_s$
4: Replace all uses of x with its value number v:
5: $\forall \text{u} \mid u$ is use of x at S_u
6: Delete $< \text{USE } x > @ S_u$
7: Insert $< \text{USE } v > @ S_u$
Fold constant a statement $x \leftarrow y$ op z at S_s
8: Delete the computation:
9: Delete $< \text{USE } y, z \ \text{ OP } op > @ S_s$
Delete a redundant Φ-instruction $x \leftarrow \Phi(x_1, x_2, \ldots)$
10: Replace all uses of x with its value number v:
11: $\forall \text{u} \mid u$ is use of x at S_u
12: Delete $< \text{USE } x > @ S_u$
13: Insert $< \text{USE } v > @ S_u$
Delete a useless copy instruction $x \leftarrow y$ at S_s
14: Delete the copy instruction:
15: Delete $< \text{DEF } x \ \text{ USE } y \ \text{ OP } copy > @ S_s$

Figure 4.7: VN optimization model

The VN optimization model, given in Figure 4.7, describes the code changes from VN. In the figure, VN[x] is the value number of x, where x can be a variable, an expression or a Φ-instruction. Each value number is a variable name. For an expression, its value number is the variable name of the first occurrence of the expression in this path in the dominator tree.

In Figure 4.7, lines 2 and 3 show that if an expression *EXP* (y op z) at S_s is redundant, it is replaced by a copy from its value number v. That is, the variables of *EXP* are deleted as uses with the operation *op* at S_s. The expression's value number v is inserted as a use with the operation *copy* at S_s. Also, all uses of the defined variable x are replaced by v (expressed in lines 5 to 7). In the example shown in Figure 4.6, at statement 4, the redundant expression $d_0 + c_0$ is deleted and a copy from its value number b_0 is inserted. At statement 5, the definition variable e_0 is used and is replaced by b_0.

In our VN algorithm, we also find statements for constant folding. Line 9 in Figure 4.7 shows if an expression *EXP* (y op z) at S_s can be simplified by constant folding, *EXP* is deleted. As given in lines 11 to 13, if a redundant Φ-instruction is deleted, all the uses of the defined variable x are replaced by the value number v. Thus, at the statement S_u where the defined variable x is used, x is deleted as a use and v is inserted as a use. The last line in Figure 4.7 models the deletion of a useless copy statement that is inserted in the step of replacing the computation. Here, the variable y is deleted as a use and the defined variable x is deleted as a definition with the operation *copy* at the location S_s.

4.2.4 Register Allocation Optimization Model

To determine the register profit of scalar optimizations, we need a model for register allocation. By applying register allocation, hardware registers are assigned to live ranges. If the number of hardware registers is not enough for all live ranges, the register allocator selects live ranges to spill to memory, which impacts overall performance. Thus, to predict the impact of optimizations on registers, we need to compute spills for the original live ranges and the live ranges changed by the optimization and compare them. This is a time consuming process. Instead, we use an incremental approach that computes how spills are changed due to each live range change. Our register allocation model reflects this incremental approach.

We model a global graph coloring register allocator. Figure 4.8 shows the register allocation optimization model. For each changed live range $LR^{c}_{[n,...,m]}$, we determine how spills are changed. As given in lines 1 to 7, if $LR^{c}_{[n,...,m]}$ is inserted or lengthened, it may introduce one more spill. Within the range [n,...,m], if the insertion of a new live range causes the number of live ranges to exceed the number of available hardware registers (HR), we select a live range to spill to memory, which introduces more loads and stores. We use $LR^{all}_{[n,...,m]}$ to represent the live ranges in [n,...,m]. To select a live range to spill, we choose the one that has the least number of uses and definitions within the range, under the assumption that the register allocator typically performs well. Thus, we need to represent all variables' uses and definitions within the range. Suppose, $LR^{s}_{[n,...,m]}$ is selected to be spilled. If there is no definition of s before a use of s or there

is no use of s within the range $[n,...,m]$, a store or load is inserted at the boundary of $[n,...,m]$. If the boundary of $[n,...,m]$ is within a loop, a store or load is inserted outside the loop. Otherwise, at all the uses or definitions of s within $[n,...,m]$, a load or store will be inserted. Alternatively, if $LR^{c}_{[n,...,m]}$ is deleted or shortened, it may decrease one spill (shown in lines 8 to 14). This register allocation model is input to the profitability engine (see Section 4.4) to predict the impact of the other optimizations on registers.

```
# Determine how spill changes for every live range change LR^c_[n,...,m]

1:   IF Inserted(LR^c_[n,...,m]) ∪ Lengthened(LR^c_[n,...,m])
2:        IF |LR^all_[n,...,m] + LR^c_[n,...,m]| > |HR|
3:             Select { LR^s_[n,...,m] } → MEM
4:             ∀d | d is definition of s at S_d ∩ S_d ∈ [n,...,m]
5:                  Insert <OP store> @ S_d
6:             ∀u | u is use of s at S_u ∩ S_u ∈ [n,...,m]
7:                  Insert <OP load> @ S_u
8:   ELSE
9:        IF |LR^all_[n,...,m] − LR^c_[n,...,m]| ⩽ |HR|
10:            Select { LR^s_[n,...,m] } → MEM
11:            ∀d | d is definition of s at S_d ∩ S_d ∈ [n,...,m]
12:                 Delete <OP store> @ S_d
13:            ∀u | u is use of s at S_u ∩ S_u ∈ [n,...,m]
14:                 Delete <OP load> @ S_u
```

Figure 4.8: Register allocation optimization model

4.2.5 Other Scalar Optimizations

We also develop optimization models for copy propagation, constant propagation and dead code elimination. These models are given in Appendix A.

4.3 RESOURCE MODELS FOR REGISTERS AND COMPUTATION

A resource model expresses the resource configuration and the cost to use the resource. It is built for a specific platform by a compiler writer.

Our resource model for registers specifies the number of available hardware registers and the cost of memory accesses (i.e., loads and stores). Thus, the compiler writer needs to specify how many hardware registers are available in the platform. For example, there are eight hardware registers that can be allocated for a byte-type variable on an Intel IA-32 machine. The compiler writer also needs to specify the average access time for a memory access. When there are not enough hardware registers to allocate variables, loads and stores (i.e., memory access) are inserted into the code. Because these loads and stores may be caches misses or cache hits, our resource model uses the average memory access time to represent the cost of registers.

Our resource model for computation describes the computational operations available in the architecture and their execution latencies. Thus, the compiler writer needs to specify what operations are available in the platform (using a form of intermediate representation), such as a move between registers or an add operation. The compiler writer also needs to give the execution latency for each operation. Some operations need the average latency, such as loads and stores. Our resource model for computation in an Intel IA-32 machine (using Mach SUIF intermediate representation [44]) is shown in Appendix B.

Resource models will be used by the profitability engine when computing the profit of applying an optimization. For example, to compute the register profit, the profitability engine uses the number of hardware registers to decide whether inserting or deleting a live range will increase or decrease spills. According to the cost of memory accesses, the profitability engine computes the cost of increasing spills or the benefit of decreasing spills (i.e., the register profit).

4.4 PROFITABILITY ENGINE

The profitability engine inputs code models, optimization models and resource models. It can also integrate profile information from offline experiments (e.g., execution frequency of basic

blocks). It then determines the changes on code models (for both registers and computation) and generates the optimized code models. Finally, it computes the register and computation profits and combines them.

Table 4.1: Incremental computation of the new register code model

Code Change	Incrementally compute the new register code model
Insert a use of variable *v* at statement *s*	IF *v is live at post-s* THEN no change; ELSE /* *lengthen v's live range*/ The original live range $LR^{v}_{[n,...,m]}$ changes to $LR^{v}_{[n,...,m]\cup[n,...,s]} = \begin{cases} LR^{v}_{[n,...,s]} & \text{s post - dominate other uses} \\ LR^{v}_{[n,...,m,...,s]} & \text{otherwise} \end{cases}$
Insert a definition of variable *v* at statement *s*	IF *v is not live at post-s* THEN no change; ELSE /* *shorten v's live range*/ The original live range $LR^{v}_{[n,...,m]}$ changes to $LR^{v}_{[n,...,m]\cap[s,...,m]} = \begin{cases} LR^{v}_{[s,...,m]} & \text{s post - dominate other definition} \\ LR^{v}_{[n,...,s,...,m]} & \text{otherwise} \end{cases}$
Delete a use of variable *v* at statement *s*	IF *v is live at post-s and v is not only use in a loop* THEN no change; ELSE /* *shorten v's live range*/ The original live range $LR^{v}_{[n,...,s]}$ changes to $LR^{v}_{[n,...,s]\cap[n,...]} = \begin{cases} LR^{v}_{[n,...,m]} & \text{m post - dominate other uses} \\ LR^{v}_{[n,...m,...]} & \text{otherwise} \end{cases}$
Delete a definition of variable *v* at statement *s*	IF *v is not live at post-s* THEN no change; ELSE /* *lengthen v's live range*/ The original live range $LR^{v}_{[s,...,m]}$ changes to $LR^{v}_{[s,...m]\cup[...,m]} = \begin{cases} LR^{v}_{[n,...,m]} & \text{n post - dominate other definition} \\ LR^{v}_{[...,n,...,m]} & \text{otherwise} \end{cases}$
Delete an edge from block *Bs* to block *Bd*	Delete all uses of any variable that is live at the beginning of *Bd* from the *Bs* and all predecessors of *Bs* where the variable is no longer live by any path.
Insert an edge from block *Bs* to block *Bd*	Insert all uses of any variable that is live at the beginning of *Bd* to the *Bs* and all predecessors of *Bs*.

An optimization model expresses the semantics of optimizations by basic edits. From an optimization model, the engine determines how the optimization changes the register code model with an incremental dataflow algorithm [41]. Table 4.1 shows how to incrementally compute the new register code model (i.e., live ranges) for each edit. Each row in the table represents an edit and shows how the profitability engine incrementally updates the register code model for this edit, considering code context. There are six basic edits shown in the table.

In Table 4.1, *post-s* means the point immediately after statement *s*. We use *n* to represent a statement where there is a *definition* of the variable *v* and use *m* to represent a statement where there is a *use* of the variable *v*. For example, the effect on the live ranges from inserting a use of *v* (1st row of the table) depends on the current code. If *v* is already live at *post-s*, there is no change. Otherwise, the original live range $LR^{v}_{[n,\ldots,m]}$, is lengthened. If the inserted use at statement *s* is the last use (i.e., *s* post-dominates other uses), the new live range for *v* becomes $LR^{v}_{[n,\ldots,s]}$. Otherwise, the new live range consists of the original live range and a range to the use statement *s*. This range is represented as $LR^{v}_{[n,\ldots,m,\ldots,s]}$. Similarly, the profitability engine updates the register code model for inserting a definition, deleting a use, deleting a definition, inserting an edge and deleting an edge.

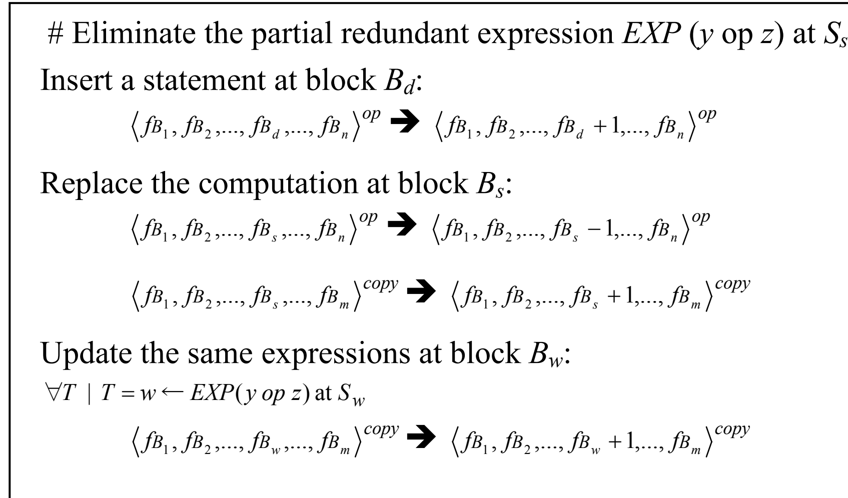
Table 4.2: Updates of the computation code model

Code Change	Update the computation code model
Insert an operation *op* at block B_s	The original operation list $\langle fB_1, fB_2,\ldots, fB_s,\ldots, fB_n \rangle^{op}$ changes to $\langle fB_1, fB_2,\ldots, fB_s+1,\ldots, fB_n \rangle^{op}$
Delete an operation *op* at block B_s	The original operation list $\langle fB_1, fB_2,\ldots, fB_s,\ldots, fB_n \rangle^{op}$ changes to $\langle fB_1, fB_2,\ldots, fB_s-1,\ldots, fB_n \rangle^{op}$

The engine also infers how an optimization changes the computation code model. Table 4.2 shows the basic changes on computation code model and how the profitability engine updates the computation code model for each basic change. As shown in Table 4.2, the code

changes from an optimization can be classified as either inserting an operation or deleting an operation. If an operation *op* is inserted at a block B_s, the number of *op* in block B_s (i.e., f_{Bs}) is increased by one. If an operation *op* is deleted at a block B_s, f_{Bs} is decreased by one. Thus, the engine can determine the impact of an optimization on computation.

For example, the impact of PRE on computation can be determined by the engine, as shown in Figure 4.9. To insert a statement, the operation *op* is inserted at block B_d (the destination code location S_d is in block B_d). To replace the computation, the operation *op* is deleted at block B_s and a copy is inserted at block B_s (the source location S_s is in block B_s). Finally, to update the same expression T at the code location S_w, a copy is inserted in block B_w, where S_w is in block B_w.


Eliminate the partial redundant expression EXP (y op z) at S_s

Insert a statement at block B_d:

$$\langle fB_1, fB_2, \ldots, fB_d, \ldots, fB_n \rangle^{op} \rightarrow \langle fB_1, fB_2, \ldots, fB_d + 1, \ldots, fB_n \rangle^{op}$$

Replace the computation at block B_s:

$$\langle fB_1, fB_2, \ldots, fB_s, \ldots, fB_n \rangle^{op} \rightarrow \langle fB_1, fB_2, \ldots, fB_s - 1, \ldots, fB_n \rangle^{op}$$

$$\langle fB_1, fB_2, \ldots, fB_s, \ldots, fB_m \rangle^{copy} \rightarrow \langle fB_1, fB_2, \ldots, fB_s + 1, \ldots, fB_m \rangle^{copy}$$

Update the same expressions at block B_w:

$\forall T \mid T = w \leftarrow EXP(y\ op\ z)$ at S_w

$$\langle fB_1, fB_2, \ldots, fB_w, \ldots, fB_m \rangle^{copy} \rightarrow \langle fB_1, fB_2, \ldots, fB_w + 1, \ldots, fB_m \rangle^{copy}$$

Figure 4.9: Impact of PRE on computation code model

After determining the changes on the code models, the engine generates the optimized code model and computes the profit for the resource under consideration. For example, to compute the profit for registers, the engine computes the benefit/cost in terms of spills (i.e., loads and stores) based on the register allocation model. That is, for each live range change, the engine determines the impacted region and compares the total number of live ranges with the available hardware registers. If the total number of live ranges is larger than the available hardware registers, inserting a live range will introduce one more spill. To select a live range to spill to memory, the engine records the uses and definitions of all variables in the region and chooses the

one that has the least number of uses and definitions. The benefit/cost associated with the spill is the profit of the optimization on registers.

Table 4.3: Computing profit on registers (R_{total}) and computation (C_{total})

Optimization	Compute the profit on registers and computation
PRE: eliminate a redundant expression	$R_{total} = R_{insertstat} + R_{replacecomp} + R_{update\exp}$ $= R_{insertuse}(EXP, S_d) + R_{insertdef}(v, S_d)$ $+ R_{deleteuse}(EXP, S_s) + R_{insertuse}(v, S_s)$ $+ \sum_w (R_{deletedef}(w, S_w) + R_{insertdef}(v, S_w)$ $+ R_{insertdef}(w, S_w + 1) + R_{insertuse}(v, S_w + 1))$ $C_{total} = C_{insertstat} + C_{replacecomp} + C_{update\exp}$ $= C_{insert}(op, B_d)$ $+ C_{delete}(op, B_s) + C_{insert}(copy, B_s)$ $+ \sum_w C_{insert}(copy, B_w)$
LICM: move an invariant statement	$R_{total} = R_{insertstat} + R_{deletestat}$ $= R_{insertuse}(EXP, S_d) + R_{insertdef}(x, S_d)$ $+ R_{deleteuse}(EXP, S_s) + R_{deletedef}(x, S_s)$ $C_{total} = C_{insertstat} + C_{deletestat}$ $= C_{insert}(op, B_d) + C_{delete}(op, B_s)$
VN: eliminate a redundant expression	$R_{total} = R_{replacecomp} + R_{replaceuse}$ $= R_{deleteuse}(EXP, S_s) + \alpha \times R_{insertuse}(v, S_s)$ $+ \sum_u (R_{deleteuse}(x, S_u) + R_{insertuse}(v, S_u))$ $C_{total} = C_{replacecomp} + C_{replaceuse}$ $= C_{delete}(op, B_s) + \alpha \times C_{insert}(copy, B_s)$
VN: fold constant a statement	$R_{total} = R_{deletecomp}$ $= R_{deleteuse}(EXP, S_s)$ $C_{total} = C_{deletecomp}$ $= C_{delete}(op, B_s)$

To compute the overall profit of an optimization, P_{total}, the engine needs to combine the register profit, R_{total} and computation profit, C_{total}. To compute R_{total}, the engine sums the register

profit associated with every step in the optimization model. Similarly, to compute C_{total}, the engine sums the computation profit for every step.

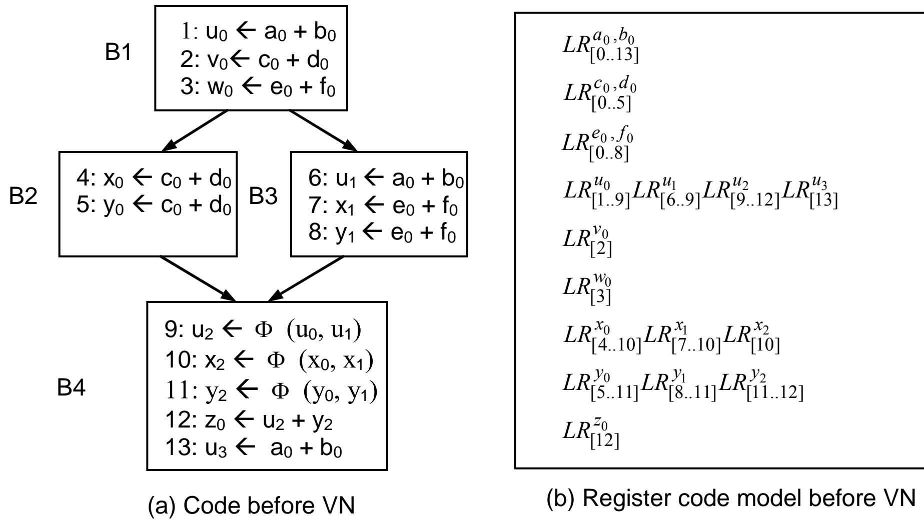
Table 4.3 shows how the profitability engine computes R_{total} and C_{total} for PRE, LICM and VN. For example, to compute the profit of eliminating a redundant expression in VN (3rd row in Table 4.3), the engine needs to compute the register profit, which includes the register profit of replacing the computation $R_{replacecomp}$ and updating of the uses of the defined variable $R_{replaceuse}$. Further, $R_{replacecomp}$ is computed by deleting a use, $R_{deleteuse}(EXP, S_s)$ and inserting a use, $R_{insertuse}(v, S_s)$. The engine also needs to compute the computation profit of replacing the computation $C_{replacecomp}$ (i.e., removing the computation and inserting a copy). However, the inserted copy statement may be deleted later as a useless statement if it is not an argument of an Φ-instruction (described in Section 4.2.3). The engine also considers the deletion. Thus, the engine multiplies $R_{insertuse}(v, S_s)$ and $C_{insert}(copy, B_s)$ by a factor of α. α is the ratio that a copy statement will stay in the code (i.e., not deleted in the later phase of VN), which is a number between zero and one. We determine α by profiling.

To combine the profits for registers and computation, they must have the same metric. If the computation profit considers the frequency of a node, the register profit also needs to consider the execution frequency of the loads or stores.

4.5 AN EXAMPLE OF PROFIT-DRIVEN VN

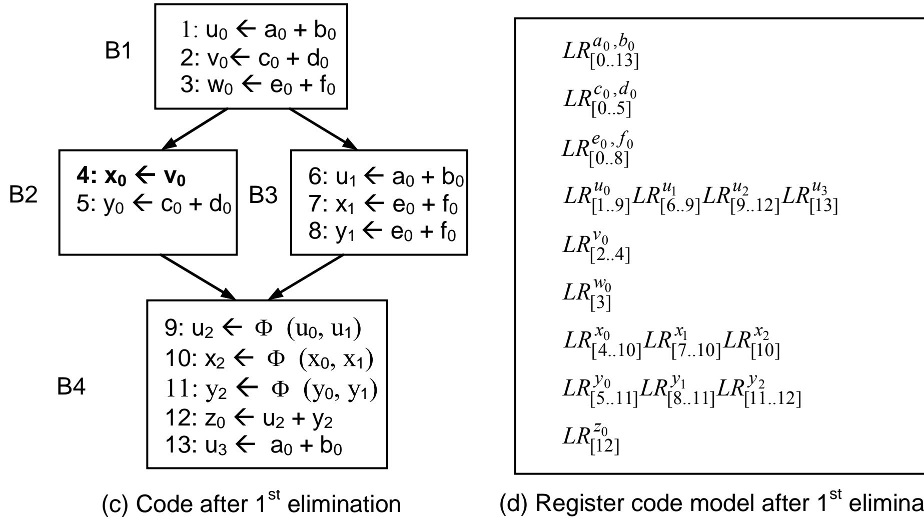
To illustrate how FPSO works, we show an example of profit-driven VN applied to a code segment, shown in Figure 4.10 (a). Figure 4.10 (b) gives the corresponding register code model, where all the live ranges are expressed.

VN processes each block in the dominator tree. The first block processed is B_1. Since none of the expressions in B_1 has been seen, the value number of the defined variables and the expressions are the defined variables themselves. For example, VN[u_0] is u_0 and VN[a_0+b_0] is u_0.



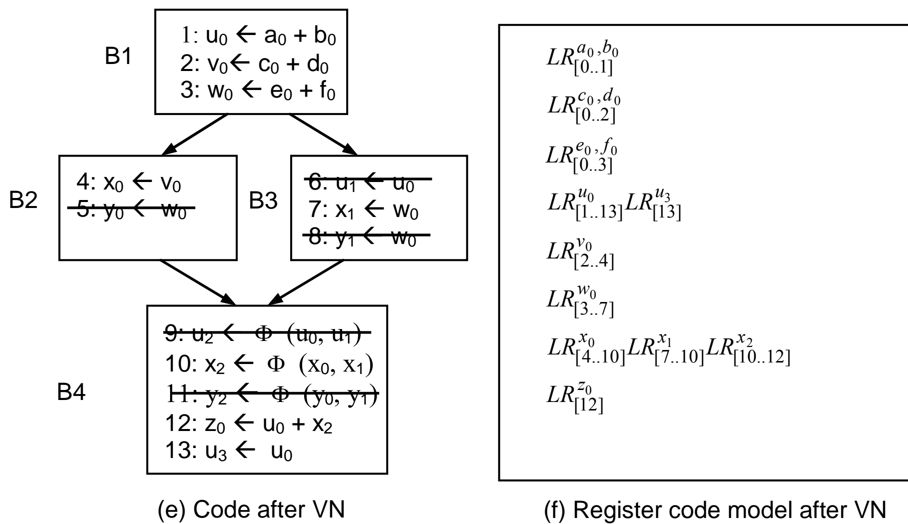
(a) Code before VN

(b) Register code model before VN



(c) Code after 1st elimination

(d) Register code model after 1st elimination



(e) Code after VN

(f) Register code model after VN

Figure 4.10: An example of model-driven VN

The next block processed is B_2. Since the expression $c_0 + d_0$ is defined in block B_1, the first redundant expression, $x_0 \leftarrow c_0 + d_0$, is found. The optimizer calls the engine to predict the profit of eliminating this redundancy. The engine computes the profit on both registers and computation. To predict the profit on registers, the engine first takes the register code model (shown in Figure 4.10 (b)) and the VN optimization model. The engine generates the optimized code model using the incremental dataflow algorithm (shown in Table 4.1). In this case, c_0 and d_0 are deleted as uses. Because c_0 and d_0 are live after statement 4, there is no change on the register code model for the deletions. Also, v_0 is inserted as a use. Thus, the live range of v_0 is lengthened from $LR^{v0}_{[2]}$ to $LR^{v0}_{[2..4]}$. Figure 4.10 (d) shows the updated register code model after replacing this redundancy.

Using the register allocation optimization model, the engine determines how the spills change based on the live range updates. For this example, there is no spill change from deleting c_0 and d_0. But inserting v_0 will increase the spills by one if the number of hardware registers is less than 8. Indeed, the number of live ranges at statement 3 changes from 7 to 8. Choosing which variable to spill depends on the register allocator's spill strategy. In our register allocation model, we select the one that has the fewest number of uses and definitions, which is u_0. This introduces a store before statement 2 and a load after statement 4. The cost associated with the inserted load and store is the register profit as predicted by the engine.

The profit on computation is more easily predicted, which includes the benefit of removing an add statement and the cost to insert a copy statement. To compute the overall profit, the engine uses the functions described in the previous section. If the overall profit is positive, redundancy elimination is applied. Otherwise, it is not applied.

There are 6 redundant expressions that can be eliminated in this example. For every redundant expression, the profitability engine is triggered to predict the profit of applying the redundancy elimination. Figure 4.10 (e) shows the code after VN (assuming all 6 redundant expressions are profitable). The register code model after VN is shown in Figure 4.10 (f), where all the live ranges are changed except for $LR^{z0}_{[12]}$.

4.6 EXPERIMENTAL RESULTS

To evaluate FPSO, we implemented optimization models for six optimizations, including PRE, LICM, VN, copy propagation (CPP), constant propagation (CTP) and dead code elimination (DCE). We integrated FPSO into the Mach SUIF compiler [44]. Mach SUIF was chosen as each optimization in Mach SUIF is implemented and applied as a single pass. Thus, we can incorporate our models for experimentation proposed. We used the DCE pass from Mach SUIF, and implemented PRE, LICM, VN, CPP and CTP.

For the experiments, we used several programs from MiBench [34], MediaBench [21] and SPEC2K to show FPSO works for a variety of programs. We ran our experiments on a dual-processor AMD Athlon MP 1800 1.4 GHz machine and a Pentium III 1.4G machine, running RedHat Linux. The experimental results show the same trend for both machines. We report the results on the Pentium III machine in this chapter. The results for the AMD Athlon are given in Appendix C. We performed node profiling on training data sets to get the basic block frequency counts that were used by the engine. In all experiments, each benchmark was run three times on a lightly loaded machine and the average execution time was computed to factor out system effects.

We show the experimental results of FPSO for two uses. First, we show that profitability is useful for selectively applying optimizations. Second, we show that profitability is also useful in searching for code-specific optimization sequences.

4.6.1 Selectively Applying Optimizations

Always applying an applicable optimization can sometimes lead to a performance degradation. Such a simple heuristic of “always applying” is not sufficient in making decisions about when to apply an optimization. Work has been done to develop heuristics to decide when to apply optimizations [20]. Heuristics can work well in general. However, heuristics tend to be ad hoc and focus specifically on a single or a small class of optimizations. Heuristics also require tuning parameters to select appropriate threshold values. The success of a heuristic can depend on these values and the best choice can vary for different optimizations and code contexts. Instead of

using heuristics, we can determine the profitability of an optimization and selectively apply profitable optimizations using FPSO.

In the following sections, we present an approach that uses heuristics to decide when to apply optimizations. We compare it with our profit-driven approach. We compare the two approaches in terms of prediction accuracy, performance improvement and compile-time overhead. The experimental results show that FPSO is accurate in predicting profitability, which is useful for deciding when to apply optimizations.

4.6.1.1 A heuristic approach

Previous work used heuristics to decide when to apply optimizations, such as register pressure sensitive redundancy elimination, which sets upper limits on allowable register pressure and performs redundancy elimination within these limits [20]. We implemented a similar heuristic. We set the upper limit on allowable live ranges at the places where the redundant expressions will be moved. Redundancy elimination is performed only when the number of live ranges is within the limit. In VN, we eliminate full redundancies and there is no code movement. Thus, the heuristic described here is not useful for VN. In this section, we show the experimental results for heuristic-driven PRE and LICM.

One challenge with a heuristic-driven approach is how to select a limit that can achieve good performance across all benchmarks. Our experiments show that different benchmarks need different limits to achieve the best performance. Figure 4.11 and Figure 4.12 show the run-time performance improvement of heuristic-driven PRE and LICM over the baseline. The baseline compiler applies register allocation and simple instruction scheduling. Also, to enable more opportunities for PRE and LICM, we apply copy propagation, constant propagation and dead code elimination before PRE and LICM. We varied the limit on register pressure from zero to sixteen. For PRE, if the limit is zero, only full redundancies are eliminated. In practice, the limits are usually chosen to be the number of available hardware registers. Hence, eight may be a good limit because there are eight hardware registers that can be allocated for a byte-type variable on the Intel IA-32. Zero, four and sixteen are used to examine stricter or higher limits.

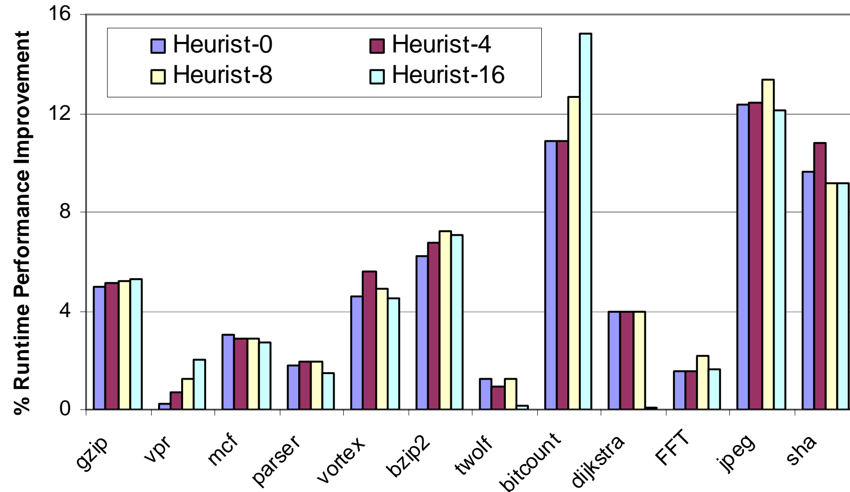


Figure 4.11: Improvement of heuristic-driven PRE with different limits

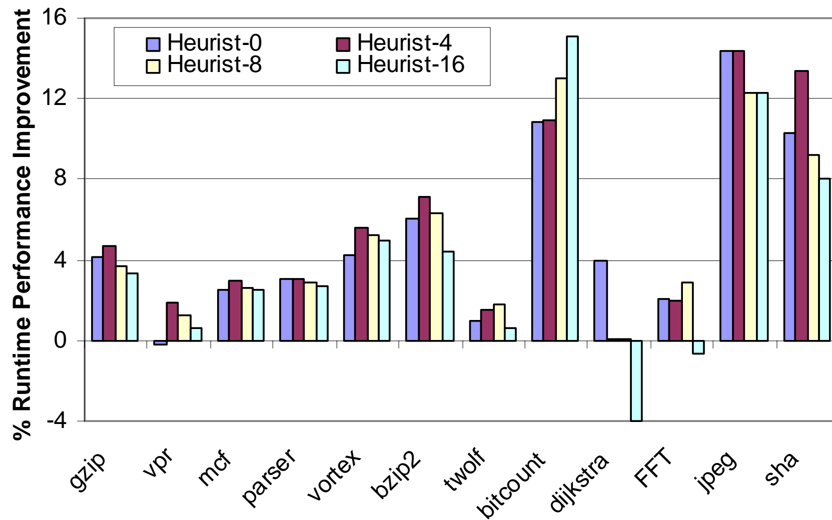


Figure 4.12: Improvement of heuristic-driven LICM with different limits

From the figures, we can see that different benchmarks need different limits to perform the best. For example, for PRE, *gzip* can achieve an improvement of 5.25% when the limit is set to sixteen, while *mcf* needs the limit set to zero to achieve the best improvement of 3.01%. Also, some benchmarks are sensitive to the limit (e.g., *bitcount*), while others are not (e.g., *mcf*). Further, we see that different optimizations may need different limits for the same benchmarks. For example, *gzip* needs the limit set to sixteen for PRE but needs the limit set to four for LICM. If we fix the limit, then we can not always achieve the best improvement with a heuristic.

4.6.1.2 Comparing prediction accuracy

To use FPSO, we must ensure it has good prediction accuracy. We evaluated the prediction accuracy by considering only registers. We did not evaluate the accuracy of the computation profit because the computation is exact in terms of instruction count, given relative node frequencies from a profile. If the relative frequencies in the profile do not match what happens in an actual run, then there can be an inaccuracy in predicting the computation profit. However, this inaccuracy is a property of the profile – not of the models that compute the computation profit.

For deciding whether an optimization should be applied, a correct prediction is one in which we predict there is a benefit/cost for registers (i.e., if register profit is positive, it indicates a spill reduction; otherwise, it shows a spill increase) and actual execution has the same result. For those cases where the actual execution shows there was no impact on registers, we consider the prediction to be correct. The accuracy prediction is measured by how often we make a correct prediction. To validate the prediction accuracy, we checked every prediction and compared the value predicted with the actual execution (i.e., we use the number of memory accesses before and after applying an optimization to reflect the spill changes).

Table 4.4: Prediction accuracy of H-PRE and P-PRE

Benchmark	Heuristic-8 PRE		Profit-driven PRE	
	TP	accuracy%	TP	accuracy%
gzip	43	79.07%	48	89.58%
vpr	290	80.34%	303	96.04%
mcf	51	88.23%	51	86.27%
parser	239	75.73%	293	87.87%
vortex	513	79.72%	530	81.13%
bzip2	58	81.03%	56	78.57%
twolf	484	76.03%	475	91.12%
bitcount	5	100%	5	100%
dijkstra	2	100%	2	100%
FFT	3	33%	3	100%
jpeg	58	96.55%	58	100%
sha	5	100%	5	100%
average	--	82.48%	--	92.55%

Table 4.5: Prediction accuracy of H-LICM and P-LICM

Benchmark	Heuristic-8 LICM		Profit-driven LICM	
	TP	accuracy%	TP	accuracy%
gzip	53	88.68%	45	84.44%
vpr	251	75.70%	230	94.35%
mcf	68	76.47%	52	82.69%
parser	89	79.78%	75	90.67%
vortex	361	77.56%	346	87.57%
bzip2	92	82.60%	88	89.77%
twolf	367	77.93%	345	88.70%
bitcount	3	66.67%	3	100%
dijkstra	5	40%	5	80%
FFT	23	86.96%	23	95.65%
jpeg	82	97.56%	79	100%
sha	21	76.19%	21	95.24%
average	--	77.18%	--	90.76%

Table 4.4 and Table 4.5 show the prediction accuracy of PRE and LICM. In the tables, "TP" is the total number of predictions and "accuracy%" is the prediction accuracy for both heuristic-driven and profit-driven approaches. In the heuristic-driven PRE and LICM, we set the limit to eight.

As Table 4.4 shows, in some cases heuristic-driven PRE had a different number of predictions than profit-driven PRE because of the interactions among PRE instances. The prediction accuracy for heuristic-driven PRE varies from 75% to 100%, with an average of 82.5%. Compared with heuristic-driven PRE, profit-driven PRE makes more correct predictions generally, with the prediction accuracy from 78% to 100% (average 92.6%). Profit-driven PRE considers the impact on register pressure in a more precise way. In some cases, such as *mcf*, although the prediction accuracy of P-PRE is lower than H-PRE, P-PRE achieves a better performance benefit than H-PRE because P-PRE also considers computation (shown in Figure 4.14).

A similar trend can be seen in Table 4.5 for LICM. The prediction accuracy for heuristic-driven LICM varies from 40% to 97%, with an average of 77%. Profit-driven LICM has a higher prediction accuracy, varying from 82% to 100% (average 91%). Because profit-driven PRE and

LICM can make more correct predictions than the heuristic-driven approach, the performance improvement of P-PRE and P-LICM is generally better than heuristic-8 PRE and heuristic-8 LICM.

Table 4.6 shows the prediction accuracy of FPSO for profit-driven VN. It varies from 81% to 100%, with an average of 87%. In some cases, there is no applicable VN, so no accuracy is reported (i.e., *bitcount*, *dijkstra* and *sha*).

Table 4.6: Prediction accuracy of P-VN

Benchmark	Profit-driven VN	
	TP	accuracy%
gzip	30	93.33%
vpr	77	87.01%
mcf	35	82.86%
parser	32	84.38%
vortex	71	94.37%
bzip2	48	87.5%
twolf	101	81.19%
bitcount	0	--
dijkstra	0	--
FFT	4	75%
jpeg	1	100%
sha	0	--
average	--	87.29%

On average, FPSO made inaccurate predictions about 10% of the time. The inaccuracy is primarily from a simplified assumption used in the register optimization model about how the register allocator spills registers. The model assumes that the allocator will select the spill priority based solely on the number of uses and definitions in a live range. However, Mach SUIF's register allocator also uses the number of conflicting edges in the interference graph to make spill decisions. Even without detailed implementation information, our models achieve good accuracy. If more accuracy is needed, the models can be improved by incorporating more implementation information. Also, in FPSO, the prediction inaccuracy does not accumulate, which is important in predicting the profitability of a sequence of optimizations. The engine incrementally updates the code models. The incremental update is accurate. That is, the updated

code model is the same as performing the optimization and reconstructing the code models. The inaccuracy of the prediction only comes from computing the profit associated with every update in an optimization. Thus, the prediction of an optimization does not impact the prediction accuracy of later optimizations.

4.6.1.3 Comparing performance improvement

Using FPSO, we can determine the profitability of an optimization and selectively apply profitable optimizations without setting threshold limits. The cases where optimizations degrade performance can be avoided. In this section, we first compare profit-driven PRE and LICM with always applying PRE and LICM and the heuristic-driven PRE and LICM. We then compare profit-driven VN and always applying VN.

Figure 4.13 and Figure 4.14 show the comparisons of four PRE approaches, in terms of the improvement in the dynamic number of memory accesses and run-time performance over the baseline. The comparison on the number of memory accesses shows how these approaches impact the use of registers. It also helps to explain the run-time performance difference. In the figures, A-PRE is the improvement of always applying PRE when it is applicable. Heuristic-driven PRE is described as above and has two versions based on the register pressure allowed. Best-heuristic is the best case among the limits for each benchmark, while Heuristic-8 uses a fixed limit of eight. Lastly, P-PRE is the performance benefit of profit-driven PRE. Figure 4.15 and Figure 4.16 show the comparisons with the same configurations except for LICM.

As Figure 4.13 shows, the problem with always applying PRE when it is applicable is it may increase register pressure and incur more spills. In most cases, A-PRE increases the number of memory accesses. For example, in *vpr*, A-PRE increases the memory accesses by 5.11%. Both the heuristic approach and P-PRE can avoid the unprofitable instances of PRE, thus decreasing the memory accesses. However, P-PRE considers the registers in a more accurate way (as demonstrated by the prediction accuracy in Section 4.6.1.2). It improves the memory access count more than the heuristic approach. For example, in *gzip*, the best-heuristic, which is unattainable, increases the memory access by 1.1%, while P-PRE decreases the memory accesses by 0.82%. Due to the mispredictions, P-PRE increases the memory accesses more than the heuristic approach for *mcf* and *bzip2*.

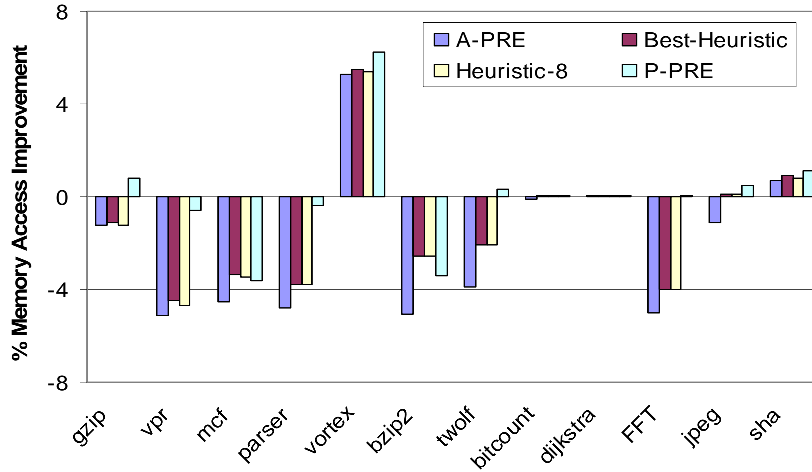


Figure 4.13: Memory access improvement for PRE

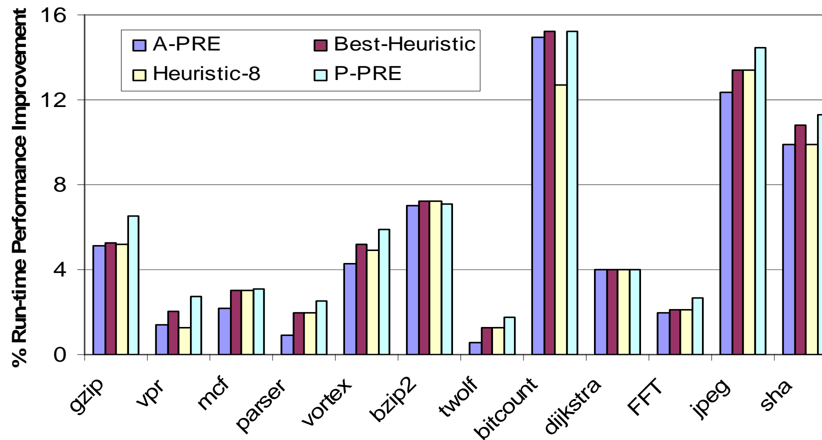


Figure 4.14: Run-time performance improvement for PRE

Figure 4.14 shows the run-time performance improvement for different PRE approaches over the baseline. Both H-PRE and P-PRE achieve performance benefits over always applying PRE. However, the choice of the limits in heuristic-PRE is very important (as described in Section 4.6.1.1). For example, in *vortex*, when the limit is set to 4, H-PRE improves performance by 5.61%. While when the limit is 8, H-PRE improves performance by 4.89%. P-PRE considers both register pressure and computation to predict the profitability of PRE. Thus, in the case where P-PRE increases memory accesses more than H-PRE (*mcf*), P-PRE still improves the overall run-time performance. P-PRE consistently performs as good as or better than the Best-Heuristic for PRE, except for *bzip2*, where predictions are sometimes incorrect. In the cases where P-PRE decreases the number of memory accesses, it improves the run-time performance

more (e.g., *gzip*, *twolf* and *jpeg*). That is, the performance benefit comes from the careful consideration of register pressure. On a register limited machine, like the Intel IA-32, it is particularly important to consider the register pressure as these results indicate.

Figure 4.15 and Figure 4.16 show a comparison of the different approaches for applying LICM. As shown in Figure 4.15, A-LICM can increase register pressure greatly. For example, in *sha*, A-LICM increases the memory accesses by 19.17%. Heuristic LICM and profit-driven LICM selectively choose profitable LICM instances to apply. Thus, in *sha*, Best-Heuristic LICM decreases the memory accesses by 0.74% and P-LICM decreases the accesses by 1.24%.

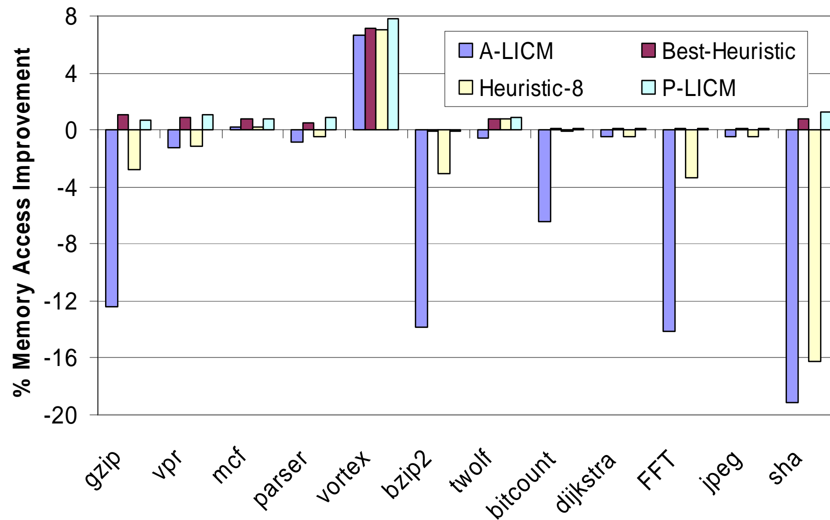


Figure 4.15: Memory access improvement for LICM

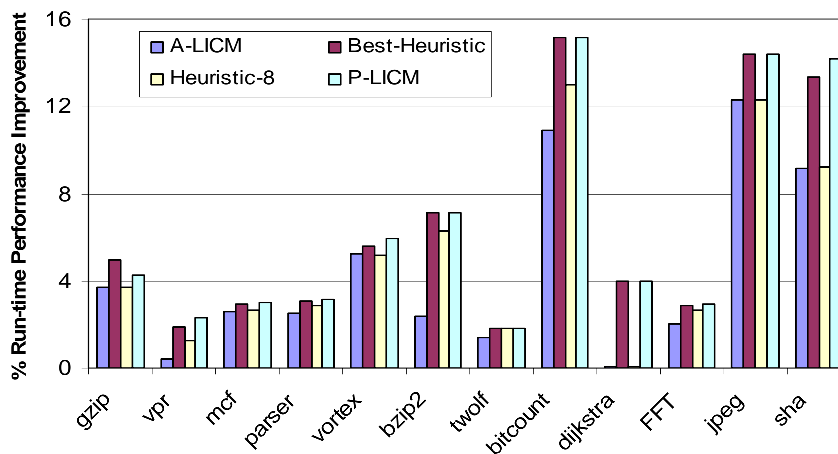


Figure 4.16: Run-time performance improvement for LICM

Figure 4.16 shows the run-time performance improvement for different LICM approaches over the baseline. From the figure, we can see that overall performance of A-LICM can be improved by not applying unprofitable LICMs. Although the heuristic-driven LICM achieves a performance improvement over always applying LICM, it is important to choose the right limit. For example, in *vortex*, with a register pressure limit of eight, the heuristic-driven LICM is worse than always applying LICM. While in the Best-Heuristic (where the limit is sixteen), it is better than always applying LICM. P-LICM can perform at least as well as the best-heuristic LICM in most cases, without tuning the parameters used in H-LICM. However, in one case (*gzip*), due to incorrect predictions, P-LICM has worse performance than the heuristic-driven approach.

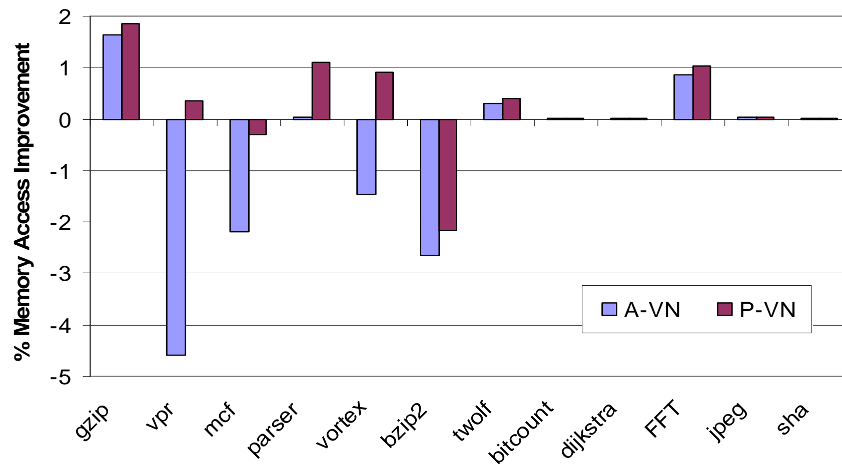


Figure 4.17: Memory access improvement for VN

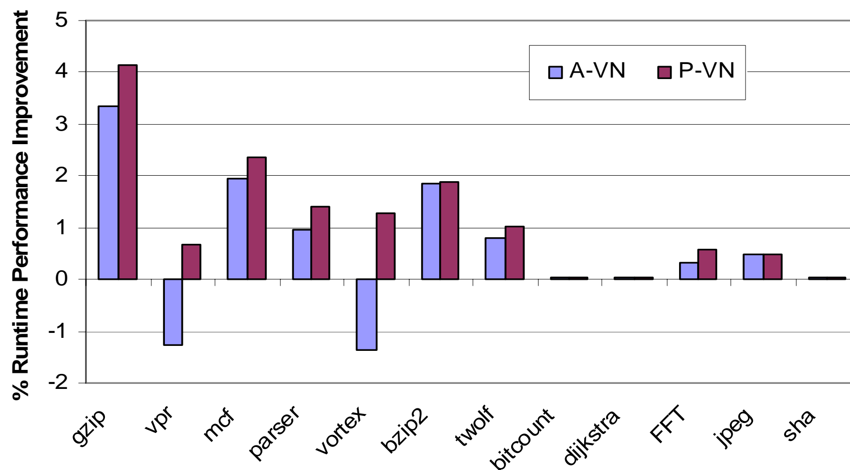


Figure 4.18: Run-time performance improvement for VN

Figure 4.17 and Figure 4.18 show the improvement of memory accesses and run-time performance of profit-driven VN over the baseline, compared to always applying VN. Unlike PRE and LICM, we did not apply other optimizations (e.g., copy propagation or constant propagation) before VN because VN eliminates redundancies by value, not by name. Constant or copy propagation cannot enable more opportunities for VN. Always applying VN degraded performance in some cases because of the increased register pressure caused by eliminating some redundancies, as shown in Figure 4.17. For example, for *vortex*, A-VN increases the memory accesses by 1.46% and thus, the run-time performance was degraded by 1.37%. However, using FPSO, P-VN can selectively apply only profitable redundancy elimination, achieving a performance benefit. For *vortex*, P-VN decreases the memory accesses by 0.91%, and thus, improves run-time performance by 1.28% over the baseline.

From these figures, we see that FPSO is useful for a variety of optimizations, whether the optimization operates on SSA or non-SSA intermediate code formats. In comparison with the always applying approach, our profit-driven approach achieved a better performance improvement. The performance degradation from always applying optimizations was avoided. In comparison with the heuristic approach, our profit-driven approach performed as good as the Best-Heuristic approach, which is unattainable in practice. We conclude that our model-based approach can be effectively used to explore and determine the profitability of optimizations. The profitability property is useful in deciding when to apply optimizations.

4.6.1.4 Comparing compile-time overhead

Because our approach uses models to make decisions, we investigated how compile-time is impacted. We need to ensure that evaluating the models does not overly increase compile-time. Table 4.7, Table 4.8 and Table 4.9 show the compile-time for different optimization strategies for PRE, LICM and VN. In the tables, the compile-time for all compilation passes, including the front-end, optimizations and back-end passes ("Full Compile-time"), and for the optimization pass under consideration ("One Pass Compile-time") are shown. In Table 4.7 and Table 4.8, there are three columns for compile-time comparison. The first column shows the compile-time of always applying approach. The second one gives the percentage of the compile-time increased by the heuristic approach over always applying approach. The third column shows the percentage of compile-time increased by the profit-driven approach over always applying

approach. In Table 4.9, we compare the compile time of always applying VN and profit-driven VN.

Table 4.7: Compile-time for PRE

Benchmark	Full Compile-time			One Pass Compile-time		
	A-PRE	H over A	P over A	A-PRE	H over A	P over A
gzip	44.99	9.18%	17.63%	10.44	36.78%	65.90%
vpr	142.46	52.23%	61.86%	37.61	77.45%	103.56%
mcf	21.84	37.36%	48.49%	4.68	57.39%	72.91%
parser	106.74	25.10%	34.00%	26.7	69.06%	94.23%
vortex	518.5	19.11%	29.64%	88.49	56.78%	79.76%
bzip2	35.58	22.85%	27.15%	10.77	68.25%	86.56%
twolf	767.27	46.05%	58.24%	199.49	90.29%	104.82%
bitcount	6.59	7.13%	10.93%	1.79	56.98%	61.45%
dijkstra	1.15	11.30%	13.91%	0.29	24.14%	48.28%
FFT	4.61	8.89%	13.02%	1.07	41.12%	55.14%
jpeg	35.08	40.34%	53.62%	7.49	80.32%	104.74%
sha	3.04	10.53%	15.13%	0.66	21.21%	36.36%
average	--	24.17%	31.97%	--	56.65%	76.14%

From Table 4.7, the full compile-time for A-PRE varies from approximately 1.2 seconds to 767.3 seconds. The compile-time shown for the heuristic approach is the average for the different limits. It increases from 7% to 52% over A-PRE, with an average of 24%. The heuristic-driven PRE has to compute and update live range information, which causes the compile-time increase. The compile-time for profit-driven PRE increased over A-PRE by 11% to 62%, with an average of 32%. Because P-PRE considers computation and register pressure in a more precise way than the heuristic-driven PRE, it incurs a modest overhead increase. Table 4.7 also shows compile-time for only the PRE optimization pass. The one pass compile-time for A-PRE varies from approximately 0.3 seconds to 199.49 seconds. The compile-time for H-PRE increases from 21% to 90% over A-PRE, with an average of 57%. The compile-time for P-PRE increases over A-PRE by 36% to 105%, with an average of 76%.

Similar compile-time trends can be seen for A-LICM, H-LICM and P-LICM in Table 4.8. The full compile-time for A-LICM varies from approximately 1.2 seconds to 579.9 seconds. The

heuristic-driven LICM increases compile-time over A-LICM from 5% to 38% (average 21%) and profit-driven LICM increases compile-time over A-LICM by 7% to 56% (average 28%). The one pass compile-time for A-LICM varies from approximately 0.35 seconds to 165.49 seconds. The compile-time for H-LICM increases from 11% to 88% over A-LICM, with an average of 49%. The compile-time for P-PRE increases over A-PRE by 14% to 132%, with an average of 68%.

Table 4.8: Compile-time for LICM

Benchmark	Full Compile-time			One Pass Compile-time		
	A-LICM	H over A	P over A	A-LICM	H over A	P over A
gzip	45.97	23.65%	27.65%	12.94	57.26%	69.63%
vpr	127.84	18.80%	27.35%	32.36	58.19%	79.49%
mcf	20.51	32.42%	9.10%	4.73	49.89%	72.94%
parser	106.08	21.86%	30.82%	29.53	58.42%	88.93%
vortex	511.8	11.34%	15.48%	98.87	36.41%	47.25%
bzip2	34.63	22.81%	30.26%	11	57.55%	79.55%
twolf	579.97	37.73%	55.50%	165.49	88.14%	132.64%
bitcount	6.63	4.52%	7.39%	1.88	16.49%	25.53%
dijkstra	1.19	7.56%	10.08%	0.35	11.43%	14.29%
FFT	4.58	35.37%	41.48%	1.21	60.33%	85.12%
jpeg	25.26	20.23%	28.82%	6.38	56.99%	70.82%
sha	2.78	7.63%	25.90%	0.81	38.27%	54.32%
average	--	21.16%	28.32%	--	49.11%	68.38%

From Table 4.9, the full compile-time for A-VN varies from 1.7 seconds to 512 seconds. The profit-driven VN increases the compile-time over always applying VN from 12% to 18%, with an average of 15%. The one pass compile-time for A-VN is from 0.25 to 21 seconds. The P-VN increase compile-time over A-VN from 22% to 49%, with an average of 32%. Compared with P-PRE and P-LICM, the compile-time increased by P-VN is smaller. One reason is that there are fewer instances of VN than PRE and LICM (shown in the next section). The overhead

of the profit-driven approach depends on how many instances of the optimization appear in the code and the impact of every instance.

Table 4.9: Compile-time for VN

Benchmark	Full Compile-time		One Pass Compile-time	
	A-VN	P over A	A-VN	P over A
gzip	47.02	15.82%	6.82	26.83%
vpr	127.93	14.88%	18.17	26.25%
mcf	25.98	15.97%	3.61	22.44%
parser	97.2	17.78%	13.56	33.48%
vortex	511.68	14.72%	61.95	27.44%
bzip2	28.59	17.59%	3.47	48.99%
twolf	284.34	16.93%	40.4	34.16%
bitcount	7.33	12.55%	1.81	26.52%
dijkstra	1.67	13.17%	0.25	24.00%
FFT	5.66	17.49%	0.84	44.05%
jpeg	29.11	15.94%	4.27	37.24%
sha	3.58	12.29%	0.55	27.27%
average	--	15.43%	--	31.56%

As the tables show, the increase in compile-time of our profit-driven approach is modest and about the same as the heuristic-driven approach. These small increases show that our approach is feasible and efficient. However, our prototype has several implementation artifacts that negatively impact performance; a production implementation could decrease the compile-time further. We conclude that the compile-time increase is worth the benefit of applying the optimizations more effectively without tuning parameters.

4.6.2 Searching for Code-specific Optimization Sequences

It is known that the order to apply optimizations can have an impact on performance. However, traditional compilers typically choose a fixed order to apply optimizations. It is almost impossible that this single order can work best for every application.

Previous work has focused on experimentally searching for code-specific optimization sequences. Almagor et al. presented the promising results of using a genetic algorithm to find effective optimization sequences [1]. However, the profitability of a sequence of optimizations is evaluated by experimentation. That is, they perform the optimizations in a sequence and execute the optimized code to evaluate a candidate sequence. Thus, it is very costly to find an effective sequence, even for small kernel applications.

Instead of executing the code, we can predict the profitability of a sequence of optimizations using FPSO. The compile-time overhead will be greatly reduced because the time spent to execute the code can be avoided.

In our experiments, we compared three approaches (i.e., fixed-order approach, empirical approach and model-based approach) to find a good order of applying optimizations. The fixed order that we used in our experiments is "VN, CPP, CTP, DCE, PRE, CPP, LICM, CPP, CTP, DCE". The choice of the order was based on the interactions studied in Whitfield and Soffa's work [50]. Their study can order some of optimizations, for example constant propagation should apply before dead code elimination. However, the order for other optimizations can not be decided. According to code context, different order maybe needed for the best performance.

The genetic algorithm that we used has a similar configuration as in Almagor's work [1]. We performed the search on each *file* using 10 generations. Each generation had a population of 20 sequences. Every sequence had ten optimization passes, chosen from these six optimizations. At each generation, the best 10% of the sequence survive without any change. The rest of the new generation is created by the crossover operation, followed by the character-by-character mutation with the mutation rate is 5%. A hash table is maintained to keep track of the sequence evaluated to avoid evaluating the same sequence twice. For the empirical approach, we ran the code with the train input set to evaluate the candidate sequences. For our approach, we used FPSO to predict profitability.

In the following sections, we show the compile-time and performance improvement comparison among the three approaches: fixed-order approach, empirical approach and model-based approach.

4.6.2.1 Comparing compile-time overhead

Our approach uses models to predict the profit of a sequence of optimizations (instead of executing the code). Thus, the compile-time can be greatly reduced. We investigated the compile-time for the empirical approach and our model-based approach. Figure 4.19 shows the compile-time for both approaches in hours.

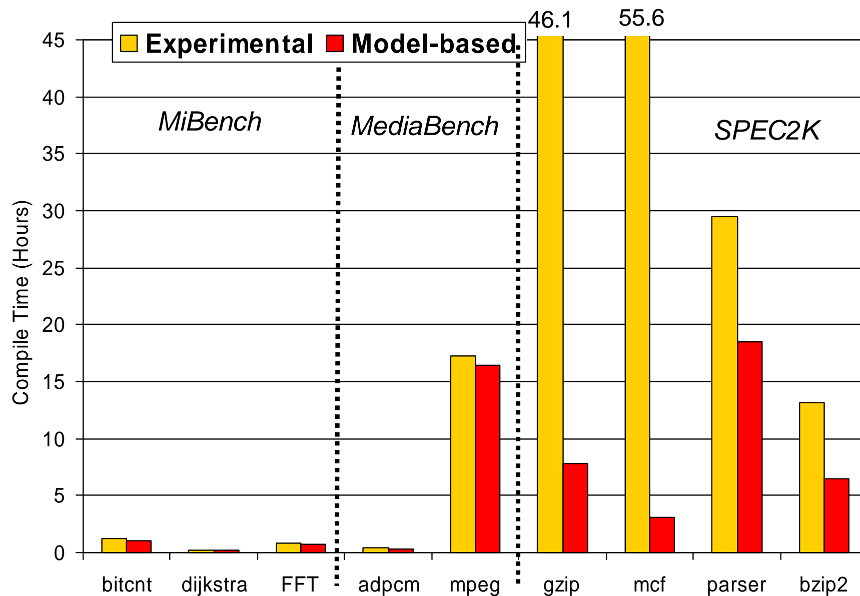


Figure 4.19: Compile-time of the experimental and model-based approaches

From Figure 4.19, the compile-time for the empirical approach varies from approximately 0.24 to 55.6 hours. The empirical approach has to perform the optimizations and execute the code to evaluate the sequences. For benchmarks with a long execution time, (e.g., SPEC2K benchmarks), most of the compile-time was spent on executing the code. For example, the empirical approach spent 55.6 hours to find effective sequences for *mcf*, among which 53.4 hours were for executing the code.

The compile-time for our model-based approach varies from 0.19 hours to 18.5 hours. For the SPEC2K benchmarks, the compile-time for our model-based approach is much smaller than the empirical approach, with up to 17 times compile-time savings. For example, for *mcf*, our

model-based approach spent 3.1 hours to find good sequences, while the empirical approach spent 55.6 hours. On the other hand, for the benchmarks from Mibench and Mediabench, the compile-time savings of our model-based approach is not much. For example, our approach used 0.99 hours to find good sequence for *bitcount* while the empirical approach used 1.21 hours.

4.6.2.2 Comparing performance improvement

Using the genetic algorithm, we can find an effective optimization sequence for each file. Thus, by applying those sequences, the program should have better performance than a fixed-order sequence.

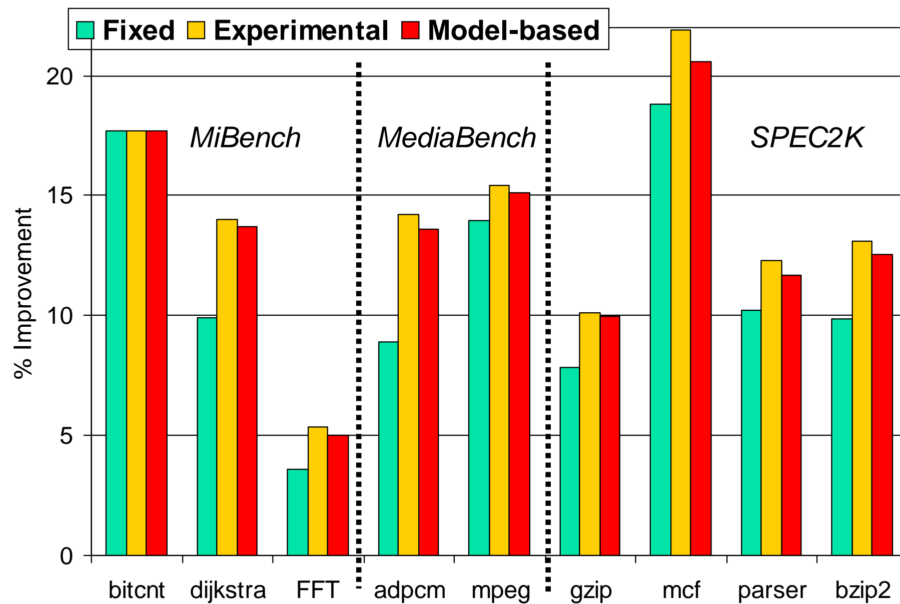


Figure 4.20: Performance of three approaches

Figure 4.20 shows the performance improvement for the three approaches over the baseline. Performance was measured by the number of instructions executed. As the figure shows, the problem with the fixed-order approach is that the fixed order may not be a good order for some files. The genetic algorithm, on the other hand, can find the code-specific sequences. In most cases, the empirical and model-based approaches improve performance more than the fixed- order approach. For example, in *dijkstra*, the fixed-order approach improves performance by 9.9% while the empirical approach and the model-based approach have improvement of

14.0% and 13.7%. From the figure, we can see that our model-based approach can achieve similar improvements as the empirical approach.

As experimental results given in this section show, the compile-time of the empirical approach to search for code-specific optimization sequences is huge for large benchmarks (e.g., SPEC2K benchmarks), which makes the empirical approach not scalable. Our model-based approach is practical and scalable: It can find good sequences for large benchmarks with much less compile-time. We conclude that the profitability property is useful in searching for code-specific optimization sequences.

5.0 FPLO: PREDICTING PROFITABILITY OF LOOP OPTIMIZATIONS

In this chapter, we describe a framework instance, called FPLO, for predicting the profitability of loop optimizations. Because loop behavior dominates data cache performance [37], we consider cache performance as our indicator for overall performance of loop optimizations.

As the disparity between processor and main memory speed increases by approximately 50 percent per year, the use of caches with high hit rates has become critical for performance [18]. Data caches are designed to exploit locality, and naturally, they work best for programs that have high locality. Some optimizations are designed to improve cache performance by rearranging code to have better locality. Other optimizations are not designed specifically for this purpose and may negatively impact cache performance and overall performance. We need to determine the profit of an optimization on cache performance and overall performance.

In the following sections, we describe the code model, loop optimization models and cache resource model. Next, we describe the profitability engine that uses the models to make predictions. Lastly, we show the experimental results.

5.1 CODE MODEL FOR CACHE

To predict the cache profit of loop optimizations, we need to express those code characteristics that affect the cache, which are a loop's header and the sequence of array references in a loop body. Loop headers give the total number of memory accesses for an array reference. The loop organization and array reference pattern determine how the memory accesses are ordered. Different orders result in different data reuse and amounts of cache misses.

Our code model for cache represents the loop nests in the code as a sequence of loop nests, $\langle ln1, ln2, \ldots \rangle$. The order of loop nests in the sequence is as they appear in the code. Each

loop nest ln is represented as $\underset{N-1}{\oint}\overset{ub}{\underset{lb}{step}} \cdots \underset{1}{\oint}\overset{ub}{\underset{lb}{step}}\underset{0}{\oint}\overset{ub}{\underset{lb}{step}} \langle R \rangle$, where $\underset{N-1}{\oint}\overset{ub}{\underset{lb}{step}} \cdots \underset{1}{\oint}\overset{ub}{\underset{lb}{step}}\underset{0}{\oint}\overset{ub}{\underset{lb}{step}}$ corresponds to the loop headers and $\langle R \rangle$ is the array reference sequence. For convenience, we put a number under each loop header to express its order in the loop nest.

A loop header, $\oint\overset{ub}{\underset{lb}{step}}$, consists of a lower bound (*lb*), upper bound (*ub*), and iteration step (*step*). The array reference seqeuence, $\langle R \rangle$, consists of all array references in a loop body in the order that they appear textually in the code. An array reference refers to an array element and includes the name of the array and its access function (subscript). Because optimizations usually change the access functions (and not the name of the array), we use an equation, $\overrightarrow{Ref} = A \times \vec{I} + \vec{C}$, to represent the access function of an array reference. A is the access matrix, $\vec{I}$ is the loop index vector and $\vec{C}$ is the constant vector [22]. This equation can be written as:

$$\begin{bmatrix} sub0 \\ \vdots \\ subd-1 \end{bmatrix} = \begin{bmatrix} A00 & \cdots & A0(N-1) \\ \vdots & \ddots & \vdots \\ A(d-1)0 & \cdots & A(d-1)(N-1) \end{bmatrix} \times \begin{bmatrix} I0 \\ \vdots \\ IN\text{-}1 \end{bmatrix} + \begin{bmatrix} C0 \\ \vdots \\ Cd-1 \end{bmatrix}$$

Although we consider only perfect loop nests, our technique can be extended to handle non-perfect nested loops by including the loop index $\vec{I}$ in every array reference.

```
for ( i=0; i<N; i++)
     for ( j=0; j<N; j++)
          a[i] = a[i] + b[j][i]*c[i][j]+c[i+1][j];
```

(a) Original loop nest

$$\underset{1}{\oint}\overset{N-1}{\underset{0}{1}}\underset{0}{\oint}\overset{N-1}{\underset{0}{1}} \langle (Aa, Ca), (Ab, Cb), (Ac1, Cc1), (Ac2, Cc2), (Aa, Ca) \rangle$$

(b) Code model for the loop nest

Figure 5.1: A loop nest and its code model

Figure 5.1 shows an example of a loop nest and its code model, where (*Aa*, *Ca*) represent the access matrix and constant vector of the array reference a[i] (same for the array references b and c).

5.2 OPTIMIZATION MODELS

Loop optimizations change the loop headers and array references. Thus, our optimization model for a loop optimization captures these changes. We use an *impact function*, $f_{opt}(\langle ln1, ln2, \ldots\rangle) = \langle ln1', ln2', \ldots\rangle$, to map an original sequence of loop nests to a new sequence.

We develop an optimization model for each loop optimization considered in this research. In the following sections, we present our models for loop interchange, unrolling, tiling, reversal, fusion, and distribution [3].

5.2.1 Loop Interchange

Loop interchange exchanges the position of two loops in a loop nest. The optimization model for loop interchange is illustrated in Figure 5.2. The impact function, $f_{interchange}$, maps an original loop nest to a new loop nest, according to the semantics of loop interchange. Essentially this function exchanges *lb, ub* and *step* of loop *i* with that of loop *j*. It also changes the array reference sequence $\langle R\rangle$ by a function $g(\langle R\rangle)$. This function determines the new array reference sequence for the transformed loop by applying *h(r)* on every reference *r* in $\langle R\rangle$. Function *h(r)* computes a new array reference by exchanging column *i* and *j* in the access matrix *A* from *r*'s reference equation. *l(A)* handles the column interchange. The constant vector, *C*, for *r* is unchanged.

Consider the example in Figure 5.1. Using the model in Figure 5.2, we determine the new loop nest. The new header is determined by exchanging *lb*, *ub*, and *step* for loop l_i and l_j. The new array reference sequence, $\langle R'\rangle = \langle r0', r1', r2', \ldots, r4'\rangle$, is determined by changing the access matrix of every array reference in $\langle R\rangle$. For example, the access matrix of a[i] is changed from $[1 \quad 0]$ to $[0 \quad 1]$ and b[j][i] is changed from $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

INPUT: $\oint_{N-1} \cdots \oint_{1} \oint_{0} \langle R \rangle$ and interchange is legal for loops *i, j*;

$$f_{Interchange}(\oint_{N-1} \cdots \oint_{i} \cdots \oint_{j} \cdots \oint_{0} \langle R \rangle) = \oint_{N-1} \cdots \oint_{j} \cdots \oint_{i} \cdots \oint_{0} g(\langle R \rangle)$$

where $g(\langle R \rangle) = \langle \forall (r \in \langle R \rangle)\, h(r) \rangle$,

$h(r) = (l(A), C)$, and $l(A) = A[:][i] \leftrightarrow A[:][j]$

Figure 5.2: Loop interchange optimization model

5.2.2 Loop Unrolling

Loop unrolling duplicates a loop's body a number of times [3]. It is commonly understood that loop unrolling has little impact on data cache performance when register pressure is not considered. However, we model loop unrolling to demonstrate the effectiveness of our models. The optimization model for loop unrolling is shown in Figure 5.3.

INPUT: $\oint_{N-1} \cdots \oint_{1} \oint_{0} \langle R \rangle$ and unroll factor U;

$f_{unroll}(\oint_{N-1} \cdots \oint_{1} \oint_{0} \langle R \rangle) = \langle ln_{unroll}, ln_{rest} \rangle$ where

$ln_{unroll} = \oint_{N-1} \cdots \oint_{1} \oint_{0}^{step \times U} g(\langle R \rangle)$ and

$ln_{rest} = \oint_{N-1} \cdots \oint_{1} \oint_{0 \left\lceil \frac{ub+1}{U} \right\rceil \times U} \langle R \rangle$

$g(\langle R \rangle) = \langle R \rangle \wedge (\bigwedge_{i=1}^{U-1} \langle \forall (r \in \langle R \rangle)\, h(r,i) \rangle)$ and

$h(r,i) = (A, l(C,i))$ and

$l(C,i) = \forall (s \in \{a \mid A[a][N-1] \neq 0\})\, C[s] + i$

Figure 5.3: Loop unrolling optimization model

The impact function $f_{unrollling}$ maps an original loop nest to two nested loops (one for the unrolled loop and one for possible leftover iterations) according to the semantics of loop

unrolling. In the unrolled loop nest, the step of the innermost loop is changed to $step \times U$ (U is the unroll factor) and the array reference sequence, $\langle R \rangle$, is changed by a function g, which combines $\langle R \rangle, \langle R1 \rangle, \langle R2 \rangle, \cdots, \langle RU-1 \rangle$ together. A reference $\langle Ri \rangle$ is determined by applying the function $h(r,i)$ on every array reference, r, in $\langle R \rangle$. Function $h(r,i)$ models how the access matrix and constant vector of a reference are changed. It keeps the access matrix unchanged and applies $l(C,i)$ on the constant vector. Essentially, $l(C,i)$ changes C by adding i to those dimensions that have the innermost loop control variable. In the loop nest for the leftover iterations, the lower bound of the innermost loop is changed to $\left\lceil \frac{ub+1}{U} \right\rceil \times U$ and the array reference sequence, $\langle R \rangle$, is unchanged.

Using the example from Figure 5.1, suppose that the unroll factor is two. With the model from Figure 5.3, the unrolled loop's header becomes, $\overset{N-1}{\underset{1}{\oint}} \underset{0}{1} \overset{N-1}{\underset{0}{\oint}} \underset{0}{2}$, from the rolled loop's header, $\overset{N-1}{\underset{1}{\oint}} \underset{0}{1} \overset{N-1}{\underset{0}{\oint}} \underset{0}{1}$, by doubling the step of the innermost loop. The array reference sequence for the unrolled loop is $\langle r0, r1, \cdots, r5, \cdots, r9 \rangle$, where r5 to r9 is determined by keeping the access matrix and changing the constant vector of r0 to r4 in $\langle R \rangle$. For example, r6 (b[j+1][i]) has the same access matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ as $r1$ (b[j][i]) , but a different constant vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Second, we determine the loop nest for the leftover iterations. Its loop header is $\overset{N-1}{\underset{1}{\oint}} \underset{0}{1} \overset{N-1}{\underset{0}{\oint}} \underset{\left\lceil \frac{N}{2} \right\rceil \times 2}{1}$ and its array reference sequence is unchanged.

5.2.3 Loop Tiling

Loop tiling improves cache reuse by dividing an iteration space into tiles and transforming the loop nest to iterate over them [3]. The optimization model for loop tiling is shown in Figure 5.4.

$$\textbf{INPUT}: \oint_{N-1} \cdots \oint_{1} \oint_{0} \langle R \rangle \text{, tiling loops } t1, \cdots, tn \text{, with tile size } ts1, \cdots, tsn \text{ respectively;}$$

$$f_{tiling}(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0} \langle R \rangle) = g(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0}) \; f\langle R \rangle \text{ where}$$

$$g(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0}) = \oint_{N+n-1} {}^{ubn}_{lbn} tsn \cdots \oint_{N} {}^{ub1}_{lb1} ts1 \oint_{N-1} \cdots \oint_{tn} {}^{h(n)}_{xn} \cdots \oint_{t1} {}^{h(1)}_{x1} \cdots \oint_{0}$$

$$h(i) = \min(ubi, xi + tsi - 1) \text{ and}$$

$$f\langle R \rangle = \langle \forall (r \in \langle R \rangle)\ (l(A), C) \rangle \text{ where } l(A) = [0A]$$

Figure 5.4: Loop tiling optimization model

The impact function, f_{tiling}, maps an original loop nest to a new loop nest by changing its loop header by function *g* and changing its array reference sequence $\langle R \rangle$ by function *f*. Essentially, function *g* adds $\oint_{N+n-1} {}^{ubn}_{lbn} tsn \cdots \oint_{N} {}^{ub1}_{lb1} ts1$ to the outermost and changes lower bound and upper bound of loops to be tiled. (The input to the model specifies the number of loops to be tiled, *n*, their index in the header sequence $t1, t2, \cdots, tn$ and their tile size, $ts1, ts2, \cdots, tsn$.) The lower bound of l_{ti} changes to the control variable of *lN+ti-1* (represented as *xi*). The upper bound of l_{ti} changes to a function *h(i)*, which gets the minimum number of original upper bound and ($xi + tsi - 1$). On the other hand, function $f(\langle R \rangle)$ changes the access matrix (*A*) by function *l(A)* of every array reference in $\langle R \rangle$, where function *l(A)* adds *n* columns of zero to *A*'s first n columns. The constant vector (*C*) does not change.

For the example in Figure 5.1, if we tile *li* and *lj* with tile size 64, using the model shown in Figure 5.4, we get the new loop header as $\oint_{3} {}^{N-1}_{0} 64 \oint_{2} {}^{N-1}_{0} 64 \oint_{1} {}^{\min(N-1, x2+63)}_{x2} 1 \oint_{0} {}^{\min(N-1, x1+63)}_{x1} 1$. The access matrix of every array reference is changed; e.g., b[j][i] is changed from $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

5.2.4 Other Loop Optimizations

We also develop impact function for loop reversal, loop fusion and loop distribution. The detailed optimization models for these optimizations are described in Appendix A.

5.3 CACHE MODEL

We use a cache model to estimate the cache cost of executing a loop nest. This model indicates how a given reference pattern affects cache misses (and hits) under the assumption of a single issue in-order pipelined processor with a blocking cache. To improve locality, we want to reduce the number of cache misses, and in evaluating the impact of an optimization, we want to know whether the number of cache misses is decreased by the optimization.

Because some array references may access the same cache line in the same or different iteration (due to group temporal or spatial reuse), we group references to avoid over estimating the number of cache misses when a reference may access a cache element that has been previously loaded. We adapt Mckinley et al.'s RefGroup algorithm [37] to formulate RefSet using our code model representation to calculate group spatial and temporal reuse with respect to the innermost loop. We consider two references $r1$ (A1, C1) and $r2$ (A2, C2) that refer to the same array that belongs to the same RefSet if:

1) $A1 = A2$, $\forall ik$ (ik is the row index of the none-zero elements in the last column of *A1*) $|C1[ik]\text{-}C2[ik]| = p \times stepN-1$ (p is a positive integer and $p \leq 2$, $stepN-1$ is the iteration step of the innermost loop) , and all other ip ($ip \neq ik$), $C1[ip] = C2[ip]$ or
2) $A1 = A2$, $C1[i] = C2[i]$ ($0 \leq i < d\text{-}1$), and $|C1[d-1] - C2[d-1]| < cls$ (cls is the cache line size, and d is the dimension of the array.

Condition 1 accounts for group temporal reuse, and condition 2 accounts for group spatial reuse.

Once we account for group reuse, we can calculate the cache misses of a representative array reference, say *Rα*, in a RefSet. Initially, we used McKinley et al.'s cache cost model. While their model accurately estimated cache misses under some circumstances, it did not have sufficient overall accuracy needed to achieve good results for all of our optimization models. The

reason is that it handles cache conflict misses in a simple manner and did not accurately reflect all possible sources of conflict misses.

Cache conflicts are difficult to predict and estimate [45]. From our own experiments, we found that cache conflict misses can vary widely with slight variations in the problem input size. Ghosh et al. [18] proposed a precise algorithm, Cache Miss Equation (CME), to generate a set of equations for cold and replacement misses. The solutions to these equations represent all compulsory and conflict misses. However, finding all reuse vectors and setting up complete cache miss equations is very complex. Instead, our goal was to develop a more feasible and practical model that tailors Ghosh's scheme to our specific problem of predicting the impact of loop optimizations on cache performance. We simplified Ghosh's model to calculate the cache misses of *Rα*. Table 5.1 explains the terms that are used in computing the cache misses of an array reference, *Rα*.

Table 5.1: Terms used in cache model

Term	Meaning
TI	Total number of iterations in the loop nest
cls	Cache line size
FP	Footprint of *Rα* (i.e., how many different elements *Rα* access over all iterations)
CRT	Fraction of *Rα*'s self temporal-reuse that cannot be realized (realizing a reuse means a reuse can result a cache hit)
CRS	Fraction of *Rα*'s self spatial-reuse that cannot be realized

We estimate the cache misses of *Rα* to be:

$$CM(R\alpha) = TI \times (\frac{FP}{TI} \times (1 - CRT) + CRT) \times (\frac{1}{cls} \times (1 - CRS) + CRS) \qquad (1)$$

We compute *CRS* and *CRT* in a way similar to the CME approach by solving a set of equations that sets the cache block address of Rα equal to that of other references within its reuse distance to find possible conflicts. The reuse distance is the number of iterations between a reuse and its previous access. For example, in Figure 5.1, b[j][i]'s spatial reuse distance is N, because an access in iteration (i, j) can be spatially reused by another access in iteration $(i+1, j)$, which is N iterations behind. With this approach, we take into account the cache conflicts in an accurate manner. We illustrate how to compute *CRS* and *CRT* for b[j][i] in Figure 5.1. Suppose that we

have direct-mapped cache. First according to b[j][i]'s spatial reuse distance N, we set up a set of equations to get *CRS* for b[j][i], including:

$$\forall t \in [0, N-1] \; Addr(b[j][i]) = Addr(c[i][j+t]) \quad (2)$$

$$\forall t \in [0, N-1] \; Addr(b[j][i]) = Addr(c[i+1][j+t]) \quad (3)$$

$$\forall t \in [1, N] \quad Addr(b[j][i]) = Addr(b[j+t][i]) \quad (4)$$

$$Addr(b[j][i]) = Addr(a[i]) \quad (5)$$

The solutions to every equation represent all the iterations where b[j][i] conflicts with another reference. Because of direct mapping, the total number of iterations that b[j][i] will be evicted by another reference will be the union of these solution sets. We compute *CRS* by dividing the total number of conflict iterations by the total number of iterations. As b[j][i] has no temporal reuse, *CRT* equals one.

5.4 PROFITABILITY ENGINE

The profitability engine inputs the code model for cache, loop optimization models and cache model to predict the profitability of loop optimizations.

When a loop optimization is applicable, the optimizer extracts the loop nests from the original code and expresses them using the code model (described in section 5.1). The optimizer then triggers the engine. When the engine is triggered, it inputs the code model, optimization models and cache model. It applies the optimization model on the code model and generates a new code model that represents the optimized code. Finally, the engine applies the cache model on the original and optimized code models. With a cache configuration, the cache model estimates the cache misses according to the representation of the code model. The engine outputs the profit of applying a loop optimization by determining the difference between cache misses of the original and optimized code models.

5.5 EXPERIMENTAL RESULTS

To evaluate FPLO, we implemented the models and extensively tested them using a number of loop benchmarks that were commonly used in other researches [22]. There are two types of benchmarks: those with a single loop nest (*alv, irkernel, lgsi, smsi, srsi, tfsi, tomcat3, biquad_N, lms, gdevcdj* and *pegwit*) and those with multiple loop nests (*adi, aps, eflux, tomcat, vpenta,* and *bmcm*). The benchmarks have from one to nine loop nests and from four to thirty two array references in a loop nest.

In the compiler infrastructure we used for scalar optimizations, Mach SUIF [44], there are no loop optimizations implemented. Thus, we implemented a stand-alone tool for FPLO, which inputs the cache code model, loop optimization models and cache resource model and outputs the profit of applying an optimization. Based on the output from the tool, we manually apply loop optimizations. To experimentally evaluate our approach, we measured the actual execution behavior by simulating the code using SimpleScalar sim-outorder microarchitecture simulator [9]. We simulate a processor pipeline with in-order single issue and a critical-word first non-blocking cache. The processor has a two entry load-store queue and can sustain up to two cache misses before stalling. This model is similar to several popular embedded processors, including MIPS' 4Kp (R4000), ARM's 94x series, and IBM's PowerPC 405. The cache that is used in our simulation is a direct-mapped data cache with 32-byte block size. Using a small cache with scaled working sets allows us to investigate the impact of different sized working sets without suffering the high simulation times required for large data sets. The performance numbers that we present will scale to other cache configurations and working set sizes.

We first present the prediction accuracy of FPLO. Then we compare our profit-driven approach with an approach that always applies an optimization if applicable. We also show other uses of FPLO in selecting the most beneficial loop optimizations. Finally, we present the compile-time overhead of FPLO.

5.5.1 Model Accuracy

To use FPLO to drive the application of loop optimizations, we must ensure that it has good prediction accuracy, as we did for scalar optimizations. To measure the prediction accuracy, we

ran the original benchmarks and optimized ones with our simulation framework and compared the predictions from FPLO against the simulation results.

First, we compared the predictions of cache miss reductions against the simulation results. When replaced by a simpler cache model [36], FPLO could not make correct predictions in some cases. With our cache model, FPLO predicted more accurately. Figure 5.5 shows an example of how FPLO with different cache models compares with the simulation results for loop interchange on *irkernel* with varying trip counts. With a simple cache model, wrong predictions about whether to apply interchange were made in some cases. For example, when the trip count equals 128, FPLO with a simple cache model predicts that interchange reduces the number of cache misses by 8224. But the simulation result showed that interchange increased the number of cache misses by 3937. Using our cache model, FPLO correctly predicted the trend of cache misses increased to 3810. Other benchmarks and other optimizations showed a similar trend.

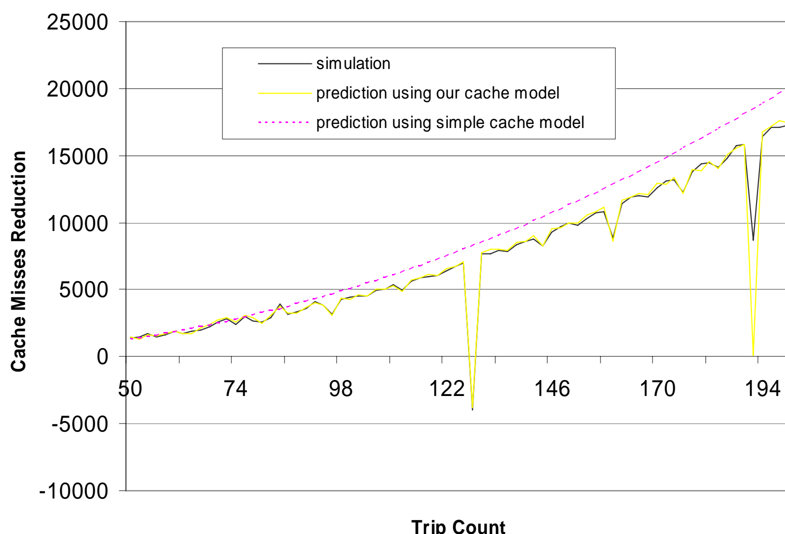


Figure 5.5: Loop interchange on *irkernel* with different cache models

We computed prediction accuracy for FPLO integrated with our cache resource model. If an optimization improves cache performance shown in the simulation results, and our model predicted that the optimization should be applied, then we consider this to be a correct prediction. If the simulation result does not match our predicted result, then it is a misprediction. Prediction accuracy captures how often FPLO gives the correct prediction. Table 5.2 shows prediction accuracy for the single nest benchmark loops with varying trip counts. For each benchmark, the trip count was varied from 50 to 200. From the table, the prediction accuracy

ranged from 81.6% to 100% across all benchmarks and optimizations with an average of 97%. Although there is high accuracy across all optimization models, loop reversal has the lowest accuracy. The reason is that loop reversal has a minimal impact on data cache locality (i.e., the cache miss reduction of applying reversal is very small), and as such, it is difficult to predict its benefit. Although FPLO chose not to apply loop reversal in these cases, this choice did not degrade the effectiveness of FPLO because the benefit of applying reversal was so small that it can be ignored (see Figure 5.6).

Table 5.2: Prediction accuracy for single-loop nest benchmarks

Benchmark	Interchange	Tiling	Reversal
alv	100%	100%	97.4%
irkernel	98.7%	100%	93.4%
lgsi	100%	100%	82%
smsi	100%	100%	86.8%
srsi	100%	100%	86.8%
fsi	100%	97.4%	100%
tomcat3	98.7%	92.1%	93.4%
biquad_N	89.5%	88.2%	100%
gdevcdj	100%	100%	97.4%
lms	97.4%	100%	94.7%
pegwit	100%	100%	81.6%
average	99%	98%	92%

We also investigated the prediction accuracy of FPLO for the benchmarks with multiple loop nests. Table 5.3 shows the choices made with our models and how the choices compare with actual performance as reported by simulation. For each optimization in the table, there are three columns. The first column, A, indicates on how many loop nests in a benchmark an optimization is applicable. The second column, M, indicates the number of loops for which our framework predicts a benefit to applying an optimization. The final column, S, indicates the number of loops in a benchmark in which an optimization should have been applied (i.e., it had an actual performance improvement). As an example, consider loop reversal for *vpenta*. On this benchmark, there are eight loops where reversal could be applied and FPLO predicted to apply it

in seven cases. The simulation results indicate that the optimization had a benefit on seven loops. In all cases in the table where there are mispredictions, our model selected the same set of loop nests for optimization as the simulation results, except for the one case where there was a misprediction. Although not shown in the table, our model also always made the correct choice for loop unrolling, fusion, and distribution.

Table 5.3: Prediction accuracy for multi-loop nest benchmarks

Benchmark	Interchange			Tiling			Reversal		
	A	M	S	A	M	S	A	M	S
adi	2	0	0	2	0	0	2	0	1
aps	1	1	1	1	1	1	3	1	1
eflux	5	5	5	5	1	1	6	2	3
tomcat	6	5	5	6	3	2	9	7	6
vpenta	3	3	3	3	2	2	8	7	7
bmcm	2	2	2	2	2	2	4	3	3

5.5.2 Comparing with Always-applying Approach

Always applying an applicable optimization can lead to a performance degradation in some cases. Such a simple heuristic of "always applying" is not sufficient in making decisions about when to apply an optimization. Figure 5.6 shows how always applying an optimization can lead to significant performance penalties. This figure shows the percentage change in performance (i.e., cycle count) when always applying an optimization versus not applying the optimization. Several benchmarks were run with varying trip counts to explore the effect of different configurations of a loop on whether to apply an optimization or not. For the benchmarks where the configuration was varied, only two trip counts are shown. One trip count comes directly from the benchmark, while the other is at a point that has significant conflict cache misses.

The figure demonstrates that across all benchmarks and optimizations that we considered, applying loop optimizations has significantly different performance impacts based on both a specific loop nest and the exact configuration of a loop nest. For example, loop interchange has a performance impact that varies from a 120% degradation to a 55% improvement. Also, for a specific configuration of a loop nest (i.e., different trip counts), the impact varies. In the case of

interchange for the *lgsi* benchmark, there is a 4% performance degradation for a trip count of 98 and a 8.3% performance improvement for a trip count of 128. Although the figure does not show loop unrolling, distribution, or fusion, we used our models to predict their impact. First, as expected, loop unrolling had no benefit on data cache locality. Of course, it had other non-cache related benefits such as reducing the total number of branch tests, improving the scheduling window and changing register pressure. Second, loop distribution had a 17.8% degradation when applied to *alv* with a trip count of 100 and a 1.2% improvement when applied to *alv* with a trip count of 128. Finally, on *tomcat3*, loop fusion had a very small benefit (0.8%) for a trip count of 100 and a 2.8% degradation for a trip count of 128. Optimizations may improve the performance for one trip count while degrade the performance for another. This trend for the single loop nest benchmarks is also true for the complex benchmarks with multiple loop nests. Here, interchange has a performance range from a 2.5% degradation to a 55% improvement. Tiling shows a similar trend, with the *aps* having a 26.2% performance improvement and *vpenta* having a 1% performance degradation.

As this figure shows, the strategy of always applying an applicable loop optimization is a dangerous one that may indeed lead to significant performance degradations. Of course, in some cases, this strategy works, but it is hard to know when it will work and when it will not. Instead of blindly applying an optimization, a more selective approach can be taken using FPLO. It can be used to predict when to apply an applicable optimization without actually applying it.

Using our profit-driven approach, the cases where performance is degraded can be avoided, which can have a significant effect. Figure 5.7 shows the performance improvement of selectively applying an optimization over always applying it. The improvement is relative to always applying the optimization and demonstrates the effect of selectivity. For the single nest benchmarks, a performance improvement implies that an optimization was not applied. For example, the benchmark *alv* with a trip count of 100, selectively deciding not to apply loop interchange has twice the performance of applying it. When performance is not improved both always applying and selectively applying an optimization had the same effect.

For interchange on the single nest benchmarks, optimization selectivity has a performance improvement of 0 to 120%. The large improvements in this case are due to the large degradations from always applying interchange (see Figure 5.6). Although loop tiling shows a slight improvement due to selectivity, it does not have as much an improvement as interchange

because the degradation from always applying the optimization is less. Reversal is similar to the tiling case. Distribution and fusion also showed improvements when applied with selectivity. With selectivity, unrolling was not applied since it does not have any benefit to cache performance. For all single nest benchmarks and optimizations considered, a selective approach using FPLO never results in a performance degradation over always applying an optimization. Indeed, the model captures the points at which an optimization is harmful as well as the points at which an optimization is helpful.

The rightmost bars in the figure show the effect of selectivity on benchmarks with multiple loop nests. In these cases, interchange with selectivity has a small performance improvement for *adi* and *tomcat*. A similar trend is true for loop reversal. However, in the case of loop reversal, two points (*eflux* and *adi*) are shown where our model mispredicts the benefit of applying an optimization and results in a small performance degradation over always applying reversal. The situation is different for tiling where selectivity has a significant difference. For *eflux*, tomcat, and *vpenta*, there is a performance improvement of 1.12.

While Figure 5.7 shows the advantage of selectively applying an optimization, it does not show the actual improvement in execution time due to selectivity. Figure 5.8 shows how cycle count is improved. For the single nest benchmarks, performance is improved by deciding not to apply an optimization when it would be harmful and by applying an optimization when it would help. In Figure 5.8, the cases with multiple loop nests are very compelling with selectivity resulting in a cycle count improvement over always applying an optimization for some cases. Consider the *tomcat* benchmark and the tiling optimization. Tiling results in a 15.5% improvement in cycle count by selectively applying the optimization to some loop nests and not to others within the same program. In comparison, always applying tiling achieved only a 5.4% improvement in cycle count.

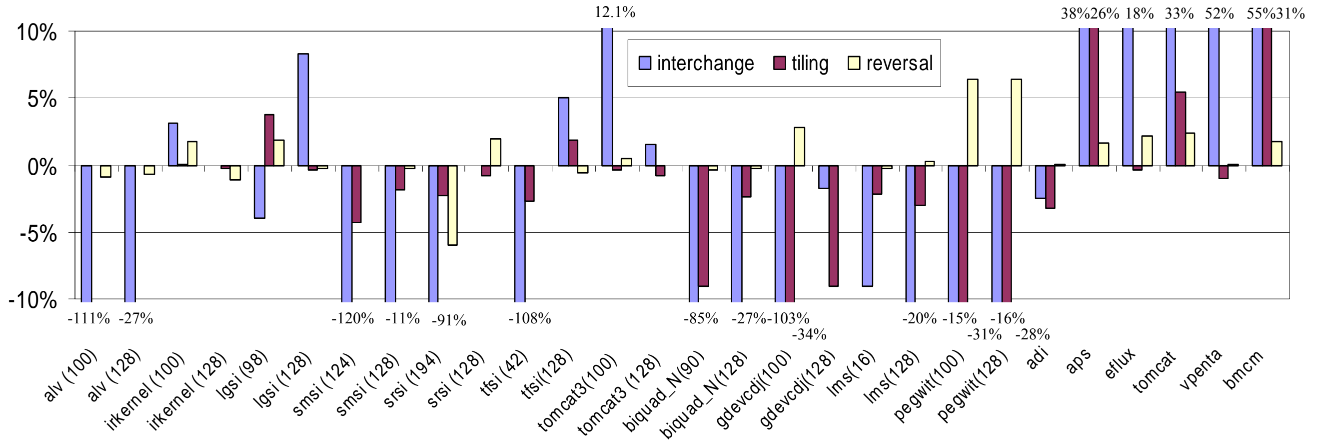


Figure 5.6: Performance impact of always-applying approach

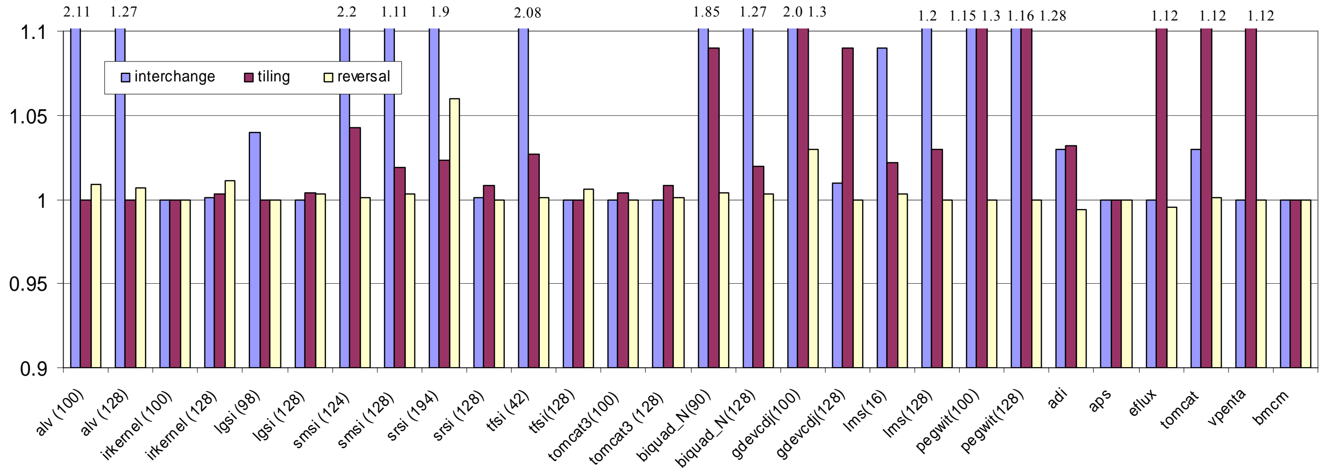


Figure 5.7: Improvement of profit-driven approach vs. always-applying

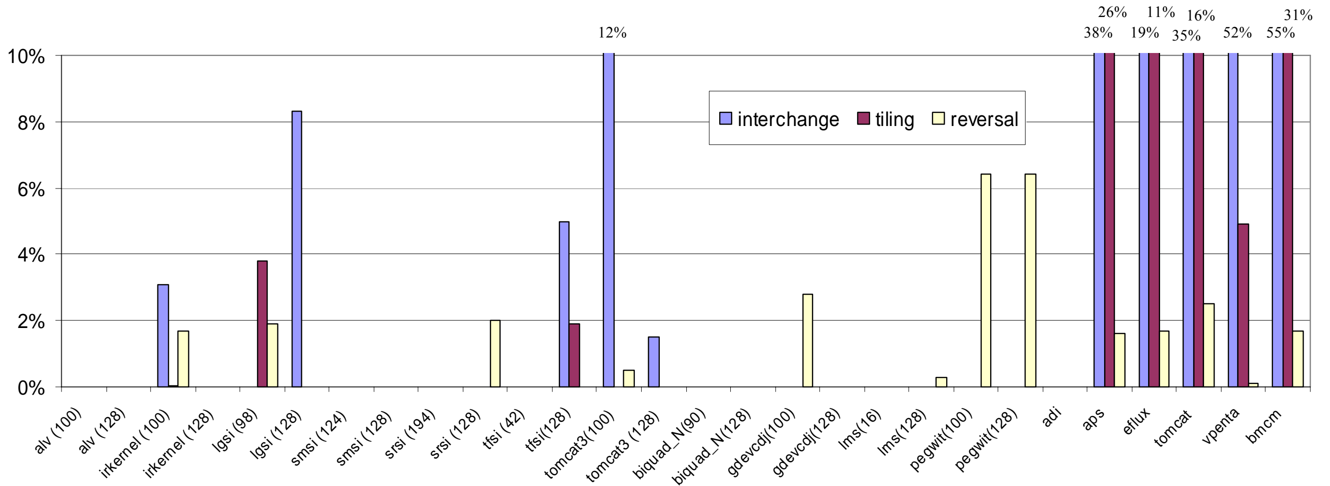


Figure 5.8: Performance impact of profit-driven approach

5.5.3 Choosing the Best Optimization

Not only can FPLO be used to decide whether an optimization should be applied or not, but it can also be used to select among several applicable optimizations. We can use FPLO to predict the profit of applying each optimization on a loop and then select the one with the maximum benefit, which is useful for constructing the good optimization sequences. Choosing the best optimization is particularly interesting in our single nest benchmarks when varying the trip count. Here, the trip count (the loop configuration) has a big impact on which optimization is the most beneficial. Figure 5.9 shows the accuracy and distribution of optimizations selected for each single nest benchmark with the trip count varied from 50 to 200. The figure shows the percentage of times that a particular optimization was chosen as the best one to apply. When all optimization models predicted a performance degradation (or no benefit), our model decided not to apply any optimization (the "not applying" case in the figure).

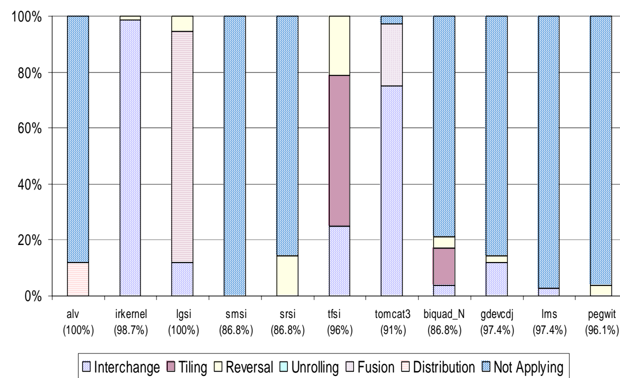


Figure 5.9: Accuracy and distribution of the most beneficial optimizations

For several of the benchmarks, only a couple of choices were made. For example, in *alv*, loop distribution was applied for 11% of the trip counts. For the other 89% of the trip counts, no optimization was applied. The benchmarks *tfsi* and *tomcat3* are interesting since they have three different choices. In *tfsi*, loop reversal, interchange, and tiling were applied, with tiling being applied the most often. For *tomcat3*, loop interchange was most often the best optimization, followed by fusion.

The figure also shows the accuracy of the choices made by our models (in parenthesis below each benchmark name). For most of the benchmarks, the accuracy was above 96%. For the others, such as *smsi* and *srsi*, the accuracy was lower due to mispredictions from our loop

reversal model. For example, in *smsi*, the model predicted no benefit to loop reversal, yet there was a very small actual benefit. Notice that from Table 5.2 we see that reversal had an accuracy of 86%, and as described earlier, the actual benefit was so small that our model did not capture it. Here, the performance improvement due to reversal was minimal.

5.5.4 Compile-time Overhead for Prediction

FPLO uses models to make predictions and thus the cost of predicting profitability needs to be evaluated. Thus, we need to evaluate the compile-time overhead, as done earlier. Table 5.4 shows for several loop benchmarks the compile time overhead (in milliseconds) of our tool. From the table, we see that the overhead depends on the loop configuration and the array references. For example, *irkernel* is a triple loop nest with five references and *srsi* is a double loop nest with 25 references. The compile-time overhead is high in these programs due to their complexity. On average, our compile-time for predicting is reasonable.

Table 5.4: Compile-time overhead for prediction (millisecond)

Benchmark	Interchange	Tiling	Reversal
alv (100)	24	29	23
irkernel (100)	2150	2637	2140
lgsi (98)	40	49	38
smsi (124)	118	137	117
srsi (194)	541	630	541
tfsi (42)	8	10	7
tomcat3 (100)	136	160	137
biquad_N (90)	30	36	29
gdevcdj (100)	11	15	11
lms (16)	1	1	1
pegwit (100)	7	10	6

In this section, we described FPLO for predicting the profitability of loop optimizations. Our experimental results demonstrate the predication accuracy and the usefulness of FPLO. On average, with our accurate cache model FPLO can make correct predictions for 97% of the time.

Using FPLO, compilers can selectively apply loop optimizations. Thus, the performance degradation cases in always-applying approach were avoided. FPLO can also be used to select the most beneficial optimization among a set of optimizations, which will be useful in constructing a good optimization sequence.

6.0 FIO: DETERMINING THE INTERACTION PROPERTY

In this chapter, we present the framework instance, FIO, for automatically determining the interactions among a set of optimizations, considering code context. The key idea of our technique is to model the pre- and post conditions of optimizations and code context, and determine how the pre-conditions of one optimization interact with the post conditions of another optimization. In our approach, there is no need to actually apply the optimization on the code or to recompute data and control flow information after each optimization.

To understand how FIO works, we present an overview of our approach in Figure 6.1. The figure shows the components of FIO and how FIO detects the interactions among a set of optimizations.

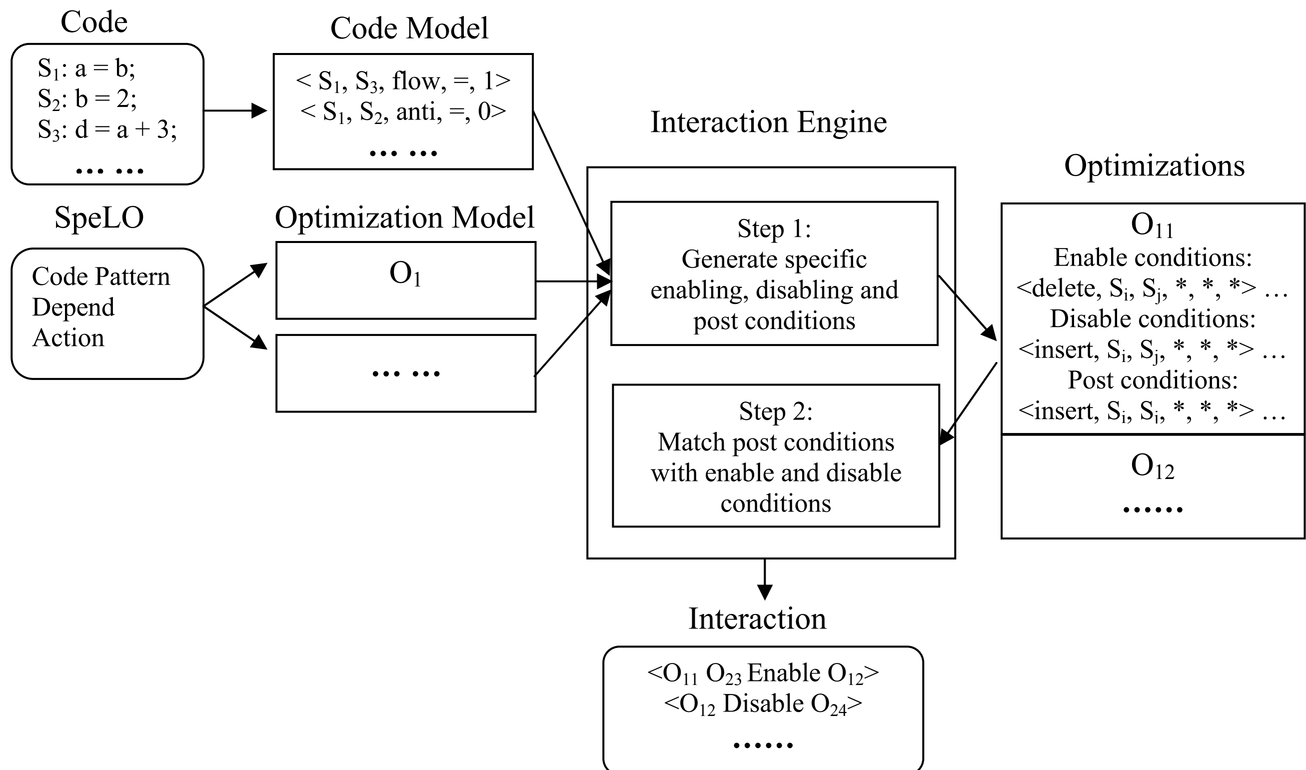


Figure 6.1: Overview of FIO

In Figure 6.1, the code model is extracted from the code and automatically generated by the optimizer. Its representation is the control flow graph with explicit data and control dependence information, which is needed to verify the pre-conditions of optimizations and determine the changes by the actions of optimizations. A specification language, SpeLO, is designed to express the conditions under which an optimization can be safely applied and the actions of the optimization. Compiler writers use SpeLO to develop models for optimizations. As part of FIO, there is an interaction engine, which uses models to determine the interaction property. In the first step, the interaction engine inputs the code and optimization models to generate the specific enabling, disabling and post conditions for each optimization at a program point. In the second step, these enabling and disabling conditions are matched with the post conditions of other optimizations to determine the enabling and disabling interactions. The output of our framework is the interactions among optimizations.

In the following sections, we describe the code model of FIO. We then present the specification language, SpeLO. A number of optimization models are described, followed by the interaction engine. We also describe how to use the interaction property to determine a code-specific optimization sequence. Finally, we show experimental results.

6.1 CODE MODEL FOR INTERACTION

The code model for interaction analysis represents the dependences for each statement in the code. We represent a dependence by $< Ss, Sd, type, dir, pos >$. There are four *types* of dependencies, including flow, anti-, output, and control dependencies [54]. A flow dependence is a dependence between statement S_s that defines a variable and statement S_d that uses the definition. An anti-dependence exists between statement S_s that uses a variable that is then defined in statement S_d. An output dependence defines a dependence between a statement S_s that defines (or writes) a variable that is later defined (or written) by S_d. A control dependence exists between a control statement S_s and all of the statements S_d under its control. The *dir* records the direction of the dependence, which can be forward, backward or equivalent, represented by <, >, or =, respectively. The direction is needed in optimizations for parallelization. The *pos* records

the position of dependence between S_s and S_d. Except for the dependences, we also need the control flow graph for the code model to verify the path related information.

6.2 A SPECIFICATION LANGUAGE

In prior work, a number of specification languages have been introduced to specify optimizations and formally analyze the properties of optimizations. Whitfield and Soffa [50] presented a specification language, Gospel, to specify a class of scalar and loop optimizations. Gospel has been used to automatically generate the implementation of optimizations and detect the interactions among optimizations. Lerner et al. [35] introduced a domain specific language, Cobalt, for automatically checking the correctness of optimizations. Lacey [33] introduced a specification language, TRANS, to prove the soundness of optimizations and detect the disabling interaction among optimizations. Both Cobalt and TRANS are based on temporal logic.

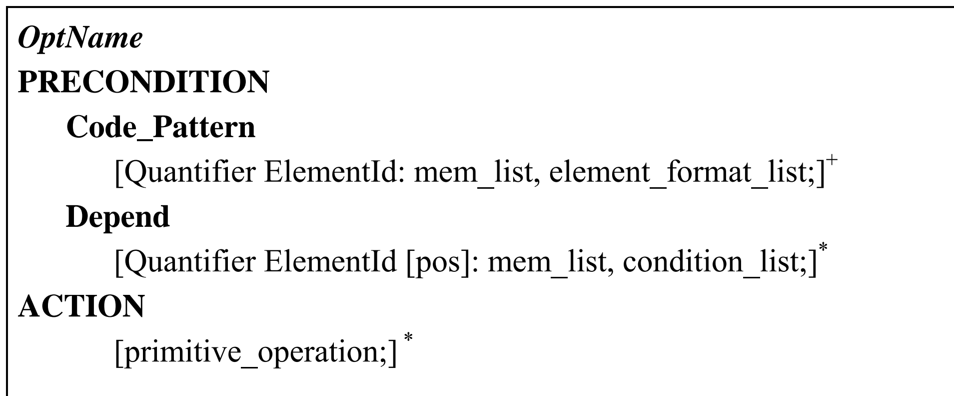


Figure 6.2: The format of SpeLO specification

We design a specification language, SpeLO, to specify optimizations for determining the interaction property. Our SpeLO language extends Gospel by introducing path related conditions, and thus, we can express path based optimizations such as PRE. The format of a SpeLO specification is shown in Figure 6.2. The PRECONDITION section contains the code pattern and dependence conditions needed before applying an optimization to maintain the semantics. The ACTION section consists of a series of primitive operations to perform an optimization. In SpeLO, an *elementId* starting with *S* represents a statement, an *L* represents a

loop, and a *B* represents a block. $B(S_i)$ represents the block of S_i. The general form of the code is three-address code with loop structure information. A basic three-address code statement has the form:

dst := $opnd_1$ opcode $opnd_2$

The names (e.g., $opnd_1$, or opcode) are used to specify the attributes of the operands or operator.

6.2.1 SpeLO PRECONDITION Section

Previous research demonstrated that dependence relationships can be used to efficiently determine the applicability of optimizations [24]. Thus, we use the code pattern and dependence conditions to specify the conditions under which an optimization is applicable. Our approach is the same as Gospel [50]. There are two parts in the pre-condition section.

Code_Pattern: This part identifies the code pattern of program elements such as a statement or loop. If the element is a statement, then the code pattern expresses the statement's operator and operands required. If the element is a loop, then the code pattern expresses the particular loop's header, trip count, etc. needed. The quantifier can be one of ANY, ALL or NO with the following meaning:

- **ANY** returns the set of matching elements and each element is considered separately;
- **ALL** returns the set of matching elements and all elements are considered together;
- **NO** returns a null set if there is no matching element.

The *mem_list* specifies a predefined set to which the element belongs, such as a path or a loop. Format expressions are used to give the specific format of the code element and can be combined in *element_format_list* using AND and OR. To help in generating the enabling and disabling conditions, SpeLO requires that all the combined expressions are in disjunctive normal format (DNF).

Depend: The second part of the PRECONDITION section specifies the necessary data and control dependence conditions for applying the optimization. The quantifier operators can be one of ANY, ALL or NO. The *condition_list* consists of the condition expressions combined by AND and OR operators in DNF. A condition expression can be a dependence condition in the form of $type(S_s, S_d, dir)$. As in code model, the dependence *type* can be flow, anti-, output or control dependence. The direction, *dir*, can be forward, backward, equivalent or any. A condition

expression can also be a predefined condition, such as *in_any_path*(S_i, S_j, S_k), which means a statement S_i should appear in a path from S_j to S_k, and *in_every_path*(S_i, S_j, S_k), which means a set of statements S_i should appear in every path from S_j to S_k. A position tag can also be specified in a dependence rule to show whether the position of the dependence should be checked or not.

6.2.2 SpeLO ACTION Section

The ACTION section describes the modifications on code or code properties (e.g., value number) of applying optimizations. We decompose these effects on code into four primitive operations (move, add, delete and modify). The semantics of the primitive operations are shown in Table 6.1, which are similar to Gospel [50]. The effect on code properties can be assigning a new value to the property, hash the properties, etc. There can also be some conditions associated with the actions. In Section 6.3.3, we describe the optimization model for global value numbering, whose ACTION section expresses the modification on the value number of the statements and has conditions associated with the actions.

Table 6.1: Semitics of primitive operations

Operation	Parameter	Semantics
Move	(Obj, After_Obj)	move Obj to the place after After_Obj
Add	(Obj_Description, After_Obj)	add an Obj with Obj_Description after After_Obj
Delete	(Obj)	delete Obj
Modify	(Obj, Obj_Description)	modify Obj with the Obj_Description

6.3 OPTIMIZATION MODELS

Optimization models for interaction analysis express the conditions under which an optimization can be safely applied and the actions of the optimization. We describe the optimization model for dead code elimination, partial redundancy elimination and value numbering, using SpeLO.

6.3.1 Dead Code Elimination

Figure 6.3 presents a SpeLO specification for dead code elimination (DCE). Because DCE requires no path specific conditions, the optimization model of DCE is the same as what is in Gospel [50]. To facilitate the discussion of the example (shown in Section 6.5), we describe DCE optimization model. The specification uses two variables S_i and S_j whose values are statements. The Code_Pattern section specifies the code pattern consisting of **any** statement, which is a copy statement or binary expression operation (i.e., +, -, *, /). S_i will have its value as such a statement if it exists. The Depend section ensures that there is **no** statement that is flow dependent on S_i.

If an S_i is found that meets the code pattern, and no S_j is found that meets the specified requirements, then the operation expressed in the ACTION section is performed. The action is to delete the statement S_i.

```
DCE
PRECONDITION
    Code_Pattern
1:      ANY Si: Si.opcode = copy OR Si.opcode = binary_exp;
    Depend
2:      NO Sj: flow_dep(Si, Sj, any);
ACTION
3:      Delete (Si);
```

Figure 6.3: DCE optimization model

6.3.2 Partial Redundancy Elimination

Figure 6.4 presents the optimization model of partial redundancy elimination (PRE). The first line in Figure 6.4 shows when we find that a statement S_i is a binary expression operation, there is a possible PRE opportunity. We need to find all the same expressions S_j, executed on a path to S_i without a redefinition between them (lines 2 and 3). We also find some definitions S_p of this statement where there is a path that does not include the same expressions found (line 4). In this specification, common subexpression elimination is a separate optimization from PRE. We save

the immediate predecessors of the statement on the path that does not include the same expression, which will be the places to insert the computation. At the same time, we must make sure that at these insertion places, the expression is anticipated (i.e., the block of statement S_i post-dominates the insertion place), as shown in line 5 of Figure 6.4.

When applying PRE, we insert the computation at the insertion places and before the same expressions S_j and replace the same expressions S_j and the statement S_i with the assignment (lines 6 to 9 in Figure 6.4).

```
PRE
PRECONDITION
    Code_Pattern
1:      ANY Si: Si.opcode = binary_exp;
2:      ALL Sj:  mem(path(Entry, Si)), Sj.opcode = Si.opcode AND
                    Sj.opnd1 = Si.opnd1 AND Sj.opnd2 = Si.opnd2;
    Depend
3:      NO Sk: anti_dep(Sj, Sk, =) AND flow_dep(Sk, Si, =);
4:      ALL Sp: flow_dep(Sp, Si, =) AND
            ¬in_every_path(Sj, Sp, Si, save pred(Si)  ∧ ¬ in_any_path(pred(Si), Sj, Si) to Bq)
5:          NO Bl: mem(Bq), ¬post_dom(B(Si), Bl);
ACTION
6:      Add ((new_temp= Si.opnd1 Si.opcode Si.opnd2), Bq);
7:      Add (new_temp=Si.opnd1 Si.opcode Si.opnd2), Sj);
8:      Modify (Sj, (Sj.dst = new_temp));
9:      Modify (Si, (Si.dst = new_temp));
```

Figure 6.4: PRE optimization model

6.3.3 Value Numbering

Figure 6.5 presents the optimization model of global value numbering (VN), which operates on SSA code [8]. We separate the optimization into two passes. First, we assign a value number to each assignment statement. Second, we remove the redundancy based on the value number. The

first pass is a preparation for the pass that uses the code property (i.e., value number). Thus, it is always performed in the beginning and not involved in selecting a good order for optimizations.

```
VN
Pass 1: Assigning a value number
PRECONDITION
    Code_Pattern
1:      ANY Si: Si.opcode = Ø OR Si.opcode = assign
    Depend
2:      ALL Sj: flow_dep (Sj, Si)
ACTION
        // useless Ø-operation
3:      IF ((Si.opcode = Ø) AND (equal (Sj.VN)))
4:              Si.VN = Sj.VN;
        // redundant Ø-operation or assign
5:      ELSE IF (hash (Sj.VN, Si.opcode) != NULL)
6:              Si.VN = hash (Sj.VN, Si.opcode);
7:      ELSE
8:              hash (Sj.VN, Si.opcode, Si.VN);

Pass 2: Redundancy elimination
PRECONDITION
    Code_Pattern
9:      ANY Si: Si.opcode = binary_exp
    Depend
10:     ALL Sj: Sj.VN = Si.VN
ACTION
11:     Delete (Sj);
```

Figure 6.5: VN optimization model

In the first pass, the specification uses two variables S_i and S_j whose values are statements. The Code_Pattern section specifies the code pattern consisting of **any** statement, which is an assignment or Ø-operation (as shown in line 1 of Figure 6.5). The Depend section finds **all** the statements that S_i is flow dependent on. The ACTION section specifies the modification on the code property, value number, which is initialized to the destination operator of the statement. Associated with the actions, there are conditions. For example, if the value numbers for all S_j are the same and S_i is an Ø-operation, S_i is a useless Ø-operation, as shown in

line 3 of Figure 6.5. Then, we assign the value number of S_i to be S_j's value number. If there is an item that has the same operation and operators as S_i in the hash table, S_i is a redundant computation and assigned the hashed value as its value number. Otherwise, we insert an item into the hash table. In the second pass, the redundancy is eliminated based on the value number.

6.3.4 Other Optimizations

We also develop optimization models for CPP, LICM, CTP, branch chaining (BRC), branch elimination (BRE), loop interchange (LPI), and loop fusion (LPF). Their optimization models are shown in Appendix A.

6.4 INTERACTION ENGINE

The interaction engine of FIO inputs the code model and optimization models and determines the enabling and disabling interactions among optimizations. Here, we focus on the interactions among scalar optimizations; our technique also works for loop optimizations. The algorithm to detect the enabling and disabling interactions among scalar optimizations is shown in Figure 6.6. The algorithm for detecting the interactions of loop optimizations is similar, but the element checked is a loop instead of a statement.

Lines 1 and 2 in Figure 6.6 show the data structures used in the algorithm. *SetTable* is used to store the set of objects, *ObjSet*, which matches the *element_format_list* or *condition_list* for each rule in the optimization specification. *OptTable* stores the information about each optimization opportunity. Each element in *OptTable* includes an identifier, optimization type, whether the optimization is applicable or not, the list of enabling conditions, the list of disabling conditions and the list of post conditions.

As shown in Figure 6.6, the interaction engine uses two steps to detect the interactions among a set of optimizations. In the first step, from line 3 to line 13, the interaction engine executes a loop over every statement in the code model and every optimization specification and

generates the enabling, disabling and post conditions for each possible optimization opportunity. The optimization opportunities are identified by looking for the code pattern and dependence relations in the code model. In the second step, from line 14 to line 17, the interaction engine matches the enabling and disabling conditions of an optimization with the post conditions of other optimizations to compute the enabling and disabling interactions. The interaction engine outputs a list of interaction relations, represented by "$<O_1 \ldots O_n>$ Enable/Disable O_j". The next two sections describe in detail the algorithm for the interaction engine.

```
Data Structure
     // SetTable records ObjSet found that matches the condition
1:  SetTable: structure (Quantifier, ElementId, ObjSet)
     // OptTable records the optimization opportunities
2:  OptTable: structure (OptId, OptType, Applicable, Enable, Disable, Post)
Algorithm
     //Step1: generating specific conditions
3:       foreach statement S in the code model {
4:           foreach optimization O under consider {
5:               if (check_code_pattern (S, O) = = match | possible) {
6:                   check_depend (S, O);
7:                   foreach related opt in OptTable {
8:                       if (opt.enable is empty)
9:                           opt.applicable = true;
10:                      else
11:                          opt.applicable = false;
12:                  generate_postcondition(S, O);
13:      } } } }
     //Step2: matching the conditions
14:      foreach opt in the OptTable {
15:          postcondition_match(opt.enable);
16:          postcondition_match(opt.disable);
17:      }
```

Figure 6.6: The overview algorithm for the interaction engine

6.4.1 Generating Specific Conditions

For each optimization and program point, the interaction engine checks the conditions described in the PRECONDITION section and generates the specific enabling and disabling conditions. Because there are two parts, Code_Pattern and Depend, in the PRECONDITION section, we have two functions, check_code_pattern and check_depend (as shown in line 5 and line 6 of Figure 6.6).

Table 6.2 shows how to generate enabling and disabling conditions for checking conditions described in Code_Pattern (i.e., function check_code_pattern). Each row in the table shows a different case when checking the conditions. There are three columns for each row. The first column shows a case. The second and the third columns show the enabling and disabling conditions generated for the case. The condition expressions in Code_Pattern are combined by AND and OR operators in DNF. Without loss of generality, we represent a condition expression as (A AND B OR C) in our discussion.

Table 6.2: Generating enabling and disabling conditions for check_code_pattern

Case	Enabling conditions	Disabling conditions
Match	True	(delete S) $\vee$ ($\neg$A $\wedge$ $\neg$ C) $\vee$ ($\neg$B $\wedge$ $\neg$ C)
Possible match	A if A not mach	(delete S) $\vee$ ($\neg$A $\wedge$ $\neg$ C) $\vee$ ($\neg$B $\wedge$ $\neg$ C)

As shown in Table 6.2, there are two cases when comparing a statement with the conditions specified in Code_Pattern:

Case 1: The statement matches the conditions specified in Code_Pattern. The interaction engine stores the statement in *SetTable* by calling SetTable_insert with (Quantifier = ANY, ElementId = ElementId in the rule, ObjSet = {StatId}). It also creates an optimization opportunity in OptTable by calling OptTable_insert with (OptId = cur_opt ++, OptType = O). It then generates enabling and disabling conditions. As shown in second row of Table 6.2, the

enabling condition is true. The disabling conditions include a condition to delete the statement and the conditions to modify the operands or operation to *not* match *element_format_list*. The table shows a general form of the disabling conditions. "$\neg A \wedge \neg C$" means modifying the statement to not match the conditions A and C.

Case 2: The statement can be modified (by other optimizations) to match the conditions specified in Code_Pattern. It is possible that the statement can be modified by other optimization code changes, making this optimization applicable. For example, constant folding requires that both operands are constant. But if the statement has a variable operand, it is still possible to perform constant folding on this statement if the operand can be changed to a constant by constant propagation. In the case that the operands or the operation can be changed by another optimization to match the code pattern, the interaction engine stores the statement in the *SetTable* and creates an optimization opportunity in *OptTable*. Here it generates both disabling and enabling conditions, as shown the third row of Table 6.2. The disabling conditions are the same as case 1. The enabling conditions are the conditions in which the code model does not match with the code pattern. When it is impossible that any code change by another optimization matches the code pattern, the interaction engine does not create an optimization opportunity.

The quantifier (ANY or ALL) specified in the code pattern does not change the generation of the enabling and disabling conditions. When the quantifier is ANY, the generator will create an optimization opportunity for each statement that matches or possibly matches with the code pattern.

After checking the conditions specified in Code_Pattern, the interaction engine needs to check the conditions given in the Depend section, as shown in Table 6.3. The table shows the enabling and disabling conditions generated for different cases. Each row represents a case for matching the conditions with code context. There are four columns. The first column shows the quantifier of the conditions. The second column indicates whether the matching objects can be found or not. The third and the fourth columns give the enabling and disabling conditions generated. We still use (A AND B OR C) to represent a general condition in our discussion.

Table 6.3: Generating enabling and disabling conditions for check_depend

Quantifier	Match	Enabling Conditions	Disabling Conditions
ALL	Yes	True	(delete obj_1) ^ ... ^ (delete obj_n) ∨ (insert A AND B)* ∨ (insert C)*
ALL	No	(insert A AND B)* ∨ (insert C)*	None
ANY	Yes	True	(delete obj_i) ∨ (insert A AND B)* ∨ (insert C)* for every element in OptTable
ANY	No	(insert A AND B)* ∨ (insert C)*	None
NO	Yes	(delete obj_1) ^ ... ^ (delete obj_n) ∨ (delete dep_i) if dep_i not match	(insert A AND B) ∨ (insert C)
NO	No	True	(insert A AND B) ∨ (insert C)

The second row in Table 6.3 shows the first case, where the quantifier of this condition is **ALL** and there **are** objects that match the condition. Because this is a match case, the enabling condition is true. The disabling conditions generated show that if deleting all of these matching objects, the application of this optimization will be destroyed. The disabling conditions also include inserting a dependence that matches the conditions, (insert A AND B)* or (insert C)*. The stars on these disabling conditions show that the dependencies need to be updated before the interaction engine can determine whether other optimizations disable this optimization because of this condition. In most cases, we do not need to update the code model. However, there are two cases when it is needed. In one case, a statement is inserted by an optimization. We need to temporarily update the dependencies (i.e., the code model). For example, considering the following code:

S_1: a = b;

S_2: d = a + 3;

…

S_6: c = a + 6; ← newly inserted

Suppose O_1 is an optimization opportunity of copy propagation applied to statement S_1. It is applicable for this code segment. Another optimization O_i inserts a statement S_6 that is flow dependent on S_1, which will match the disabling condition of O_1. However, it is unknown whether S_6 has other definitions or there is redefinition of *b* between S_1 and S_6. So it cannot be decided whether O_1 is applicable after inserting this dependence. Thus, the code model needs to be updated to determine the interactions. The other case that needs to update the code model is when the interaction engine considers a combination of optimizations, which will be discussed in Section 6.4.2.

The third row in Table 6.3 shows the case where the quantifier of this condition is **ALL** and there is **no** matching object. The interaction engine needs to generate the enabling conditions, showing that dependencies can be inserted to match the condition A AND B or match the Condition C.

Other cases are similar. One major difference is that when the quantifier of the condition is ANY and there are matching objects, it needs to generate an optimization opportunity in *OptTable* for each object and store *OptId* into *ObjSet*. The reason is that the objects defined by ANY quantifier should be considered separately.

The enabling condition is combined with the other enabling conditions generated for the previous rules by the AND operator, while the disabling condition is combined by the OR operator. Finally we standardize all the enabling and disabling conditions to DNF in order to match them with the post conditions.

When an enabling condition is deleting dependence, the generator needs to follow the output dependences to generate **all** enabling conditions. For example, consider the following code:

S_1: a = b;

S_2: b = 2;

S_3: b = 3;

S_4: d = a + 3;

In order to perform copy propagation at S_1, it needs to delete the anti-dependence between S_1 and S_2 and the output dependence between S_2 and S_3 as well.

After checking conditions specified in the PRECONDITION section, the interaction engine needs to generate the specific post conditions for an optimization opportunity, as shown

in line 12 of Figure 6.6. The primitive operations in the ACTION section specify the code modifications of the optimizations. The interaction engine decomposes them when generating the specific post conditions. It also generates the post conditions that describe the changes on dependences after applying the optimization.

Table 6.4 shows how to generate specific post conditions for each primitive operation in a SpeLO ACTION section. Each row in the table represents a primitive operation. There are two columns for each primitive operation. The first one gives the code modifications. The second one gives the modification on the dependence.

For example, the move operation can be decomposed to delete the object at its original place and insert the new object at a new place. Deleting an object needs to delete all its dependences. Inserting an object will insert the dependences that relate to the new object at the new place.

Table 6.4: Generating post conditions for primitive operations

Operation	Code Modifications	Dependence Modifications
Move	delete (Obj) insert (NewObj, After_obj)	delete_dep (any_type, any_stat, Obj, any_dir) insert_dep (any_type, any_stat, NewObj, any_dir) insert_dep (any_type, NewObj, any_stat, any_dir)
Add	insert(Obj, After_obj)	insert_dep (any_type, Obj, any_stat, any_dir) insert_dep (any_type, any_stat, Obj, any_dir)
Delete	delete (Obj)	delete_dep (any_type, any_stat, Obj, any_dir)
Modify	modify_opnd(Obj,opnd, new_opnd)	delete_dep (any_type, any_stat, Obj, any_dir) where dep_position = opnd insert_dep (any_type, any_stat, Obj, any_dir) where dep_position = new_opnd insert_dep (any_type, Obj, any_stat, any_dir) where dep_position = new_opnd
	modify_opcode(Obj, new_opcode)	--

6.4.2 Matching Conditions

In the first step, the interaction engine generates an optimization table, *OptTable*, which has all the possible optimization opportunities (including their disabling, enabling and post conditions).

In this step, the interaction engine determines the interactions among these optimizations by matching the enabling and disabling conditions of each optimization with the post conditions of other optimizations in *OptTable*.

The algorithm for matching O_i's enabling and disabling (E/D) conditions with the post conditions is shown in Figure 6.7. Because the E/D conditions are already in DNF, we represent them using the general form (A AND B OR C).

```
// Suppose the general form of E/D conditions of Oi is (A AND B OR C)
1:      foreach optimization Oj (Oj ≠ Oi)  {
2:              foreach postcondition Pj of Oj {
                        // match Pj with the condition A
3:                      if (Pj match A) {
4:                              if ((A has a star) && (update_match(Oj, Oi))
5:                                      Oj → {match1};
6:                              elseif (A has no a star)
7:                                      Oj → {match1};
8:                      }
                        // same for conditions B and C to get {match2} and {match3}
9:      } }
10:     foreach Oa in {match1} {
11:             foreach Ob in {match2} {
12:                     { Oa + Ob } →(E/D) Oi;
13:     }}
14:     foreach Oc in{match3} { { Oc } → (E/D) Oi; }
```

Figure 6.7: Matching O_i's E/D conditions with post conditions

As Figure 6.7 shows, the interaction engine checks each post condition of other optimizations. It finds all optimizations whose post conditions match the condition A, B, or C. Then the set of optimizations whose post conditions match conditions A and B enable/disable O_i together. The optimization whose post conditions match condition C enables/disables O_i. Matching the post condition with condition A (or others) is straightforward. The condition action (i.e., delete, insert, delete_dep, insert_dep, modify_opnd, or modify_opcode) and the object (e.g., statement, or dependence) are compared. For example, if A is <delete S_3>, the post condition that deletes S_3 matches with A. If A is <delete_dep, type, S_i, S_j, dir, other_condition>, the post

condition that deletes the same type of dependence between S_i and S_j with the same direction as well as meets the other condition can match A. The other condition specifies other requirements for this dependence, such as the statement should be in a path.

In Section 6.4.1, we discussed that in one case when an optimization's enabling or disabling condition (a *star* condition) cannot fully determine the enabling and disabling interactions, the interaction engine needs to update the code model to determine their interactions, by calling *update_match*. Another case is when the interaction engine considers the interactions of a combination of optimizations with other optimizations, in which it also needs to update the control flow and dependencies. In these cases, the interaction engine creates a new element in *OptTable* which represents the combination of O_2O_1. It then applies O_2's post conditions to temporarily update the code model and checks the conditions specified in O_1's dependence section under the modified code model. The interaction engine generates the enabling and disabling conditions for this combination. According to whether O_1 is applicable after O_2 is applied, the interaction engine determines whether O_2 enables or disables O_1. The engine considers how the enabling and disabling conditions of this combination interact with the post conditions of other optimizations to determine the interactions of a combination of optimization with other optimizations.

6.5 AN EXAMPLE OF DETERMINING THE INTERACTION

In this section, we use an example to show how FIO automatically determines the enabling and disabling interactions for two optimizations, dead code elimination (DCE) and copy propagation (CPP). The code is shown in Figure 6.8(a). The optimizer generates the code model, as shown in Figure 6.8(b). The code model describes the dependences in the code. Each dependence is expressed as <S_i, S_j, type, dir, pos>. For example, there is a flow dependence between S_1 and S_2. It has equal direction. The dependence exists on the first operand. Thus, this dependence can be represented as <S_1, S_2, flow, =, 1>. The optimizer inputs the code model and optimization model for DCE and CPP into the interaction engine, which determines the interactions among these two optimizations.

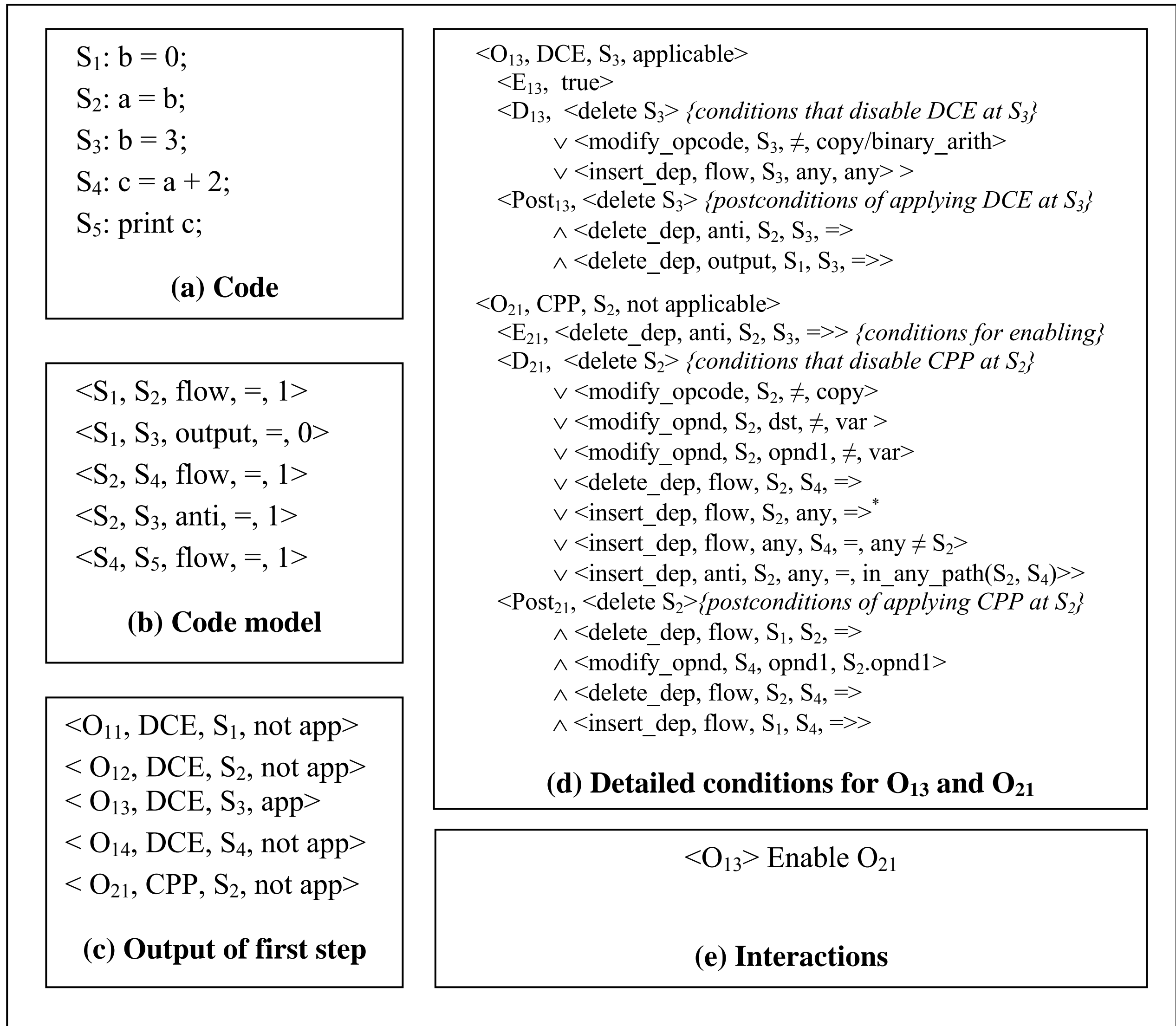


Figure 6.8: An example of determining the interaction

When the engine is triggered, it first generates the specific enabling, disabling and post conditions for every possible optimization opportunity in the code. Figure 6.8(c) shows all the possible optimizations, generated by the engine. Here, we only show the details of the conditions for two optimizations, O_{13} and O_{21} in Figure 6.8(d). O_{13} is a dead code elimination that operates on S_3. O_{13} is applicable for this code segment. Thus, the enabling condition for O_{13} is true. There are three disabling conditions for destroying the application of O_{13}. The first one is deleting S_3. The second one is modifying its operation. The third one is inserting a flow dependence that has S_3 as the source. The post conditions for O_{13} show how it changes the code model, which

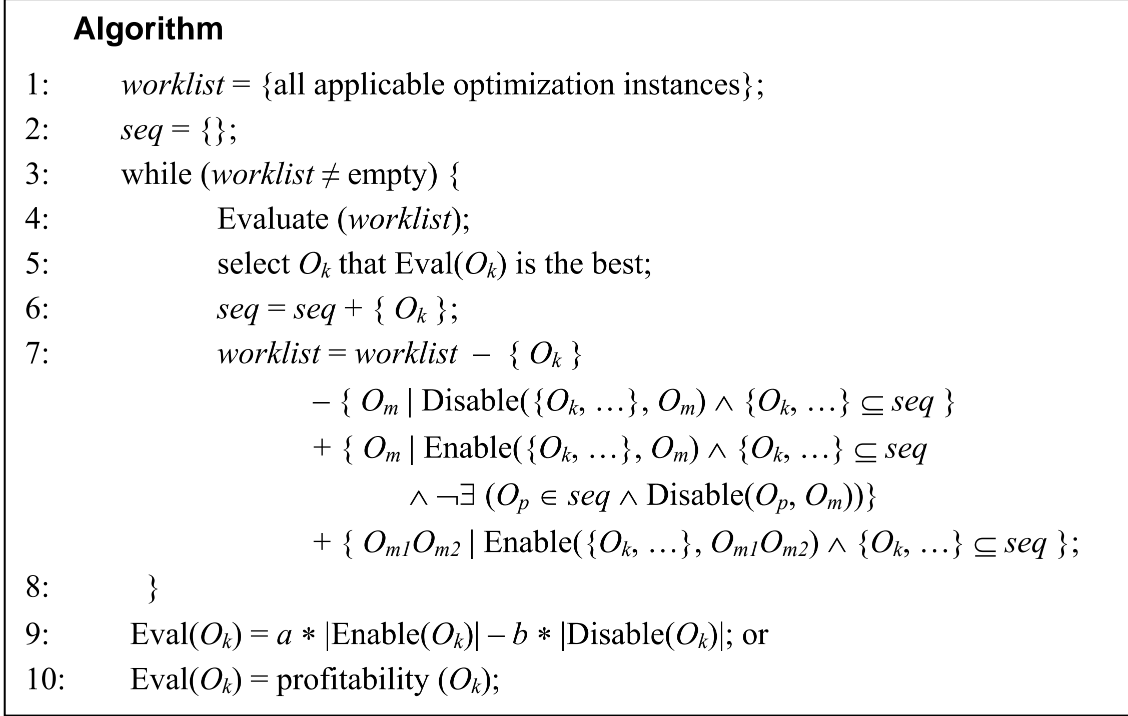
includes deleting S_3, deleting the anti-dependence between S_2 and S_3 and deleting the output dependence between S_1 and S_3. Similarly, the enabling, disabling and post conditions are generated for O_{21} according to the CPP specification.

In the second step, the interaction engine compares the enabling and disabling conditions with the post conditions of other optimizations and determines the interactions. For example, the engine needs to determine the enabling interaction for O_{21}. There is only one condition needed for O_{21} to be applicable, i.e., <delete_dep, anti, S_2, S_3, =>. When the interaction engine checks each condition in O_{13}'s post conditions, it finds that O_{13} changes the dependency by deleting the anti-dependence between S_2 and S_3. This condition matches with the enabling condition of O_{21}. Thus, O_{13} enables O_{21}, shown in Figure 6.8(e).

6.6 USING INTERACTION TO ORDER OPTIMIZATIONS

FIO can be used to determine a good order to apply a set of optimizations. Instead of blindly searching the optimization space, we can determine what optimizations are legal after applying an optimization based on the interaction property. We design an algorithm to construct a code-specific optimization sequence using the interaction. Our algorithm is shown in Figure 6.9.

In the algorithm, *worklist* is initialized to the applicable optimizations and *seq* is initialized to the empty sequence. We evaluate every optimization in *worklist* by some evaluation function, *Eval(O)*. Then we select O_k with the largest *Eval* value as the next optimization in the sequence. As shown in line 7 of Figure 6.9, we modify *worklist* according to what optimizations are disabled by this optimization O_k, and what optimizations are enabled by O_k. We require that when O_k along with other optimizations together disable O_m and only if all the other optimizations are already in the sequence, then we can remove O_m from *worklist*. For the enabling, we also require that optimizations already in *seq* do not disable O_m, and then we can add O_m to *worklist*. As discussed in Section 6.4.2, we also consider the interactions between the individual optimization and the combination of two optimizations. Thus we add the combination of two optimizations that are enabled by this optimization. We evaluate *worklist* until it becomes empty. And then we achieve the best sequence that maximizes the evaluation function.



Algorithm

1: $worklist$ = {all applicable optimization instances};
2: $seq = \{\}$;
3: while ($worklist \neq$ empty) {
4: Evaluate ($worklist$);
5: select O_k that Eval(O_k) is the best;
6: $seq = seq + \{ O_k \}$;
7: $worklist = worklist - \{ O_k \}$
$- \{ O_m \mid \text{Disable}(\{O_k, \ldots\}, O_m) \wedge \{O_k, \ldots\} \subseteq seq \}$
$+ \{ O_m \mid \text{Enable}(\{O_k, \ldots\}, O_m) \wedge \{O_k, \ldots\} \subseteq seq$
$\wedge \neg\exists (O_p \in seq \wedge \text{Disable}(O_p, O_m))\}$
$+ \{ O_{m1}O_{m2} \mid \text{Enable}(\{O_k, \ldots\}, O_{m1}O_{m2}) \wedge \{O_k, \ldots\} \subseteq seq \}$;
8: }
9: $\text{Eval}(O_k) = a * |\text{Enable}(O_k)| - b * |\text{Disable}(O_k)|$; or
10: $\text{Eval}(O_k) = \text{profitability}(O_k)$;

Figure 6.9: Determining a good optimization sequence using interaction

The evaluation function, *Eval(O)* can be to maximize the number of optimizations in the sequence. We use the weighted number of optimizations enabled and disabled by the optimization (line 9 of Figure 6.9). We can also use profitability as the evaluation function (line 10 of Figure 6.9), which combines profitability and the interaction property to search for code-specific optimization sequence.

There are some other possible search algorithms that can use interactions in finding code-specific optimization sequences. The more complicated the search algorithm, the longer search time it may take. Our experimental results show that this simple constructive algorithm achieves good optimization sequences.

6.7 EXPERIMENTAL RESULTS

To evaluate FIO, we compare three approaches to apply optimizations: a fixed-order approach, an empirical approach that uses a genetic algorithm to search for effective optimization sequences [1] and our model-driven approach. We performed two sets of experiments. One used

the number of optimizations applied as the evaluation function in the search. The other used profitability as the evaluation function. We ran our experiments on an Intel Pentium IV 2.4GHz machine, with 512MB of memory running RedHat Linux.

There are eight optimizations in our experiments, including CPP, CTP, DCE, PRE, LICM, VN, branch chaining (BRC) and branch elimination (BRE). The fixed-order sequence that we used is "VN, BRC, BRE, CPP, CTP, DCE, PRE, LICM, VN, BRC, BRE, CPP, CTP, DCE, PRE, LICM". The selection of the fixed order was based on the study of interactions among these optimizations [50]. The empirical approach used in the experiments has the following configuration. We performed a search for each *function* of a program using 10 generations. Each generation had a population of 20 sequences. Every sequence had 16 optimization passes, choosing from eight optimizations. At each generation, the best 10% of the sequences survive without any change. The rest of the new generation is created by the crossover operation, followed by the character-by-character mutation with the mutation rate is 5%. This configuration is the same as the experiments in Section 4.6.2, only here search is for each function.

6.7.1 Evaluation Function: the Number of Optimizations

In the first set of our experiments, the evaluation function is the number of optimizations applied. In the empirical approach, the optimizations in a sequence are performed on the code. Then, the number of optimizations applied is measured to evaluate the sequence. In our model-driven approach, we construct a code-specific optimization sequence as described in Section 6.6. The evaluation function is the number of optimizations enabled and disabled by an optimization. We compare three approaches (the fixed-order approach, the empirical approach and our model-driven approach) in terms of compile-time overhead and performance improvement. We also show the memory requirements for our approach.

6.7.1.1 Compile-time overhead

For each sequence, the genetic algorithm determines the interactions by applying the optimizations and recomputing the data flow needed for other optimizations. In our approach, the interaction engine is used to determine the optimization property and thus the good sequences.

By determining the interaction property, the time to find a good order is greatly reduced. The compile-time comparison among the fixed-order, the empirical (i.e., GA approach) and our model-driven approaches is shown in Table 6.5.

From the table, the compile-time for the fixed-order approach is small. It varies from 0.05 minutes to 2.34 minutes. The compile-time for the GA approach varies from 3 minutes to 5.5 hours, while the compile-time for our model-driven approach is from 0.4 to 65 minutes. In the GA approach, each function is compiled for 200 sequences and evaluated by the number of optimizations applied. The compile-time for the GA approach is related to the average compile-time for each function. For example, there are 106 functions in *gzip* and the average compile-time for a function is about 0.8 seconds. Adding the GA search time, it took 327 minutes for the GA approach to find code-specific sequences for *gzip*. In our approach, we use FIO to identify all the possible optimization opportunities in a function and determine their interactions. Then we use these interactions to find a good order to apply these optimization instances. The compile-time of our approach depends on the time for the interaction engine to determine the interactions and the search time using the interactions. As the number of the optimization opportunities in each function increases, the compile-time of our approach increases. For example, on average, there are about 957 optimization opportunities for a function in *mpeg* while about 423 in *gzip*. Thus, the average time for the interaction engine to determine the optimization interactions for mpeg is 20 seconds while it takes 10 seconds for *gzip*. This is why the interaction engine took more time for *mpeg* than for *gzip* in our approach.

Table 6.5: Compile-time overhead of three approaches (minutes)

Benchmarks	Fixed-order	Empirical	Model-driven
adpcm.rawcaudio	0.05	3.01	0.89
mpeg2.enc	1.92	308.96	65.25
bitcount	0.15	16.84	1.03
dijkstra.large	0.05	8.21	0.36
FFT	0.11	10.13	1.02
gzip	1.52	327.60	35.33
mcf	0.53	41.7	4.02
bzip2	2.34	250.35	29.67

6.7.1.2 Performance improvement

Besides compile-time, we also compare performance of three approaches. Next, we first show the comparison on the number of optimizations applied and then the run-time performance using dynamic instruction counts.

In Table 6.6, the number of optimizations applied is shown for the fixed-order, empirical and model-driven approaches. For example, for *adpcm*, using the fixed order sequence, 146 optimizations are applied. Using the sequences found by the GA approach, 155 optimizations are applied. While using the sequences found in our approach, 155 optimizations are applied. On average, the number of optimizations applied in our approach is 2.7% less than the empirical approach.

Table 6.6: Comparing the number of optimization applied

Benchmarks	Fixed	Empirical	Model-based
adpcm.rawcaudio	146	155	155
mpeg2.enc	10009	11686	11031
bitcount	302	335	326
dijkstra.large	113	154	148
FFT	251	291	283
gzip	5138	5589	5493
mcf	2020	2280	2218
bzip2	3509	3883	3802

In Figure 6.10, we compare the performance improvement of three approaches over the unoptimized code (only register allocation is applied). In the figure, the performance improvement is measured using dynamic instruction count. The empirical approach and model-driven approach use a code-specific order to apply optimizations. Thus, they improve the performance more than the fixed-order approach. For example, for *bitcount*, by applying fixed-order sequences, there is an improvement of 22.2%. While using the sequences found by the empirical and model-driven approaches, the improvement is 24.5% and 24.1% respectively. In most cases, our approach achieves similar performance improvements as the empirical approach, yet compile-time is much lower.

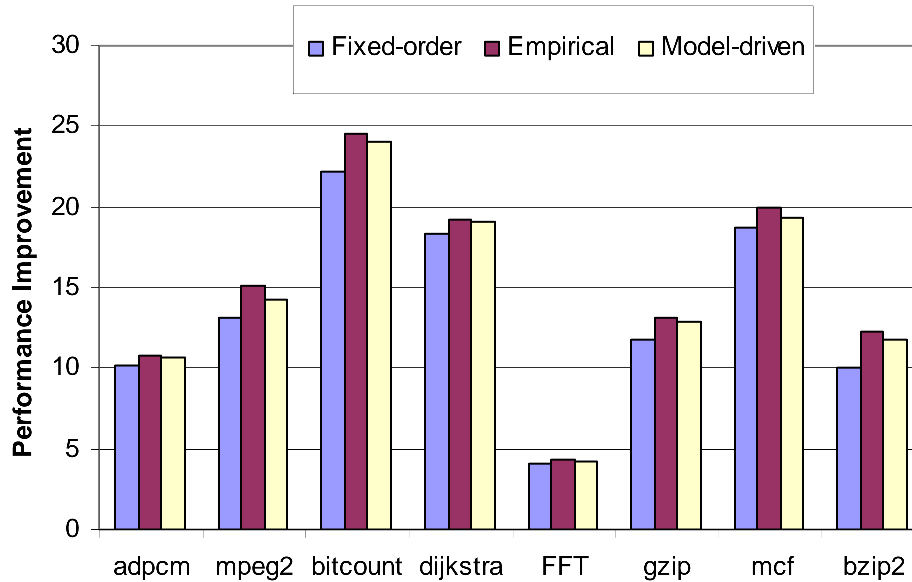


Figure 6.10: Comparing performance improvement

6.7.1.3 Memory requirement

In our approach, FIO needs tables to store the data and control dependence information and information about each optimization opportunity and their interactions. We measured the memory requirements of FIO to ensure the information can be stored in memory.

Table 6.7: Memory requirement of our approach (KB)

Benchmarks	Min	Max	Average
adpcm.rawcaudio	22	1102	710
mpeg2.enc	1	9714	601
bitcount	1	164	58
dijkstra.large	9	98	43
FFT	1	888	205
gzip	1	3417	289
mcf	4	1646	227
bzip2	1	3938	527

Table 6.7 shows the minimum, maximum and average memory requirements for the functions in each benchmark. For example, there are 3 functions in *adpcm*. They required 1102

KB, 1004 KB and 22 KB memory. These three functions needed 710 KB memory on average. Most of the memory consumed is for storing information about each optimization opportunity and their interactions. As the number of optimization opportunities in each procedure increases, the memory requirements also increase. For the largest procedure *putpict* (in *mpeg2*), it has 7321 optimization opportunities and required 9714 memory. From the table, we can see that the memory requirements of FIO are reasonable and the information generated in FIO can be sufficiently stored in memory.

From these experimental results, we can see that our model-driven approach achieves the similarly good sequences as in the empirical approach with much less compile-time, using reasonable memory.

6.7.2 Evaluation Function: Profitability

In the second set of our experiments, the evaluation function is profitability. In the empirical approach, the optimizations in a sequence are performed on the code. Then, the code is executed to evaluate the profitability of the sequence. In our model-driven approach, we construct a code-specific optimization sequence as described in Section 6.6. The evaluation function used is the profitability of optimizations, predicted by FPSO. We compare compile-time and performance of three approaches (the fixed-order approach, the empirical GA approach and our model-driven approach).

6.7.2.1 Compile-time overhead

When using profitability as the evaluation function, the empirical approach needs to apply optimizations and execute the code to evaluate the profitability. For the SPEC benchmarks (i.e., *gzip*, *mcf*, and *bzip2*), the test input was used to execute the code. In our approach, the interaction property is detected by FIO and profitability is determined by FPSO. Thus, the compile-time to search for a good order is greatly reduced. The compile-time overhead of these three approaches is shown in Table 6.8.

Table 6.8: Compile-time overhead of three approaches (minutes)

Benchmarks	Fixed-order	Empirical	Model-driven
adpcm.rawcaudio	0.05	5.41	1.14
mpeg2.enc	1.92	726.67	82.24
bitcount	0.15	18.97	1.66
dijkstra.large	0.05	11.63	0.68
FFT	0.11	13.20	1.81
gzip	1.52	1180.67	53.82
mcf	0.53	74.64	19.54
bzip2	2.34	2618.79	58.68

From the table, the compile-time for the fixed-order approach is the same as in Table 6.5. It varies from 0.05 minutes to 2.34 minutes. Because the empirical approach needs to execute the code, its compile-time is large, varying from 5 minutes to 43.6 hours. Using our approach, the compile-time is greatly reduced compared with the empirical approach. It varies from 0.7 to 82 minutes. In the empirical approach, each function is compiled for 200 sequences and evaluated by executing the code. The compile-time for the empirical approach is related to the compile-time and execution time for each function. For example, there are 106 functions in *gzip*. The average compile-time for a function is about 0.8 seconds. The execution time for test input is about 2.4 seconds. Adding the GA search time, it took 1181 minutes for the GA to find code-specific sequences for *gzip*. In our approach, we use FIO to determine the interactions among optimizations and FPSO to predict the profitability of optimizations. The compile-time of our approach depends on the time for FIO to determine the interaction property and the time for FPSO to predict profitability. For example, for *mpeg*, the average compile-time for FIO to determine the optimization property is about 20 second and the compile-time for FPSO to determine profitability is about 6 seconds. Thus, it took 82.24 minutes for our approach to determine good optimization sequences for *mpeg*.

6.7.2.2 Performance improvement

Besides compile-time, we also compare performance of three approaches, as shown in Figure 6.11. In the figure, the performance improvement is measured using dynamic instruction count.

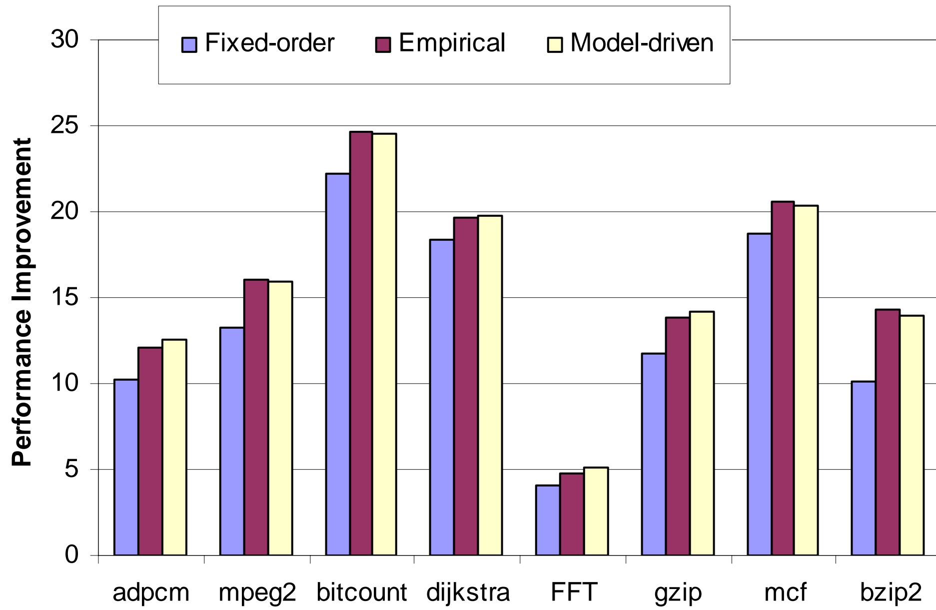


Figure 6.11: Comparing performance improvement

In the figure, the empirical approach and our model-driven approach improve the performance more than the fixed-order approach. For example, by applying fixed-order sequences, there is an improvement of 22.2% for *bitcount*. While using the sequences found by the empirical and model-driven approaches, the improvement is 24.6% and 24.5% respectively. Comparing with the results in Figure 6.10 (the number of optimizations applied as the evaluation function), both the empirical approach and our model-driven approach have better performance. For example, the empirical approach improves performance by 10.8% for *adpcm* if using the number of optimizations as the evaluation function. However, if profitability is used as the evaluation function, the empirical approach improves performance by 12.1%.

In most cases, our model-driven approach achieves similar performance improvements as the empirical approach. In some cases, performance of our model-driven approach is even better than the empirical approach. For example, for *adpcm*, using the empirical approach, the improvement is 12.1%, while using our model-driven approach, the improvement is 12.6%. This is because an optimization is not applied if it is predicted as unprofitable by FPSO in our approach.

6.7.2.3 Memory requirement

Similarly as in Section 6.7.1.3, we measured the memory requirements of our approach to ensure the information can be stored in memory. Here, not only the dependence and the optimization table, but also the code models for determining profitability need to be stored.

Table 6.9: Memory requirement of our approach (KB)

Benchmarks	Min	Max	Average
adpcm.rawcaudio	23	1131	723
mpeg2.enc	2	9815	659
bitcount	1	181	66
dijkstra.large	10	106	51
FFT	1	985	232
gzip	2	3476	332
mcf	5	1675	253
bzip2	2	4328	582

Table 6.9 shows the minimum, maximum and average memory requirements for the functions in each benchmark. For example, there are 3 procedures in *adpcm*. They required 1131 KB, 1014 KB and 23 KB memory. These three procedures needed 723 KB memory on average. The information needed for determining profitability is small. Thus, comparing with the results in Table 6.7, the memory requirements do not increase much. For example, here the smallest function in *adpcm* needs 23KB memory, while it needs 22KB memory without determining profitability. From the table, we can see that the memory requirements of our approach are reasonable and the information can be sufficiently stored in memory.

Our experiments show that the interaction property is very useful in finding code-specific optimization sequences. Comparing with the empirical approach, our model-driven approach can find similarly good optimization sequences in much less compile-time. Our techniques make the search for good order to apply optimizations practical.

7.0 CONCLUSIONS

Compilers apply code optimizations to improve the quality of generated code (e.g., running faster, consuming less memory or less power). However, it is known that there are problems with the application of optimizations that keep compilers from achieving the full potential benefit of optimization. For example, optimizations may degrade performance in certain circumstances. Also, optimizations enable and disable each other. The order to apply optimizations impacts performance. So far there is no systematic and efficient way to decide when, where and in what order to apply optimizations to be effective. The continued growth of embedded systems, the application of dynamic optimizations and the shrinking performance gains from developing new optimizations demand us to handle these long-standing problems.

Most prior work has focused on developing heuristics or empirical approaches to handle some of these application problems. However, heuristics tend to be ad hoc and focus specifically on a single or a small class of optimizations. Heuristics also require tuning parameters to select appropriate threshold values. The major disadvantage of an empirical approach is its high cost. Although there has been some work that uses the models to explore the application problems, the work is very limited; it works for a small set of optimizations and a single machine resource. Ideally, we need a general, effective and efficient model-driven approach, which uses models to determine the optimization properties and to intelligently apply optimizations.

7.1 SUMMARY OF CONTRIBUTIONS

The benefits of this dissertation are twofold. The theoretical benefits include developing a foundation that determines two optimization properties: profitability and interaction. The

practical benefits include developing an optimizing compiler that uses model-driven techniques developed in the framework to effectively apply optimizations.

Effectively applying optimizations is hampered by the difficulties in understanding the properties of optimizations. This research presents a novel model-based framework to determine optimization properties. The focus is to accurately predict profitability and automatically detect the interaction property without applying optimizations or executing the code. The scope of this research covers a wide range of optimizations and machine resources.

This dissertation presents framework instances, FPSO and FPLO, to predict the profitability of scalar and loop optimizations. FPSO and FPLO include models of code, optimizations and machine resources. For machine resources, FPSO considers registers and functional units and FPLO considers data cache. In FPSO and FPLO, there is a profitability engine that uses models to predict the profit of applying an optimization at any code point where the optimization is applicable.

This dissertation also describes a framework instance, FIO, to detect the interactions among a set of optimizations. A specification language, SpeLO, is developed to express the conditions under which an optimization can be safely applied and the actions of the optimization. Optimization models are developed using SpeLO. The code model in FIO is the control flow graph with explicit data and control dependence information. As part of FIO, there is an interaction engine that uses models to generate the specific enabling, disabling and post conditions for each optimization at a program point. These enabling and disabling conditions are then matched with the post-conditions of other optimizations to determine the enabling and disabling interactions.

By determining these optimization properties, compilers will apply optimizations more effectively. Compilers can perform profit-driven optimization, which applies only profitable optimizations. Also, compilers can determine a code-specific order or configuration to apply optimizations with practical compile-time overhead.

We implemented our framework instances and performed experiments to evaluate their effectiveness and efficiency. We evaluated prediction accuracy of FPSO and FPLO. On average, they can make correct predictions about 90% of the time. We compared our profit-driven approach with other two approaches. One approach always applies applicable optimizations. The other uses a heuristics to decide whether an optimization should be applied. The model-driven

approach and the heuristic approach achieved better performance improvement than the always-applying approach. The model-driven approach is practical because it does not require tuning the parameters necessary in the heuristic approach. For FIO, we compared the model-driven approach with other two approaches for searching for code-specific optimization sequences. One approach uses a fixed order to apply optimizations. The other approach experimentally searches for good order to apply optimizations to get the most benefit. The model-driven approach and the empirical approach can find similarly good optimization sequences. Thus, they achieve similar improvement, better than the fixed order approach. However, compile-time of the model-driven approach is greatly reduced, when compared with the empirical approach (up to 43 times better). The model-driven approach is scalable.

This dissertation demonstrates that analytic models can be used to address the effective application of optimizations. Our model-driven approach is practical and scalable. With model-driven optimizations, compilers can produce higher quality code in less time than what is possible with current approaches.

7.2 LIMITATIONS

This dissertation has several limitations, including limitations of the models, limitations of what can be automatically generated and limitations in the experiments.

This research covers a number of machine resources, including cache, registers and computation without code scheduling. Our resource models are most suitable for the Intel IA-32 and other processors where there are few registers or with in-order single issue pipeline. However, other machine resources such as computation with code scheduling are important and need to be modeled. The optimizations considered in this research include a number of scalar and loop optimizations. However, there are other important optimizations that have not been studied, such as procedure inlining and code scheduling. Our specification language, SpeLO, can specify a wide range of scalar and loop optimizations, including path-based optimizations. However, some optimizations cannot be expressed by SpeLO, for example conditional constant propagation [49]. In conditional constant propagation, a program needs to be symbolically

executed, which cannot be expressed using the current SpeLO specification. Modeling more machine resources and optimizations is deferred to future work.

In this research, code models are automatically generated by the optimizer. However, optimization models are developed separately and manually by a compiler writer. To predict profitability, the compiler writer needs to express the semantics of optimizations using basic edits. To determine the interaction property, the compiler writer requires to represent the conditions under which an optimization can be applied and the actions of the optimization, using SpeLO. When the profitability and interaction properties are needed, the compiler writer needs to write optimization models based on different specifications. A unifying specification is needed that optimization models can be uniformly developed. A tool is also needed to automatically generate optimization models based on the unifying specification and provide the compiler writer a simpler interface to use our framework.

Regarding the experiments, there are two major limitations. First, due to the restrictions of the compiler infrastructure we use, Mach SUIF [44], we ran experiments on the Intel IA-32 machines. We have not investigated how our approach applies to other machine architectures. Secondly, some empirical investigation would have to be undertaken to compare our model-driven approach with other approaches, such as the Optimization-Space Explore compiler [46], where only analytic resource models are used for effectively applying optimizations. Experiments are needed to show the compile-time advantages of modeling code and optimizations in our approach.

7.3 FUTURE WORK

There are a number of open research problems related to this research. Although only profitability and interaction were studied in this dissertation, our model-based framework can be used to study other optimization properties. Also, although we focus on profit-driven optimization and finding a code-specific order to apply optimizations, there are other uses of our framework. In the future, we can extend the work in the following ways.

1) Modeling more resources. In this dissertation, the resources that we model are cache, registers and computation without code scheduling. Our models for the

registers are more suitable for the Intel IA-32 and other processors where there are few registers. In the future, we may need to model resources based on different machine architectures. For example, we may need to predict the profit on computation with code scheduling. To do so, a code model (e.g., dependence graph), a resource model, and an optimization model for code scheduling are needed. Also, the profitability engine should be able to infer the changes of an optimization on the computation code model directly from the optimization model. For some architecture, we may also need to combine all the resources (cache, registers and computation) to make more accurate predictions.

2) Modeling more optimizations. In this dissertation, we developed models for several scalar and loop optimizations. Although they cover a wide range of optimizations, there are some other important optimizations (e.g., procedure inlining, code scheduling) needed to be studied. Also the optimizations studied in this work are global optimizations. We may also need to model the inter-procedural optimizations.

3) Determining other optimization properties. In this dissertation, we focus on two optimization properties, profitability and interaction. Our framework can also be used to study other optimization properties. For example, we can study the impact of optimizations on code size and power consumption. In this work, we combined the profitability and interaction to find a code-specific order to apply optimizations. In the future, we may also need to combine the profitability and interaction properties with other optimization properties to find a way to apply optimizations to balance multiple constraints. For example, in embedded systems, in addition to performance, memory and power consumption are also important. We need to consider the impact of optimizations on all these factors and determine a way to apply optimizations to balance these constraints. Another optimization property that needs to be studied is the cost of applying optimizations, which includes the cost for applicability analysis and the actions to perform the optimizations. The cost of applying optimizations is important for deciding when and how to apply dynamic optimizations.

4) Using optimization properties for other applications. Profit-driven optimization and finding a code-specific order of applying optimizations are two applications experimentally evaluated in this dissertation. There are other applications for our framework. For example, we can find a code-specific configuration to apply optimizations similarly as finding effective optimization order. Another interesting application is to reconfigure the hardware. Based on the optimization properties determined in the framework, we can choose a hardware configuration (e.g., cache configuration) that fits better the application.

5) Develop software tools to enable the automatic generation of models and model-based optimizations. In this dissertation, optimization models are developed separately and manually by a compiler writer. Work can be done to design a unifying specification language to express optimizations and resources, from which all the models could be automatically created. A tool and algorithm are also needed to automatically generate the implementation of model-based optimizations.

APPENDIX A OPTIMIZATION MODELS

A.1 SCALAR OPTIMIZATION MODELS

Optimization	Optimization Model
Copy Propagation	# Propagate $x \leftarrow y$ Modify the statement: $\text{Delete} < \text{USE } x > @ S_d$ $\text{Insert} < \text{USE } y > @ S_d$ Delete the statement: $\text{Delete} < \text{DEF } x \text{ USE } y \text{ OP } copy > @ S_s$
Constant Propagation	# Propagate $x \leftarrow const$ Modify the statement: $\text{Delete} < \text{USE } x > @ S_d$
Dead Code Elimination	# Eliminate the dead code x $\leftarrow$ *EXP* (*y* op *z*) at S_s Delete a statement: $\text{Delete} < \text{DEF } x \text{ USE } y, z \text{ OP } op > @ S_s$
Loop Invariant Code Motion	# Move a loop invariant statement $x \leftarrow y$ op z Insert a statement: $S_d' = S_d + 1$ $\text{Insert} < \text{DEF } x \text{ USE } y, z \text{ OP } op > @ S_d'$ Delete the statement: $\text{Delete} < \text{DEF } x \text{ USE } y, z \text{ OP } op > @ S_s$

Optimization	Optimization Model
Partial Redundancy Elimination	# Eliminate the partial redundant expression *EXP* (y op z) at S_s Insert a statement: $S_d' = S_d + 1$ $\text{Insert} < \text{DEF } v \ \ \text{USE } y, z \ \ \text{OP } op > @ S_d'$ Replace the computation: $\text{Delete} < \text{USE } y, z \ \ \text{OP } op > @ S_s$ $\text{Insert} < \text{USE } v \ \ \text{OP } copy > @ S_s$ Update the same expressions: $\forall T \ \mid T = w \leftarrow EXP(y \ op \ z) \text{ at } S_w$ $\text{Delete} < \text{DEF } w > @ S_w$ $\text{Insert} < \text{DEF } v > @ S_w$ $S_w' = S_w + 1$ $\text{Insert} < \text{DEF } w \ \ \text{USE } v \ \ \text{OP } copy > @ S_w'$
Value Numbering	#Replace a redundant statement $x \leftarrow y$ op z with $x \leftarrow$ VN[x] at S_s Replace the computation: $\text{Delete} < \text{USE } y, z \ \ \text{OP } op > @ S_s$ $\text{Insert} < \text{USE } v \ \ \text{OP } copy > @ S_s$ Replace all uses of x with its value number v: $\forall \text{u} \mid \text{u is use of x at } \text{S}_\text{u}$ $\text{Delete} < \text{USE } x > @ S_u$ $\text{Insert} < \text{USE } v > @ S_u$ #Fold constant a statement $x \leftarrow y$ op z at S_s Delete the computation: $\text{Delete} < \text{USE } y, z \ \ \text{OP } op > @ S_s$ #Delete a redundant Φ-instruction $x \leftarrow \Phi(x_1, x_2, \ldots)$ Replace all uses of x with its value number v: $\forall \text{u} \mid \text{u is use of x at } \text{S}_\text{u}$ $\text{Delete} < \text{USE } x > @ S_u$ $\text{Insert} < \text{USE } v > @ S_u$ #Delete a useless copy instruction $x \leftarrow y$ at S_s Delete the copy instruction: $\text{Delete} < \text{DEF } x \ \ \text{USE } y \ \ \text{OP } copy > @ S_s$

A.2 LOOP OPTIMIZATION MODELS

Optimization	Optimization Model
Loop Interchange	INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and interchange is legal for loops *i, j*; $f_{Interchange}(\oint_{N-1}\cdots\oint_{i}\cdots\oint_{j}\cdots\oint_{0}\langle R\rangle) = \oint_{N-1}\cdots\oint_{j}\cdots\oint_{i}\cdots\oint_{0} g(\langle R\rangle)$, where $g(\langle R\rangle) = \langle\forall(r\in\langle R\rangle)\,h(r)\rangle$ and $h(r) = (l(A), C)$ and $l(A) = A[:][i] \leftrightarrow A[:][j]$
Loop Reversal	INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and reversal loop *i*; $f_{reversal}(\oint_{N-1}\cdots\oint_{i}\cdots\oint_{0}\langle R\rangle) = \oint_{N-1}\cdots\oint_{i}\substack{lb\\ -step\\ ub}\cdots\oint_{0}\langle R\rangle$
Loop Tiling	INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ tiling loops $t1,\cdots,tn$, with tile size $ts_1,\cdots,ts_n$ respectively; $f_{tiling}(\oint_{N-1}\cdots\oint_{tn}\cdots\oint_{t1}\cdots\oint_{0}\langle R\rangle) = g(\oint_{N-1}\cdots\oint_{tn}\cdots\oint_{t1}\cdots\oint_{0})\ f\langle R\rangle$, where $g(\oint_{N-1}\cdots\oint_{tn}\cdots\oint_{t1}\cdots\oint_{0}) = \oint_{N+n-1}\substack{ub_n\\ ts_n\\ lb_n}\cdots\oint_{N}\substack{ub_1\\ ts_1\\ lb_1}\oint_{N-1}\cdots\oint_{tn}\substack{h(n)\\ \\ x_n}\cdots\oint_{t1}\substack{h(1)\\ \\ x1}\cdots\oint_{0}$, $h(i) = \min(ub_i, x_i + ts_i - 1)$, and $f\langle R\rangle = \langle\forall(r\in\langle R\rangle)\ (l(A), C)\rangle$ where $l(A) = [0A]$

Optimization	Optimization Model
Loop unrolling	INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and unroll factor U; $f_{unroll}(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle)=\langle ln_{unroll},ln_{rest}\rangle$ $ln_{unroll}=\oint_{N-1}\cdots\oint_{1}\oint_{0,\,step\times U} g(\langle R\rangle)$ $\quad ln_{rest}=\oint_{N-1}\cdots\oint_{1}\oint_{0,\,\lceil \frac{ub+1}{U}\rceil\times U}\langle R\rangle$ $g(\langle R\rangle)=\langle R\rangle\wedge(\bigwedge_{i=1}^{U-1}\langle\forall(r\in\langle R\rangle)\,h(r,i)\rangle)$ $h(r,i)=(A,l(C,i))$ $l(C,i)=\forall(s\in\{a\mid A[a][N-1]\neq 0\})\,C[s]+i$
Loop fusion	INPUT: $ln_1(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R_1\rangle)$, $ln_2(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R_2\rangle)$, …, $ln_m(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R_m\rangle)$ $f_{fusion}(\langle ln_1,ln_2,\cdots ln_m\rangle)=\oint_{N-1}\cdots\oint_{1}\oint_{0}f(\langle R_1\rangle,\langle R_2\rangle,\cdots,\langle R_m\rangle)$ $f(\langle R_1\rangle,\langle R_2\rangle,\cdots,\langle R_m\rangle)=\bigwedge_{i=1}^{m}\langle R_i\rangle$
Loop distribution	INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and the sets of reference index which will be in lni , $\{i_1,\ldots,i_p\}$; $f_{distribution}(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle)=\langle ln_1,ln_2,\cdots ln_m\rangle$ $ln_i=\oint_{N-1}\cdots\oint_{1}\oint_{0}f_i(\langle R\rangle)$, where $f_i(\langle R\rangle)=\langle r_{i1},r_{i2},\ldots,r_{ip}\rangle$

A.3 OPTIMIZATION MODEL FOR INTERACTION

Optimization	Optimization Model
copy propagation	***CPP*** **PRECONDITION** **Code_Pattern** ANY Si: Si.opcode = copy AND type(Si.opnd1) = var AND type(Si.dst) = var; **Depend** ALL Sj, pos: flow_dep(Si, Sj, =); NO Sk: flow(Sk, Sj, =) AND (Sk != Si); NO Sp: mem(Sp, path(Si, Sj)), anti_dep(Si, Sp, =); **ACTION** Modify (operand(Sj, pos), Si.opnd1); Delete (Si);
dead code elimination	***DCE*** **PRECONDITION** **Code_Pattern** ANY L; **Depend** ANY Sk: mem(Sk, L), NOT flow_dep(Sk, Sk) AND NOT anti_dep(Sk, Sk) AND NOT flow_dep(L.head, Sk) NO Sm: mem(Sm, L) AND Sm != Sk, flow_dep(Sm, Sk) OR anti_dep(Sk, Sm) OR out_dep(Sm, Sk) OR out_dep(Sk, Sm) OR anti_dep(Sm, Sk) OR ctr_dep (Sm, Sk) **ACTION** Move (Sk, L.preheader);
loop invariant code motion	***LICM*** **PRECONDITION** **Code_Pattern** ANY Si: Si.opcode = copy AND type(Si.opnd1) = var AND type(Si.dst) = var; **Depend** ALL Sj, pos: flow_dep(Si, Sj, =); NO Sk: flow(Sk, Sj, =) AND (Sk != Si); NO Sp: mem(Sp, path(Si, Sj)), anti_dep(Si, Sp, =); **ACTION** Modify (operand(Sj, pos), Si.opnd1); Delete (Si);

Optimization	Optimization Model
partial redundancy elimination	***PRE*** **PRECONDITION** **Code_Pattern** ANY Si: Si.opcode = binary_exp; ALL Sj: mem(path(Entry, Si)), Sj.opcode = Si.opcode AND Sj.opnd1 = Si.opnd1 AND Sj.opnd2 = Si.opnd2; **Depend** NO Sk: anti_dep(Sj, Sk, =) AND flow_dep(Sk, Si, =); ALL Sp: flow_dep(Sp, Si, =) AND ¬in_every_path(Sj, Sp, Si, save pred(Si) ∧ ¬ in_any_path(pred(Si), Sj, Si) to Bq) NO Bl: mem(Bq), ¬post_dom(B(Si), Bl); **ACTION** Add ((new_temp= Si.opnd1 Si.opcode Si.opnd2), Bq); Add (new_temp=Si.opnd1 Si.opcode Si.opnd2), Sj); Modify (Sj, (Sj.dst = new_temp)); Modify (Si, (Si.dst = new_temp));
constant propagation	***CTP*** **PRECONDITION** **Code_Pattern** ANY Si: Si.opcode = copy AND type(Si.opnd1) = const AND type(Si.dst) = var; **Depend** ALL Sj, pos: flow_dep(Si, Sj, =); NO Sk: flow(Sk, Sj, =) AND (Sk != Si); **ACTION** Modify (operand(Sj, pos), Si.opnd1);
branch chaining	***BRC*** **PRECONDITION** **Code_Pattern** ANY Si: Si.opcode = jmp AND B(Si) – Si = Ø; **Depend** ALL Sj: ctrl_dep(Sj, Si, =); **ACTION** Modify (Sj.target, Si.target); Delete (Si);

Optimization	Optimization Model

global value numbering	***VN***
	Pass 1: Assigning a value number
	PRECONDITION
	    Code_Pattern
	        ANY Si: Si.opcode = Ø OR Si.opcode = assign
	    Depend
	        ALL Sj: flow_dep (Sj, Si)
	ACTION
	        // meaningless Ø-operation
	        IF ((Si.opcode = Ø) AND (equal (Sj.VN)))
	            Si.VN = Sj.VN;
	        // redundant Ø-operation or assign
	        ELSE IF (hash (Sj.VN, Si.opcode) != NULL)
	            Si.VN = hash (Sj.VN, Si.opcode);
	        ELSE
	            hash (Sj.VN, Si.opcode, Si.VN);
	Pass 2: Redundancy elimination
	PRECONDITION
	    Code_Pattern
	        ANY Si: Si.opcode = binary_exp
	    Depend
	        ALL Sj: Sj.VN = Si.VN
	ACTION
	        Delete (Sj);

branch elimination	***BRE***
	PRECONDITION
	    Code_Pattern
	        ANY Si: Si.opcode = branch;
	    Depend
	        ALL Sj: ctrl_dep(Sj, Si, =);
	        ALL Sp: flow_dep(Sp, Sj, =) AND type(Sp.opnd1) = const;
	ACTION
	        IF match(Sp.opnd1, Si.opcode)
	            Modify (Sj.target, Si.target);
	        ELSE
	            Modify (Sj.target, Si.fall_through);
	        Delete (Si);

Optimization	Optimization Model

loop interchange	***LPI***
	PRECONDITION
	Code_Pattern
	// find tightly nested loops
	ANY L1, L2, L3: tight_loop(L1, L2, L3);
	Depend
	// perfectly nested without flow dependence with <, >
	no L1.head: flow_dep(L1.head, L2.head);
	no L2.head: flow_dep(L2.head, L3.head);
	no Sm, Sn: mem(Sm, L3) AND mem(Sn, L3), flow_dep(Sn, Sm, (<,>));
	ACTION
	move (L1.head, L3.head);
	move (L1.end, L3.end.prev);

loop fusion	***LPF***
	PRECONDITION
	Code_Pattern
	// find adjacent loops with equivalent heads
	ANY L1, L2: adjacent_loop(L1, L2) AND L1.initial = L2.initial AND
	L1.final = L2.final AND L1.lcv = L2.lcv;
	Depend
	// No dependence with backward direction
	no Sn, Sm: mem(Sn, L1) AND mem(Sm, L2), flow_dep(Sn, Sm, any) OR
	out_dep(Sn, Sm, any) OR anti_dep(Sn, Sm, any);
	// No definition reaching prior to loops
	no Si, Sj, Sk: mem(Sj, L1) AND mem(Sk, L2), flow(Si, Sj, any) AND
	anti_dep(Sj, Sk, any) AND (Si ≠ Sk);
	ACTION
	modify(L1.head.opr1, L2.head.label);
	modify(L2.end.opr1, L1.end.label);
	delete(L1.end);
	delete(L2.head);

APPENDIX B RESOURCE MODEL FOR COMPUTATION

Table B.1: Computation resource model for an Intel IA-32 machine using Mach SUIF representation

Operation	Latency
CVT	1
LDA	1
LDC	1
ADD	1
SUB	1
NEG	1
MUL	3
DIV	19
REM	19
MOD	19
ABS	1
MIN	1
MAX	1
NOT	1
AND	1
IOR	1
XOR	1
ASR	1
LSL	1
LSR	1
ROT	1

Operation	Latency
MOV	1
STR	2
MEMCOPY	1
SEQ	1
SNE	1
SL	1
SLE	1
BTRUE	1
BFALSE	1
BEQ	1
BNE	1
BGT	1
BLE	1
BLT	1
JMP	1
JMPI	1
MBR	1
RET	1

APPENDIX C EXPERIMENTAL RESULTS FOR ATHLON MACHINE

C.1 HEURISTIC-DRIVEN APPROACH

Table C.1: Improvement of heuristic-driven PRE and LICM with different limits

Benchmark	Heuristic-driven PRE				Heuristic-driven LICM			
	0	4	8	16	0	4	8	16
gzip	3.50	3.75	3.78	**4.10**	2.90	3.29	**5.40**	3.27
vpr	1.22	0.75	1.81	**1.83**	-	-0.38	0.52	**0.69**
mcf	**2.37**	2.35	2.31	2.22	2.50	**2.62**	2.58	2.47
parser	1.25	1.50	**1.70**	1.35	2.55	**2.86**	1.99	2.23
vortex	4.73	**5.25**	4.66	3.86	4.88	**5.69**	4.99	5.28
bzip2	7.35	7.52	**8.19**	7.91	7.02	**7.35**	6.70	4.57
twolf	1.07	0.88	**1.14**	0.02	0.52	0.38	**2.14**	1.91
bitcount	6.8	6.8	8.69	**9.53**	6.35	6.35	8.99	**10.2**
dijkstra	3.1	3.5	**3.6**	0	**3.2**	0	0	-3.1
FFT	1.12	1.21	**1.69**	1.23	2.13	1.93	**2.85**	-0.3
jpeg	9.13	9.16	**10.0**	8.69	10.1	**10.5**	9.5	9.23
sha	8.64	**10.7**	8.2	8.2	9.34	**11.2**	8.24	7.33

C.2 PERFORMANCE BENEFIT OF PROFIT-DRIVEN PRE, LICM, AND VN

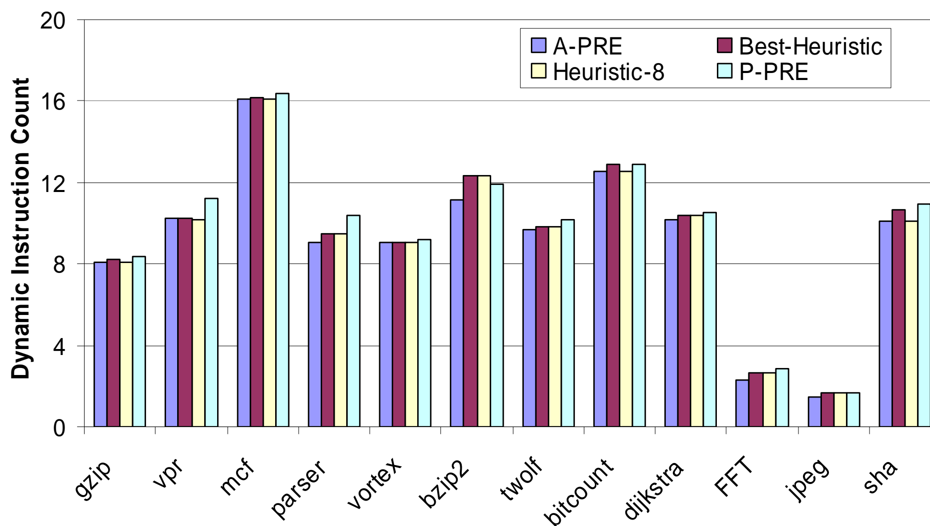


Figure C.1: Dynamic instruction count improvement of PRE

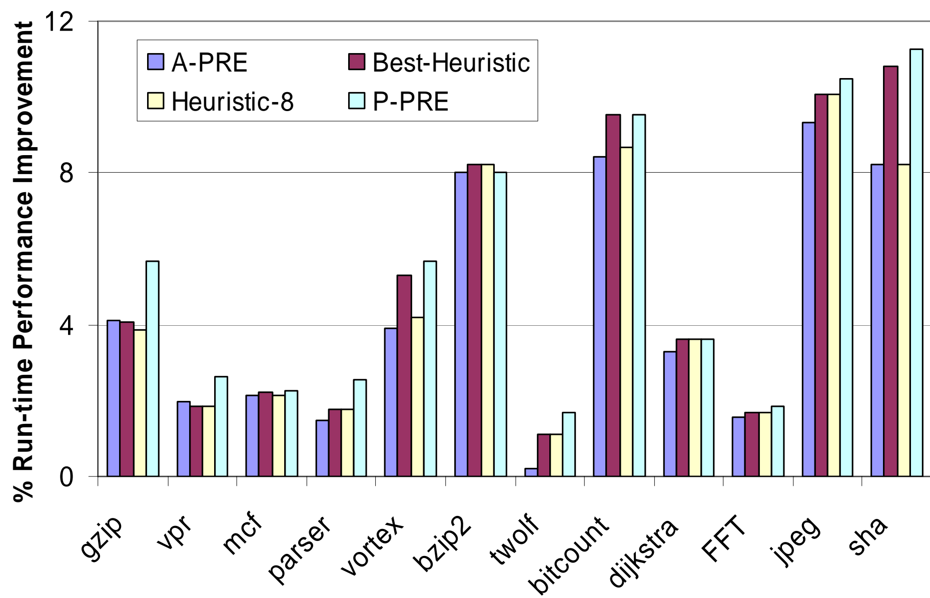


Figure C.2: Run-time performance improvement of PRE

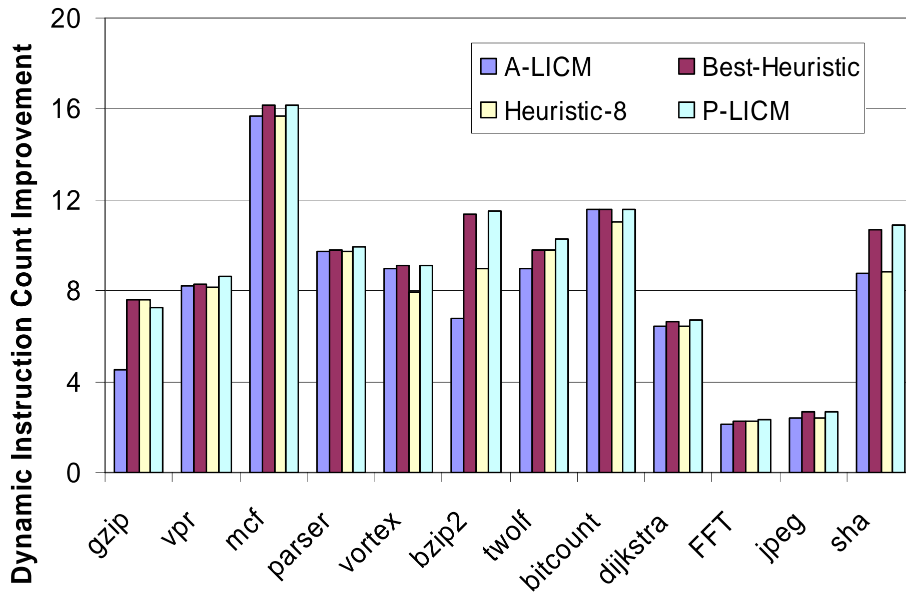


Figure C.3: Dynamic instruction count improvement of LICM

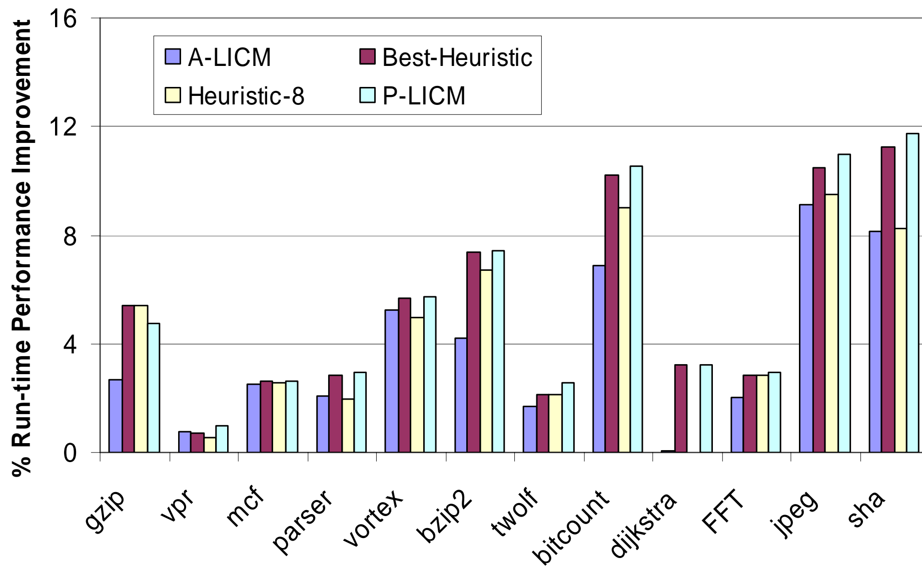


Figure C.4: Run-time performance improvement of LICM

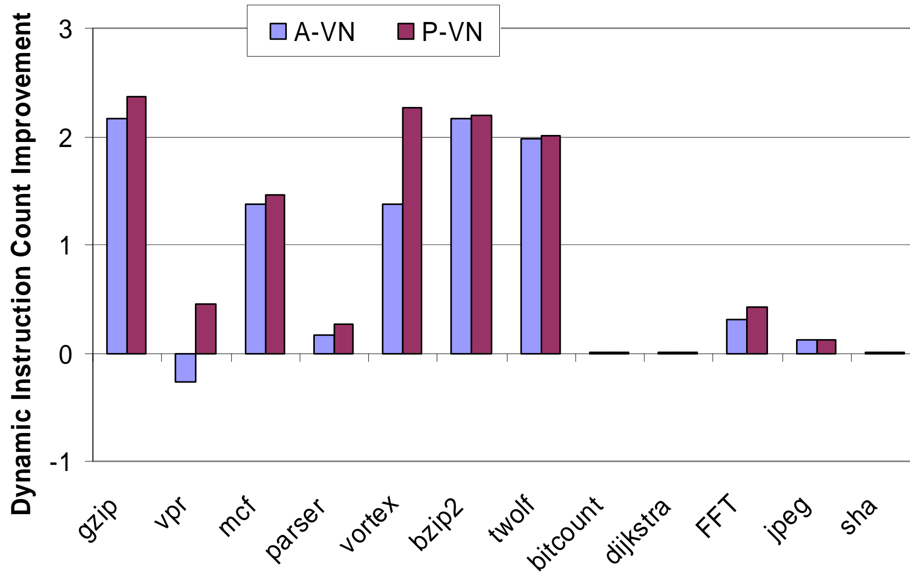


Figure C.5: Dynamic instruction count improvement of VN

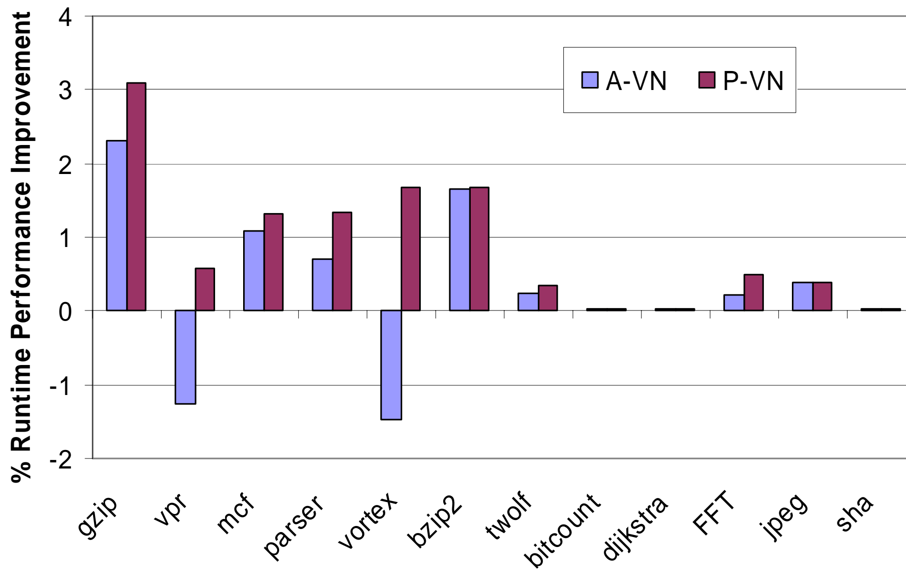


Figure C.6: Run-time performance improvement of VN

C.3 COMPILE-TIME OVERHEAD

Table C.2: Compile-time for PRE

Benchmark	Full Compile-time			One Pass Compile-time		
	A-PRE	H over A	P over	A-PRE	H over	P over A
gzip	42	7.48%	16.14%	9.14	35.26%	61.34%
vpr	128.38	50.33%	68.25%	36.32	76.35%	101.18%
mcf	20.89	38.82%	46.00%	3.96	59.39%	71.15%
parser	100.67	22.18%	35.40%	25.72	66.68%	91.23%
vortex	490.48	17.30%	29.08%	83.23	55.82%	76.96%
bzip2	33.77	26.15%	30.59%	9.97	70.15%	88.91%
twolf	755.55	43.87%	57.13%	192.19	89.93%	102.12%
bitcount	6.33	7.03%	10.65%	1.23	57.19%	63.24%
dijkstra	1.13	10.93%	13.86%	0.23	25.66%	49.32%
FFT	4.59	9.12%	13.93%	1.01	42.23%	56.18%
jpeg	34.34	39.89%	51.78%	6.18	79.13%	101.43%
sha	2.99	10.59%	15.88%	0.59	24.26%	38.39%
average	--	23.64%	32.39%	--	56.84%	75.12%

Table C.3: Compile-time for LICM

Benchmark	Full Compile-time			One Pass Compile-time		
	A-LICM	H over	P over A	A- LICM	H over	P over A
gzip	47.8	23.51%	27.80%	13.45	59.36%	70.13%
vpr	128	14.84%	25.78%	33.52	56.92%	75.42%
mcf	20.8	32.69%	39.28%	4.93	46.23%	71.35%
parser	109.3	22.11%	26.43%	30.15	59.27%	86.39%
vortex	492.1	11.24%	15.63%	90.18	38.21%	49.51%
bzip2	38.59	25.81%	33.89%	13.14	55.65%	76.15%
twolf	591	38.37%	55.04%	160.14	89.04%	130.49%
bitcount	6.82	4.32%	7.54%	1.68	19.58%	28.13%
dijkstra	1.13	7.55%	10.12%	0.31	12.49%	16.38%
FFT	4.66	34.37%	40.49%	1.31	61.13%	84.62%
jpeg	25.23	21.23%	29.03%	6.23	57.69%	73.18%
sha	2.89	18.98%	26.61%	0.89	39.76%	56.23%
average	--	21.25%	28.14%	--	49.61%	68.17%

Table C.4: Compile-time for VN

benchmark	Full Compile-time		One Pass Compile-time	
	A-LICM	P over A	A- LICM	P over A
gzip	46.93	15.77%	6.15	28.13%
vpr	127.06	15.14%	17.98	26.52%
mcf	25.57	15.02%	3.13	22.64%
parser	96.25	18.02%	14.16	32.85%
vortex	508.94	14.73%	60.52	26.94%
bzip2	28.35	17.39%	3.25	49.39%
twolf	283.35	16.83%	40.12	35.08%
bitcount	7.25	13.12%	1.81	25.25%
dijkstra	1.89	13.19%	0.23	25.02%
FFT	5.54	18.24%	0.89	43.35%
jpeg	30.11	16.65%	4.72	38.04%
sha	3.12	12.19%	0.53	28.13%
average	--	15.52%	--	31.78%

C.4 MODEL VERIFICATION

The prediction accuracy for PRE, LICM and VN is the same as what are reported in Table 4.4, Table 4.5 and Table 4.6.

BIBLIOGRAPHY

[1] L.Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon and T. Waterman. Finding Effective Compilation Sequences. *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004.

[2] M. Arnold, S. Fink, D. Grove, M. Hind and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, October 2000.

[3] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys, Vol. 26 No. 4, pp 345-420,* December 1994.

[4] M.P. Bivens and M.L. Soffa. Incremental register reallocation. *Software Practice and Experience, Vol. 20, No. 10,* October 1990.

[5] R. Bodik. *Path-Sensitive, Value-Flow Optimizations of Programs.* PhD dissertation. University of Pittsburgh, 1999.

[6] P. Briggs and K. D. Cooper. Effective Partial Redundancy Elimination. *Proceedings of ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[7] P. Briggs, K. D. Cooper, T. J. Harvey and L. T. Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software Practice and Experience, Vol. 28, No. 8,* July 1998.

[8] P. Briggs, K. Cooper and L. T. Simpson. Value Numbering. *Software Practice and Experience, Vol. 27, No. 6,* June 1997.

[9] D.C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *UW Computer Sciences Technical Report 1342,* June 1997.

[10] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symposium on Compiler Construction,* June 1982.

[11] B. Chandramouli, J. Carter, W. Hsieh, and S. McKee. A Cost Framework for Evaluating Integrated Restructuring Optimizations. *International Conference on Parallel Architectures and Compilation Techniques,* September 2001.

[12] C. Click. Combining Analyses, Combining Optimizations. *PHD Thesis, Rice University,* 1995.

[13] C. Click and K.D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems,* March 1995.

[14] S. Coleman and K.S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation,* June 1995.

[15] K. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing, vol. 23, no. 1, pp 7-22,* August 2002.

[16] E. Duesterwald. Meta-Analysis: a Unifying Framework for Optimizing Data Flow Analysis. *PhD thesis, University of Pittsburgh, 1996.*

[17] G.G. Fursin, M.F.P. O'Boyle and P.M.W. Knijnenburh. Evaluating Iterative Compilation. *Languages and Compilers for Parallel Computers*, 2003.

[18] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Behavior. *ACM Transactions on Programming Languages and Systems, 21(4): 703-746,* July 1999.

[19] T. Gross, D. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel & Distributed Technology, vol. 2, no 2, pp 16-26,* 1994.

[20] R. Gupta and R. Bodík. Register Pressure Sensitive Redundancy Elimination. *The 8th International Conference on Compiler Construction,* March 1999.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization,* December 2001.

[22] J. S. Hu, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, H. Saputra, and W. Zhang. Compiler-Directed Cache Polymorphism. *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems,* June 2002.

[23] C. Jaramillo, R. Gupta and M.L. Soffa. Comparison Checking: An approach to avoid debugging of optimized code. *Proceedings of Foundation of Software Engineering*, June 1999.

[24] R. Johnson and K. Pingali. Dependence-Based Program Analysis. *ACM Conference on Programming Language Design and Implementation,* June 1993.

[25] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers, Vol. 48, No. 2,* February 1999.

[26] T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle. Combined Selection of Tile Size and Unroll Factors Using Iterative Compilation. *International Conference on Parallel Architectures and Compilation Techniques,* October 2000.

[27] J. Knoop, O. Rüthing and B. Steffen. Lazy Code Motion. *In Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, June 1992.

[28] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. *Computational problems in abstract algebra, Pergamon Press*, 1970.

[29] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, D. Jones. Fast Searches for Effective Optimization Phase Sequences. *In Proceedings of SIGPLAN'04 Conference on Programming Language Design and Implementation,* June 2004.

[30] P. Kulkarni, D. B. Whalley, G. S. Tyson and J. W. Davidson. Exhaustive Optimization Phase Order Space Exploration. In Proceedings of International Symposium on Code Generation and Optimization, March 2006.

[31] P. Kulkarni, W. Zhao, H. Moon, K. Cho, and et al. Finding Effective Optimization Phase Sequence. In *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems,* June 2003.

[32] D. Lacey. Program Transformation using Temporal Logic Specifications. *PhD thesis, University of Oxford,* August 2003.

[33] D. Lacey, N. D. Jones, E. Wyk and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* January 2002.

[34] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *30th International Symposium on Microarchitecture (MICRO-30),* December 1997.

[35] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of compiler optimizations. *Proceedings of SIGPLAN'03 Conference on Programming Language Design and Implementation*, June 2003.

[36] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems, vol. 18, no. 4, pp 424-453,* July 1996.

[37] K. McKinley and O. Temam. A Quantitative Analysis of Loop Nest Locality. *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems,* October 1996.

[38] S.S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[39] George C. Necula. Translation validation for an optimizing compiler. *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, June, 2000.

[40] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems, Vol. 21, No. 5,* September 1999.

[41] L. Pollock and M.L. Soffa. An Incremental Version of Iterative Data Flow Analysis. *IEEE Transactions on Software Engineering, Vol. 15, No. 12,* December 1989.

[42] V. Sarkar. Automatic Selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM Journal of Research and Development,* May 1997.

[43] V. Sarkar and N. Megiddo. An Analytic Model for Loop Tiling and its Solution. *International Symposium on Performance Analysis of Systems and Software,* April 2000.

[44] D. Smith and Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization. URL: *http://www.eecs.harvard.edu/hube/software/nci/overview.html*

[45] O. Temam, C. Fricker and W. Jalby. Cache Interference Phenomena. *In Proceedings of SIGMETRICS Conference on Measurement and Modeling Computer Systems*, 1994.

[46] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler Optimization-space Exploration. *First International Symposium on Code Generation and Optimization*, March 2003.

[47] S. Triantafyllis, M. Vachharajani, and D.I. August. Compiler Optimization-space Exploration. *Journal of Instruction-Level Parallelism vol. 7, pp 1-25,* 2005.

[48] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. *Technical Report UT CS-97-366, LAPACK Working Note No. 131, University of Tennessee,* 1997.

[49] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems, vol. 13, no. 2, pp 181-210,* April 1991.

[50] D. Whitfield and M.L. Soffa. An Approach to Ordering optimizing transformations. *In Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, March 1990.

[51] D. Whitfield and M.L. Soffa. An Approach for Exploring Code Improving Transformations. *ACM Transactions on Programming Languages, vol. 19, no. 6, pp 1053-1084,* November 1997.

[52] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. *In Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* June 1991.

[53] M.E. Wolf, D.E. Maydan and D. Chen. Combining Loop Transformations Considering Caches and Scheduling. *29th Annual IEEE/ACM International Symposium on Mircoarchitecture,* December 1996.

[54] M. Wolfe. High Performance Compilers for Parallel Computing. *Addison-Wesley,* 1996.

[55] K. Yotov, X. Li, G. Ren, and M. Cibulskis. A Comparison of Empirical and Model-driven optimization. *In Proceedings of SIGPLAN'03 Conference on Programming Language Design and Implementation,* June 2003.

[56] W. Zhao, B. Cai, D. Whalley, M.W. Bailey, and et al. VISTA: A System for Interactive Code Improvement. *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems,* June 2002.