AUTOMATICALLY CONSTRUCTING COMPILER OPTIMIZATION HEURISTICS USING SUPERVISED LEARNING

A Dissertation Presented

by

JOHN CAVAZOS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2005

Department of Computer Science

© Copyright by John Cavazos 2005

All Rights Reserved

AUTOMATICALLY CONSTRUCTING COMPILER OPTIMIZATION HEURISTICS USING SUPERVISED LEARNING

A Dissertation Presented

by

JOHN CAVAZOS

Approved as to style and content by:

J. Eliot B. Moss, Chair

Andrew G. Barto, Member

Wayne P. Burleson, Member

Emery D. Berger, Member

Andrew G. Barto, Department Chair
Department of Computer Science

This dissertation is dedicated to my mom, Maria,
whose love and support made it possible.

ACKNOWLEDGMENTS

Eliot Moss has been a great thesis advisor. He has helped me to become a better researcher by shaping my critical thinking as well as by improving my expressive skills. I would like to thank the members of my thesis committee, Andy Barto, Emery Berger, and Wayne Burleson for their feedback and advice that helped to improve the overall quality of this dissertation.

I gratefully acknowledge the friendships and interactions from all members of the Architecture and Language Implementation group (ALI). Beginning with my first lab meeting talk, I have received helpful feedback on the best way to present myself and my work. The ongoing discussions in the lab helped to stimulate my research. Thanks especially to M. Tyler Maxwell for some of the amazing diagrams in this dissertation. Robbie Moll was helpful at stimulating my research interests in the applications of machine learning and for believing in me as an instructor.

I especially would like to acknowledge Emmanuel Agu, who has been a good friend and with whom I have had many rewarding discussions on research and life. Finally, I am extremely grateful for the love and support of my entire family. Overall, I am extremely lucky to be part of such a close and wonderful family. I would like to express my sincerest gratitude to my mother, Maria. As a young child I remember my mother always telling me that I could accomplish anything that I set my mind to. She was right as always. Her confidence in me gave me the strength both to overcome any difficulties and to maintain high goals.

This work was supported by National Physical Science Consortium and Lawrence Livermore National Laboratory.

ABSTRACT

AUTOMATICALLY CONSTRUCTING COMPILER OPTIMIZATION HEURISTICS USING SUPERVISED LEARNING

FEBRUARY 2005

JOHN CAVAZOS

B.S., TEXAS A&M UNIVERSITY-CORPUS CHRISTI

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Optimizing compilers use heuristics to control different aspects of compilation and to construct approximate solutions to hard compiler problems. Finding a heuristic that is effective for a broad range of applications is time-consuming and one of the most difficult tasks faced by compiler writers. In this dissertation, I introduce a new methodology called *LOCO* (Learning for Optimizing COmpilers) for constructing compiler heuristics automatically. In the context of Java JIT compilation, the efficiency of an optimization algorithm is important because time spent optimizing a program is part of the total running time of that program. We apply LOCO to construct new compiler heuristics that improve the efficiency of two classical compiler optimizations: instruction scheduling and register allocation.

Instruction scheduling is one of the most effective compiler optimization algorithms, sometimes improving program speed by 10% or more—but it can also be expensive. We

found that instruction scheduling often does not produce significant benefit and sometimes degrades speed. Using LOCO, we induced heuristics to predict which blocks benefit from scheduling. These ”filters” choose whether or not to schedule each block. Using filters, we can dramatically reduce scheduling effort while only slightly degrading the benefit of scheduling.

Like instruction scheduling, register allocation is an important and expensive optimization. However, unlike instruction scheduling, register allocation is a mandatory phase of compilation. We use LOCO to construct a ”hybrid” register allocator, that controls the application of two different register allocation algorithms. One algorithm is expensive, but sometimes more effective. The other is more efficient, but sometimes less effective. A hybrid allocator chooses which algorithm to apply for each method. Using hybrid allocators we can dramatically reduce register allocation effort while still obtaining much of the additional benefit of the more expensive algorithm.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . . **v**

ABSTRACT . . . . . . **vi**

LIST OF TABLES . . . . . . **xiii**

LIST OF FIGURES . . . . . . **xvi**

CHAPTER

1. INTRODUCTION **1**

1.1 My Thesis 1
1.2 The Problem Statement 1
1.3 State of the Art 2
1.4 Overview of the Solution 3
1.5 Contributions 5

1.5.1 Learning Whether to Optimize 7
1.5.2 Learning Which Algorithm To Use 8
1.5.3 Learning How to Optimize 9
1.5.4 Summary of Contributions 9

1.6 Organization 10

2. BACKGROUND **11**

2.1 Introduction 11
2.2 The Java Compilation Process 11

2.2.1 Jikes Research Virtual Machine (RVM) 12
2.2.2 Related Work on JVMs 14

2.3 Experimental Infrastructure 15

2.3.1 Benchmarks 16

2.4 Instruction Scheduling 17

2.4.1 Instruction Scheduling in Jikes RVM 18
2.4.2 Related Work on Instruction Scheduling 20

2.5 Register Allocation 22

2.5.1 Register Allocation and Spilling 23
2.5.2 Linear Scan Register Allocation 23
2.5.3 Graph Coloring Register Allocation 25
2.5.4 Liveness Analysis 26
2.5.5 Related Work on Register Allocation 26

2.6 Model and Empirical-driven Optimization Feedback 28

2.6.1 Basic Block Simulation 29

2.7 Machine Learning 31

2.7.1 Supervised Learning Algorithms 31
2.7.2 Overfitting, Pruning, and Stopping Criteria 34

3. THE LOCO METHODOLOGY 36

3.1 LOCO Methodology Description 36
3.2 Phrasing the Learning Problem 36
3.3 Feature Construction 38
3.4 Generating Training Instances 39

3.4.1 Threshold Parameters 40
3.4.2 Pruning Features and Training Data 41

3.5 Training the Learning Component 42
3.6 Integration and Evaluation of the Induced Heuristic 43
3.7 Key Properties 44
3.8 Conclusion 45

4. LEARNING WHETHER TO OPTIMIZE 46

4.1 Introduction 47
4.2 Problem and Approach 48

4.2.1 Features 48
4.2.2 Learning Methodology 50
4.2.3 The Learning Algorithm 51
4.2.4 Integration of the Induced Heuristic 51

4.3 Evaluation Methodology 52
4.4 Experimental infrastructure 53
4.5 Experimental Results 53

4.5.1 Classification Accuracy 54
4.5.2 Simulated Execution Times 54
4.5.3 Efficiency and Effectiveness 55
4.5.4 Thresholding 58
4.5.5 Filtering Applied to Other Benchmarks 61
4.5.6 A Sample Induced (Learned) Filter 61
4.5.7 The Cost of Evaluating Filters 64
4.5.8 Filtering with Adaptive Optimization 64
4.5.9 Time Compiling versus Time Running 67

4.6 Summary 68

5. LEARNING WHICH OPTIMIZATION ALGORITHM TO USE 71

5.1 Introduction 72
5.2 Motivation 73
5.3 Problem and Approach 73

5.3.1 Features 75
5.3.2 Learning Methodology 77
5.3.3 Thresholds 78
5.3.4 Integration of the Induced Heuristic 79

5.4 Experimental infrastructure 79
5.5 Evaluation Methodology 80
5.6 Experimental Results 81

5.6.1 Classification Accuracy 82
5.6.2 Spill Loads 82
5.6.3 Efficiency and Effectiveness 84
5.6.4 A Sample Induced (Learned) Heuristic 86
5.6.5 The Cost of Evaluating a Hybrid Heuristic 87
5.6.6 Hybrid Allocation with Optimization Level O1 87
5.6.7 Time Compiling versus Time Running 90

5.7 Summary 91
5.8 Related Work 92

6. LEARNING HOW TO OPTIMIZE 94

6.1 Introduction 94
6.2 The Problem 94

6.3 The Representation of Scheduling Preference 96
6.4 Features 96
6.5 Obtaining Examples and Counter-Examples 97
6.6 Learning Algorithms 98
6.7 Benchmarks 98
6.8 Empirical Results 99

6.8.1 Experimental Infrastructure 100
6.8.2 Experimental Procedures 101
6.8.3 Statistical Analysis Procedures and Results 102
6.8.4 A Sample (Induced) Heuristic for the Alpha 21064 106

6.9 Learning How to Schedule for Out-Of-Order Processors 106

6.9.1 Features 107
6.9.2 Results 108
6.9.3 A Sample (Induced) Heuristic for the PowerPC 7410 108

6.10 Related Work 109

6.10.1 Genetic Algorithms 110
6.10.2 Reinforcement Learning 111
6.10.3 Iterative Repair 113

6.11 Summary 114

7. RELATED WORK IN APPLYING LEARNING TO COMPILATION 116

8. DISCUSSION 120

8.1 Features and Heuristics: Simple and Cheap 121
8.2 Supervised Learning: Excellent Function Inducers 121
8.3 Compiler Optimizations 122
8.4 Speculations 123

9. CONCLUSIONS 124

9.1 Key Contributions 124

9.1.1 Learning Whether to Optimize 124
9.1.2 Learning Which Optimization to Use 125
9.1.3 Learning How to Optimize 125

9.2 Future Directions 125

APPENDICES

A. HYBRID ALLOCATOR RESULTS FOR OPTIMIZATION LEVEL O3 127
B. HYBRID ALLOCATOR RESULTS FOR OPTIMIZATION LEVEL O1 132

BIBLIOGRAPHY 137

LIST OF TABLES

Table **Page**

2.1 Characteristics of the SPECjvm98 benchmarks. 16

2.2 Characteristics of a set of benchmarks that benefit from scheduling. 17

4.1 Features of a basic block. 49

4.2 Classification error rates (percent misclassified) for different threshold values. 54

4.3 Predicted execution times for different threshold values. 56

4.4 Effect of *t* on training set size of SPECjvm98. NS is constant at 37280. 60

4.5 Effect of *t* on run time classification of blocks for SPECjvm98. 60

4.6 Induced Heuristic Generated By Ripper. 63

4.7 Cost breakdowns: gm = geometric mean; SB = scheduled blocks; SI = scheduled instructions; f = time to evaluate features and heuristic function; s = time spent in scheduling; c = compile time excluding scheduling. 65

4.8 Cost breakdowns for Pseudo-Adaptive runs: gm = geometric mean; SB = scheduled blocks; SI = scheduled instructions; f = time to evaluate features and heuristic function; s = time spent in scheduling; c = compile time excluding scheduling. 66

4.9 Application execution time ratio (versus no scheduling) for List and Pseudo-Adaptive runs, geometric mean across six benchmarks. 67

4.10 Percent of compile time recovered by each iteration of the application, and application time as percentage of compile time for NS. 68

4.11 Percent of Pseudo-Adaptive compile time recovered by each iteration of the application, and application time as percentage of compile time for NS. 69

5.1 Running time statistics for GC and LS 74

5.2 Features of a method. 76

5.3 Effect of different threshold values on training set size for SPECjvm98. 79

5.4 Classification error rates (percent misclassified) for different threshold values. 82

5.5 Spill loads for different hybrids and linear scan. 83

5.6 Induced Heuristic Generated By Ripper. 86

5.7 Cost breakdowns: GCM = GC allocated methods; GCI = GC allocated instructions; LSM = LS allocated method; LSI = LS allocated instructions; f = time to evaluate features and heuristic function; a = time spent allocating methods; c = compile time excluding allocation time. 88

5.8 Percent of O3 compile time recovered by each iteration of the application, and application time as percentage of compile time for LS. 90

5.9 Percent of O1 compile time recovered by each iteration of the application, and application time as percentage of compile time for LS. 91

6.1 Features for Instructions and Partial Schedule 97

6.2 Characteristics of SPEC95 FORTRAN benchmarks. 99

6.3 Characteristics of SPEC95 C benchmarks. 100

6.4 Normalized Measured Execution Times 103

6.5 Predicted Execution Time 104

6.6 Experimental Results: Optimal Choices in Small Blocks 105

6.7 Features for Instructions and Partial Schedule 107

6.8 LIST = list scheduling heuristic; RULE = induced scheduling heuristic. 108

LIST OF FIGURES

Figure **Page**

2.1 Jikes Research Virtual Machine (RVM) System. 13

2.2 High level view of List Scheduling 19

3.1 LOCO Methodology to Constructing Compiler Heuristics 37

3.2 Threshold Process 41

4.1 Efficiency and Effectiveness Using Filters. 57

4.2 Efficiency and Effectiveness Using Filter Thresholds. 59

4.3 Efficiency and Effectiveness Using Filters On Other Benchmarks. 62

5.1 Algorithm running time versus method size for GC and LS 74

5.2 Procedure for labeling instances with GC and LS 78

5.3 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3. 85

5.4 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1. 89

6.1 Normalized Measured Execution Times 103

6.2 Predicted Execution Time 104

8.1 Percentage of Time Spent in Compiler Optimizations 122

A.1 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (12 Regs). 128

A.2 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (16 Regs). 129

A.3 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (20 Regs). 130

A.4 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (24 Regs). 131

B.1 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (12 Regs). 133

B.2 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (16 Regs). 134

B.3 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (20 Regs). 135

B.4 Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (24 Regs). 136

CHAPTER 1

INTRODUCTION

1.1 My Thesis

The central thesis of this dissertation is: we can design a new technique for automatically constructing compiler optimization heuristics that has several benefits and no significant drawbacks over other methods of constructing heuristics.

1.2 The Problem Statement

Most compiler optimizations are controlled through functions called *heuristics*. A heuristic for a particular optimization uses different properties of the code being optimized and of the target architecture to control the optimization algorithm. Optimization heuristics make a difference as to whether the optimization improves or degrades the performance of a program. Unfortunately, constructing compiler heuristics is difficult and tedious.

Compiler writers are expected to find effective and efficient heuristics to compute approximate solutions to NP-hard problems, such as instruction scheduling and register allocation. Further complicating the construction of good heuristics is that other compiler phases may interact with the optimization adversely affecting its effectiveness. This makes it difficult to construct sets of compiler heuristics that mesh well together.

As computer architectures become increasingly complex, we need ever more sophisticated compiler optimizations to exploit new features of the architecture. Previous architectures were simple enough that it was easy to discern the relevant details that affect performance, but today's architectures are intractably complex. Also, the fast-paced development of new processor technology precludes spending a large amount of time hand-crafting

these optimizations. Hand-crafting back-end optimizations means that the compiler writer has less time to spend on other parts of the compiler. Further, computer engineers desire the ability to experiment with many different designs of an architecture to measure gains of architectural alternatives over one another. This requires not only quick prototyping of the architectures in question, but also quick prototyping of optimizing compilers for those architectures. Hand tuning any part of the compiler, including the back end, is ineffective at supporting quick prototyping of optimizing compilers.

Compiler writers not only must learn what features of the problem affect a particular optimization, they need to discover the relative importance of the features and the combinations of these features that work well. The compiler writer first decides upon a set of features they think are important to the problem and constructs an initial heuristic based on those features. The programmer then tests this heuristic by compiling a suite of benchmarks and measuring the performance of these benchmarks. If the performance after applying the optimization meets the compiler writer's desires, they stop iterating the loop and the heuristic can be included in the production version of the compiler. However, this feedback loop is typically iterated several times until either the compiler writers are satisfied with the performance of the heuristic or give up because they were unable to achieve acceptable performance.

Because today's architectures appear intractably complex and new architectures are rapidly being introduced, the timely construction of good heuristics is difficult. Techniques for automating and simplifying the process of constructing heuristics have become necessary.

1.3 State of the Art

Over the last ten years, new techniques have emerged that automate the process of constructing compiler heuristics, namely genetic algorithms [52, 21, 22, 28, 8] and reinforcement learning [36, 37]. These techniques work by performing a massive search over the

parameter space of features of a particular optimization heuristic. In effect, they automate the process of a heuristic construction feedback loop, blindly testing thousands of heuristics to find ones that perform well. These techniques have the advantage over manually constructing heuristics, that they automate the process of constructing and testing heuristics. However, there are several drawbacks to these approaches. The search can take hours or days, even on multi-processor machines. Additionally, because these techniques employ randomization in their search, they are not guaranteed to converge to the same solution every time. This can be frustrating to a compiler writer, who is looking for a stable heuristic. These systems are also hard to get working right and the form of the heuristics they produce can be hard to interpret. This thesis reports on a new technique that also automates the process of constructing compiler heuristics, but does not have these drawbacks.

1.4 Overview of the Solution

We introduce a new technique called *LOCO*, Learning for Optimizing COmpilers. LOCO uses *supervised learning* to automate and simplify the construction of compiler heuristics. Supervised learning refers to a wide variety of algorithms that can "induce" (automatically construct) general functions from training data.

With LOCO, we were successful at inducing heuristic functions for two back end compiler optimization algorithms, namely instruction scheduling and register allocation. Finding optimal solutions for these optimizations is also known to be NP-hard, so one requires heuristics to solve them efficiently. These compiler optimizations are among the most important compiler optimizations for most architectures. It is therefore essential that the heuristic be effective. Where possible we compare the heuristics produced by LOCO to hand-crafted heuristics, and found that they perform equally well.

LOCO also offers the following benefits over the traditional methods of manually constructing heuristics:

1. LOCO speeds up heuristic construction. Using supervised learning we can construct competent heuristics in a matter of minutes. On the other hand, manual construction of heuristics can take days and even weeks to construct and fine tune compiler heuristics.

2. LOCO discovers the relative importance of the different properties of an optimization problem automatically. In contrast, the manual technique of discovering what features is done through tedious and complicated trial-and-error experimentation.

3. LOCO's heuristic inducing algorithms have a firm theoretical foundations. Thus, the relative importance of features used in the heuristics is developed using sound principles. The relative importance of features in manually constructed heuristics is often derived through ad hoc experimentation, often based on intuition or inadequate fine-tuning, which can lead to sub-par effectiveness of an optimization.

LOCO has the following important advantages over other automated techniques of constructing heuristics:

1. LOCO is more efficient at constructing compiler heuristics than these other techniques. As mentioned, LOCO can construct heuristics in minutes. Other automated techniques can take hours to days, even on multi-processor machines.

2. LOCO is easier to use successfully than these techniques. The supervised learning algorithms used in LOCO are easily tuned to the particular problem. These techniques are more difficult to get working and often require expert tuning.

3. LOCO produces heuristics that are more readable. Unsupervised learning algorithms and reinforcement learning algorithms use representations that are generally less readable.

4. LOCO's learning algorithms always produce the same result for the same set of training data. The other automated techniques use randomization, thus the output of their searches can differ from one run to the next.

The other automated techniques are effective in certain situations, such as when the underlying architecture is constantly changing, thus requiring an adaptive learning solution. However, in most situations, LOCO is suitable for constructing compiler heuristics and the other methods should be used only after attempts to use LOCO are unsuccessful.

1.5 Contributions

Heuristics are found throughout compilers. They control how a program is compiled—controlling everything from the order in which to apply certain optimizations, to whether an optimization is applied at all, or even how an optimization is applied.

I describe four different categories of compiler optimization heuristics. We used LOCO to construct compiler heuristics for three of the four categories of this taxonomy, and it should be applicable to the fourth as well (left for future work).

The first category of heuristic controls *whether* to apply an optimization or not. Typically a compiler writer decides a priori whether or not to apply an optimization to all parts of a program or to none. However, an optimization can have a large impact on some parts of a program while having little or no impact on other parts. If an optimization is expensive and only partially effective, it would be useful to apply it selectively. We developed a novel technique, called *filtering*, that uses features of the code to predict whether to apply an optimization. Instruction scheduling is one optimization, in particular, that can be expensive to apply and effective only some of the time. We developed filter heuristics to control *whether* to apply instruction scheduling. Filters uses features of a block to predict whether the block will benefit from instruction scheduling.

The second category of heuristics are ones that control *which* optimization algorithm to apply. This kind of decision is typically decided beforehand and often hard-coded into

the compiler. Instead we used LOCO to construct heuristics for a new kind of optimization strategy, which we call *hybrid optimization*. A hybrid optimization chooses among different implementations of the same optimization to find the best tradeoff between effectiveness and efficiency of those various implementations.

The third category of heuristics are ones that control decisions within an optimization algorithm. These decision functions are sometimes called *priority functions* because they prioritize different properties of interest. Some researchers have applied machine learning to construct these kinds of heuristics automatically. We discuss this related work in Chapter 7. We used LOCO to construct heuristics to control how to schedule instructions in an instruction scheduler.

Finally, the fourth category of heuristic controls the ordering of transformations in a compiler. This heuristic is referred to as a *phase-ordering* heuristic in the compiler literature. Cooper *et al.* [21] successfully apply genetic algorithms to this class of heuristics. It would be interesting to apply LOCO to this class of heuristics, but this is left to future work.

Compilers also require optimizations that are *efficient*. Efficient optimizations are ones that consume little resources to perform. Efficient optimizations are especially important in *just-in-time* (JIT) compilers because they compile programs at run time, thus the time spent compiling and optimizing a program is a percentage of the total running time of the program. In JIT compilers, it is essential that optimizations not only produce good quality code, but also execute as fast as possible.

Most previous work in compiler optimization research has focused on the effectiveness of compiler optimizations. In contrast, the efficiency of optimizations has received little attention. As JIT compilers have become more popular, some research in efficient optimizations has begun to emerge [56, 41, 18, 16]. Still, there remains an important gap in the study of efficient optimization algorithms. The first two categories of heuristics control compiler optimizations to improve an optimization's efficiency.

The first three categories of heuristics use models to evaluate their effectiveness. These models are used in the construction of learning data. Models are simplified estimators of a program's performance on a real machine. We show that a technique, we call *thresholding*, is essential to inducing the best performing heuristics.

1.5.1 Learning Whether to Optimize

We use LOCO to construct heuristics for *filtering*, a new technique for controlling whether to apply compiler optimizations.

Optimizations are most commonly applied in an all-or-nothing fashion—either they are always applied or not at all. However, an optimization doesn't always have a positive impact on a program's performance, and in some cases can have a negative impact. There may be parts of a program where an optimization will have a positive effect and other parts of the program where it will have negative or no effect. Applying an optimization when it produces no benefit unnecessarily increases compilation time.

If applying an optimization is costly and not always beneficial, a program's compilation time (and therefore a program's running time under a JIT compiler) could be reduced by applying the optimization *selectively*. Thus, we desire the ability to predict effectively which parts of a program will benefit from an optimization, and then apply the optimization only to those parts. This will reduce compilation time while still retaining most or all of the benefit of an optimization.

We use filtering to control the application of instruction scheduling. Filtering uses features of the block of code about to be scheduled to predict whether scheduling will benefit that piece of code, and schedules the code only if it is deemed beneficial.

Instruction scheduling is an expensive optimization, the overall performance of which is usually dominated by the time spent constructing the data structures required to perform scheduling [40] and by the costly modeling of the target processor's timing. We constructed cheap-to-compute features that we believed were predictive of whether scheduling would

benefit a block of code. We then used LOCO to construct our filter heuristics. Then we used the resultant filters to select when to apply instruction scheduling, namely to those blocks the filter predicted would benefit from scheduling.

We used filtering to control instruction scheduling for several Java benchmarks. We were able to reduce scheduling time by as much as 75% while still retaining most of the benefit of always scheduling.

1.5.2 Learning Which Algorithm To Use

We also used LOCO to construct heuristics automatically for a new compiler optimization that we call hybrid optimization.

For many optimization, there are several different algorithms that differ in efficiency and effectiveness. Some of these algorithms are more efficient, but not always as effective as their slower counterparts. For some parts of a program, the faster, less effective optimization algorithm are often sufficient, but for other parts, we would achieve a significant benefit by applying a slower, but more effective algorithm. It would be desirable to apply the slow algorithm selectively only on the parts of the program that will benefit from it. We developed predictive heuristics that chooses when to apply the slower more effective algorithm over the faster, less effective one.

Specifically, we develop a hybrid allocator that chooses between two register allocation algorithms, *graph coloring* and *linear scan*. Graph coloring is an effective allocation algorithm, but can very expensive. On the other hand, linear scan, while often effective, is sometimes less effective than graph coloring, but is generally more efficient(in one compiler more than 80% faster).

We used LOCO to construct heuristics that control our hybrid register allocator. This hybrid allocator chooses between an implementation of graph coloring and an implementation of linear scan. Our hybrid allocator was able to reduce the running time of register

allocation so that it is competitive with the faster linear scan algorithm, while still achieving much of the effectiveness of graph coloring.

1.5.3 Learning How to Optimize

Finally, we use LOCO to construct heuristics for a popular instruction scheduling algorithm called *list scheduling*. The heuristic in list scheduling dictates the order in which to add available instructions to a schedule under construction. List scheduling has been well-studied for several decades and there are now excellent heuristics for this algorithm. Stefanović [51] has shown that one popular heuristic used for list scheduling is almost always optimal for blocks that he could feasibly explore for its optimality. Using LOCO, we induced heuristics automatically for list scheduling, whose performance was comparable to that of hand-tuned list scheduling heuristics for two different architectures.

1.5.4 Summary of Contributions

We summarize our contributions:

1. We introduce a technique for automatically constructing compiler optimization heuristics that has many benefits over traditional and state of the art methods of constructing heuristics.

2. We introduce a new optimization technique, called *filtering*, that significantly reduces the effort of instruction scheduling while still obtaining most of the benefit.

3. We introduce a new optimization technique, called *hybrid optimization*, that automatically chooses between two different register allocation algorithms. By using hybrid optimization, we achieve most of the efficiency of the faster linear scan register allocation algorithm, while still obtaining much of the benefit of the slower but more effective graph coloring algorithm.

4. We show that LOCO can induce heuristics automatically that are comparable in effectiveness as hand-tuned heuristics.

5. We show that a technique, we call *thresholding* is essential to obtaining the best performing heuristics out of LOCO.

1.6 Organization

We organize the remainder of this dissertation as follows. Chapter 2 provides background information. In the first part, it presents the JIT compilation model that is the foundation of today's typical Java JIT compilers, and the two back end compiler algorithms we worked with, instruction scheduling and register allocation. In the second part, it discusses supervised learning algorithms in general, and also the specific algorithms we used.

Chapter 3 describes the main components of our solution, called LOCO, and gives an example to illustrate the technique. It compares that solution to traditional methods as well as new state of the art methods aiming to provide similar services to heuristic construction.

Chapters 4, 5, and 6 describe three important heuristics we developed using LOCO: (1) whether to optimize, (2) which optimization algorithm to use, and (3) how to optimize. Our solution is the first one automatically and quickly to induce effective and efficient heuristics for these three problems.

Finally, Chapter 9 concludes the dissertation, discusses the limitations of our work, and suggests directions for future work.

CHAPTER 2

BACKGROUND

2.1 Introduction

Because this research is a synthesis of two different areas of computer science, compilers and machine learning, we give background in both of these areas to help in understanding the research described in this dissertation. We describe the Java compilation process and then the Jikes Research Virtual Machine (RVM), the particular system we used for this research. We also describe instruction scheduling and register allocation, the two optimizations we used for our experiments. Finally, we describe machine learning algorithms in general and elaborate on the specific learning algorithms we used. We discuss related work pertaining to these different areas throughout this chapter. Chapter 7 discusses related work pertaining to machine learning applied to compiler heuristics.

2.2 The Java Compilation Process

Programs written in Java are first compiled into machine-independent bytecodes using a Java bytecode compiler, such as *javac*. These bytecodes are stored in class files to be later read by a *Java Virtual Machine* (JVM). The process of executing a Java method is as follows: A JVM reads in the bytecodes for each method as each method is invoked. The JVM transforms the bytecodes into an *intermediate representation* (IR) and a variety of optimizations are applied. The IR is then transformed into assembly code which is then transformed into machine code and executed. The core of a JVM implementation contains an execution engine that executes the bytecodes. There are two popular ways of implementing the execution engine, interpretation and *Just-In-Time* (JIT) compilation.

In interpretation, a loop repeatedly fetches, decodes, and executes the next bytecode. The interpreter has code to interpret, i.e., to accomplish the effect of each form of bytecode instruction. Many Java JIT systems provide an interpreted mode as the initial mode of execution or for infrequently invoked methods. For *hot* (frequently executed) methods, typically JVMs also provide a JIT that compiles those methods to native codes, possibly with optimizations. Jikes RVM is unusual among JVMs in that it does not include a bytecode interpreter, but always compiles to native code any method that gets invoked. Therefore, we do not discuss interpretation further.

2.2.1 Jikes Research Virtual Machine (RVM)

Jikes RVM, formerly known as Jalapeño, is a Java virtual machine (JVM) for servers, written in the Java language [4, 13, 3, 2]. As previously mentioned, Jikes RVM takes a *compile-only* approach to program execution. That is, the first time a method is invoked, Jikes RVM compiles it into executable machine code. It then caches the method's compiled code so that it can execute this code on subsequent invocations of that method. Jikes RVM currently runs on the PowerPC and Intel x86 architectures.

Jikes RVM currently offers two compilers: the *Baseline* compiler, which makes little attempt to produce high quality code, but runs very quickly, and the *Opt* compiler, which has a large number of optimizations available to be applied. We are only interested in the optimization compiler so we do not discuss the *Baseline* compiler further.

When a method is *Opt* compiled, Jikes RVM converts its bytecodes to an *internal representation* (IR). An IR pertains to the particular data structures and techniques for representing each Java method being compiled. Jikes RVM has multiple IRs that are used to support various levels of optimizations. Figure 2.1 depicts the three different phases in the *Opt* compiler and some of the different optimizations that can be applied in each phase. During each phase, the code is represented by a different IR, though these IRs are similar. Each phase applies optimizations and continually lowers the IR from a high-level IR (HIR)

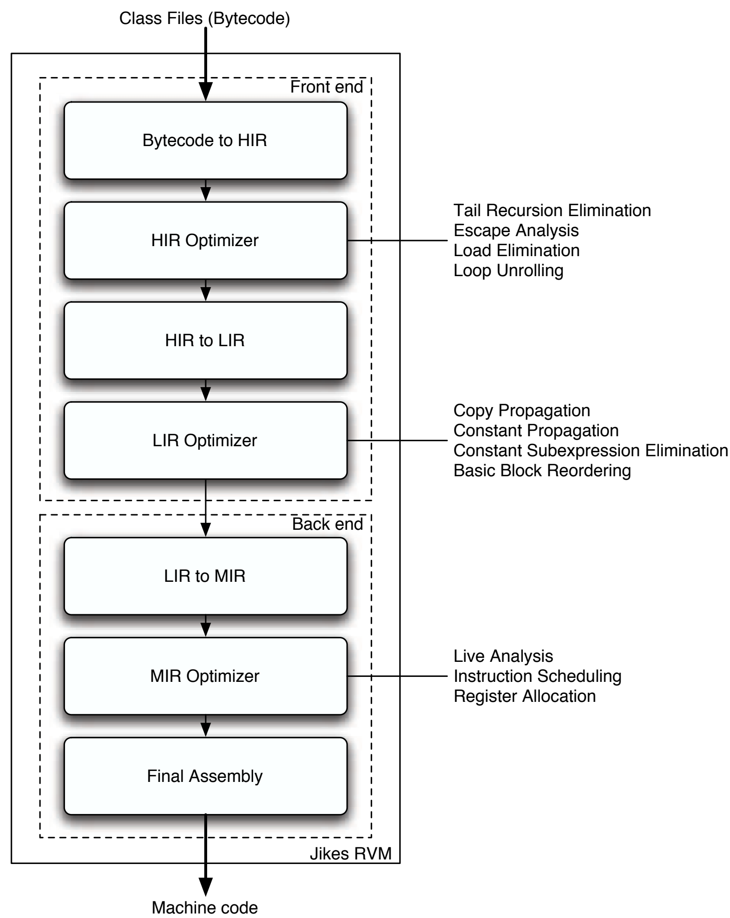


Figure 2.1. Jikes Research Virtual Machine (RVM) System.

to a low-level IR (LIR), and finally to a machine-level IR (MIR). The research in this dissertation involves optimizations applied in the final phase of compilation while the code is in MIR.[1] Even though this research restricts itself to the last phase of compilation, we believe the concepts and techniques presented here are relevant to optimizations applied in other phases.

Later we offer some comparison with compilation techniques that identify and optimize only frequently executed (*hot*) methods.

2.2.2 Related Work on JVMs

The HotSpot JVM [53] begins by interpreting all code, but it monitors the execution of that code. Most programs spend a large percentage of their time executing a small percentage of the code. By monitoring program execution, HotSpot can figure out which methods represent the program's "hot spots"- that small percentage of code that is executed most of the time. When HotSpot decides that a particular method is a hot spot, it fires off a background thread that compiles those bytecodes to native code and heavily optimizes that native code. Meanwhile, the program can still execute that method by interpreting its bytecodes. Because the program isn't held up and because HotSpot is only compiling and optimizing the "hot spots" (perhaps 10 to 20 percent of the code), HotSpot has more time than a JIT only system does to perform optimizations.

BEA WebLogic JRockit [7] is a JVM that operates in a similar fashion to HotSpot with its selective optimizations of "hot spots." We discuss the research JRockit later in Chapter 7. Here we discuss its compilation process. JRockit continuously monitors applications and uses the application's ongoing characteristics to adapt and upgrade performance. Its JIT compiles all methods it encounters during startup, continuously optimizing application code at run time to improve performance. A "bottleneck detector" runs in the background

[1]This fi nal phase is generally known as the *back-end* and the optimizations applied during this phase are called *back-end optimizations*.

and collects run time statistics to detect frequently executed methods. Then the system aggressively optimizes these methods even as the system runs to deliver on-the-spot improvements.

OpenJIT [35] is a reflective Just-In-Time compiler, allowing various language extension and compiler optimizations to be added in as Plug-ins written in Java. OpenJIT allows customization of the JIT compiler for dynamic adaptation to the computing environment. Similar to Jikes RVM, almost all of OpenJIT is written in Java, save for a small amount of native code for runtime support and adding APIs to the JVM. The back end of OpenJIT uses the JIT interface API to convert the Java bytecodes into SPARC V8 native code, and controls the execution between the JDK and the native code. The back end currently does not perform extensive optimizations performing only *economical* optimizations such as good code generation and effective peephole optimizations.

2.3 Experimental Infrastructure

We ran the experiments in this dissertation on two different platforms. We ran experiments for Chapters 4 and 5 primarily on an Apple PowerPC Macintosh system with two 533 MHz G4 processors, model 7410. This is an aggressive superscalar architecture and represents the current state of the art in processor implementations.

For the PowerPC, all measurements are elapsed (wall clock) times. The compiler infrastructure also measures elapsed time spent in the compiler, broken down by phase and individual optimization. These measurements use the bus clock rate time counter and thus give sub-microsecond accuracy; this clock register is also cheap to read, so there is little overhead in collecting the information.

For instruction scheduling, the 7410 *implementation* of the PowerPC is interestingly complex, having two dissimilar integer functional units and one each of the following functional units: floating point, branch, load/store, and system (handles special system instruc-

Program	Description
`compress`	Java version of 129.compress from SPEC 95
`jess`	Java expert system shell
`db`	Builds and operates on an in-memory database
`javac`	Java source to bytecode compiler in JDK 1.0.2
`mpegaudio`	Decodes an MPEG-3 audio file
`raytrace`	A raytracer working on a scene with a dinosaur
`jack`	A Java parser generator with lexical analysis

Table 2.1. Characteristics of the SPECjvm98 benchmarks.

tions). It can issue one branch and two non-branch instructions per cycle, if a complicated set of conditions holds. Instructions take from one to many tens of cycles to execute.

In Chapter 6 we report on experiments on instruction scheduling heuristics done on an Alpha 21064 (in-order issue) processor. It is widely thought that instruction scheduling has more effect on in-order processors, so it makes sense to report numbers for both an in-order and an out-of-order processor.

2.3.1 Benchmarks

We examine 7 programs drawn from the SPECjvm98 suite [50] for most of the experiments in this dissertation. We detailed these benchmarks in Table 2.1. We ran these benchmarks with the largest data set size (called 100).

For Chapter 4 we gathered another suite of programs where scheduling gives more of an improvement in running time. Table 2.2 offers details about these benchmarks. Note that this suite of benchmarks consists solely of numerically intensive (floating-point) computations. For this architecture, instruction scheduling is an important optimization for removing stalls caused by floating point instructions having long latencies.

Program	Description
`linpack`	A numerically intensive floating point kernel
`power`	Power pricing optimization problem solver
`bh`	Barnes and Hut N-body physics algorithm
`voronoi`	Computes Voronoi diagram of a set of points recursively on the tree
`aes`	Tests vectors from the NIST encryption tests
`scimark`	A scientific and numerical computation

Table 2.2. Characteristics of a set of benchmarks that benefit from scheduling.

In Chapter 6 we develop the heuristic used for deciding how to schedule instructions. In this chapter, we examine C and FORTRAN programs from the SPEC95 benchmark suite. Since these benchmarks were only used in this chapter, we describe these benchmarks in more detail there.

2.4 Instruction Scheduling

In Chapters 4 and 6, we experiment with instruction scheduling. Instruction scheduling is used to tolerate instructions with a latency of more than one cycle. *Latency* refers to the number of clock cycles after an instruction has begun execution until its result can be used. The latency of memory and floating point instructions is often multiple cycles, and can cause the processor to stall if instructions follow too closely that depend on their results or use the same resources. We tolerate long latencies by scheduling other instructions between the issue of a *long latency* instruction and the issue of any dependent instructions.

Scheduling reorders instructions in a block. However, two instructions cannot be reordered if there is a dependence relation between them. There are several different types

of dependence relations including dependences on a register, on memory, control flow, and possible exception conditions. The appendix discusses these dependences in more detail.

Compilers represent this dependence information in a directed acyclic graph (DAG); each instruction is a node in the graph, and each dependence relationship is an edge in the graph. If node j depends on node i, there must be an edge in the DAG from i to j. Node i is called a predecessor of j and node j is called the successor of i. Nodes in the DAG that do not have predecessors are called *root* nodes and nodes that do not have successors are called *leaf* nodes. Edges are assigned weights, giving to the latency of the functional units for register def-use edges and memory true dependence edges; otherwise the weight is zero. These weights give an estimate of how long the dependent instruction will have to wait before it can execute after its predecessor has been issued.

2.4.1 Instruction Scheduling in Jikes RVM

Here we are concerned with *prepass* instruction scheduling, which takes place after the code has been transformed into MIR (machine-dependent IR) but before final register assignments have been made. After register allocation another round of scheduling is sometimes performed (called *postpass* scheduling), but this is currently not supported by Jikes RVM. A previous study [33] has shown that prepass scheduling extracts more instruction-level parallelism than postpass scheduling on wide-issue machines.

The scheduler we are working with in Jikes RVM is a *top-down* scheduler. A top-down scheduler works by constructing a dependence DAG, then considering first the instructions that are roots, working down the DAG towards the leaves. A bottom-up scheduler works in the opposite direction. These algorithms can produce different results. The top-down approach is conceptually easier to understand and is the more common implementation.

The traditional method of performing instruction scheduling, and the method used in this research, is called *list scheduling*. The algorithm starts by constructing the dependence DAG. Each node (instruction) in the DAG is given a priority; there are a number of ways

```
listScheduler(block)
    Ready ← DAG roots
    While (Ready not empty)
    // i will contain best candidate so far
        i := instruction in Ready;
        for (j in Ready) do
            if (prefer-candidate(j, i))
                // j preferred over i
                i := j;
            endif
        endfor
        schedule(i)
        remove i from Ready
        Ready ← any new candidates
    end
```

Figure 2.2. High level view of List Scheduling

to assign these priorities. Instructions are scheduled only when all their predecessors in the DAG have been scheduled. The instructions that are available to be scheduled are kept in a *ready list*. The roots of the DAG are used to initialize the ready list. At each step, the list scheduling algorithm evaluates the quality of each instruction in the ready list based on some *preference function*, then chooses to schedule next the one with the highest priority. When there is more than one instruction with the highest priority, some disambiguation rule is used.

Also, we are concerned with *local* or basic block scheduling. Basic blocks are straight-line sequences of code with a single entry point and exit point. Local list scheduling has been the dominant instruction scheduling algorithm for 20 years. It is a greedy, heuristic, local technique that works well for many processors. There are other scheduling algorithms that employ backtracking [1], but these algorithms are expensive and would typically not be appropriate for JIT environments.

As mentioned previously, we apply our technique to local (basic block) scheduling, not global scheduling. We have investigated superblock scheduling in our compiler setting, and

with it one can get slight (1-2%) additional improvement over local scheduling. However, superblock formation requires detailed profiling information and we did not want to require that. Also, it is in a way beside the point: we are *not* trying to build a better scheduler, but trying to decide whether to apply whatever scheduler we have. We could apply our same procedure to the superblock case, and it might provide additional evidence that we can induce heuristics that greatly reduce scheduling effort while preserving most of the benefit.

Instruction scheduling is a well-known issue and others have proposed a variety of heuristics and techniques. It is also known that optimal instruction scheduling for today's complex processors is generally NP-hard. There has been some recent work on trying to find optimal schedules [62] using integer programming. The scheduling problem is posed as an integer programming problem and it is solved using an off-the-shelf integer programming package. Again, these techniques are expensive and not practical for most JIT compilation systems.

We are interested in Java compilation environments that strive for for efficient JIT compiler optimization. In Chapter 6 we show how to construct heuristics automatically that are as efficient and effective as hand-tuned heuristics. Chapter 4 focuses on drastically improving the efficiency of an instruction scheduler, while not degrading the effectiveness much.

2.4.2 Related Work on Instruction Scheduling

This section describes some prior work on instruction scheduling. We describe related work on applying machine learning to instruction scheduling in Section 6.10.

Muchnick [40] presents a comprehensive look at the design of real-world compilers for a broad range of architectures. He presents the design of several categories of both local and global instruction schedulers. He devotes one chapter to code scheduling. In the beginning of this chapter he states:

“methods for scheduling or reordering instructions to improve performance, an optimization that is among the most important for most programs on most machines.”

This statement lends further support to the importance of constructing good heuristics for this optimization.

A classic paper by Gibbons *et al.* [26] describes an algorithm for instruction scheduling for a pipelined architecture that significantly reduces the number of run time pipeline stalls. The algorithm is effective in practice while having an $O(n^2)$ worst-case run time.

A survey paper by Smotherman *et al.* [47] describes several categories of heuristics and DAG construction techniques for instruction scheduling. The heuristics are categorized based on different features, such as difficulty of implementation, what kind of pass (forward or backward) through the block is required to compute the heuristic, and how costly the heuristic is to compute.

Joseph Fisher [25] introduced a global scheduling technique called *trace scheduling*. John Ellis [24] also experimented with trace scheduling and showed its utility in a static compiler called Bulldog. In trace scheduling, a function is divided into a set of *traces*. These traces represent the most frequently executed loop-free paths through the function, and are treated like large basic blocks during scheduling. Instructions are scheduled within each trace ignoring control-flow transitions. After scheduling, some tricky adjustments are required (compensation code) to ensure the correct execution of off-trace code.

Richard E. Hank *et al.* [29] introduce another technique for global scheduling, called superblock scheduling, that reduces the amount of bookkeeping required compared with trace scheduling. A superblock is a trace which has no side entrances. Control may enter only from the top, but may leave at one or more exit points. Eliminating side entrances simplifies the bookkeeping needed to ensure the correctness of off-trace code. The construction of superblocks consists of two steps. First, traces are identified using static program analysis or profile information as in trace scheduling. Second, a process called tail duplication is performed to eliminate any side entrances to the trace. A copy is made of the tail portion

of the trace from the first side entrance to the end. All those side entrances into the trace are then redirected to the corresponding duplicate basic block. Duplicating code can lead to greater opportunity for instruction-level parallelism (ILP) at the expense of increasing static code size.

2.5 Register Allocation

Instructions that access operands in registers are usually cheaper than those that access operands from memory. Also (especially in RISC architectures), many instructions cannot access operands from memory. In either case, it is usually advantageous to have the most frequently used operands in registers. However, there are often a smaller number of available registers than there are candidates to place in registers. Register allocation is the phase of a compiler that specifies which candidates will go into registers and at what points in a piece of code. It is easy to estimate the benefit of allocating a value to a register; however, it is difficult to estimate how a specific allocation will affect subsequent allocations. Most previous approaches have focused on casting register allocation as a graph coloring problem. Because of the expense of building and coloring a graph and because of the current popularity of generating code at run time (due in part to Java JIT compilers) there has been a recent trend in faster ways of allocating registers, such as linear-scan allocation. This register allocation algorithm is sometimes less effective than graph coloring, but is usually more efficient, since it does not require an interference graph for its execution. We present results in Chapter 5 that show the performance difference between graph coloring and linear scan register allocation algorithms for a small register set size (8 registers) on two different optimization levels. The appendix contains more results for different register set sizes.

2.5.1 Register Allocation and Spilling

Jikes RVM performs register allocation for each method, using intra-procedural allocation.[2] Before register allocation the intermediate representation uses an unbounded number of registers, called *symbolic registers*. The task of the register allocator is to pack symbolics into *available machine registers*, while satisfying some important constraints:

- If two symbolics are live at the same time, i.e., both have been loaded or computed and both have possible future uses, then the allocator must assign them to different machine registers.
- Integers and pointers must go into integer registers and floating point numbers into floating point registers.
- Some values are bound to specific machine registers, notably arguments to a method and the method's result if any. Also, some registers are reserved for special use (e.g., the frame pointer).

It is not always possible to assign every symbolic to a register while satisfying these constraints. In that case some symbolics are *spilled*, i.e., assigned a location in memory. Spilled symbolics must be fetched from memory (into one of two registers reserved specifically for holding spilled operands) each time they are used, and immediately written to memory each time their value is updated.

2.5.2 Linear Scan Register Allocation

Linear scan register allocation [41] is part of the standard Jikes RVM optimizing (Opt) compiler. The basic algorithm processes the code of each method as follows.[3] First it determines, for each symbolic register, the instructions at which that register is live. We define the *live range* of a symbolic to be from the first instruction (in linear sequence) at which the symbolic is live to the last instruction at which it is live. Next, the algorithm scans

---

[2] This is often called "global" allocation in the literature, to distinguish it from "local" allocation, which treats each basic block separately. However, now, since some compilers perform inter-procedural or even whole-program allocation, "global" might be confusing.

[3] Linear scan requires a linear form of the code, as opposed to, say, a control flow graph.

the instructions of the method from beginning to end, maintaining a set of live symbolics as it goes. (This set is initially empty.) At a given instruction, if the live range of any symbolic ends, it drops that symbolic from the live set, and any register allocated to that symbolic is made *free*. Then, if any symbolic's live range begins at the instruction, the algorithm attempts to allocate a suitable register to that symbolic.

If a symbolic's live range includes a call, then we will not allocate that symbolic to a volatile (caller-save) register (one the callee is allowed to overwrite without preserving). Given this constraint, the algorithm determines, based on the current live set of symbolics and the registers allocated to them, whether there is a suitable register available for the new symbolic. If there is, the algorithm allocates that register to the symbolic, adds the symbolic to the live set, and continues.

If there is no suitable register available, then the algorithm must spill some symbolic. We describe this process with the following pseudo-code:

```
// s will contain best spill candidate so far
s := new-symbolic;
for (o in live-set) do
    if (prefer-to-spill(o, s))
        // gives o's register to s
        register(s) := register(o);
        s := o;
spill(s);
if (s != new-symbolic)
    add new-symbolic to live-set;
    remove s from live-set;
```

The `prefer-to-spill` call in the pseudo-code marks where we use a *spill heuristic*.

2.5.3 Graph Coloring Register Allocation

In addition to the linear scan algorithm already available in Jikes RVM, we implemented a version of Brigg's graph coloring algorithm [12]. This works as follows. First the algorithm builds an *interference graph*. This has one node for each symbolic register, and an undirected edge between each pair of symbolics that are live at the same time. The goal is to assign registers ("colors") to the nodes in such a way that the same color does not appear at both ends of any edge in the interference graph.

Given the interference graph, the algorithm makes passes over the graph, removing certain nodes (and the edges incident to those nodes). In particular, it removes a node if the node's degree (number of incident edges) is less than the number of registers suitable for that node. Note that such a node can *always* be assigned a non-conflicting register (color). When the algorithm removes a node in this way, it pushes it on a *coloring stack*. This overall process is called *simplifying* the interference graph.

It is possible that simplification produces an empty graph, in which case one moves on to coloring. But if simplification leaves a non-empty graph, then every node in that graph has degree higher than the number of registers suitable for that node, so some node must be spilled. We apply the spilling pseudo-code shown for linear scan, with new-symbolic being an arbitrary node in the remaining interference graph, and live-set being the rest of the nodes. Once we choose a node, we mark it for spilling and permanently remove it from the graph. We then iterate the entire process by rebuilding the graph without any nodes marked as spilled. We then attempt to simplify the graph again.

Eventually, the interference graph is empty and we proceed to *coloring*. In this phase the algorithm pops from the coloring stack each node that it pushed (in the opposite order, of course) and colors the node (i.e., assigns it a suitable register). By design of the algorithm a suitable register will always be available. As it pops each node, it adds it back to the interference graph, along with any of its incident edges that connect to nodes currently in

the graph. It uses the edges to check for conflicting registers and thus determine which available register to allocate to the node.

2.5.4 Liveness Analysis

Without getting into low-level implementation details of the algorithms, we observe that both linear scan and graph coloring require calculation of where each symbolic register is *live*. A symbolic is live at a program point if there is an execution path forward from that point on which the symbolic is used before it is assigned. Liveness is a data flow problem, readily solved with well known techniques. Solving the data flow problem may be computationally expensive, but both algorithms pay this cost, and in Jikes RVM it is calculated in another part of the compiler for other purposes, so we do not include its computation costs in any of our register allocation measurements.

We also do not include the cost of computing some of the information used in some of the heuristics, such as loop nesting depth. Again, this is computed elsewhere in the compiler for other purposes, so we do not charge "extra" for using it in register allocation spill heuristics.

2.5.5 Related Work on Register Allocation

The first to implement a graph coloring algorithm to solve register allocation was Chaitin [15]. The approach builds an interference graph that represents the interferences or overlaps between the live ranges of the variables in the code for which one is allocating registers. The live range of a variable is the contiguous region of the program where definitions flow to uses of that live variable. Formally, a variable, V, is live at point P if there exists a control path that includes P consisting of a definition of V before P which reaches a use of V after P. The algorithm then tries to find a k-coloring for the interference graph where k represents the number of registers available on the processor being targeted. Determining whether a k-coloring exists is known to be an NP-complete problem for $k \geq 3$, therefore Chaitin's algorithm, as in all global register allocation algorithms, uses heuristics

to find a k-coloring or to change the graph in order to make it k-colorable. The algorithm proceeds by repeatedly removing all those nodes (and their corresponding edges) that have degree less than k and placing them in a stack to be colored later. If nodes remain after this graph reduction phase, the graph cannot easily be k-colored. The algorithm therefore *spills*, i.e., copies values to memory, one or more live ranges in order to transform the graph into one that has a simple k-coloring. Thus, each spill causes a successive graph to be built. The cost of the algorithm is dominated by the building and processing of these successive graphs, which is in the worst case quadratic in the number of register candidates.

Briggs *et al.* [12] use the same order as Chaitin to remove nodes, but they continue removing nodes and pushing them on the stack even if they have $> k$ neighbors. This optimistic algorithm hopes some neighbors will get the same color so that this node can still trivially get a color. During coloring if there is not a color available (for a node with $> k$ neighbors), Briggs *et al.* spill, rebuild the interference graph, and try again. It has the same worst case complexity as Chaitin but tends to perform better in practice.

Recently Omri Traub *et al.* [56] formulated the register allocation problem as a bin-packing problem. This algorithm allocates registers for a code sequence by traversing the sequence in a linear scan. When a temporary variable is encountered it is placed in a bin. The constraint on a bin is that it can contain only one valid value at any given point in a program's execution. If a new temporary is encountered and all bins are full, one of the values in a bin must be spilled. The spilled value marks a split in the temporary's live range. Picking a spill candidate employs a heuristic that looks at the distance to the candidate's next reference. This algorithm improves on a previous bin-packing allocation implementation [10] by being able to allocate and rewrite the instruction stream in a single pass.

Vegdahl [61] describes a modification to Chaitin's algorithm that leads to a reduction in the number of colors (and therefore number of registers needed) to color an interference graph. Chaitin's coloring algorithm blocks during the simplification stage, that is after all

nodes that have degree less than K have been removed from the graph. At this point, the algorithm splits a live range (which introduces spilling), or (in Briggs's algorithm) optimistically continues hoping K-coloring will still be found. Vegdahl observed that merging two nodes in the graph that are not neighbors, but that share common neighbors, causes a reduction in the degree of any node that had been adjacent to both. In some cases, this reduction can allow simplification to continue without introducing spilling. Also, Vegdahl noticed that there were two aspects in Chaitin's algorithm that were non-deterministic. First, during simplification there may be many nodes that have less than degree K, and the order of their removal indicates what color they will get. Second, during coloring, there may be more than one color to choose from when coloring a node. Thus, there are many different colorings of the same graph and these colorings can differ in the number of colors that are used. This led to the insight of applying Chaitin's algorithm repeatedly to the interference graph, and using random choices whenever a non-deterministic choice was available. Node merging colors interference graphs with fewer colors than Chaitin's algorithm and applying Chaitin's algorithm repeatedly by 8% and 0.6% respectively. The fact that node merging and applying Chaitin's algorithm repeatedly produce improvements over Chaitin's original algorithm are evidence that applying heuristics in several phases of the original algorithm proves beneficial.

2.6 Model and Empirical-driven Optimization Feedback

There are two methods of evaluating the transformations performed by an optimization. Typically, compilers use simple architectural models that are abstractions of the complex hardware of modern processors. This approach is known as *model-driven* optimization. Recently, another approach, called *empirically-driven* optimization, has gained popularity. This approach evaluates the performance of optimizations by generating different program versions for each transformation and running the resulting code on the actual hardware. This is the approach taken by Monsifrot *et al.* [38] (discussed in Chapter 7). Although

LOCO could be adapted to use an empirically-driven approach, we have used only a model-driven approach in our methodology.

A recent study [63] showed that using a model to evaluate the performance of an optimization can be equally as effective as evaluating its performance on a real machine. From our experience working with compilers and profiling tools, we feel it is easier to use a model-driven approach than an empirical-driven approach. Since the model-driven approach works well for the problems in this dissertation and is easy to use, we feel it is preferable.

For our instruction scheduling research in Chapters 4 and 6, we use a basic block simulator that is described in the next section. For Chapter 5, we use the number of spill loads and stores added through register allocation as an estimate of the register allocator's performance.

We emphasize that a model's estimates need not be precise in an absolute sense. The estimates need only be good in a relative sense. For example, our basic block simulator need not return precisely how many cycles a block will take to execute. However, it should accurately predict whether one particular schedule is better than another schedule.

2.6.1 Basic Block Simulation

We need a fair comparison for the different schedules that each scheduler outputs as well as the ability to evaluate quickly different derived heuristics. While actual execution times are more accurate than simulation results, they are more difficult to obtain because the blocks of instructions are so small that accurate and meaningful cycle times are difficult to extract from the hardware. For Chapter 4 we implemented a basic block simulator within Jikes RVM. For Chapter 6 we used a simulator provided by the target's manufacturer.

A block simulator gives a predicted execution time for a given sequence of instructions on a particular architecture implementation. The simulators we used assume that all resources and functional units are available at the start of simulating a block and that all

memory instructions take a constant amount of time. Although these assumptions are not realistic for actual executions of the program, it has been previously demonstrated that the results obtained from these simulators are comparable to actual execution times [37]. Block simulators do not simulate actual execution of the code, but instead rely on the known timings for each machine instruction on the different architectures. The dependence DAG is used to enforce constraints between the instructions.

At the start of a sequence of instructions, each functional unit is assumed to be empty and available for executing new instructions. The block timer then iterates through the instructions and sends them to the appropriate functional unit. The simulator keeps an overall cycle count and sends a parameterized number of instructions to the functional units on each cycle. This simulates the ability to fetch and decode multiple instructions in one clock cycle. Once an instruction is ready to be issued to a functional unit, the simulator chooses the functional unit from the instruction's preferred functional unit list with the minimum time before the instruction will be executed.

Once the instruction has been passed to a functional unit, that unit accesses the timings for the current instruction. Each instruction has two cycle counts associated with it. The first is the number of cycles that the instruction takes to issue completely. The second is the number of cycles until the instruction has completed its execution and the results are available to the other instructions. The functional unit checks the DAG to ascertain at which cycle all of the constraints will be satisfied. If that time is later than the current clock time, then the functional unit is stalled until the instruction can execute. The functional unit keeps track of when it can issue new instructions as well as when all instructions in its pipeline will be completed. All children of the current instruction in the DAG are updated with the earliest time that they can begin to execute.

At the end of each issue cycle, each of the functional units is notified that a clock tick has occurred. Once all of the instructions have been processed, the simulator returns the total number of clock cycles that the sequence of instructions took to execute. This can

be used by the schedulers or the learning methods to evaluate the performance of different sequences.

2.7 Machine Learning

Machine learning is the study of computer algorithms that learn (improve their behavior) through experience. There are many successful applications of machine learning, ranging from data mining programs that discover interesting properties from large databases, to programs that learn how to play world-class backgammon, to systems that learn to recognize spoken words. Machine learning algorithms are useful in complicated, poorly understood, or new application domains where humans might not have the knowledge needed to develop effective algorithms. The complexity of today's processors, the poorly understood interaction between different compiler optimizations, and the development of new optimization algorithms that require new, highly-tuned heuristics make compiler construction a domain appropriate for the application of machine learning. We used a specific class of machine learning algorithms called *supervised learning* algorithms.

There are many machine learning algorithms that are suitable for inducing compiler heuristics. We describe several algorithms that we used in this research. We also describe a few algorithms used by other researchers for automating the construction of compiler heuristics, briefly noting their advantages and disadvantages over the algorithms we used.

2.7.1 Supervised Learning Algorithms

The problem of inducing a general function (hypothesis[4]) from specific training examples is central to supervised learning. Supervised learning algorithms involve search: searching through a predefined space of potential hypotheses for the hypothesis that best fits the set of training examples and generalizes to new examples. The chosen hypothesis representation implicitly defines the space of hypotheses that the learning algorithm can

[4]In machine learning parlance, the induced function that we use as our heuristic is called a 'hypothesis".

represent and therefore can learn. The learning algorithms differ in the way they represent the hypotheses and in the way they search through the hypothesis space. It is important to note that several of the programs we used contain techniques (called *pruning*) for simplifying their output, with the aim of improving accuracy on unseen data (generalization, i.e., avoiding over-specialization, also called over-fitting).

Decision tree learning is a widely used and practical method of learning that induces heuristic functions and is robust in handling *noisy* data (that is, the data may contain errors). Decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the root of the tree to a leaf corresponds to a conjunction of attribute tests; the tree is a disjunction of these conjunctions. Learned trees can be converted to sets of if-then rules, allowing them to be human readable and to be converted easily into executable heuristic functions. However, even simplified trees can grow to unwieldy proportions. The induced trees can be extremely accurate, but too complex to be understood by anyone. Also decision tree methods tend to be unstable, that is they tend to have high variance in that small perturbations in their training sets or in construction may result in large changes in the constructed predictor. We used the decision tree induction packages (C4.5 [42] and ITI [58]) for some of our research.

Another method we experimented with is table lookup (TLU), using a table indexed by the feature values. The table has one cell for every possible combination of feature values, with continuous-valued features suitably discretized. Each cell records the number of positive and negative instances from a training set that map to that cell. The table lookup function returns the most frequently seen value associated with the corresponding cell. It is useful to know that the data set used is large and generally covers all possible table cells with multiple instances. Thus, table lookup is "unbiased" and one would expect it to give the best predictions possible for the chosen features, assuming the statistics of the training and test sets are consistent.

We also experimented with *artificial neural networks* (ANNs). ANNs provide an approach to learning real-valued, discrete-valued, and vector-valued functions from examples. They are inspired from biological learning systems that are connected through a complex interconnected web of neurons. ANNs are typically represented as directed acyclic graphs of nodes (perceptrons, linear units, and sigmoid units). Learning corresponds to choosing weights for the edges that connect these nodes. ANNs are well-suited to problems where the training examples may contain errors or where the function being learned is highly nonlinear. However, ANNs require long training times (typically longer than other supervised learning algorithms) and the induced function, consisting of the edge weights in the network, is often difficult for humans to interpret.

Another method that we experimented with is called the ELF function approximator [57]. It uses a multi-layer network similar to ANNs. ELF constructs additional features (much like a hidden unit) as necessary while it updates its representation of the function that it is learning. The function is represented by two layers of mapping. The first layer maps the features of a training example (which must be boolean for ELF) to a set of boolean feature values. The second layer maps those features to a single scalar value by combining them linearly with a vector of real-valued coefficients called weights. Though the second layer is linear in the features of a training example, the boolean features are nonlinear in the example's features.

Finally, the last method we experimented with is *rule induction*. Rule induction learns sets of rules that contain variables[5] from a set of training examples. Many rule learning systems generate hypotheses using a greedy method in which rules are added to the rule set one by one in an effort to form a small cover of the positive examples. This approach to building rule sets is termed *separate and conquer*, by analogy with the *divide and conquer*

[5]The rules are expressed as first-order Horn clauses

approach commonly used in top-down decision tree induction. The particular rule induction package we used is the *Ripper* [20] program.

We initially tried all the supervised learning algorithms described above on the problem of learning how to schedule. We noticed that all these algorithms performed equally well at solving that learning task. We also tried some of these algorithms for the other problems and found that the error rates were not significantly different.

However, rule induction had the following benefits over the other supervised learning algorithms we tried. It is easy and fast to tune Ripper's parameters (typically an important part in obtaining the best result). Ripper generates sets of if-then rules that are more expressive, more compact, and more human readable (hence good for compiler writers) than the output of other learning techniques, such as neural networks and decision tree induction algorithms. Also, rule sets are representationally more powerful than competing representations such as decision trees, which is reflected in the fact that rule induction systems significantly outperform standard decision tree induction systems on many problems [42]. Finally, the time to induce a heuristic (i.e., the learning time) was faster than the other algorithms. We settled on using Ripper to solve the learning problems in Chapters 6 and 5.

There are other machine learning algorithms that have been used to construct compiler heuristics. We introduce these algorithms in Section 6.10 and describe the related work that uses these algorithms in Section 6.10 and Chapter 7. We now describe three concepts important in machine learning: *overfitting*, *pruning*, and *stopping criteria*.

2.7.2 Overfitting, Pruning, and Stopping Criteria

If a learning algorithm adapts so well to a training set that random disturbances in the training set end up being treated as significant rather than as noise, the learned heuristic is said to *overfit* the data, similar to fitting curves with too many parameters to too few points. In the presence of overfitting, performance on test sets, which have their own, but definitely other, random variations, suffers. Thus a heuristic that has been overfit to training data is

specialized to that data and will not generalize well to unseen data (i.e., data not used for training).

There are several approaches to avoid overfitting. We can choose to stop learning early, before it reaches the point where it perfectly classifies the training data. Alternatively, we can allow training to proceed until the data is overfit, and then *prune* the induced hypothesis. Typically, one takes the latter approach because of the difficulty of estimating when to stop growing the learned hypothesis. Regardless of the approach taken, a key question is what *stopping criterion* to use to determine the final hypothesis size. One popular approach (the one used by Ripper) is to use a separate set of examples, called a *validation set*, that is distinct from the training examples, to evaluate the utility of pruning the hypothesis. The process involves the following steps:

1. Infer a rule set from the training data, growing the rules until the training data is fit as well as possible, allowing overfitting to occur.

2. Prune (generalize) each rule by removing any attribute test whose removal results in improving the rule set's estimated accuracy (performance on the validation set).

3. Sort the pruned rules by estimated accuracy, and consider them in this sequence when classifying subsequent instances.

CHAPTER 3

THE LOCO METHODOLOGY

The previous chapter discussed background in compilation and machine learning. This background will assist in understanding a new methodology, called LOCO, that we introduce in this chapter. LOCO provides a simple systematic approach for generating heuristics to control optimizations. We now describe the steps involved in applying LOCO to a compiler problem.

3.1 LOCO Methodology Description

LOCO involves six distinct steps, illustrated in Figure 3.1: *phrasing the learning problem*, *constructing features*, *generating training instances*, *training*, *integrating the heuristic*, and *evaluating its effectiveness*. The LOCO methodology involves a *feedback loop*. After each step is performed, the compiler writer's feedback involves the performance of the induced heuristic on a set of interesting benchmarks. If the compiler writer is not satisfied with the performance, they should reevaluate how they phrased the learning problem or their choice of features and iterate through the methodology again.

3.2 Phrasing the Learning Problem

The first step to applying LOCO is to phrase the compiler optimization problem as a classification learning problem. Generally this means that when the optimizer needs to make a decision, certain factors (the *features*) are inputs to a decision function, whose output selects a choice from two or more possibilities. For example, one way to phrase the list scheduling problem is: Given certain instructions already scheduled, and two possible

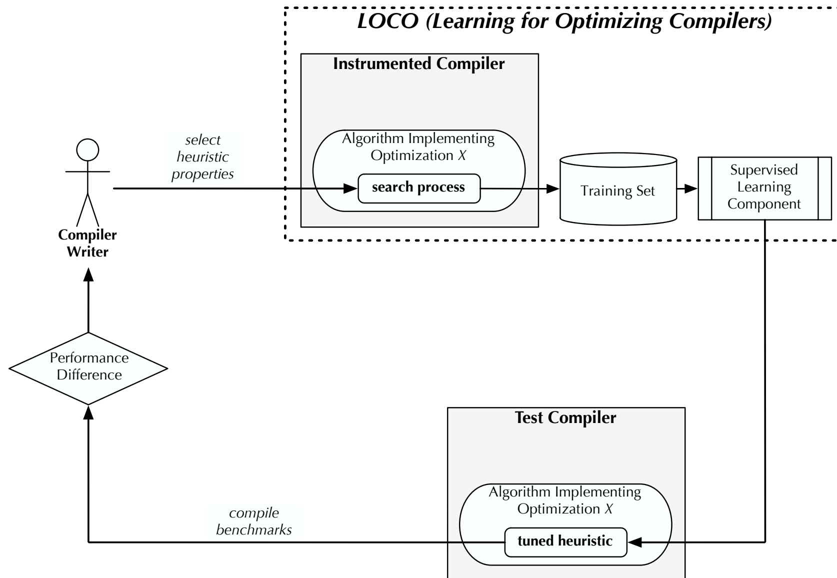


Figure 3.1. LOCO Methodology to Constructing Compiler Heuristics

instructions to schedule next, choose which of the two to add to the schedule. This decision function can be used to run a tournament among three or more instructions. Using the decision function, we would compare the first two instructions, then compare the winner from that decision with the third instruction, and so on. The winner of these comparisons would be added to the schedule under construction. Adding one instruction changes the sequence of instructions scheduled, and the set of instructions available next, so one iterates until scheduling completes. Another example involves the spill heuristic in register allocation. [1] This heuristic decides which variable to spill when there are not enough registers to allocate for all the variables live at a particular program point. We can devise a decision function that chooses which of two possible variables to spill to memory. Again, we could use this heuristic to run a tournament among three or more variables to choose to spill from.

[1]Note that we do not try to solve this learning problem, but instead leave this to future work.

3.3 Feature Construction

Feature construction involves identifying a set of properties (features) that one thinks will be predictive at solving the optimization problem of interest. This is the same first step required for constructing heuristics manually, but after this step, the approaches diverge. Finding the "right" set of features, and a good representation of the information they bear, is the most challenging, and probably least automatable, part of the LOCO methodology.

Feature construction is more an art than a step-by-step procedure. The compiler writer may require several iterations in developing features for a given problem. However, we believe it is easier for a human to *develop* features of a problem than to *rank* their relative importance. LOCO not only finds the most important features automatically, but it constructs the most predictive combinations of those features to form a heuristic.

In list scheduling, it is well known that considering the *critical path* helps, so we used features related to the critical path in Chapter 6. Chapters 4 and 5 concern two problems with no prior art, so we guessed some features that we thought might work (and were very fortunate that they did work well, first try).

Feature construction may require some "massaging" of the data including normalizing (scaling), discretizing (binning), or combining features if necessary. Normalizing or scaling the training data allows the training instances from different "size" problems to be applicable to one another. For example, in Chapter 4, we normalized the features of the problem by number of instructions in the basic block. Discretizing or binning features involves breaking features into categories. For example, a features describing temperature might be constrained to the values *hot*, *warm*, and *cold* as opposed to providing the actual values.

Many of the supervised learning programs have heuristic methods that automatically discretize one's data. However, a compiler writer may have insight into the values being constructed and may be able to more intelligently bin the data. Also, to get the best performing and most concise heuristic it may be necessary to combine multiple features

into a single feature. For example, when solving the list scheduling problem, we initially represented the features of the two instructions being compared by providing each feature separately. The results we achieved with this representation were unsatisfactory. We then combined the same feature provided by each of the two instructions into one feature. For instance, instead of providing two *wcp* features, one for each instruction, we instead provide one *wcp* feature with the possible values of *first*, *second*, or *same*, indicating whether the first instruction was larger, the second instruction was larger, or the values were the same. Combining the features this way allowed us to achieve a better result.

3.4 Generating Training Instances

First, one chooses a suite of benchmarks from which to produce training data. One should choose the benchmarks so that they represent the spectrum of programs that might be compiled by the compiler that will contain the induced heuristic. From each benchmark, the instrumented compiler will produce a set of training examples. For N benchmarks we will have N sets of training data.

Generating training data involves instrumenting the compiler to generate a training instance at every point of the optimization algorithm where the heuristic would typically be applied, or at least at a significant, randomly chosen, sample of those points. This generally involves arranging things so that one can pursue all decision possibilities at that point. One needs a way to evaluate the end result of each choice, and then one labels the instance with the best choice. One must also output the values of the features at the decision point. A training instance says: "In this situation (features) one should do X (label)."

An interesting issue arises when, in order to evaluate a decision, one needs to make more decisions. For example, in instruction scheduling, to evaluate a given choice, one needs to schedule the rest of the block. There are several ways to do this. Sometimes one can enumerate all possibilities (exhaustive search), and pick the best. (This works for short blocks in instruction scheduling.) An alternative is to consider a stochastic approach in

which one constructs samples randomly (an approach applicable to larger blocks, where exhaustive search may be intractable). Finally, one can run to the end of the problem using one or more heuristics already known to be competent (though perhaps not as good as one is aiming for). For the list scheduling example, one might use a generic critical-path based heuristic.

Another significant issue involves *evaluating* each choice. For the learning problems in Chapters 4 and 6, we used a (slightly simplified) cycle-level model of the CPU to estimate the number of cycles each schedule for a block would take when executed. Note that this estimator needs to be good only in a *relative* sense, leading to good decisions about which choice is better. Its absolute estimates do not matter as much. For the learning problem in Chapter 5, we evaluated each allocation algorithm by the dynamic number of additional loads and stores added because of spilling. In contrast, genetic algorithms typically do not use a model, but instead *empirically* evaluate the heuristic performance on the real machine. We describe both model-driven and empirical-driven evaluation in Section 2.6.

3.4.1 Threshold Parameters

For any given optimization learning problem, we generate training instances representing a decision point in the compiler. Each decision point is represented by a set of features and a label representing the "correct" choice to be made at that decision point. Note that we phrase all learning problems in this dissertation as binary classification problems.

In general, an instance might be labeled with class "X" if the action it represents is "better" than performing the action represented by class "Y" as estimated by our model. The evaluations from our models assist in the labeling process of training instances so they are important.

Since a model is a simplified estimator of the performance on actual hardware, its estimates are imprecise and will lead to "noisy" training data. To reduce this inherent noise, we use a technique called *threshold parameters*. Returning to our previous example, we would

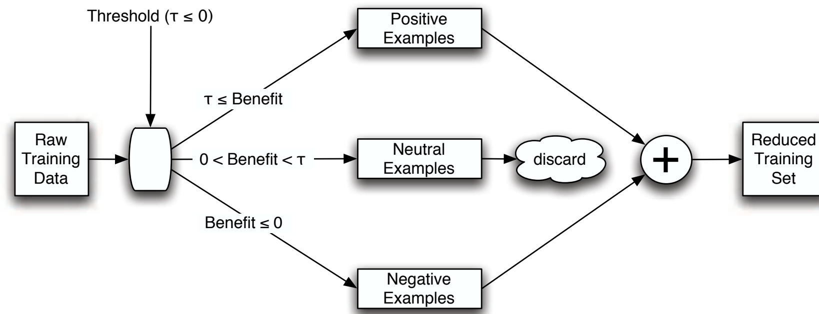


Figure 3.2. Threshold Process

only label an instance with "X" if the benefit of applying optimization "X" to a method was above a threshold τ. Those instances where there was no benefit from applying optimization "X" would be labeled "NX". And, those instances where the benefit was between 0 and the threshold τ would be removed from the training set and not considered for learning. In effect, thresholds remove instances of fine distinction, therefore creating a *threshold gap* between the two classes of training instances. The thresholds are not unlike the concept of *margins* as found in support vector machines [60].

The experiments presented in Chapter 4 and 5 evaluate several different values for each threshold variable. We note that using threshold values allows us to "slice-up" the data in different ways. Through more extensive search, one could perhaps find the "best" values for these thresholds. This kind of search could be automated based on the desired property for which one wishes to optimize (e.g., optimization cost or benefit).

3.4.2 Pruning Features and Training Data

One might consider *manually* pruning the set of features to come up with a good "reduced-set" of features. We recommend against this and instead recommend letting the learning program deal with the additional features. Unless the feature set is large and unwieldy, learning programs can easily prune out insignificant features and focus on the most important features for solving the problem. However, features sets that are too large run the risk of overfitting.

Also, there might be a tendency to want to prune the training instances manually so that training occurs only on a fraction of the original training data. We might prune the training set based to some property, such as importance, of the problem. I believe that this is unnecessary and possibly even detrimental to the learning unless the sheer volume of data is impossible for the learning algorithms to handle. Many learning algorithms can handle large noisy training sets.

3.5 Training the Learning Component

During the learning phase, we present the training instances to one of a variety of supervised learning components in order to construct a tuned heuristic. This is automatic and easy to do, given training instances in the appropriate form. In fact, many of the machine learning tools accept similar formats for training data. Machine learning tools generally provide statistics concerning their effectiveness (classification accuracy) on the training set, and on one or more test sets. A concept that is central to training is *cross validation*.

Cross validation is a method for estimating the generalization error of a hypothesis generated by a learning algorithm. There are several methods of cross validation. In k-fold cross validation, one divides the training data into k subsets of (approximately) equal size. One trains the learning algorithm k times, each time leaving out one of the subsets from training, but using only the omitted subset to compute whatever error criterion is of interest.

Another type of cross validation is called *leave-one-out* cross validation. This kind of cross validation works as follows. Given a suite of N benchmarks, we generate training instances from $N-1$ of the benchmarks, and test the accuracy of the generated heuristic on the benchmark that was left out. This type of cross validation gives the learning algorithm a broad collection of instances from which to learn, but insures evaluation on instances from a benchmark *not* used in the learning process. Thus we can check the generality of the heuristic, insuring that it is not overly specialized and thus avoiding overfitting to any

particular benchmark. Leave-one-out cross validation also makes sense for LOCO for two other reasons:

1. We envision developing and installing of the heuristic "at the factory", and it will then be applied to code it has not "seen" before.[2]

2. While the end goal is to develop a single heuristic, it is important that we test the overall procedure by developing heuristics many times and seeing how well they work. The leave-one-out cross validation procedure is a commonly used way to do this. Another way is repeatedly to choose about half the programs and use their data for training and the other half for testing. However, we want our heuristics to be developed over a wide enough range of benchmarks that we are likely to see all the "interesting" behaviors, so leave-one-out may be more realistic in that sense.

3.6 Integration and Evaluation of the Induced Heuristic

The last two steps involve using the induced heuristic in the compiler. After the learning process, we must convert the output of the supervised learning component into code, and integrate it into the compiler. Many supervised learning components already have options to produce code as output, making this particularly easy. For example, Ripper has an option that converts the induced classification rules into generic if-then-else code that is easily converted to Java code.

The final step in the LOCO methodology involves empirically evaluating the performance of the induced heuristic. While classification accuracy, usually available from the learning tool, gives a good sense of how well the features predict the labels, there is usually some degree of approximation or modeling in the generation of training instances, so one

---

[2] One could provide tools to end users so that they could develop their own training sets and retrain. This would be valuable only if they are likely to come up with a significantly different heuristic function, which would have significantly different performance on their programs. If we train over a large enough set "at the factory", then we presumably "cover" all the interesting behaviors of our compiler and a variety of blocks that present a full range of scheduling issues. Thus, it is not clear that user retraining would have much value.

must test the heuristic's effectiveness in practice. If the effectiveness is not good enough, one may need to adjust the set of features or the process for generating instances.

3.7 Key Properties

There are a few key properties to successfully framing an optimization problem as a classification problem.

First, a decision point should be expressible in terms of a *fixed* collection of properties or features. Each feature may have either discrete or numeric values, but each training instance must not vary from one case to another. Some machine learning tools allow for variable-sized training instances, but this is typically not the norm, and we discourage trying this especially if one might be experimenting with multiple learning programs.

Second, there must be *pre-defined classes* or labels established beforehand to which training instances are to be assigned. This is a specific requirement for supervised learning, in contrast to unsupervised learning (genetic algorithms and reinforcement learning) in which groupings of training instances are found by analysis.

Third, the classes with which the training instances will be labeled must be sharply delineated. A training instance either belongs to a particular class or not. Some learning problems cannot be phrased in this way. For example, trying to predict the particular cycles per instruction (CPI) of a method (which is a continuous value) would be difficult to phrase as a supervised learning problem.

Forth, it is important to frame the heuristic decision function so that it selects among a small set of classes (in the list scheduling example, the *first* instruction of the two, or the *second*), often just two classes. In general, a binary classification problem is easier to solve than a multi-classification problem. Also, the induced heuristics for a problem with a small number of labels will typically lead to a more concise heuristic and thus is easier to understand.

Fifth, there must be enough training instances to allow the statistical tests used in learning algorithms to be effective. The amount of data required is affected by the number of features and classes and the complexity of the heuristic function being induced. Also, the best way to avoid overfitting is to use lots of training data.

Through our experience with identifying features and using machine learning, we also noticed some useful and (possibly obvious) general principles.

1. Experiment with the simplest features first. In this case, it would have been moot to develop additional features.
2. Normalize features and simplify them. Prefer categorical or boolean values over integral or continuous ones. Binning of continuous values can also help the learning task: it simplifies and also tends to enhance readability of the induced heuristic.
3. Examine any relevant hand-coded heuristics. This not only helps in identifying important features to use, but allows us to see the underlying structure of successful heuristics, which will give clues as to how the features should be represented and used.
4. Apply the simplest learning algorithm possible to start with. Obviously, a procedure that is easier to get working (i.e., require less tweaking) is preferable.

3.8 Conclusion

This chapter gives a detailed description of how to use the LOCO methodology to induce heuristics for optimization algorithms. We also provide some key properties to using LOCO that we believe will increase the potential to be successful using LOCO. The next three chapters describe how we used LOCO to solve three different classes of optimization problems: *How to Optimize*, *Whether to Optimize*, and *Which Optimization to Use*.

CHAPTER 4

LEARNING WHETHER TO OPTIMIZE

In the previous chapter we introduced a methodology called LOCO for constructing compiler heuristics automatically. We described each step of the methodology detailing how it could be applied to solve any general optimization problem. In this chapter we present an illustration of using this methodology to construct automatically the first of three different kinds of optimization heuristics that we study in this dissertation.

Instruction scheduling is a compiler optimization that can improve program speed, sometimes by 10% or more—but it can also be expensive. Furthermore, time spent optimizing is more important in a Java just-in-time (JIT) compiler than in a traditional one because a JIT compiles code at run time, adding to the running time of the program. We found that, on any given block of code, instruction scheduling often does not produce significant benefit and sometimes degrades speed. Thus, we hoped that we could focus scheduling effort on those blocks that would benefit from it.

Using LOCO we induced heuristics to predict which blocks benefit from scheduling. The induced function chooses, for each block, between list scheduling and not scheduling the block at all. Using the induced function we obtained over 90% of the improvement of scheduling every block, but with less than 25% of the scheduling effort. We also experimented with using the induced function in combination with profile-based adaptive optimization. The heuristic remains effective but gives a smaller reduction in scheduling effort.

Deciding *whether* to optimize is an important open problem area in compiler research. We show that LOCO solves a specific instance of this problem well.

4.1 Introduction

It is common for compiler optimizations to benefit certain programs greatly, while having little impact (or even a negative impact) on other programs. For example, instruction scheduling is able to speed up certain programs, sometimes by 10% or more [40]. Yet on other programs, applying instruction scheduling has little impact (and in some rare cases, degrades performance). Reasons for this are that equivalent orderings happen to execute at the same speed, or because the block has only one legal order, etc.

If instruction scheduling were an inexpensive optimization we would apply it to all blocks without regard to whether it benefits a particular block. However, scheduling is a costly optimization to apply, accounting for more than 10% of total compilation time in our optimizing JIT compiler. Because it is costly and because it is not beneficial to many blocks, we want to apply it selectively.

We would prefer to apply scheduling to those blocks that have the following two properties. The block 1) accounts for a significant portion of a program's running time, and 2) will benefit from scheduling.

Determining the first property is done through profiling and is a well-studied area of research [6, 64, 17]. On the other hand, determining the second property (whether the block will benefit from scheduling) has received relatively little attention. This is the subject of the research reported in this chapter. Also, to the best of our knowledge this is the first application of supervised learning to determine whether to apply an optimization, in our case instruction scheduling.

We present in this chapter a class of heuristics, which we call *filters*, that accurately predict which blocks will benefit from scheduling. This allows us to *filter* from scheduling the blocks that will *not* benefit from this optimization. Since in practice a large fraction of blocks do not benefit from instruction scheduling, and since the filter is much cheaper to apply than instruction scheduling itself, we significantly decrease the compiler's time spent scheduling instructions.

We show that we can use *inexpensive* and *static* features successfully to determine whether or not to schedule.

4.2 Problem and Approach

We want to construct a filter that with high effectiveness predicts whether scheduling a block will benefit the application's running time. The filter should be significantly cheaper to apply than instruction scheduling; thus we restrict ourselves to using properties (*features*) of a block that are cheap to compute. To the best of our knowledge, this is the first time anyone has developed heuristics to apply instruction scheduling selectively, so we had no "good" hand-coded heuristics from which to start.

The first step in applying LOCO to this problem requires phrasing the problem as a classification problem. For this task, this means that each block is represented by a training instance and each training instance is labeled with respect to whether scheduling benefits the block or not.

For this problem we used only rule induction (Ripper) to induce filters. We emphasize that the goal of the methods discussed in this chapter is to learn to *choose between scheduling and not scheduling*, not to induce the heuristic used by the scheduler. In Chapter 6, we consider learning *how* to schedule, that is, learning to choose which instruction to schedule next. We now consider the specific features used for this research and the methodology for developing the training instances.

4.2.1 Features

The second step in applying LOCO involves identifying the set of properties that are important to the optimization problem.

What properties of a block might predict its scheduling improvement? One can imagine that certain properties of the block's dependence graph (DAG) might predict scheduling benefit. However, building the DAG is an expensive phase that can sometimes dominate the

Feature	Type	Meaning
bbLen	BB size	Number of instructions in the block
Category		Fraction of instructions that ...
Branch	Op kind	are Branches
Call	Op kind	are Calls
Load	Op kind	are Loads
Store	Op kind	are Stores
Return	Op kind	are Returns
Integer	FU use	use an Integer functional unit
Float	FU use	use a Floating point functional unit
System	FU use	use a System functional unit
PEI	Hazard	are Potentially Excepting
GC	Hazard	are Garbage Collection points
TS	Hazard	are Thread Switch points
Yield	Hazard	are Yield points

Table 4.1. Features of a basic block.

overall running time of the scheduling algorithm [40]. Since we require cheap-to-compute features, we specifically chose not to use properties of the DAG. Instead, we tried the simplest kind of cheap-to-compute features that we thought might be relevant. Computing these features requires a single pass over the instructions in the block.

We grouped the different kinds of instructions into 12 possibly overlapping categories, where instructions in each category have similar scheduling properties. Rather than examining the *structure* of the block, we consider just the *fraction of instructions of each category* that occur in the block (e.g., 30% loads, 22% floating point, 5% yield points, etc.). We also supply the block size (number of instructions in the block). See Table 4.1 for a complete list of the features. “Hazards” are a type of branch (typically not taken) that disallows reordering. These features are as cheap to compute as we can imagine, while offering some useful information. It turns out that they work well. We present all of the features (except block size) as ratios to the size of the block (i.e., fraction of instructions falling into a category, rather than the number of such instructions). This allows the learning algorithm to generalize over many different block sizes.

We could potentially have expanded our set of features, but what we have works well. We note that coming up with features for other optimizations might be easy or hard, depending on the optimization. In this case, we were "lucky" in that a little domain knowledge allowed us to develop on our first attempt a set of features that produced highly-predictive heuristics.

It is possible that a smaller set of features would perform nearly as well. However, it is doubtful that in this case reducing the feature set will make the filtering process run faster, since we calculate features values in a simple linear pass over the basic block, incrementing counters corresponding to the categories pertaining to the instruction. Calculating features and evaluating the heuristic functions typically consumed .5%-1% of compile time (a negligible fraction of total time) in our experiments, so we did not explore this possibility. For other problems, it might be important to prune the feature set, especially if the features are not as cheap to compute as these are.

One final observation is that these features are fairly generic, for the most part, and might be useful across a wide range of systems. The GC, TS, and Yield features are specific to Jikes RVM, but other systems may have similar "barriers" to code reordering. If those barriers are plain in the code being scheduled, then a feature similar to ours will probably be useful.

4.2.2 Learning Methodology

As mentioned in Chapter 3, determining what features to use is an important (and possibly the most difficult) step in applying LOCO to a problem. Once we determine the features, the next step involves generating training instances. Each training instance consists of a vector of feature values, plus a boolean classification label, i.e., *LS* (Schedule) or *NS* (Don't Schedule), depending on whether or not the block benefits from scheduling.

Our procedure for this problem is as follows. As the Java system compiles each Java method, it divides the method into blocks, which it presents to the instruction scheduler. We

instrument the scheduler to print into a trace file raw data for forming instances, consisting of the features of the block and an estimate of the block's cost (number of cycles) without scheduling, and an estimate of the block's cost with list scheduling applied. We obtain these estimates from a simplified machine simulator. We described this simulator in more detail in Chapter 2.

We label an instance with LS if the estimated time after list scheduling is more than t% less than before scheduling. We label an instance with NS if scheduling is not better (at all). We do not produce a training instance if the benefit lies between 0 and $t\%$. We call t the *threshold value*. We first consider the case $t = 0$ and discuss positive threshold values later. Typically we obtain thousands of instances for each program (one for each block in the program). Table 4.4 shows training set sizes for different threshold values for SPECjvm98.

4.2.3 The Learning Algorithm

After generating the training instances, the next step involves applying a learning algorithm. We chose the rule induction package called Ripper. It has many advantages over other learning methodologies, such as its output being easier for people to understand and therefore easier to gain insight from. We apply Ripper to the training instances. The output of Ripper is a heuristic function: given the features of the block, it indicates whether or not we should schedule the block. It is important to note that the procedure above (including learning) occurs entirely *offline*. We analyze one of the induced if-then rule sets (a filter) in Section 4.5.6.

4.2.4 Integration of the Induced Heuristic

After the training, the next step is installing the heuristic function in the compiler and applying it *online*. Each block from each method that is compiled by the optimizing compiler is considered as a possible candidate for scheduling. We compute features for the block. If the heuristic function says we should schedule a block, we do so. (The cost

of computing the features is included in all of our actual timings. It is small relative to scheduling and to the rest of the cost of compiling a method.)

4.3 Evaluation Methodology

The final step of using LOCO is to evaluate the induced heuristics. To evaluate a filter on a benchmark, we consider three kinds of results: *classification accuracy*, *scheduler running time*, and *application running time*.

Classification accuracy refers to the accuracy of the induced filter on correctly classifying a set of labeled instances. Classification accuracy tells us whether a filter heuristic has the potential of being useful; however, the real measure of success lies in whether applying the filter can successfully reduce scheduling time while not adversely affecting the benefit of scheduling to application running time. In a few cases, using a filter improved application running time over always applying the scheduler (this occurs when filters inhibit scheduling that actually degrades performance).

Scheduler running time refers to the impact on compile time, comparing against not scheduling at all, and against scheduling every block. Since timings of our proposed system include the cost of computing features and applying the heuristic function, this (at least indirectly) substantiates our claim that the cost of applying the heuristic at run time is low. (We also supply measurements of those costs, in a separate section.)

Application running time (i.e., without compile time), refers to measuring the change in execution time of the scheduled code, comparing against not scheduling and against scheduling every block. This validates not only the heuristic function but also our instance labeling procedure, and by implication the block timing simulator we used to develop the labels. The goal is to achieve application running time close to the best of the fixed strategies, and compilation time substantially less than scheduling every block.

4.4 Experimental infrastructure

We implemented our instruction schedulers in Jikes RVM, a Java virtual machine with JIT compilers, provided by IBM Research [2]. The system has two bytecode compilers, a *baseline* compiler that essentially macro-expands each bytecode into machine code, and an *optimizing* compiler.

We optimized all methods at the highest optimization setting, and with aggressive settings for inlining. We used the build configuration called OptOpt with more aggressive inlining, which increases scheduling benefit.[1]

4.5 Experimental Results

We aimed to answer the following questions: How *efficient* is scheduling using filter heuristics as compared to scheduling all blocks? How *effective* are the filter heuristics in obtaining best application performance? We ask these questions first on the SPECjvm98 standard benchmark suite and next on a suite that includes only benchmarks for which list scheduling made an impact of more than 4% on their running time. (We describe these benchmarks in Section 2.3.1.) What we can verify with this is that we have not undermined the scheduler: the scheduler can still improve some programs a lot while having little impact on others. Later in the chapter, we consider some additional questions, such as how much time does it take to apply our heuristic filters in the compiler, and what happens if we apply our filter in compilations that optimize only hot methods.

We address the first question by comparing the time spent scheduling. We answer the second by comparing the running time of the application, with compilation time removed. To accomplish the latter, we requested that the Java benchmark iterate 6 times. The first iteration will cause the program to be loaded, compiled, and scheduled according to the

[1] We set the maximum callee size to 30 bytecode instructions, the maximum inlining depth to 6, and the upper bound on the relative expansion of the caller due to inlining to be a factor of 7.

appropriate scheduling protocol. The remaining 5 iterations should involve no compilation; we use the *median* of the 5 runs as our measure of application performance.

4.5.1 Classification Accuracy

Before presenting efficiency and effectiveness results, we offer statistics on the accuracy of the induced classifiers (for threshold values t from 0 to 50). For each benchmark, we built a filter with leave-one-out cross validation. For each benchmark in our benchmark suite, we train with learning data from 6 programs, then use the induced heuristic to schedule the benchmark left out. We discuss cross validation in Chapter 3. The filter chooses between list scheduling and no scheduling.

Table 4.2 shows the classification errors rates of rules induced by Ripper on SPECjvm98 benchmark program test sets generated during the cross validation tests. We also include the geometric mean of these error rates. These impress us as good error rates, and they are also fairly consistent across the benchmarks.

Program	0%	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
compress	6.66	6.52	5.81	5.74	1.52	0.93	0.76	0.34	0.53	0.15	0.00
jess	7.68	8.33	7.51	6.65	2.04	1.16	0.96	0.49	0.30	0.03	0.03
raytrace	10.96	9.02	7.05	6.48	2.79	2.47	1.78	1.27	0.43	0.23	0.18
db	6.33	5.87	5.39	5.45	1.14	0.65	0.33	0.33	0.13	0.10	0.00
javac	8.34	8.92	8.57	6.86	4.06	3.08	2.20	1.46	0.89	0.22	0.12
mpegaudio	7.36	6.94	5.91	5.75	2.30	1.75	1.22	1.47	0.55	0.22	0.06
jack	7.63	7.03	6.07	5.36	1.69	1.34	0.92	1.09	0.20	0.03	0.00
Geo. mean	7.86	7.53	6.62	6.04	2.22	1.63	1.17	0.92	0.43	0.14	0.06

Table 4.2. Classification error rates (percent misclassified) for different threshold values.

4.5.2 Simulated Execution Times

Before looking at execution times on an actual machine, we consider the quality of the induced filters (compared with always scheduling and never scheduling) in terms of the simulated running time of each benchmark. We used the block simulator to predict (and

therefore label) whether a block will benefit from scheduling or not. Thus, we hoped that our filters would perform well on a metric based on time reported by the block simulator. Comparing our filters with simulated execution time helps us validate the learning methodology, and to separate validation of the learning methodology from validation of the block simulator's model of the actual machine.

We calculate the weighted simulated running time of each block by multiplying the block's simulated time by the number of times that block is executed (as reported by profiling information). We obtain the simulated running time of the application by summing the weighted simulated running time of each block. More precisely, the performance measure for program P is:

$$\mathrm{SIM}_{\pi}(P) = \sum_{b \in P} (\text{\# Executions of } b) \cdot (\text{cycles for } b \text{ under scheduler } \pi)$$

where b is a basic block and π is either using a filter, always scheduling, or never scheduling. Table 4.3 shows predicted execution times as a ratio to predicted time of unscheduled code. We see that the model predicts improvements at all thresholds. These improvements do not correspond exactly to our measured improvements, which is not surprising given the simplicity of the basic block time estimator. What the numbers confirm is that the induced heuristic indeed improves the metric on which we based its training instances. Thus, LOCO "worked" for this learning problem. Whether we get improvement on the real machine depends on how predictive the basic block simulator is of reality.

4.5.3 Efficiency and Effectiveness

We now consider the quality of each induced filter for threshold $t = 0$, and then present results for the rest of the threshold values. Figure 4.1(a) shows the scheduling time of the L/N filters (chooses to schedule or not) relative to LS (always perform list scheduling). NS (no scheduling) is 0 since it does no scheduling work. We find that on average (geometric

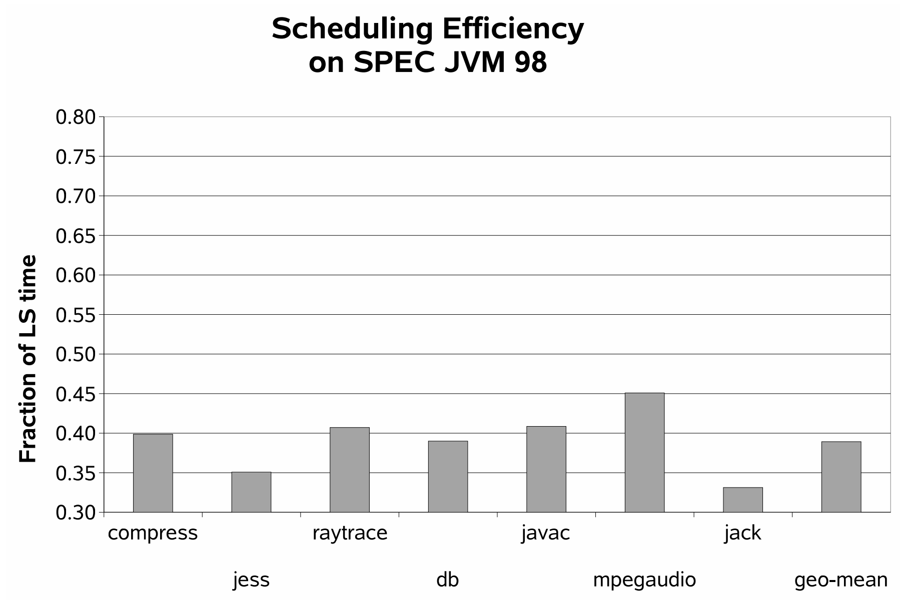
Program	0%	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
compress	84.66	83.70	83.92	83.79	87.33	84.18	85.45	98.16	92.79	99.55	100.00
jess	92.53	92.09	95.76	92.64	95.51	97.09	97.69	96.42	98.46	99.98	100.00
raytrace	93.56	92.81	84.60	86.38	89.38	92.31	89.68	99.48	99.78	99.96	99.95
db	88.53	88.53	88.53	88.54	92.61	90.24	90.34	92.74	99.94	100.00	100.00
javac	96.63	96.08	96.92	97.55	97.07	97.79	97.77	97.94	98.06	98.85	100.00
mpegaudio	90.53	89.26	86.84	89.16	87.00	92.10	92.26	99.75	97.16	89.99	97.64
jack	97.20	97.16	97.35	97.54	97.62	98.39	98.97	99.59	99.43	99.93	99.89
Geo. mean	91.85	91.27	90.39	90.67	92.26	93.04	93.04	97.70	97.92	98.26	99.64

Table 4.3. Predicted execution times for different threshold values.

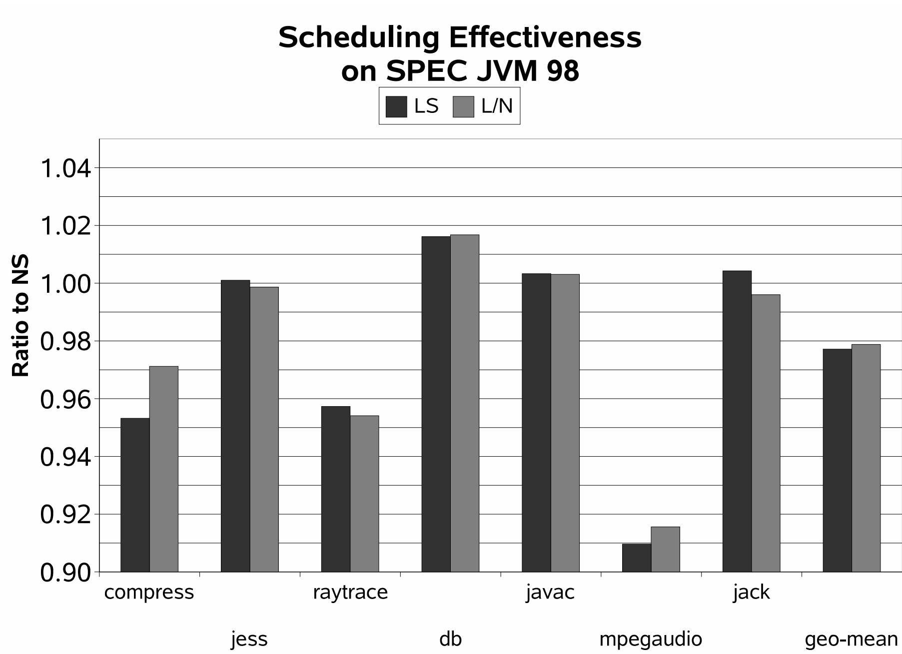
mean) L/N takes 38% of the time of LS (i.e., is 2.5 times faster). These numbers are also fairly consistent across the benchmarks.

Figure 4.1(b) shows the impact of L/N filters and LS on application running time, presented relative to NS (a value smaller than 1 is an improvement, greater than 1 a slow down). Here there is more variation across the benchmarks, with LS doing the best at .977 and L/N filters doing well at .979. Of the benefit LS obtains (2.3%), L/N obtains 93% of it. Given the substantially lower cost of L/N to run, it is preferable to running LS all the time. The results are fairly consistent across the benchmarks, though some benchmarks improve more than others.

Note that our features (and filters) do not take into account the importance of the blocks and therefore do not require profile information. Scheduling only important blocks based on profiling can do no better at improving application running time than just always scheduling (unless scheduling degrades performance). Thus, even if we used profile information to schedule only the important blocks, we could still improve application running time over L/N only by a small amount. (Note: here we are talking only about skipping *scheduling* of cold blocks; skipping all optimized compilation of cold blocks is a different matter, which we address in Section 4.5.8.)



(a) Scheduling Time Using No Thresholds



(b) Application Running Time Using No Thresholds

Figure 4.1. Efficiency and Effectiveness Using Filters.

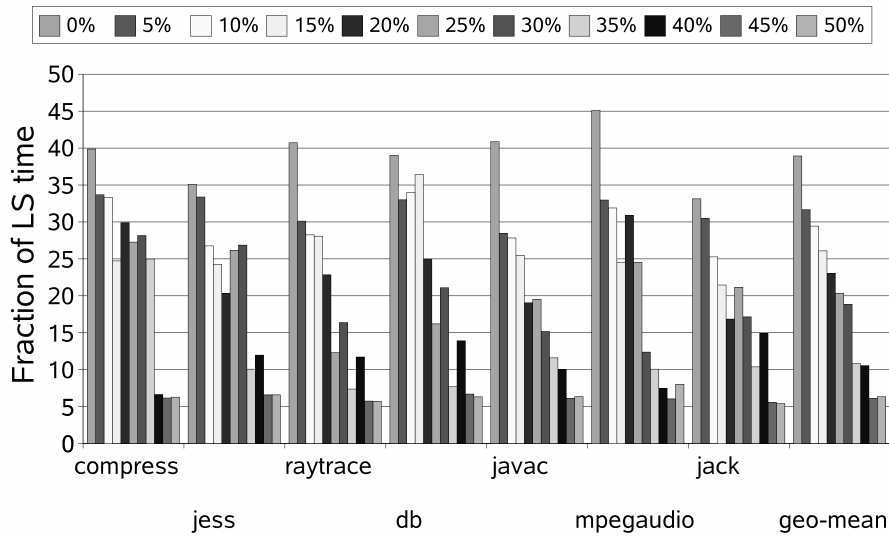
4.5.4 Thresholding

While the $t = 0$ result is not bad, we suspected that we could improve the classification error rates by increasing t. (Of course for a value large enough, only the NS category would be left and the error rate would be 0%!)

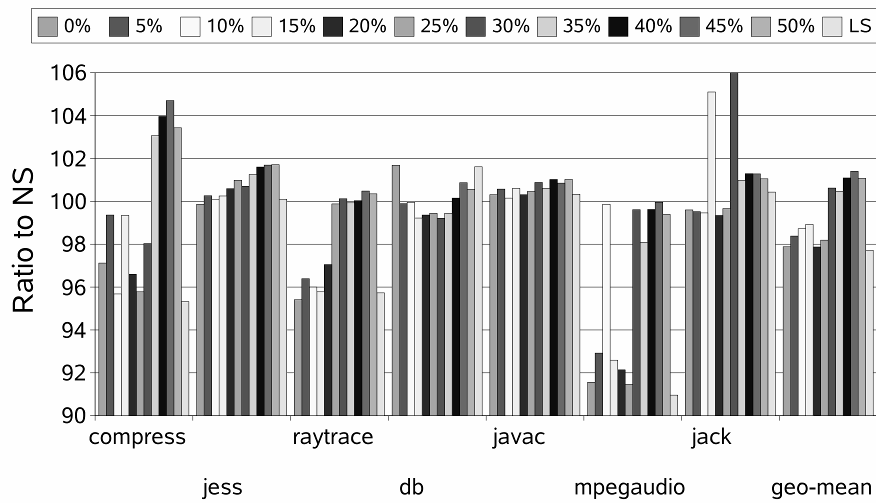
More significantly, we suspected that by eliminating instances where scheduling made little difference, and giving the learning algorithm only those points where the choice makes a significant difference, we might improve scheduler effectiveness. We were less certain of the impact on efficiency, but thought it might increase because the training sets would have fewer LS instances, but as many NS instances. We reasoned that this would tend to induce functions that would prefer scheduling blocks less often. These speculations were borne out, as can be seen in Figures 4.2(a) and 4.2(b).

Again, we performed the experiment for the L/N protocol, varying t from 0 to 50 in increments of 5. Note that $t = 0$ is the same L/N from the previous graphs. Considering first the efficiency effects, the geometric mean shows a steady improvement as t goes from 0 to 50, from 39% to 6% of the cost of LS. This is somewhat consistent across the benchmarks, but there is definite variation. We were able to cut the scheduling effort in half, but what happened to the effectiveness? First it degraded, but at $t = 20$ it improved, offering 93% of the benefit of LS. Thereafter, it generally degrades. While the results seem sensitive to the exact value of t, the value 20 improves over straight L/N ($t = 0$). At this value, scheduling is 4.3 times faster than LS.

How does thresholding affect the size of the training sets? And how does it affect the classification by the induced heuristics? We include two tables that offer some simple statistics that show what happens. Table 4.4 indicate how many instances (across all the benchmarks) have label LS at the given t value. That number is constant for NS (at 37280, so we only show statistics for instances with the LS label), but drops off steadily for LS as t increases.



(a) Scheduling Time Using Thresholds



(b) Application Running Time Using Thresholds

Figure 4.2. Efficiency and Effectiveness Using Filter Thresholds.

Table 4.5 shows how many instances *at run time* were classified with that label by the induced heuristic. We develop the numbers separately for each benchmark's heuristic (using leave-one-out cross validation), applied to that benchmark's instances; the table gives the sum across the benchmarks. The sum is the same for all t values (45453), but the number of NS instances increases, and the number of LS instances steadily decreases, as t increases. This clearly explains the efficiency results. As the threshold increases, the induced rules predict more blocks not to benefit from scheduling. This result further shows that effectiveness depends on the scheduling of a rather small minority of the methods, 7.5% of them for $t = 20$. This is not surprising: in compiler optimization it is often true that an optimization has little effect on many if not most code fragments, but is crucial to improving a certain minority of them.

t%	0%	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
LS	8173	7976	7098	4930	2438	1443	912	565	316	192	49
LS%	21.9	21.4	19.0	13.2	6.5	3.9	2.4	1.5	0.8	0.5	0.1

Table 4.4. Effect of t on training set size of SPECjvm98. NS is constant at 37280.

t%	0%	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
NS	39389	39256	40250	41065	42046	42557	43154	44061	44851	45142	45293
LS	6064	6197	5203	4388	3407	2896	2299	1392	602	311	160
LS%	13.3	13.6	11.4	9.7	7.5	6.4	5.1	3.1	1.3	0.7	0.4

Table 4.5. Effect of t on run time classification of blocks for SPECjvm98.

Since our threshold technique worked well in this case, we encourage others to explore its effectiveness in other settings. We suspect it may be helpful whenever class labels are chosen on a "best" or "better than" basis that compares a "predicted" metric (such as simulated cycle count of a block) under different treatments (scheduling and not scheduling).

4.5.5 Filtering Applied to Other Benchmarks

While the above result is not bad, we suspected that we would have better results focusing only on benchmarks where scheduling is beneficial. We gathered a suite of programs that benefit from scheduling by exploring Java programs freely available on the Internet. Instruction scheduling leads to at least a 4% improvement on the running time of these benchmarks. We describe the benchmarks in more detail in Section 2.3.1.

Our reasoning for focusing on this set of benchmarks is as follows:

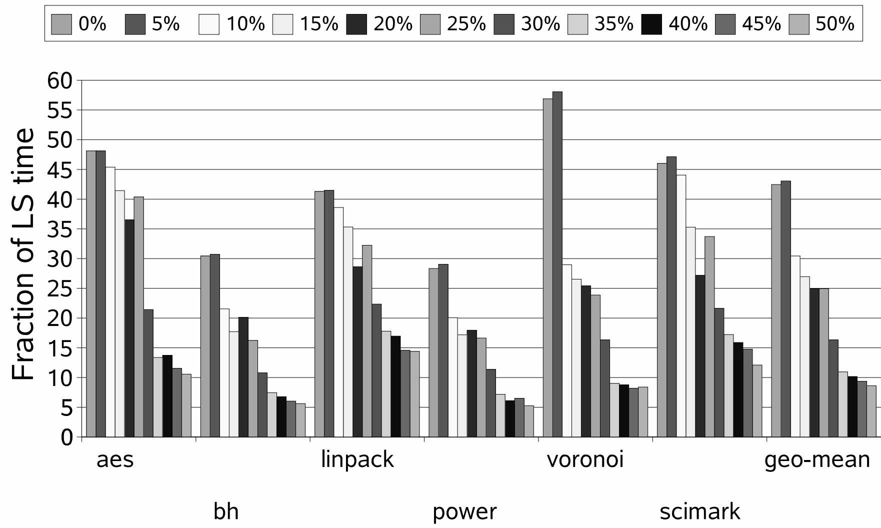
If a program gains little benefit at all from scheduling, our filtering technique can reduce the compile time, but will have no substantial impact on the application running time. We could do a poor job, or a good job, of choosing which blocks to schedule, and it won't matter because the scheduler just is not having much effect.

On the other hand, if we consider a program that gets a *lot* of benefit from scheduling, then we want to make sure that we do not seriously undermine that benefit. Focusing on benchmarks that gain scheduling benefit allows us to determine this. Suppose, for the sake of argument, we included a large number of programs with little (but barely measurable) scheduling benefit. And further suppose that we show that filtering preserves that benefit. We claim that this is not at all as interesting or useful as showing that we preserve the benefit gained by programs that benefit *a lot* from scheduling. By focusing on this set of benchmarks we are trying to be *more* critical, not *less*, of our technique.

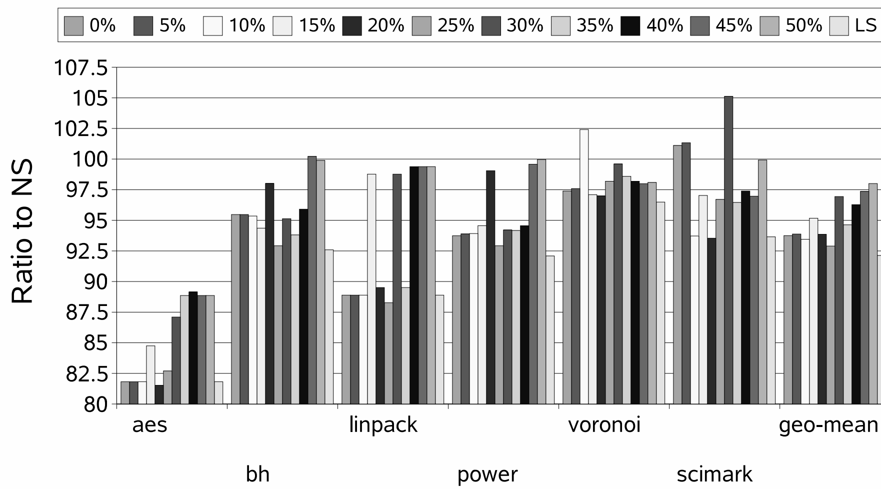
For these experiments we used a filter induced from training examples from all the 7 SPECjvm98 programs. We expected that filtering would achieve most of the benefit of scheduling all blocks, while being much more efficient. This expectation was borne out, as can be seen in Figures 4.3(a) and 4.3(b).

4.5.6 A Sample Induced (Learned) Filter

Some learning schemes produce expressions that are difficult for humans to understand, especially those based on numeric weights and thresholds such as neural networks and



(a) Scheduling Time Using Thresholds



(b) Application Running Time Using Thresholds

Figure 4.3. Efficiency and Effectiveness Using Filters On Other Benchmarks.

genetic algorithms. Rule sets are easier to comprehend and are often compact. It is also relatively easy to generate code from a rule set that will evaluate the learned filter in a scheduler.

Table 4.6 shows a rule set induced by training using examples drawn from 6 of 7 SPECjvm98 benchmark programs. If the right hand side condition of any rule (except the last) is met, then we will apply the scheduler on the block; otherwise the learned filter predicts that scheduling will not benefit the block.

The numbers in the first two columns give the number of correct and incorrect training examples matching the condition of the rule.

```
(   924/   12)  list  ← bbLen>=7 ∧ calls<=0.0857     ∧ loads<=0.3793     ∧ peis<=0.1493      ∧ integers<=0.6087
(   661/    8)  list  ← bbLen>=7 ∧ systems<=0.0889   ∧ stores<=0.05      ∧ loads>=0.1538     ∧ gcpoints<=0.0833  ∧ loads<=0.5556     ∧ loads>=0.3636
(   452/   23)  list  ← bbLen>=7 ∧ calls<=0.1034     ∧ stores>=0.1778    ∧ loads>=0.375
(   218/   14)  list  ← bbLen>=7 ∧ systems<=0.0606   ∧ integers<=0.4167  ∧ peis<=0.2361      ∧ branches<=0.1     ∧ stores<=0.0435
(   272/   19)  list  ← bbLen>=7 ∧ systems<=0.0741   ∧ branches<=0.1111  ∧ loads<=0.3667     ∧ integers<=0.3667  ∧ peis<=0.1667      ∧ fbats<=0
(   518/   41)  list  ← bbLen>=7 ∧ systems<=0.0606   ∧ gcpoints>=0.0714  ∧ integers>=0.4091
(   269/   52)  list  ← bbLen>=7 ∧ calls<=0.119      ∧ stores<=0.0667    ∧ loads>=0.2222     ∧ integers<=0.2857  ∧ loads<=0.625
(    74/    3)  list  ← bbLen>=5 ∧ stores<=0.1613    ∧ loads>=0.3        ∧ integers>=0.3438
(   166/    5)  list  ← bbLen>=7 ∧ calls<=0.119      ∧ branches<=0.0476  ∧ peis<=0.1765      ∧ stores>=0.1237    ∧ peis>=0.093
(    75/   13)  list  ← bbLen>=5 ∧ stores<=0.12      ∧ loads>=0.2083     ∧ integers>=0.3448  ∧ yieldpoints>=0.0143
(    51/   14)  list  ← bbLen>=7 ∧ systems<=0.0741   ∧ loads>=0.3        ∧ systems<=0.0465   ∧ peis<=0.2
(    39/    8)  list  ← bbLen>=5 ∧ stores<=0.1562    ∧ loads>=0.3        ∧ integers>=0.3529  ∧ gcpoints<=0.1818  ∧ peis>=0.1667
(    33/    7)  list  ← bbLen>=5 ∧ stores<=0.1562    ∧ loads>=0.3        ∧ integers>=0.3529  ∧ peis<=0.15        ∧ peis>=0.125       ∧ calls<=0
(    25/    3)  list  ← bbLen>=5 ∧ stores<=0.12      ∧ loads>=0.2222     ∧ integers>=0.3889  ∧ stores>=0.1       ∧ branches>=0.1111
(    18/    5)  list  ← bbLen>=5 ∧ stores<=0.1613    ∧ loads>=0.2941     ∧ integers>=0.3846  ∧ calls>=0.0769     ∧ stores>=0.1111
(27476/1946)  orig ←
```

Table 4.6. Induced Heuristic Generated By Ripper.

In this case we see that block size and several classes of instructions (call, system, load, and store) are the most important features, with the rest offering some fine tuning. For example, the first if-then rule predicts that it is beneficial to schedule blocks consisting of 7 instructions or more, that have a small fraction of call and PEI instructions, but possibly a larger fraction of load and integer instructions. Note (from the last line) that for this training set a large percentage of blocks were predicted not to benefit from scheduling.

As we will see shortly, determining the feature values and then evaluating rules like this sample one does not add very much to compilation time, and typically takes an order of magnitude less time than actually scheduling the blocks selected for scheduling. Thus, while we might be able to eliminate some of the features and retain most of the effectiveness of the filter heuristics, there was not much motivation in this case to do so.

4.5.7 The Cost of Evaluating Filters

Table 4.7 gives a breakdown of the compilation costs of our system, and statistics concerning the percentage of blocks and instructions scheduled. For the individual programs, we give a range of values, covering all threshold values ($t = 0$ through 50). As t increases, s decreases, and while f remains nearly constant, f/s will increase. We also give, for each threshold value, the geometric mean of each statistic across the six benchmarks.

Here are some interesting facts revealed in the table. First, the fraction of blocks and of instructions scheduled steadily decreases with increasing t, dropping significantly at $t = 30$ and $t = 35$. Second, the fraction of instructions scheduled, which tracks the relative cost of scheduling fairly well, tends to be about twice as big as the fraction of blocks scheduled, implying that the filter tends to retain longer blocks. This makes sense in that longer blocks probably tend to benefit more from scheduling. Third, the cost of calculating the filter, as a percentage of non-scheduling compilation time, is always 1% or less. Fourth, there is not a lot of variation in these statistics across the benchmarks. Finally, we always obtain substantial reduction in scheduling time compared with List (scheduling every block).

4.5.8 Filtering with Adaptive Optimization

The Jikes RVM system includes an adaptive compilation mechanism, which determines at run time which methods are executed frequently, and then optimizes those methods at progressively higher levels of optimization [5]. This system typically achieves most of the benefit of optimizing all methods, but with much lower compilation cost. There are two obvious comparisons one might make between our filtering technique and the Jikes RVM adaptive system:

1. The improvement offered by filtering alone versus the improvement offered by the adaptive system alone. Even when scheduling every basic block, scheduling costs only 13–23% of compile time, which gives an upper bound on the improvement we

Program	SB	SI	f/s	f/c	s/c
aes	0.2–14%	0.2–36%	8–20%	1.0%	4–12%
bh	0.1–19%	1.0–41%	7–27%	0.3%	1– 5%
linpack	0.4–14%	0.9–30%	9–17%	1.0%	5–11%
power	0.3–20%	0.6–40%	4–27%	0.4%	1– 9%
voronoi	0.3–19%	0.9–39%	4–27%	0.4%	1– 9%
scimark	0.6–16%	1.3–37%	8–22%	1.0%	5–13%
gm, $t = 0$	16.8%	36.3%	6.5%	0.7%	10.3%
gm, $t = 5$	16.1%	36.9%	6.3%	0.7%	10.3%
gm, $t = 10$	12.7%	28.5%	8.8%	0.7%	7.6%
gm, $t = 15$	11.2%	23.3%	9.3%	0.6%	6.7%
gm, $t = 20$	8.3%	20.5%	9.8%	0.6%	6.2%
gm, $t = 25$	9.2%	21.0%	10.3%	0.7%	6.3%
gm, $t = 30$	5.6%	10.2%	15.1%	0.6%	4.1%
gm, $t = 35$	1.7%	3.8%	20.0%	0.6%	3.0%
gm, $t = 40$	1.3%	2.9%	20.9%	0.6%	2.9%
gm, $t = 45$	0.5%	1.8%	21.8%	0.6%	2.7%
gm, $t = 50$	0.3%	0.7%	22.6%	0.6%	2.5%
List	100%	100%	0%	0.0%	13–23%

Table 4.7. Cost breakdowns: gm = geometric mean; SB = scheduled blocks; SI = scheduled instructions; f = time to evaluate features and heuristic function; s = time spent in scheduling; c = compile time excluding scheduling.

can obtain with filtering. Adaptive compilation reduces compile time much more than this, so we do not even bother to report specific numbers.

2. The improvement offered by the adaptive system alone versus the improvement offered by the adaptive system plus our filtering. We report this comparison below.

First we need to explain a methodological point. The adaptive system is generally triggered according to time-driven sampling of program counter values, which make it non-deterministic from run to run. To obtain deterministic results we proceeded as follows. First, we performed a number of adaptive runs with compilation logging turned on. This told us, for each run, the methods optimized and their optimization levels. From the logs, we determined for each method the highest optimization level achieved by that method in a majority of the runs of that benchmark in which the method was optimized. To obtain

deterministic runs similar to the adaptive system, we force optimization of each method to its majority level when the system first attempts to compile the method, and we prevent any further optimization as the system runs. We call this the *Pseudo-Adaptive* system, and its compile time and execution time behavior is very similar to the adaptive system.

Table 4.8 shows filtering cost breakdowns similar to those we presented for filtering alone. We observe that in this case our filters select a larger fraction of the instructions. This may seem surprising, but is logical in that these benchmarks tend to spend much of their time in floating point computations, and blocks with floating point instructions are more likely to benefit from careful scheduling. These blocks may also be longer, some of them resulting from loop unrolling, etc.

Program	SB	SI	f/s	f/c	s/c
aes	0.0–26%	0.0–68%	4–13%	0.5%	4–16%
bh	0.0–24%	0.0–57%	4– 7%	0.6%	7–14%
linpack	0.0–21%	0.0–63%	3– 4%	0.4%	8–13%
power	5.3–38%	12.2–71%	4%	0.4%	10–15%
voronoi	1.7–30%	5.4–59%	5–10%	0.9%	8–18%
scimark	1.3–23%	2.0–59%	4– 7%	0.7%	5–16%
gm, $t = 0$	26.0%	60.8%	4.0%	0.6%	14.6%
gm, $t = 5$	24.7%	61.3%	3.9%	0.6%	15.0%
gm, $t = 10$	23.3%	58.3%	4.2%	0.6%	14.8%
gm, $t = 15$	17.4%	42.6%	4.2%	0.6%	13.4%
gm, $t = 20$	18.3%	47.4%	4.0%	0.5%	13.4%
gm, $t = 25$	14.6%	38.3%	4.7%	0.6%	12.7%
gm, $t = 30$	6.4%	14.1%	5.7%	0.6%	9.6%
gm, $t = 35$	6.1%	11.9%	5.7%	0.5%	9.2%
gm, $t = 40$	3.9%	8.2%	6.2%	0.5%	8.7%
gm, $t = 45$	0.0%	0.0%	5.7%	0.4%	7.3%
gm, $t = 50$	0.0%	0.0%	6.7%	0.5%	7.2%

Table 4.8. Cost breakdowns for Pseudo-Adaptive runs: gm = geometric mean; SB = scheduled blocks; SI = scheduled instructions; f = time to evaluate features and heuristic function; s = time spent in scheduling; c = compile time excluding scheduling.

Efficiency: Scheduling all blocks in pseudo-adaptive runs consumed about 16% of non-scheduling compile time (geometric mean). Comparing with Table 4.8, we see that filtering

does not save as much on this population of blocks. Put another way, filtering, to some significant extent, avoids scheduling blocks that turn out to be cold. This is interesting, though it is not immediately clear how we can exploit it (since scheduling sees code in its form after most optimizations, we cannot apply the filters much earlier in the optimization process). Still, we might be able to save perhaps 5% of compilation costs, provided filtering does not undermine effectiveness.

Effectiveness: Table 4.9 shows the geometric mean execution time ratio compared with no scheduling, for list scheduling and for filtering with our various threshold values. It is notable that instruction scheduling appears less effective on the blocks optimized in (pseudo) adaptive runs. We still do well for $t = 20$ and $t = 25$, but in those cases we reduce compilation time by at most a few percent. We are forced to conclude that filtering may not be worthwhile in this adaptive compilation setting.

List	$t = 0$	$t = 5$	$t = 10$	$t = 15$	$t = 20$	$t = 25$	$t = 30$	$t = 35$	$t = 40$	$t = 45$	$t = 50$
94.3	95.5	94.8	96.0	99.4	95.3	95.9	99.1	97.4	99.1	99.6	100.0

Table 4.9. Application execution time ratio (versus no scheduling) for List and Pseudo-Adaptive runs, geometric mean across six benchmarks.

4.5.9 Time Compiling versus Time Running

It might at first seem reasonable to report comparisons of total execution time (compilation plus application execution) with and without filtering, etc. We believe this does not make much sense for benchmark programs, since we can make application execution time arbitrarily large compared with compilation by simply iterating the benchmark more times. Put another way: is there any "typical" ratio of compile time to running time? Still, we can determine a pay-back ratio for each benchmark, i.e., how much of the added compilation time does each iteration of the application recover? Table 4.10 gives these pay-back numbers for List and filtering at the different thresholds for our six benchmarks that benefit from scheduling. Some numbers are reported as "<0", which means that the optimization

actually hurt application performance in that case. We observe that compile time is generally *much* greater than application time for these benchmarks; the application times for `power` and `scimark` are closer to the compile times, hence their higher pay-back.

Filter	aes	bh	linpack	power	voronoi	scimark
LS	2.5	1.0	1.2	53	1.0	136
$t = 0$	5.1	1.3	2.7	132	1.1	<0
$t = 5$	5.1	1.6	2.7	118	1.0	<0
$t = 10$	5.5	2.1	3.0	130	<0	299
$t = 15$	5.1	3.1	0.4	156	1.5	175
$t = 20$	7.2	0.7	3.7	26	1.5	502
$t = 25$	6.0	4.7	3.7	164	1.0	207
$t = 30$	8.5	4.7	0.6	142	0.2	<0
$t = 35$	13.1	4.8	6.3	280	1.3	427
$t = 40$	11.7	5.0	0.4	305	1.5	350
$t = 45$	16.0	5.0	0.5	25	1.7	465
$t = 50$	19.8	<0	0.5	3	1.8	15
NSA/NSC	3.5	1.4	2.9	54	1.7	617

Table 4.10. Percent of compile time recovered by each iteration of the application, and application time as percentage of compile time for NS.

Table 4.11 presents analogous measurements for the Pseudo-Adaptive system. Here it is clear that instruction scheduling is not at all helpful for `bh`, `linpack`, or `voronoi`. For the remaining benchmarks, scheduling helps, at least sometimes.

4.6 Summary

Choosing *whether* to apply potentially costly compiler optimizations is an important open problem. We consider here the particular case of instruction scheduling, with the possible choices being a traditional list scheduler (LS) and no scheduling (NS). Since many blocks do not benefit from scheduling, one can obtain most of the benefit of scheduling by applying it to a subset of the blocks. What we demonstrated here is that it is possible to induce a function that is competent at making this choice: we obtain almost all the benefit of LS at less than 1/4 of the cost.

Filter	aes	bh	linpack	power	voronoi	scimark
LS	4.1	<0	<0	<0	<0	1900
$t=0$	4.3	0.6	5.1	76	<0	250
$t=5$	4.6	<0	<0	<0	<0	1484
$t=10$	4.8	<0	<0	<0	<0	119
$t=15$	3.9	<0	<0	<0	<0	<0
$t=20$	5.4	<0	<0	64	<0	<0
$t=25$	5.0	<0	<0	29	<0	689
$t=30$	2.8	0.3	<0	20	<0	1108
$t=35$	0.0	<0	<0	41	<0	2542
$t=40$	<0	<0	<0	37	<0	8991
$t=45$	0.0	<0	<0	24	<0	68
$t=50$	0.0	<0	<0	46	<0	7365
NSA/NSC	7.0	36.7	8.5	1113	20.1	1918

Table 4.11. Percent of Pseudo-Adaptive compile time recovered by each iteration of the application, and application time as percentage of compile time for NS.

On the way to this result we found that it helped to use thresholds: to remove training instances whose cost under different schedulers is within a chosen threshold value, i.e., not different enough to provide a good "signal" on which to train. Interestingly, this instance filtering improved *both* the efficiency and the effectiveness of our induced function.

Sometimes (perhaps only rarely) it is beneficial to perform instruction scheduling in a JIT, depending on how long the program runs, etc. If it is rarely worthwhile, that only emphasizes the need for our heuristic to decide whether to apply it. The general approach we took here should apply in other JIT situations. Of course all we have demonstrated rigorously is that it works for one Java compilation system. If the JIT is adaptive, applying optimization only to "hot" methods, then filtering is less effective and may not be worthwhile.

We found LOCO to work excellently for this learning task. Thus, beyond achieving good performance on the task, we obtain the additional benefits of a simple cheap learning algorithm that produces understandable heuristics. As with any machine learning tech-

nique, devising the appropriate features is critical. Choosing whether to apply an instruction scheduler turns out to require only simple, cheap-to-compute features.

CHAPTER 5

LEARNING WHICH OPTIMIZATION ALGORITHM TO USE

In the previous chapter we described a new class of compiler heuristics, which we called filters, that control *whether* to apply an optimization. Filters are highly predictive at choosing whether it will be beneficial to apply instruction scheduling to a block of instructions. The filters were successful using only simple properties of the block. We also showed how we constructed these filters automatically using LOCO.

This chapter introduces another new class of compiler heuristics, which we call *hybrid optimizations*. We call them hybrid optimizations because they (dynamically) choose which optimization algorithm to apply at run time from a set of different algorithms that implement the same optimization. Hybrid optimizations use a *heuristic* to predict the most appropriate algorithm for each piece of code being optimized. Specifically, we construct a hybrid register allocator that chooses which register allocation algorithm to apply from two different register allocation algorithms, linear scan and graph coloring. Linear scan is more efficient, but sometimes not as effective. Graph coloring, on the other hand, is generally more expensive than linear scan, but on some occasions more effective.

The hybrid allocator decides, based on features of a method, which register allocation algorithm to apply to that method. Using LOCO we induced heuristics that predict which algorithm to use. The induced function chooses, for each method, between linear scan and graph coloring. Using the hybrid allocator with our compiler set at the highest optimization level we obtained 74% of the improvement of using graph coloring, the more effective allocator, but with less than 42% of the effort. We also experimented with using a hybrid allocator for a less aggressive optimization level. The hybrid allocator remains effective

but gives a smaller reduction in allocation effort. To the best of our knowledge, this is the first time anyone has used features to select between two different optimization algorithms.

5.1 Introduction

Compiler writers are constantly inventing new optimization algorithms to try to improve over the current state-of-the-art. Often compiler writers arrive at significantly different algorithm implementations for a particular compiler optimization. Often, however, there is no clear winner among the different implementations. There are certain situations where one particular optimization algorithm is preferable to use and other situations where applying a different algorithm would be more beneficial.

We invent a new class of heuristics, which we call *hybrid optimizations*. Hybrid optimizations assume that one has implemented two or more algorithms for the same optimization. A hybrid optimization uses a heuristic to choose the best of these algorithms to apply in a given situation. In this chapter we construct a hybrid register allocator that chooses between two different register allocation algorithms, linear scan and graph coloring. Linear scan (LS) is more efficient, but sometimes not as effective at packing all the variables of a method into the available physical registers. On the other hand, graph coloring (GC) is generally more expensive than linear scan, but sometimes achieves a better packing of the variables into registers. Our hybrid allocators are more efficient than graph coloring and they are always more effective than linear scan.

This chapter discusses how we use LOCO to construct a hybrid allocator. The induced heuristics should be significantly cheaper to apply than register allocation; thus we restrict ourselves to using properties (features) of a method that are cheap to compute or have already been computed in a previous compilation phase.

We emphasize that while instruction scheduling is an *optional* optimization phase and it could be skipped entirely, register allocation is a *required* optimization phase. That is,

register allocation *must* be applied to every Java method before it can be executed on the machine.

5.2 Motivation

We measured the time to run GC and LS for the case of 24 registers (12 volatiles, 12 non-volatiles). Figure 5.1 shows scatter plots of the running time versus method size (number of instructions). Both graphs omit outliers that have very large running times. LS's running time is consistent and fairly linear. GC's time is in the same general region, but is worse and does not have as clear a linear relationship with method size (this is to be expected, given the overhead of constructing an interference graph in the GC algorithm).

Table 5.1 gives statistics on the measured running times of GC and LS. The second column gives the average time to allocate registers in units of microseconds per (low-level intermediate code) instruction. GC is nearly 7 times slower than LS on this measure. However, if we exclude a small number of methods that took more than 1 second to schedule (outliers), then the ratio is 1.7. The remaining columns give percentiles in the elapsed time to allocate registers for each method, for both GC and LS. These values highlight that a small percentage of methods strongly bias GC's average running time. A possible strategy to ameliorate this is to predict when applying GC will benefit a method over LS, and run GC only in those cases. This is exactly the goal of a hybrid optimization. A hybrid optimization will reduce compilation effort, using an efficient algorithm most of the time, but will use a more effective, but expensive, optimization algorithm seldomly, when it deems the additional benefit is worth the effort. This trade-off is especially interesting if compilation time is important, such as in a JIT compilation environment.

5.3 Problem and Approach

We want to construct a heuristic that with high effectiveness predicts which allocation algorithm is most beneficial to apply. To our knowledge, this is the first time any-

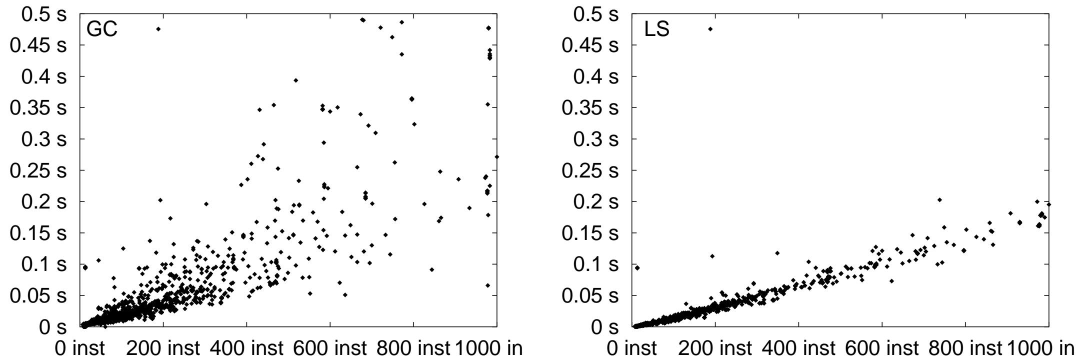


Figure 5.1. Algorithm running time versus method size for GC and LS

Algorithm	(μs)/Insts	(μs)/Insts No outliers	50%ile (seconds)	90%ile (seconds)	95%ile (seconds)	99%ile (seconds)
GC	1241	300	0.011145	0.158539	0.393485	1.955040
LS	183	176	0.008066	0.101580	0.189935	0.803425

Table 5.1. Running time statistics for GC and LS

one has used features to select between two different register allocation algorithms, so we had no "good" hand-coded heuristics from which to start. We used LOCO to construct, automatically, heuristics that choose between the two algorithms. It may be feasible to construct a heuristic manually to choose between the two algorithms, but this would be time-consuming and tedious, and it would be an inefficient way to search the space of heuristics. Also, if the task involves many different alternative algorithms from which to choose, it may require a machine learning approach because it may just be too hard to construct a heuristic by hand.

The first step in applying LOCO to this problem requires phrasing the problem as a classification problem. For this task, this means that each method is represented by a training instance and each training instance is labeled with respect to whether graph coloring achieves enough additional benefit (less spills) over linear scan to warrant applying it.

We emphasize that while register allocation involves heuristics, such as spill heuristics, the learning of which has been considered elsewhere, the goal here is to learn to *choose between* linear scan and graph coloring, not to induce a heuristic used by one particular register allocation algorithm. In other words, the research here involves learning *which* optimization algorithm (register allocation) to use. We now consider the specific features used for this research and the methodology for developing training instances.

5.3.1 Features

The second step in applying LOCO involves identifying the set of properties that are important to this optimization problem.

What properties of a method might predict which allocation algorithm to use? One can imagine that certain properties of a method's interference graph might predict whether or not to use graph coloring. However, building the interference graph is so expensive that it can sometimes dominate the overall running time of the graph coloring algorithm. Since we require cheap-to-compute features, we specifically choose not to use properties of the interference graph.

Instead, we use features that have previously been computed for other compilation phases. For instance, we use features summarizing the control flow graph, *CFG*, such as statistics pertaining to regular (non-exceptional) versus exceptional edges. We also use features that describe liveness information, such as the number of variables that are live in and out of a block. We also try the simplest kind of cheap-to-compute features that we thought might be relevant. Computing these features requires a single pass over the method.

The features can be grouped into three different categories. The first set of features pertains to edges in the control flow graph. These features include regular CFG edges and exceptional CFG edges. The second set of features pertains to the live intervals. We provide features for statistics describing the number of intervals that are live going in and out of

Feature	Meaning
Out Edges	CFG Out Edges (total, min, max, average)
In Edges	CFG In Edges (total, min, max, average)
Exception In	CFG Exceptional In Edges (total, min, max, average)
Exception Out	CFG Exceptional Out Edges (total, min, max, average)
Live on Entry	Number of edges live on entry (total, min, max, average)
Live on Exit	Number of edges live on exit (total, min, max, average)
Intervals	Number of live intervals (total, min, max, average)
Symbolics	Number of symbolics (total, average)
Block size	Size of blocks (min, max, average)
Num Insts	Number of instructions (total)
Num Blocks	Number of blocks in method (total)

Table 5.2. Features of a method.

the blocks. This set also includes features for the number of live intervals and symbolics. The third set of features describes statistics about sizes of blocks and the total number of instructions and blocks in the method. See Table 5.2 for a complete list of the features. These features were either pre-computed or as cheap to compute as we can imagine while offering some useful information. It turns out that they work well.

We present all of the features (except number of instructions and blocks) in several forms, such as minimum, maximum, total, and an average. This allows the learning algorithm to generalize over many different method sizes.

These features work well so we decided not to refine them further. Our domain knowledge allowed us to develop a set of features that on our first attempt produced highly-predictive heuristics.

It might be possible that a smaller set of features would perform nearly as well. However, calculating features and evaluating the heuristic functions typically consumed .5%-2% of compile time (a negligible fraction of total time) in our experiments, so we did not explore this possibility. We started with a larger feature set, but rule induction reduced the set to the most significant features, so we did not need to eliminate features manually.

One final observation is that these features are fairly generic for the most part, and might be useful across a wide range of systems.

5.3.2 Learning Methodology

As mentioned in Chapter 3 determining which features to use is an important step in applying LOCO to a problem. We have constructed a set of features and now take the next step, generating training instances.

Each training instance consists of a vector of feature values, plus a boolean classification label, i.e., *LS* (Linear Scan) or *GC* (Graph Color), depending on which algorithm is best for the method.

Our procedure for generating training instances is as follows. After the Java system compiles and optimizes each Java method, the last phase involves presenting the method for register allocation. As we allocate the variables in the method to registers we can compute the features we decided upon. We instrument both a graph coloring allocator and a linear scan allocator to print into a trace file, for each method, raw data for forming a training instance.

Each raw datum consists of the features of the method that make up the training instance and statistics used to calculate the label for that instance. For the particular step of computing the features of a method, we can use either algorithm since the features are not algorithm specific. However, computing the final statistics used for labeling requires allocating the method with both graph coloring and linear scan. These statistics include the time to allocate the method with each allocation algorithm, and the number of additional spills incurred by each algorithm. We discuss the use of these statistics for labeling each training instance in Section 5.3.3.

We obtain the number of spills for a method by counting the number of loads and stores added to each method after register allocation and multiplying this by the number of times each basic blocks executes. We obtain basic block execution counts by profiling the

```
if (LS_Spill − GC_Spill > Benefit_Threshold)
    Print GC;
else if (LS_Cost/GC_Cost > Cost_Threshold)
    Print LS;
else if (LS_Spill - GC_Spill <= 0)
    Print LS;
else
    { // No Label }
```

Figure 5.2. Procedure for labeling instances with GC and LS

application. We discuss this process in more detail in Chapter 2. We emphasize that these steps take place in an instrumented compiler and all happen off-line. Only the heuristics produced by LOCO are part of the production compiler and these heuristics are fast.

5.3.3 Thresholds

We label an instance based on two different thresholds, a *cost threshold* and a *benefit threshold*. The cost threshold pertains to the time it takes to allocate registers with each algorithm. The benefit threshold pertains to the number of spill loads and stores incurred by the code generated by each allocation algorithm.

For the experiments in this chapter we use the following procedure to label the training instances. We label an instance with “GC” (prefer graph coloring) if the spill benefit of graph coloring is more than B (benefit threshold) less than applying linear scan. We label an instance with “LS” (prefer linear scan) if there is no spill benefit to allocating the method with graph coloring. The spill benefit is the number of spill loads and stores saved by using GC rather than LS. We also label an instance with “LS” if the cost of using graph coloring is C% (cost threshold) more than the cost of applying linear scan. Figure 5.2 depicts this algorithm for labeling.

We experiment with threshold values of $B = 0$, $B = 8K$, and $B = 64K$ for the spill benefit and the values of $C = 0$ and $C = 50$ for the cost threshold. Varying these threshold values gives us the following hybrid allocators: $B0C0$, $B8KC0$, $B64KCO$, $B0C50$, $B8KC50$,

$B64KC50$. For example, we might construct a training set using the threshold values of $B = 8K$ and $C = 50$, for spill benefit and cost threshold values, giving us the hybrid allocator $B8KC50$.

Typically we obtain thousands of instances for each program (one for each method in the program). Table 5.3 shows training set sizes for different threshold values for SPECjvm98.

Threshold	$B0C0$	$B8KC0$	$B64KC0$	$B0C50$	$B8KC50$	$B64KC50$
LS	1796	1796	1796	1796	1964	2029
GC	313	144	79	313	144	79
GC %	17.4	8.0	4.4	17.4	7.3	3.8

Table 5.3. Effect of different threshold values on training set size for SPECjvm98.

5.3.4 Integration of the Induced Heuristic

After the training step, the next step is installing the heuristic function in the compiler and applying it *online*. Each method compiled by the optimizing compiler is allocated registers using a register allocation algorithm. We compute features for the method. The cost of computing the features is included in all of our actual timings. It is small relative to the cost of graph coloring and to the rest of the cost of compiling a method.

If the heuristic function says we should allocate a method with graph coloring, we do so, otherwise we use linear scan.

The results presented are for the SPECjvm98 benchmarks. We describe these benchmarks in further detail in Section 2.3.1.

5.4 Experimental infrastructure

We implemented our register allocation algorithms in Jikes RVM, a Java virtual machine with JIT compilers, provided by IBM Research [2]. The system provides a linear

scan allocator in its optimizing compiler. In addition, we implemented a Briggs-style graph coloring register allocator [12].

For our first set of results, we use the highest optimization setting which is O3. Optimization level O3 includes live range splitting, which increases the effectiveness of register allocation. The second set of results we report are for optimization level O1, a less aggressive optimization level.

5.5 Evaluation Methodology

To evaluate a hybrid allocator on a benchmark, we consider four kinds of results: *classification accuracy*, *spill loads*, *algorithm running time*, and *benchmark running time*.

Classification accuracy refers to the accuracy of the induced hybrid allocator on correctly classifying a set of labeled instances. Classification accuracy tells us whether a hybrid allocator has the potential of being useful. However, the real measure of success lies in whether applying the heuristic can successfully reduce allocation time while not adversely affecting the benefit of applying graph coloring to application running time. In a few cases, using a hybrid allocator improved application running time over always applying the graph coloring allocator (this occurs when the hybrid allocator inhibits the application of graph coloring in cases where it actually degrades performance).

Spill loads refers to the additional number of loads (read memory accesses) incurred by the allocation algorithm. Spill loads give an indication of how well the allocator is able to perform its task. Memory accesses are expensive, and although the latency of some additional accesses can be hidden by the overlapping execution of other instructions, the number of spill loads is highly correlated with application running time. We do not take into account spill stores because their latency can be mostly hidden with *store buffers*. (A store buffer is an architectural feature that allows computation to continue while a store executes. Store buffers are typical in most modern architectures.)

Allocator running time refers to the impact on compile time, comparing always using linear scan and always using graph coloring. Since timings of our proposed system include the cost of computing features and applying the heuristic function, this (at least indirectly) substantiates our claim that the cost of applying the heuristic at run time is low. (We also supply measurements of those costs, in a separate section.)

Application running time (i.e., without compile time), refers to measuring the change in execution time of the allocated code, comparing using linear scan and against always using graph coloring. This validates not only the heuristic function but also our instance labeling procedure, and by implication the spill model we used to develop the labels.

The goal is to achieve application running time close to always applying graph coloring, and compilation time substantially less than applying graph coloring to every method.

5.6 Experimental Results

We aimed to answer the following questions: How *efficient* is our hybrid allocator compared to allocating all methods with linear scan? How *effective* is the hybrid allocator in obtaining best application performance compared to graph coloring? We ask these questions on the SPECjvm98 benchmark suite. We then consider some additional questions, such as how much time does it take to apply the hybrid heuristic in the compiler, and what happens if we apply our hybrid allocator to a different optimization level.

We address the first question by comparing the time spent allocating. We answer the second by comparing the running time of the application, with compilation time removed. To accomplish the latter, we requested that the Java benchmark iterate 6 times. The first iteration will cause the program to be loaded, compiled, and allocated according to the allocation algorithm. The remaining 5 iterations should involve no compilation; we use the *median* of the 5 runs as our measure of application performance.

5.6.1 Classification Accuracy

Before presenting efficiency and effectiveness results, we offer statistics on the accuracy of the induced classifiers for the following threshold combinations: $B0C0$, $B8kC0$, $B64kC0$, $B0C50$, $B8kC50$, and $B64kC50$.

For each benchmark, we built a hybrid allocator with leave-one-out cross-validation. The hybrid allocator chooses between graph coloring and linear scan.

Table 5.4 shows the classification error rates of rule sets induced by Ripper on SPECjvm98 benchmark program test sets generated during the cross-validation tests. We also include the geometric mean of these error rates.

Hybrid	B0C0	B8kC0	B64kC0	B0C50	B8kC50	B64kC50
db	4.69	4.24	3.42	4.69	3.91	2.34
jack	10.59	5.80	2.51	0.59	7.66	2.55
javac	15.07	10.04	5.81	15.07	11.23	5.89
raytrace	5.53	4.84	3.26	5.53	7.04	7.54
compress	3.28	1.80	1.80	13.28	3.28	1.64
jess	10.45	4.56	2.78	10.45	5.76	2.77
mpegaudio	9.33	10.85	7.08	9.33	4.00	4.00
Geo. mean	7.49	5.23	3.45	7.49	5.63	3.36

Table 5.4. Classification error rates (percent misclassified) for different threshold values.

5.6.2 Spill Loads

Before looking at execution times on an actual machine, we consider the quality of the induced hybrid allocator (compared with always applying either graph coloring or linear scan) in terms of the number of spill loads added by register allocation. Spill loads predict whether a method will benefit from the additional effort of applying the more expensive graph coloring allocator.

Spill loads are used in the labeling process, thus we hoped that our hybrid allocator would perform well on the spill load metric. Comparing the allocators against spill loads

allows us to validate the learning methodology, independently of validating against the actual performance on the target machine.

We calculate the dynamic number of spill loads added to each method by multiplying the number of spill loads added to each block by the number of times that block is executed (as reported by profiling information). Then we sum the number of dynamic spill loads added to each block. We obtain the dynamic number of spills loads for the entire program by summing the number of dynamic spill loads added to each method. More precisely, the performance measure for program P is:

$$\text{SPILLS}_{\pi}(P) = \sum_{M \in P} \sum_{b \in M} (\text{\# Executions of } b) \cdot (\text{spill loads added to } b \text{ under allocater } \pi)$$

where M is a method, b is a basic block in that method, and π is hybrid, graph coloring, or linear scan.

Program	B0C0	B8kC0	B64kC0	B0C50	B8kC50	B64kC50	LS
db	1.19	0.99	0.99	1.19	1.01	1.00	1.20
jack	1.02	1.04	1.04	1.02	1.04	1.07	1.07
javac	1.02	1.01	1.04	1.02	1.06	1.04	1.07
raytrace	1.03	1.03	1.39	1.03	1.52	1.05	1.55
compress	1.00	1.04	1.04	1.00	1.04	2.01	2.01
jess	1.11	1.00	1.30	1.11	1.23	1.33	1.42
mpegaudio	1.07	1.05	1.16	1.07	1.36	1.49	2.13
Geo. mean	1.06	1.02	1.13	1.06	1.17	1.25	1.44

Table 5.5. Spill loads for different hybrids and linear scan.

Table 5.5 shows the spill loads for each allocator as a ratio to spill loads produced by our graph coloring algorithm. We see improvements for all hybrid allocators over linear scan. These improvements do not correspond exactly to measured execution times, which is not surprising given that the number of spill loads is not an exact measure of performance on the architecture. What the numbers confirm is that the induced heuristic indeed improves the metric on which we based its training instances. Thus, LOCO was again able to solve

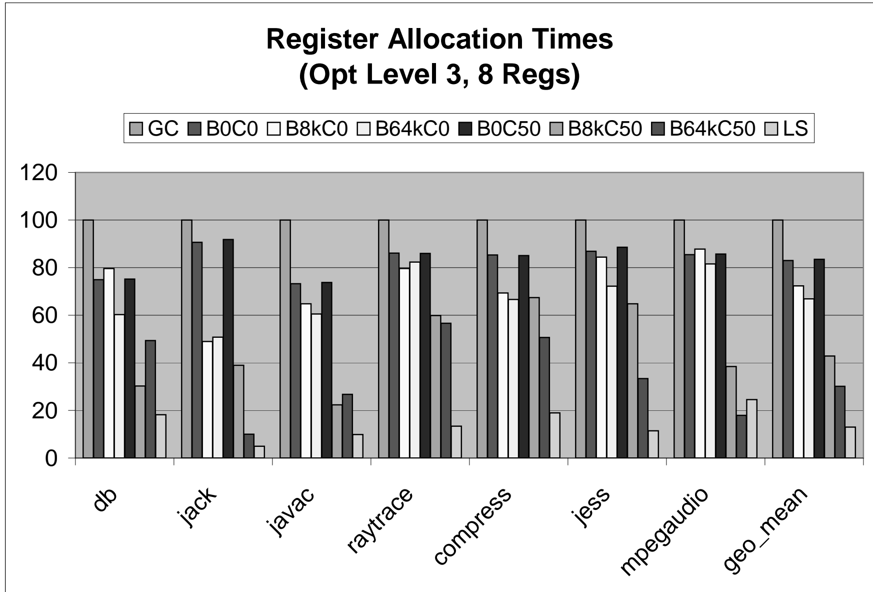
this learning problem. Whether we get improvement on the real machine is concerned with how predictive reducing spill loads are to benchmark performance.

5.6.3 Efficiency and Effectiveness

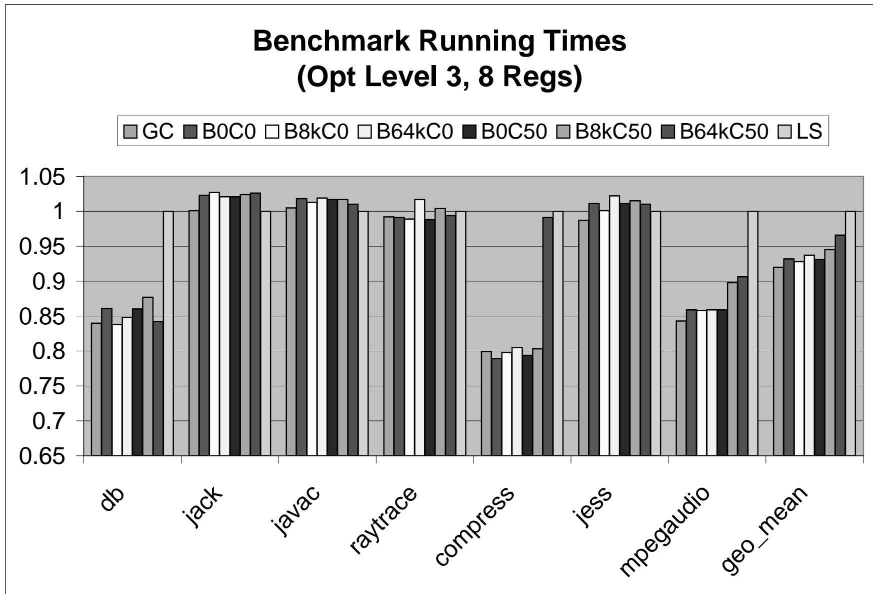
We now consider the quality of the hybrid heuristic induced using each of the different threshold parameter combinations. Figure 5.3(a) shows the allocation time of the hybrid allocators, and for LS (always perform linear scan), relative to GC (always perform graph coloring). LS (linear scan) is almost always the most efficient allocator. Each group of bars shows the performance of the different allocators and the rightmost group of bars shows the geometric mean for all the benchmarks. The leftmost bar in each group indicates the performance of graph coloring. The next six bars from left to right indicates the performance of the different hybrid allocators: B0C0, B8kC0, B64kC0, B0C50, B8kC50, and B64kC50. The rightmost bar of each group shows the performance of linear scan.

We see that for mpegaudio the *B*64*kC*50 hybrid allocator is more efficient than LS. This happens when GC is cheaper than applying LS for those methods where the hybrid allocator applies GC. This occurs only for mpegaudio.

The geometric mean for the efficiency graph shows a steady improvement as the benefit and cost thresholds are increased, from 83% to 28% of the cost of LS. This is somewhat consistent across the benchmarks, but there is definite variation. We were able to cut the allocation effort significantly, but what happened to the effectiveness? All the hybrid allocators do well (except for *B*64*kC*50), with *B*8*kC*50 offering 74% of the benefit of GC. While the results seem sensitive to the exact value of the thresholds, the value *B*8*kC*50 improves efficiency over straight *B*0*C*0 and is comparable in its effectiveness. At this value, allocation is 2.5 times faster than GC. These numbers are also fairly consistent across the benchmarks.



(a) Allocation Time Using Hybrids (opt level O3)



(b) Application Running Time Using Hybrids (opt level O3)

Figure 5.3. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3.

Figure 5.3(b) shows the impact of the hybrid allocators and GC on application running time, presented relative to LS (a value smaller than 1 is an improvement, greater than 1 a slow down). Here there is more variation across the benchmarks, with GC doing the best at .92 and the hybrid allocators doing well at .93-.97. Given the lower cost of hybrid allocators to run, they are preferable to running GC all the time. The results are fairly consistent across the benchmarks, though some benchmarks improve more than others.

5.6.4 A Sample Induced (Learned) Heuristic

As we mentioned, rule sets are easier to comprehend and are often compact. It is also relatively easy to generate code from a rule set that can be used to build a hybrid allocator.

Table 5.6 shows a rule set induced by training using examples drawn from 6 of 7 SPECjvm98 benchmark programs. If the right hand side condition of any rule (except the last) is met, then we will apply the GC on the method; otherwise the hybrid allocator predicts that GC will not benefit the method and it applies LS.

The numbers in the first two columns give the number of correct and incorrect training examples matching the condition of the rule.

```
(   20/  9)  GC ← avgLiveOnExitBB >= 3.875      ∧ avgNumbSymBB >= 13.0168
(   22/13)  GC ← avgLiveOnEntryBB >= 4.05556 ∧ avgCFGInEdgesBB >= 1.39806 ∧ avgLiveOnExitBB >= 5.52 ∧ numberInsts <= 294
(   10/  5)  GC ← avgLiveOnExitBB >= 4.28814  ∧ maxLiveOnEntry <= 13
(   12/  2)  GC ← avgLiveOnExitBB >= 3.68293  ∧ maxLiveOnEntry >= 9            ∧ numberSymbolics >= 895  ∧ maxLiveIntervals >= 38 ∧ maxLiveIntervals <= 69
(1815/78)  LS ←
```

Table 5.6. Induced Heuristic Generated By Ripper.

In this case we see that liveness information and the number of symbolics are the most important features, with the rest offering some fine tuning. For example, the first if-then rule predicts that it is beneficial to use graph coloring on methods consisting blocks with a high average number of live intervals exiting the block and a high average number of symbolics in the block. Note that for this training set a large percentage of methods ($1815+78=1893\ of\ 1986=95\%$) were predicted not to benefit from graph coloring.

As we will see shortly, determining the feature values and then evaluating rules like this sample one does not add very much to compilation time, and typically takes much less time than actually allocating the methods with GC.

5.6.5 The Cost of Evaluating a Hybrid Heuristic

Table 5.7 gives a breakdown of the compilation costs of our system, and statistics concerning the percentage of methods and instructions allocated with each allocator. The costs are given as geometric means over the 7 benchmarks.

As we increase the benefit and cost threshold values, we see that the cost of allocation (a/c) decreases. While f remains nearly constant, f/a and f/c increase.

Here are some interesting facts revealed in the table. First, the fraction of methods and of instructions allocated with GC steadily decreases with increasing threshold values, dropping significantly at $B64kC0$, $B8kC50$, and $B64kC50$. Second, the fraction of instructions allocated with GC, which tracks the relative cost of allocation fairly well, tends to be about 3.5 times as big as the fraction of methods allocated with GC, implying that the hybrid allocators tend to use GC for longer methods. This makes sense in that longer methods probably tend to benefit more from graph coloring.

Third, the cost of calculating the heuristic, as a percentage of compilation time, is always 1% or less. Finally, we always obtain reduction in allocation time compared with GC (applying graph coloring to every method).

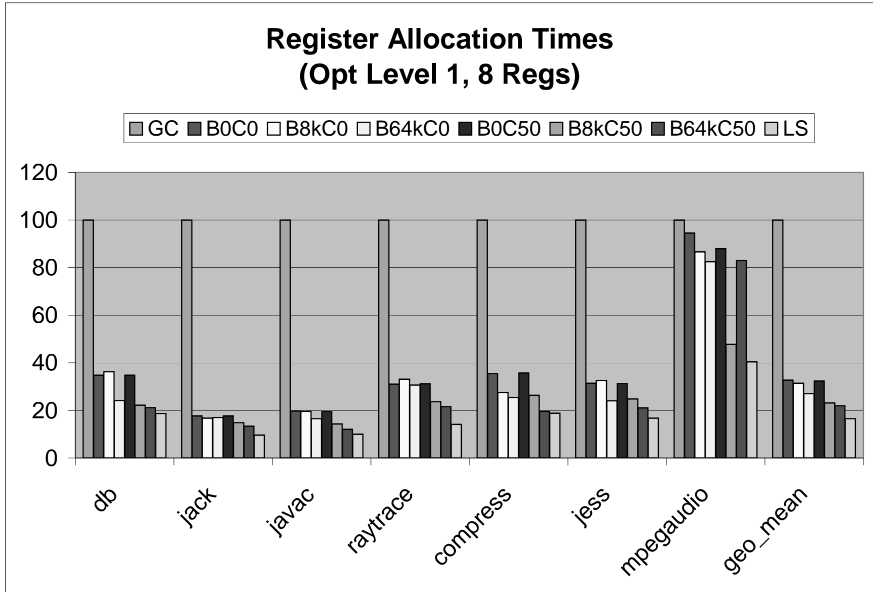
5.6.6 Hybrid Allocation with Optimization Level O1

The Jikes RVM system offers different levels of optimizations, including *O1*, which includes less aggressive, more efficient optimizations, and *O3*, the most aggressive level, which includes more costly optimizations, such as SSA-based transformations. We notice the same trends in register allocation costs found in the more aggressive optimization level of O3.

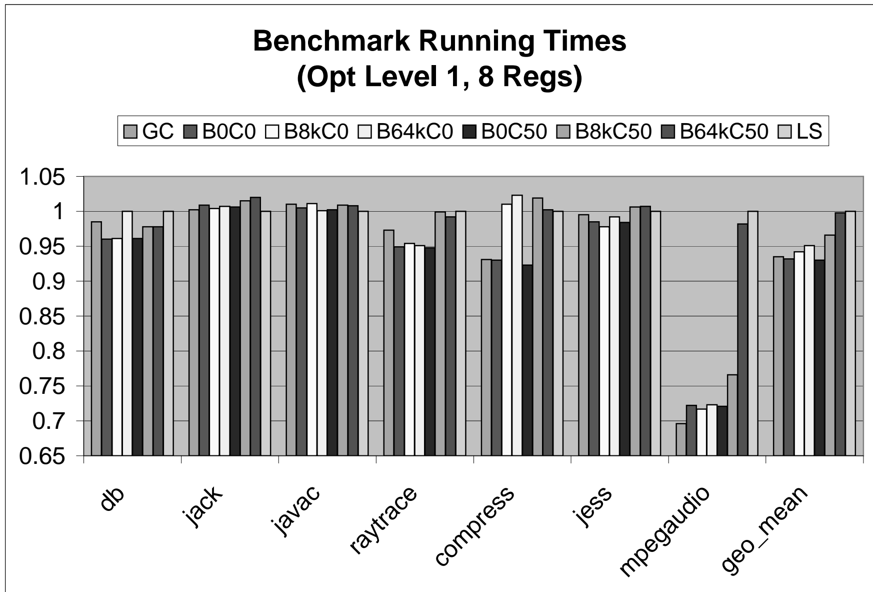
Allocator	GCM	GCI	LSM	LSI	f/a	f/c	a/c
GC	100.00%	100.00%	0.00%	0.00%	0.00%	0.00%	21.26%
*B*0*C*0	15.68%	52.91%	83.96%	45.66%	0.44%	0.07%	16.51%
*B*8*kC*0	12.03%	44.66%	87.49%	53.81%	0.50%	0.08%	15.13%
*B*64*kC*0	9.33%	33.95%	90.04%	62.77%	0.53%	0.08%	14.06%
*B*0*C*50	15.68%	52.91%	83.96%	45.66%	0.43%	0.07%	16.64%
*B*8*kC*50	6.10%	19.43%	93.78%	78.13%	0.85%	0.08%	9.81%
*B*64*kC*50	2.06%	8.50%	97.22%	89.44%	1.20%	0.09%	7.31%
LS	0.00%	0.00%	100.00%	100.00%	0.00%	0.00%	3.41%

Table 5.7. Cost breakdowns: GCM = GC allocated methods; GCI = GC allocated instructions; LSM = LS allocated method; LSI = LS allocated instructions; f = time to evaluate features and heuristic function; a = time spent allocating methods; c = compile time excluding allocation time.

As for benchmark running times, there is a clear performance degradation going to O1. For this optimization level, graph coloring does not give as large of an improvement for benchmarks `db` and `compress`. Benchmark `compress` shows a slight improvement with graph coloring and `mpegaudio` still shows a large improvement with graph coloring. In general, we notice trends in benchmark running time performance similar to those found in the more aggressive O3, just not as large.



(a) Allocation Time Using Hybrids (opt level O1)



(b) Application Running Time Using Hybrids (opt level O1)

Figure 5.4. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1.

5.6.7 Time Compiling versus Time Running

It might at first seem reasonable to report comparisons of total execution time (compilation plus application execution) with and without using hybrid allocators. We believe this does not make much sense for benchmark programs, since we can make application execution time arbitrarily large compared with compilation by simply iterating the benchmark more times. Put another way: is there any "typical" ratio of compile time to running time? Still, we can determine a pay-back ratio for each benchmark, i.e., how much of the added compilation time does each iteration of the application recover? Table 5.8 gives these pay-back numbers for GC and hybrid allocators at the different thresholds for our seven benchmarks. Some numbers are reported as "<0", which means that the the allocator actually hurt application performance in that case, compared to linear scan. We observe that compile time is generally *much* greater than application time for these benchmarks; the application times for `raytrace` and `mpegaudio` are closer to the compile times, hence their higher pay-back. We see that hybrid allocators have a better pay-back for `db`, `raytrace`, and `compress`. GC is not at all helpful for `jack` or `javac`. Hybrid allocators are not helpful for these benchmarks as well as `jess`.

Allocator	db	jack	javac	raytrace	compress	jess	mpegaudio
GC	13.32	<0	<0	1281.76	10.69	1019.72	26.27
B0C0	17.35	<0	<0	1224.12	11.71	<0	32.00
B8kC0	15.05	<0	<0	989.64	10.00	<0	31.88
B64kC0	15.37	<0	<0	<0	10.62	<0	31.68
B0C50	17.29	<0	<0	1002.82	11.98	<0	31.48
B8kC50	17.57	<0	<0	<0	10.37	<0	35.01
B64kC50	12.31	<0	<0	1588.34	229.21	<0	40.95

Table 5.8. Percent of O3 compile time recovered by each iteration of the application, and application time as percentage of compile time for LS.

Table 5.9 presents analogous measurements for a optimization level O1. Here it is clear that hybrid allocators show better pay-back numbers for `db`, `raytrace`, `compress`, and

`jess`. Again, GC and our hybrid allocators are not at all helpful for `jack` or `javac`, but this time they show a benefit for `jess`.

Allocator	db	jack	javac	raytrace	compress	jess	mpegaudio
GC	66.81	< 0	< 0	150.74	15.49	843.01	3.55
B0C0	17.87	< 0	< 0	54.78	10.46	214.48	4.07
B8kC0	18.40	< 0	< 0	59.27	< 0	152.88	3.79
B64kC0	< 0	< 0	< 0	56.98	< 0	394.78	3.84
B0C50	17.97	< 0	< 0	53.73	9.61	201.45	3.85
B8kC50	28.59	< 0	< 0	2562.00	< 0	< 0	4.16
B64kC50	29.35	< 0	< 0	359.72	< 0	< 0	57.71

Table 5.9. Percent of O1 compile time recovered by each iteration of the application, and application time as percentage of compile time for LS.

5.7 Summary

Choosing *which* optimization algorithm to apply among different optimization algorithms that differ in efficiency and effectiveness can avoid potentially costly compiler optimizations. It is an important open problem.

We consider here the particular case of register allocation, with the possible choices being linear scan (LS), graph coloring (GC), and hybrid allocators that choose between these two algorithms. Since many methods do not gain additional benefit from applying graph coloring over linear scan, a hybrid allocator applies graph coloring only to a subset of the methods.

What we demonstrated for an aggressive optimizing compiler (optimization level O3) is that it is possible to induce a function that is competent at making this choice: we obtain almost all the benefit of GC at less than 42% of the cost.

On the way to this result we found that it helped to use thresholds. For this work, we varied two different threshold parameters: a *cost* threshold that takes into account the cost of allocation with the different algorithms, and a *spill* threshold that considers the benefit of

spilling with the different algorithms. We removed training instances whose benefit using graph coloring over linear scan was more than a chosen threshold value. That is, the number of spills added by using the less effective algorithm, linear scan, is not difference enough to make a different in the running time of the program. Interestingly, this training instance filtering improved *both* the efficiency and the effectiveness of our induced function.

Sometimes (perhaps only rarely) it is beneficial to perform graph coloring in a JIT, depending on how long the program runs, etc. If it is rarely worthwhile, that only emphasizes the need for our heuristic to decide when to apply it. The general approach we took here should apply in other JIT situations. Of course all we have demonstrated rigorously is that it works for one Java compilation system. If the JIT is compiling programs at a less aggressive optimization level then hybrid allocation is less effective and may not be worthwhile.

We found LOCO to work well for this learning task. In addition, the learning algorithm we use produces understandable heuristics. As with any machine learning technique, devising the appropriate features is critical. Choosing which register allocation algorithm to apply turns out to require only simple, cheap-to-compute features.

5.8 Related Work

Lagoudakis *et al.* [32] describe an idea of using features to choose between algorithms for two different problems, order statistics selection and sorting. The order statistics selection problem consists of an array of n (unordered) numbers and some integer index i, $1 <= i <= n$. The problem involves selecting the number that would rank i-th in the array if the numbers were sorted in ascending order. The authors used reinforcement learning to choose between two well-known algorithms: Deterministic Select and Heap Select. The learned algorithm outperformed both of these algorithms at the task of order statistics selection. The second problem they look at is the *sorting problem*, that is, the problem of rearranging an array of n (unordered) numbers in ascending order. Again, the authors used

reinforcement learning to choose between two algorithms: Quicksort and Insertion Sort. The learned algorithm again was able to outperform both of these well-known algorithms.

Bernstein *et al.* [9] describe an idea of using three heuristics for choosing the next variable to spill, and choosing the best heuristic with respect to a cost function. This is similar to our idea of using a hybrid allocator to choose which algorithm is best based on properties of the method being optimized. There are important differences between their work and ours. Their technique applies all the spill heuristics and measures the resultant code with the cost function. Our technique, on the other hand, does not try each option, but instead uses features of the code to make a prediction. By making a prediction using simple properties of the code, our heuristics are more efficient while still remaining effective. In fact, our technique could be used as an alternative to Bernstein *et al.* cost function. It is worth noting that their technique is able to improve over the methods of Chaitin [15], Chow [19], Brélaz [11], resulting in about a 10% reduction in spills and a 3% improvement in running time.

CHAPTER 6

LEARNING HOW TO OPTIMIZE

6.1 Introduction

The aim of this chapter is to show how to cast the local instruction scheduling problem as a machine learning task so that one obtains a heuristic scheduling algorithm automatically. Once this specific problem is cast as a machine learning problem, we can use supervised learning to generate the list scheduling heuristic. This heuristic controls *how* to schedule a block. More specifically, this is the heuristic that decides which instruction to add to the schedule under construction in the list scheduling algorithm. We also show that the induced heuristics are "effective". That is, the heuristics produced using LOCO are comparable to ones hand-coded and hand-tuned by compiler experts.

Our empirical results demonstrate that just a few features are adequate for competence at this task for a real processor, and that any of several supervised learning methods perform nearly optimally with respect to these features. The learning goal is to minimize the cycles needed for each basic block, in isolation and assuming all memory accesses hit in the cache. We found that competence at this task resulted in good measured execution times for entire scheduled programs. In the end, we essentially match the performance of commercial compilers for the same processor.

6.2 The Problem

Execution speed of programs on modern computer architectures is sensitive, sometimes by 10% or more [40], to the order in which instructions are presented to the processor. However, as mentioned in Chapter 2, scheduling instructions optimally on modern computer

architectures is NP-hard. To realize potential execution gains with an efficient scheduling algorithm, it is customary for an optimizing compiler to use a heuristic algorithm for instruction scheduling. Currently, compiler writers hand-craft instruction scheduling heuristics for each compiler and target processor.

We wish to construct a heuristic algorithm to use in the performance task of scheduling the instructions of *basic blocks*, sequences of instructions with one entry and one exit.

In reality, execution cost is affected by preceding blocks (which may leave functional units busy, etc.), by the contents of the cache, etc. To simplify our scheduling algorithms and the learning task, we ignore these effects. Our results demonstrate that this simplification does not prevent good local instruction scheduling, as measured in terms of execution times of scheduled benchmark programs.

We consider the class of schedulers that work by repeatedly selecting the apparent best of those instructions that could be scheduled next, proceeding from the beginning of the block to the end. While one can imagine other approaches to scheduling (such as permuting non-dependent instructions), this *greedy* approach is practical for production use, corresponding to *list scheduling* as used in production compilers.

Because the scheduler selects the apparent best from those instructions that could be selected next, the task is to learn to make this selection accurately. Hence, the learner needs to acquire the notion of 'apparent best instruction'. The process of selecting the best alternative is like finding the maximum of a list of numbers. One keeps in hand the current best, and proceeds with pairwise comparisons, always keeping the better of the two.[1] One can view this as learning a relation over triples (P, I_i, I_j), where P is the partial schedule (the total order of what has been scheduled, and the partial order remaining), and I is the set of instructions from which the selection is to be made. Those triples that belong to the relation define pairwise preferences in which the first instruction is preferable to the second. This

[1] One can consider other ways to run a tournament based on pairwise comparison, but from our results this approach appears adequate.

formulation is similar to work done by Utgoff *et al.* [59] where they use a decision tree induction algorithm to learn a boolean preference predicate P(x;y) that indicates whether a state x should be preferred to a state y in a search control problem.

We now consider in turn the representation of our training instances, the features, the methodology for developing training instances, the learning algorithm, and the benchmarks.

6.3 The Representation of Scheduling Preference

Because we have framed the determination of the apparent best instruction as learning a logical relation, the learning algorithm must construct a heuristic that evaluates to *true* if and only if (P, I_i, I_j). We note that if (P, I_i, I_j) is considered to be a member of the relation, then it is safe to infer that (P, I_j, I_i) is *not* a member (i.e., this relation should be anti-symmetric).

6.4 Features

What are the primitive facts from which we can construct the heuristic expression? In this case we need *features* of the candidate instructions I_i and I_j and of the partial schedule P. There is considerable art to choosing useful features. We were greatly assisted by having in hand a good hand-crafted heuristic scheduler (called "DEC" below) supplied by the processor vendor (Digital Equipment Corporation, at that time), which suggested potentially useful features. We now describe our feature set in detail.

By examining the DEC scheduler, applying intuition, and considering the results of various preliminary experiments, we chose the features shown in Table 6.1. The possible values for the features are: **odd:** *yes* or *no*; **wcp** and **e:** *first*, *second*, or *same*, indicating whether the first instruction (I_i) has the larger value, the second instruction (I_j) has the larger value, or the values are the same; **d:** *first*, *second*, *both*, or *neither*, indicating which

of I_i and I_j can dual-issue with the previous instruction; **ic0** and **ic1:** both instruction's values, expressed as one of twenty named categories (*load*, *store*, *iarith*, etc.).

Heuristic Name	Heuristic Description	Intuition for Use
Max Delay (e)	The earliest cycle when the instruction can begin to execute, relative to the current cycle	We want to schedule instructions that will have their data and functional unit available earliest.
Instruction Class (ic)	We divide the target's instructions into about 20 classes equivalent with respect to timing.	The instructions in each class can be executed only in certain pipelines, etc.
Weighted Critical Path (wcp)	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one), with edges weighted by expected latency of the result produced by the instruction.	Instructions on longer weighted critical paths should be scheduled first, because they directly affect the lower bound of the schedule cost.
Odd Partial (odd)	Is the current number of instructions scheduled odd or even?	If 'yes', we're interested in scheduling instructions that can dual-issue with the previous instruction.
Actual Dual (d)	Can the instruction dual-issue with the previous scheduled instruction?	If Odd Partial is 'yes', it is important to know whether a candidate instruction can issue with the previous scheduled instruction.

Table 6.1. Features for Instructions and Partial Schedule

This mapping of triples to feature values loses information. The loss does not affect learning much (as shown by preliminary experiments omitted here), but it reduces the size of the input space, and tends to improve both speed and accuracy of learning for some learning algorithms. In any case, one must summarize features of P, I_i, and I_j into fixed length vectors.

6.5 Obtaining Examples and Counter-Examples

How can we construct a set of example triples (P, I_i, I_j)? For local instruction scheduling, we can search the entire space of schedules for basic blocks that are not too long. We found it practical to do this only for blocks of ten or fewer instructions. For such blocks, for each partial schedule P and each candidate next instruction I_i, given a timing model of the processor we can determine the optimal cost of schedules starting with P and choosing I_i next. If the best cost for P followed by I_i is less than the cost of P followed by I_j, we generate an example triple (P, I_i, I_j) and a counter-example (P, I_j, I_i). We had no trouble generating millions of examples and counter-examples.

Is this procedure biased because we considered only small blocks? For the benchmark programs we studied, 92% of the basic blocks were small enough, and the average block size was 4.9 instructions. However, we did see that large blocks tend to be executed more often and thus have a disproportionate impact on execution time. To determine what effect omitting the large blocks from the training data had on learning, we generated examples for large blocks by examining a large sample of schedules using Monte Carlo techniques. We found that using the large block examples resulted in no scheduling improvement over using the short block examples.

6.6 Learning Algorithms

There are many learning algorithms one could use to learn our preference relation. We tried a decision tree induction package (ITI [58]), a rule-set induction package (Ripper (RULE) [20]), table lookup (TLU),[2] a function approximator (ELF [57]), and a feed-forward artificial neural network (NN [44]). A conference paper [39] includes a detailed description of each learning algorithm and some comparisons of all schemes except for RULE. Since all the schemes performed about the same, here we report results only for ELF and RULE.

6.7 Benchmarks

We used the SPEC95 benchmark suite, which consists of 18 programs, 10 written in FORTRAN, which tend to use floating point calculations heavily, and 8 written in C, which focus more on integers, character strings, and pointer manipulations. These were compiled with the vendor's compilers, set at the highest level of optimization offered, which includes compile- or link-time instruction scheduling. We call these the *Orig* schedules

---

[2] This basically memorizes all the examples and counter-examples, and interpolates for cases not seen. If there are conflicting examples and counter-examples, then it chooses the majority class (positive or negative).

Benchmark	Description	Source Lines	Number of Blocks	Number of Instructions	Avg Size of Blocks
applu	Parabolic and elliptic partial differential equations	3 817	25 475	129 853	5.097
apsi	Solves for the mesoscale and synoptic variations of potential temperature, wind, velocity, and distribution of pollutants	4 211	29 077	159 482	5.485
fpppp	Quantum chemistry	2 122	25 693	132 139	5.143
hydro2d	Astrophysics: hydrodynamical Navier-Stokes equations are solved to compute galactical jets	2 522	26 789	129 568	4.837
mgrid	Multi-grid solver in a 3D potential field	368	25 555	121 750	4.764
su2cor	Quantum physics: Monte Carlo calculation of elementary particle masses	1 614	26 972	135 837	5.036
swim	Shallow water model with 512*x*512 grid	259	25 109	119 333	4.753
tomcatv	A mesh-generation program	107	23 856	117 515	4.926
turb3d	Simulates isotropic, homogeneous turbulence in a cube	1 280	26 285	127 884	4.865
wave5	Plasma physics: solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions	6 430	28 932	152 655	5.276
Total		22 730	263 743	1 326 016	5.028

Table 6.2. Characteristics of SPEC95 FORTRAN benchmarks.

for the blocks. The resulting collection has 447,127 basic blocks, composed of 2,205,466 instructions. Tables 6.2 and 6.3 offer more details about these benchmarks.

6.8 Empirical Results

We designed our experiments to answer the following questions: Can we schedule as well as hand-crafted algorithms in production compilers? Can we schedule as well as the best hand-crafted algorithms? We answer these questions with comparisons of program execution times, as predicted from simulations of individual basic blocks (multiplied by the number of executions of the blocks as measured in sample program runs) and with actual running time measurements. The predicted execution time measure seems fair for local instruction scheduling, since it omits the other execution time factors being ignored in developing training instances. That is, the predicted time measure indicates how well the learning algorithms learned the task as posed. The actual running time measurements indicate whether doing well on the learning task carries over to actual performance improvements, which of course is what compiler implementers ultimately care about.

Benchmark	Description	Source Lines	Number of Blocks	Number of Instructions	Avg Size of Blocks
compress	Reduces the size of fi les using adaptive Lempel-Ziv coding	1 422	4 596	20 152	4.385
gcc	Based on the GNU C compiler version 2.5.3, builds SPARC code	133 049	77 269	332 184	4.299
go	Artifi cial intelligence: plays the game of go	25 362	16 095	80 900	5.026
ijpeg	Graphic compression and decompression	17 449	12 033	70 928	5.894
li	LISP interpreter running the Gabriel benchmarks	4 323	8 056	36 668	4.552
m88ksim	Motorola 88100 microprocessor simulator runs test program	12 026	10 121	46 438	4.588
perl	Manipulates strings (anagrams) and prime numbers in Perl	21 078	22 590	111 849	4.951
vortex	Subset of a full object oriented database program called VORTEx (Virtual Object Runtime EXpository)	41 034	32 624	180 331	5.528
Total		255 743	183 384	879 450	4.902

Table 6.3. Characteristics of SPEC95 C benchmarks.

The reason we used simulated cycle counts, too, was to consider inter-block interference effects separately from cache effects. We set the simulator to treat every memory access as a cache hit, so the simulated times reflect only inter-block effects. Answering the optimality question is more difficult, since it is infeasible to generate optimal schedules for long blocks. We offer a partial answer by measuring the number of optimal choices made within small blocks.

To proceed, we selected a computer architecture implementation and a standard suite of benchmark programs (SPEC95) compiled for that architecture. We extracted basic blocks from the compiled programs and used them for training, testing, and evaluation as described below.

6.8.1 Experimental Infrastructure

We chose the Digital Alpha [46] as our architecture for the instruction scheduling problem. When introduced it was the fastest scalar processor available, and from an instruction dependence analysis standpoint its instruction set is simple. The 21064 *implementation* of the instruction set [23] is interestingly complex, having two dissimilar pipelines and the ability to issue two instructions per cycle (also called *dual issue*) if a complicated collection of conditions hold. Instructions take from one to many tens of cycles to execute.

Researchers at Digital made publicly available a simulator for basic blocks for the 21064, which indicates how many cycles a given block requires for execution, assuming all memory references hit in the caches and translation look-aside buffers, and no resources are busy when the basic block starts execution. When presenting a basic block one can also request that the simulator apply a heuristic greedy scheduling algorithm. We call this scheduler *DEC*.

6.8.2 Experimental Procedures

From the 18 SPEC95 programs, we used the ATOM tool [49] to extract all basic blocks, and also to determine, for sample runs of each program, the number of times each basic block was executed. For blocks having no more than ten instructions, we used exhaustive search of all possible schedules to (a) find instruction decision points with pairs of choices where one choice is optimal and the other is not, and (b) determine the best schedule cost attainable for either decision. Schedule costs are *always* as judged by the DEC simulator. This procedure produced over 13,000,000 distinct choice pairs, resulting in over 26,000,000 triples (given that swapping I_i and I_j creates a counter-example). We selected 1% of the choice pairs at random (always insuring we had matched pairs of example/counter-example triples).

For each learning scheme we used leave-one-out cross-validation, previously described. We evaluated two measures of execution time of the resulting schedules. The first measure, *predicted execution time*, was computed as the sum of predicted basic block costs weighted by execution frequency, with the frequencies taken from sample program runs as described above. The predicted basic block costs are as produced by the DEC simulator. The second measure is machine cycles as measured using the Alpha hardware performance counters. The first measure is from a simulator and is hence repeatable. The second measure is on actual hardware, and though we ran in single-user mode, we still saw variation from run to

run. Therefore we performed 35 runs of each program to obtain reliable estimates of the average running time and its variation.

6.8.3 Statistical Analysis Procedures and Results

Consider first our analysis of actual running times. In examining the distributions of running times for each benchmark as scheduled by each scheduler, we found the distributions were not normal, so we used a non-parametric procedure (Friedman Repeated Measures Analysis of Variance (ANOVA) on Ranks) to compare schedulers. Since the absolute running times of the 18 benchmarks vary widely, we first normalized all runs of each benchmark to the median time of the runs for the DEC scheduler (on the same benchmark). The ANOVA test sorts the 3150 (18 benchmarks $\times$ 35 runs $\times$ 5 schedulers) normalized execution times and gives them ranks according to the sort. It then considers the rank numbers for the 630 runs associated with each scheduler, and determines whether that scheduler's runs were faster, slower, or essentially the same as runs of each other scheduler, using a test of statistical significance.

Table 6.4 shows the median (50th percentile) and 25th and 75th percentile normalized execution times resulting from each scheduler's runs, in order of increasing scheduler quality as determined by the ANOVA analysis.[3] The difference in each pair of schedulers was statistically significant ($p < .05$), but the table shows that the differences are small and perhaps not of great practical significance, except that all heuristic schedulers noticeably outperform random scheduling. (The Rand scheduler simply flips a coin at each decision point.) The same information is presented graphically in Figure 6.1, which shows both the variation over all runs, and the variation of the medians of the benchmarks.

We present the predicted execution time results in Table 6.5 and Figure 6.2. Again we used an ANOVA test, but this time on the 18 predicted execution times, normalized to the

[3] The lines of the table really are in the proper order; the RULE scheduler produced enough times worse than DEC times to be ranked slower than DEC.

Scheduling Scheme	Percentile		
	50th	25th	75th
Rand	1.087	1.030	1.130
RULE	0.993	0.979	1.015
DEC	0.998	0.989	1.005
ELF	0.993	0.976	1.002
Orig	0.989	0.976	1.000

Table 6.4. Normalized Measured Execution Times

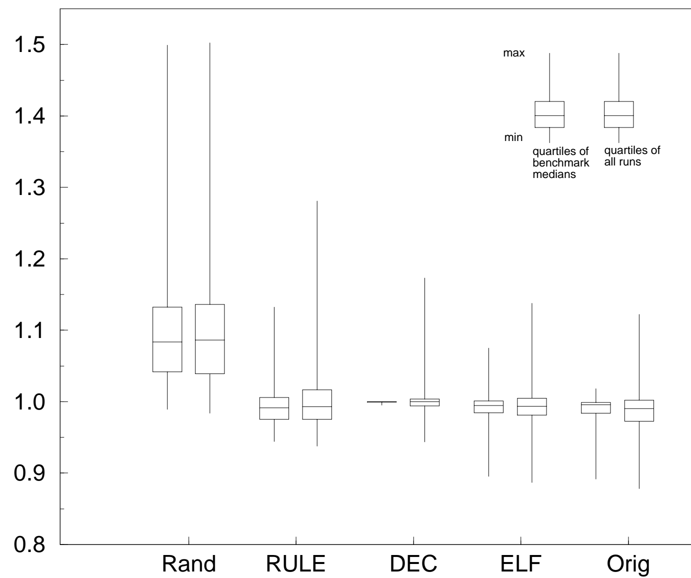


Figure 6.1. Normalized Measured Execution Times

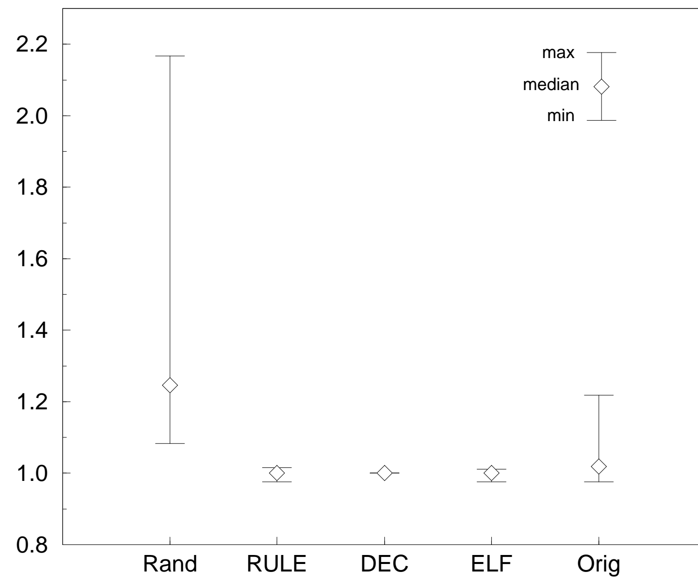


Figure 6.2. Predicted Execution Time

DEC scheduler predicted time. The analysis ranked the schedulers Rand $<$ Orig $<$ DEC $=$ ELF $=$ RULE in quality (Rand worst, Orig next, and the other three equivalent).

Scheduling Scheme	Percentile		
	50th	25th	75th
Rand	1.246	1.134	1.540
Orig	1.019	1.012	1.031
RULE	1.000	0.999	1.002
DEC	1.000	1.000	1.000
ELF	1.000	0.995	1.002

Table 6.5. Predicted Execution Time

While there are a number of differences in the ranking of schedulers by measured execution time, predicted execution time, and the ANOVA analysis, overall the heuristic schedulers perform similarly, and noticeably better than random.

Another way of understanding the results is to see how often the schedulers make optimal choices. As previously explained, we can determine optimality only for small basic blocks. We present these results in Table 6.8.3; it does not include the RULE scheduler.

We see that there is little room for improvement, at least in terms of our local instruction scheduling goal. In separate experiments we determined that the DEC scheduler produces optimal schedules for small blocks 99.5% of the time.

Scheduler	% Optimal Choices (95% conf. int.)
ITI	97.777 (97.474, 98.081)
NN	97.496 (97.058, 97.936)
TLU	96.683 (96.229, 97.140)
ELF	96.568 (96.083, 97.055)

Table 6.6. Experimental Results: Optimal Choices in Small Blocks

We tried to improve effectiveness of our learned schedulers by adding more features, in hopes of making optimal choices closer to 100% of the time. The features we added offered no significant improvement. Indeed, with performance this close to optimal already, it may be unrealistic to expect more.

The results show that all supervised learning techniques produce schedules predicted to be better than the production compilers, but not as good as the DEC heuristic scheduler. This is a striking success, given the small number of features. As expected, table lookup performs the best of the learning techniques. Curiously, relative performance in terms of making optimal decisions does not correlate with relative performance in terms of producing good schedules. This appears to be because in each program a few blocks are executed very often, and thus contribute much to execution time, and large blocks are executed disproportionately often. Still, both measures of performance are quite good.

What about reinforcement learning? McGovern *et al.* [36, 37] ran experiments with temporal difference (TD) learning and the results were not as good. This problem appears to be tricky to cast in a form suitable for TD, because TD looks at candidate instructions in

isolation, rather than in a preference setting. It is also hard to provide an adequate reward function and features predictive for the task at hand. We describe this work in more detail in Section 6.10.

6.8.4 A Sample (Induced) Heuristic for the Alpha 21064

Some of the learning schemes we used produce boolean expressions that are difficult for humans to understand, especially those based on numeric weights and thresholds, such as ELF and NN. Decision trees (C4.5 and ITI) and rule sets are easier to comprehend, and it is also relatively easy to generate code that will evaluate the learned preference expression in a scheduler. We show below a rule set produced by Ripper. This rule set was learned by training using examples drawn from all 18 SPEC benchmark programs. If the right hand side condition of any rule (except the last) is met, then the first instruction of a pair is preferred; otherwise the second is preferred. The numbers in the first two columns give the number of correct and incorrect training examples matching the condition of the rule.

```
(77838/ 600)  first   ← e=first
(15036/ 490)  first   ← e=same ∧ wcp=first
(  412/ 264)  first   ← e=same ∧ wcp=same ∧ d=first ∧ ic0='load'
(   82/  15)  first   ← e=same ∧ wcp=same ∧ d=first ∧ ic0='store'
(  450/ 115)  first   ← e=same ∧ wcp=same ∧ d=first ∧ ic0='ilogical'
(  190/  20)  first   ← e=same ∧ wcp=same ∧ d=first ∧ ic0='fpop'
(   81/  39)  first   ← e=same ∧ wcp=same ∧ d=first ∧ ic0='iarith'   ∧ ic1='load'
(  127/  41)  first   ← e=same ∧ wcp=same ∧ d=first                  ∧ ic1='iarith'
(  247/   0)  first   ← e=same ∧ wcp=same ∧ d=first                                   ∧ odd='yes'
(  225/  93)  first   ← e=same ∧ wcp=same ∧ d=both ∧ ic0='store'
(  320/  44)  first   ← e=same ∧ wcp=same ∧ d=both ∧ ic0='ilogical' ∧ ic1='fpop'
(   22/   0)  first   ← e=same ∧ wcp=same ∧ d=both ∧ ic0='fpop'     ∧ ic1='load'
(  762/ 135)  first   ← e=same ∧ wcp=same ∧ d=both                  ∧ ic1='load'    ∧ odd='yes'
(96367/2130)  second ← true
```

In this case we see that *e*, the time before the instruction can start executing, and *wcp*, the length of the critical path from the instruction to the end of the basic block, are the most important features, with the rest offering fine tuning when *e* and *wcp* are identical for both instructions.

6.9 Learning How to Schedule for Out-Of-Order Processors

What value does static instruction scheduling have in the face of *out-of-order* execution, etc.? We have done some investigation on a newer processor (PowerPC 7410, described

in Chapter 2), which has more "dynamic" scheduling (reordering of execution in the hardware). We describe the features used to induce heuristics for this architecture. We then describe some results we obtained using LOCO's induced heuristics compared to hand-tuned heuristics for this architecture. Finally, we examine the heuristic induced by LOCO.

6.9.1 Features

What properties will be predictive of how to schedule instructions on a newer, more complex architecture? We used some of the same features from Table 6.1 that we used for the older Alpha architecture, plus we added some additional features we thought might help. Table 6.7 describes all the features we used.

Heuristic Name	Heuristic Description	Intuition for Use
Max Delay (e)	The earliest cycle when the instruction can begin to execute, relative to the current cycle	We want to schedule instructions that will have their data and functional unit available earliest.
Instruction Class (ic)	We divide the target's instructions into about 20 classes equivalent with respect to timing.	The instructions in each class can be executed only in certain pipelines, etc.
Weighted Critical Path (wcp)	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one), with edges weighted by expected latency of the result produced by the instruction.	Instructions on longer weighted critical paths should be scheduled first, because they directly affect the lower bound of the schedule cost.
Critical Path (cp)	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one), with edges weighted by 1.	Instructions on longer critical paths should be scheduled first, because they directly affect the lower bound of the schedule cost. This feature is slightly cheaper than wcp because it does required looking up an instruction's latency.
Latency	The expected time it takes for an instruction's result to become available.	We want to schedule instructions that have a longer latency first.
Descendents	The number of instructions that are below this instruction in the DAG.	We want to schedule instructions that are the root of a large sub-tree in the DAG.
Parents	The number of instructions that this instruction is immediately dependent on.	We may want to postpone scheduling an instruction with a large number of parents, because it might take longer for its result to be available.
Children	The number of instructions that are immediately dependent on this instruction.	We want to schedule instructions that have a potentially large number of instructions that it can make available.

Table 6.7. Features for Instructions and Partial Schedule

6.9.2 Results

Table 6.8 shows results using a list scheduling heuristic (LIST) and scheduling using an induced heuristic (RULE) as compared to not scheduling. Static scheduling does give smaller percentage improvement on this architecture; however, we still see useful improvements for some programs. Benchmark `javac` shows a degradation in performance when using the list scheduling heuristic versus our induced heuristic! In fact, our induced heuristic is on average 1% better than the hand-tuned heuristic. This slight improvement is, however, small, and could be explained by cache or other architectural effects. We do see that our induced heuristics for these more recent machines are comparable to hand-tuned heuristics.

Program	LIST	RULE
compress	91.9%	91.9%
db	100.1%	99.5%
jack	100.6%	100.5%
javac	108.9%	100.6%
mpegaudio	88.2%	88.0%
raytrace	94.5%	94.6%
jess	100.3%	100.5%
geo_mean	97.6%	96.4%

Table 6.8. LIST = list scheduling heuristic; RULE = induced scheduling heuristic.

6.9.3 A Sample (Induced) Heuristic for the PowerPC 7410

Using Ripper, we generated preference functions that determined how to schedule for the PowerPC architecture. Analyzing the automatically generated heuristics, we found that certain features, namely *e* and *wcp*, were most important, with other features offering fine-tuning when *e* and *wcp* were identical for both instructions. This shows that Ripper can find the most important features included in the feature set. One interesting special case (third to last rule) that LOCO finds is to schedule an instruction first if the second instruction is on the weighted critical path, the second instruction has more parents, and

the first instruction is of the “pseudo” class. Instructions that are of “pseudo” class are “fake” instructions added by the compiler to respect certain dependencies in the DAG. Thus a “pseudo” instruction disallows movement of instructions above or below it, so it makes sense to prefer to schedule it first.

```
first  ← e=first   ∧ wcp=first
first  ← e=same    ∧ cp=first         ∧ onWcp1=true
first  ← e=first   ∧ cp=first
first  ← e=same    ∧ wcp=first        ∧ cp=first          ∧ iclass1=’integer’  ∧ onWcp2=true
first  ← e=same    ∧ cp=first         ∧ latency=first
first  ← e=same    ∧ wcp=first        ∧ cp=first          ∧ latency=same
first  ← e=first   ∧ latency=same     ∧ parents=same      ∧ children=first
first  ← e=first   ∧ latency=same     ∧ parents=second
first  ← e=second  ∧ latency=same     ∧ children=second   ∧ onWcp1=false
first  ← wcp=first ∧ cp=first         ∧ latency=second    ∧ parents=second     ∧ iclass2=’integer’
first  ← wcp=first ∧ latency=same     ∧ parents=second    ∧ decendents=same
first  ← wcp=first ∧ parents=second   ∧ iclass1=’psuedo’
first  ← wcp=first ∧ parents=second   ∧ onWcp2=true
second ← true
```

6.10 Related Work

We previously reported [39] some of the results presented here. To that prior work we have added the RULE scheduler, the measured execution times and improved statistical analyses. The validation of prior estimated execution times with actual and simulated measurements is crucial to demonstrating the effectiveness of our technique in practice. We also reran some of the experiments in this chapter on an out-of-order processor (that dynamically schedules instructions as they execute in the processor). These experiments were run under the Jikes RVM compilation environment. These experiments further validate that the LOCO methodology can produce heuristics that are comparable to hand-tuned heuristics.

We also found a few pieces of work closely related to the work in this chapter. One is a patent [55]. From the patent’s claims it appears that the inventors used a scheduler with heuristics weighted by the weights in something like a neural network. They evaluate each weight setting by scheduling an entire benchmark suite, running the resulting programs, and using the resulting times to choose among a set of possible weight adjustments. The inventors’ approach appears to us to be potentially very time-consuming. It does have two advantages over our technique: in the learning process it uses measured execution times

rather than simulated times, and it does not require a timing simulator. Being a patent, this work does not state any experimental results.

We now describe some related work on constructing scheduling heuristics using machine learning algorithms, namely genetic algorithms and reinforcement learning.

6.10.1 Genetic Algorithms

Genetic algorithms (GAs) [31] provide an approach to learning that is loosely based on evolution. GAs often describe their hypotheses as bit strings (although they can also be described by symbolic expressions or even computer programs) whose interpretation depends on the application. The search for the best hypothesis begins with a population, or collection, of initial hypotheses. At each step, the hypotheses in the current population are evaluated according to a fitness function. The current population is then updated by replacing a fraction of the population by the offspring of the most fit hypotheses. GAs are capable of searching high-dimensional spaces and can search spaces of hypotheses containing complex interacting parts, where the impact of each part on hypothesis fitness may be difficult to model. This latter characteristic can be especially useful to compiler writers because of the complex interaction of different optimizations and the complexity of developing a model to simulate today's processors. However, evaluating the fitness of a hypothesis (especially, in the domain of compiler construction) can be costly—the time required to iterate a population through several generations is often measured in days.

Beaty *et al.* [8] applied genetic algorithms to the problem of instruction scheduling. They presented a method in which a genetic algorithm is allowed to choose the order of instructions to be placed in the schedule. The genetic algorithm trains and schedules simultaneously, and it must be retrained for every new block. Beaty shows that genetic algorithms are able to schedule code as well as or better than existing scheduling methods on a few benchmarks. Like Boltzmann machines, genetic algorithms are prohibitively expensive, requiring several hundred iterations before a good schedule is found. This method

also produces invalid schedules, which adds to its high cost. Given the high computation cost of genetic algorithms and the large number of iterations needed to schedule a block, this is not a viable solution for commercial compilers. In contrast, our technique would allow learning to be turned off when performance had reached an acceptable level. A consequence of this is that our technique would be as efficient as hand-coded techniques in the speed in which code was scheduled and allocated. In defense of Beaty *et al.* and other approaches that use expensive scheduling techniques: these techniques might be applied only to the very "hottest" blocks.

6.10.2 Reinforcement Learning

Another area of machine learning that has been used for constructing scheduling heuristics is reinforcement learning. Reinforcement learning is learning how to map situations to actions so as to maximize a numerical reward signal [54]. The learner or agent is not told what action to perform, as in most other forms of machine learning, but instead *discovers* which actions yield the most reward, through trial and error. Supervised learning, in contrast, is learning from examples that are provided by some external knowledgeable supervisor. Learning from examples is an important kind of learning, but alone it is inadequate for learning in large or uncertain environments. In environments such as these, it is often impractical to obtain correct and representative examples of all situations an agent may encounter. For example, in instruction scheduling it may be easier to learn through trial and error which of a number of schedules is best as opposed to having a knowledgeable scheduler give the best schedule possible from a sequence of instructions. It is beneficial that the agent be allowed to interact with its environment and to learn from its own experiences. Beyond an agent and its environment, the main components of a reinforcement learning systems are *policy*, a *reward function*, and a *value function*.

A policy is a mapping of the states of the environment to actions the agent will perform when in those states. The policy defines the way the agent behaves in the environment.

A reward function is a mapping of the states of the environment to a number, a *reward*, indicating the desirability of being in that state. The reward function defines the goal of the reinforcement learning problem. Whereas reward functions indicate to the agent which states are good in the short term, value functions indicate to the agent the long term goodness of starting from a state. The value of a state is the total amount of reward expected to accumulate starting from that state. Rewards are immediate while values are predictions of future rewards. We build predictions of values with rewards, and the purpose of estimating values is to achieve more reward.

McGovern *et al.* [36, 37] had some success in applying reinforcement learning (RL) to the same learning task we solved in this chapter. Using RL, they achieved performance within a few percent of commercial compilers. However, they never induced a heuristic that beat the tuned heuristics produced by LOCO. RL has the advantage of not requiring a model of the environment from which to learn. Instead, it learns from past experience and via exploratory search. Thus, there may be some problems that one can solve more effectively using RL, in contrast to using LOCO.

Other researchers have used reinforcement learning in other scheduling domains. For instance, Zhang and Dietterich [65] describe a technique using reinforcement learning to learn a control policy for a NASA scheduling problem. They showed that their technique was more effective than the previous best known method, which used a non-learning search procedure—a method based on simulated annealing. The NASA scheduling problem is a specific type of resource-constraint problem. The NASA scheduling problem consists of certain tasks that must be scheduled before and after a space shuttle launch. Each task has a duration, a set of prerequisite tasks, and one or more resource requirements. The constraints are that the starting time of each task must come after each of its prerequisite tasks have completed and the sum of all resources allocated of each type must not exceed the total amount of resources available. The problem is solved as a state space search where the starting state is a critical path schedule, in which each task prior to launch is scheduled

as late as possible and every task after landing is scheduled as early as possible, subject to prerequisite tasks but ignoring resource constraints.

6.10.3 Iterative Repair

The general framework of *iterative repair* comes from the AI community [34]. The technique has shown promise for several scheduling applications including space shuttle mission scheduling [65]. The general idea is to construct a schedule that begins each operation as early as possible with respect to precedence constraints, ignoring resource constraints. In effect, the algorithm constructs a schedule assuming unlimited resources. The process continues by removing operations, called *unscheduling*, that conflict and inserting them back into the schedule, called *rescheduling* at a later point. The process of unscheduling and rescheduling is called *repair*. The algorithm then tries to repair the schedule by finding the first operation that will stall due to a resource constraint and moving it, along with its predecessors, to a point in the schedule that satisfies its resource constraints (always respecting precedence constraints). We continue repairing the schedule, iterating until there are no more resource constraints.

Philip Schielke [45] used iterative repair (IR) methods for local and global scheduling. Schielke was able to find only small improvements in performance compared to hand-tuned heuristics. Schielke suggests improvements to the base algorithm of IR which he terms IR-RDF and IR-BIAS.

IR-RDF uses an evaluation function to evaluate whether moving an instruction will produce a better schedule than the previous one. The RDF equals the number of cycles in the schedule multiplied by the number of functional units plus the number of operations that need to be moved due to a resource conflict. If moving an instruction produces a new schedule with a lower RDF than the previous schedule, the instruction is moved, otherwise it stays in its position in the schedule regardless of resource constraints.

Schielke introduces another improvement to the basic IR algorithm, called IR-BIAS. IR-BIAS uses the same rescheduling technique as in IR-RDF, but it biases the selection of which conflicting instruction to move. Before performing scheduling, Schielke assigns priorities to instructions using list scheduling. Operations with a higher priority are less likely to be selected for moving, thus high priority nodes have a better chance of appearing earlier in the schedule. Schielke has performed some preliminary experiments that show IR-RDF and IR-BIAS outperform list scheduling at the task of global scheduling.

Our technique for constructing heuristics differs in several important ways. First, our methodology searches for heuristics only during the learning phase of our algorithm. Once our algorithm has found a "good" heuristic, the heuristic is used to construct only one sequence of instructions. In contrast, iterative repair methods construct many different schedules and can therefore be expensive. Also, iterative methods are stochastic and therefore the heuristics produced are unstable.

6.11 Summary

The majority of the effort in this research was in determining how to cast this problem in a form suitable for machine learning. Once that form was accomplished, supervised learning produced quite good results on this practical problem. Here are some significant points about this research.

- Using LOCO we constructed heuristics comparable to two vendor production compilers, as shown on a standard benchmark suites.

- Learning with the goal of minimizing each basic block's execution time, ignoring other blocks and the effects of memory hierarchy, worked well in the presence of those effects. That is, local instruction scheduling improvements generally carried over to overall execution time.

- Several supervised learning schemes performed about as well, with only minor performance differences between them.
- Adding examples from large blocks, obtained using Monte Carlo techniques to limit the search for optimal schedules, offered no significant improvement.

Our primary conclusion is that machine learning can find, automatically, quite competent local instruction scheduling heuristics.

CHAPTER 7

RELATED WORK IN APPLYING LEARNING TO COMPILATION

For each chapter we include a related work section, when appropriate that reviews research relevant to that chapter. This chapter reviews research that is not so much related to any individual chapter as much as it is related to the entire dissertation. This chapter reviews several pieces of work related to applying machine learning to compilation.

Calder *et al.* [14] used supervised learning, namely decision trees and neural networks, to induce static branch prediction heuristics. Their approach gave a misprediction rate of 20%, versus 25% from the best hand-crafted branch predictors at the time. Their learning methodology is similar to ours, but there are important differences. First, their technique was not applied to a compiler optimization. Static branch prediction is a technique that can enable optimizations. For example, it can assist in predicting which basic blocks should be concatenated in superblock formation. Second, their technique made it inherently easy to determine a label for their training instances. The optimal choice for predicting a branch was easily obtained by instrumenting their benchmarks to observe each branch's most likely direction. However, solving our optimization problems involved using models to predict what the best choice to make is. The inherent inaccuracies in using a model required us to use a technique we called *thresholding* to achieve the best results in learning. The last and final difference from this work and ours was that there was a lot of prior work in solving this problem by hand. This gave Calder *et al.* plenty of features from which to draw, which assisted them in applying learning successfully to this problem.

Monsifrot *et al.* [38] use a classifier based on decision tree learning to determine which loops to unroll. As in Calder *et al.* [14], there was a lot of previous work in solving this

problem, giving the researchers many hand-coded heuristics from which to draw features. In contrast to our approach, they obtain labels by using real machine measurements. Their learning methodology is as follows. They measure the effect of unrolling and not unrolling each loop in isolation by measuring the performance benefits of unrolling the loop on a real machine. They use a technique similar to thresholding to label their training instances. If the effect of unrolling the loop is beneficial by more than 10%, they create a positive training example pertaining to the loop. If unrolling the loop causes a 10% degradation in performance, they generate a negative training example. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC and an IA64. They showed an improvement over the hand-tuned heuristic of 3% and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively.

A group of researchers at MIT used genetic algorithms to tune heuristic priority functions in three compiler optimizations [52]. They randomly generated expressions for a priority function for a specific compiler optimization, forming an initial population for a genetic algorithm. They performed crossovers and mutations by modifying the expressions with relational and/or real-valued functions of random expressions. They derived priority functions for these tasks: hyperblock selection, spilling in register allocation, and data prefetching. Their generated heuristics outperformed hand-crafted ones on an architectural simulator. (However, simply by producing 399 heuristics at random and choosing the best they were able to outperform the hand-crafted heuristics.) Iterating the genetic programming produced a significantly better result only for the spilling priority function in register allocation, and it stabilized to the best performing genomes in a few iterations. Unsupervised learning techniques, such as genetic algorithms, have one main advantage over LOCO in that they do not require labels for their training examples. In contrast, supervised learning techniques learn by matching training inputs with known outcomes. In all 3 cases, determining a good outcome is not a problem. Supervised learning algorithms

are easier to get working right and producing good results, and the resulting functions are typically more human-readable. This genetic programming work took days of computer time to derive a heuristic, whereas our supervised learning procedure completes in seconds (once we have developed the training instances).

The same MIT researchers also introduced a technique similar to the research performed by Monsifrot *et al.* [38]. This work presents an incremental, though important, improvement over Monsifrot's research. Previous work by Monsifrot *et al.* solved the binary classification problem of determining whether a loop would benefit from unrolling. This paper solves this same problem, then goes one step further in solving the multi-classification problem of choosing the appropriate factor by which a loop should be unrolled. They use a supervised learning algorithm called nearest neighbor. The heuristic induced by the classifier is able to choose the optimal or near optimal factor for unrolling 74% of time compared to the hand-tuned heuristic found in the *Open Research Compiler* (ORC), which chooses the near optimal factor only 37% of the time. Using nearest neighbor, the authors can construct heuristics that realize running time speedups on the SPEC benchmarks of 6% over ORC with loop unrolling.

Cooper *et al.* [21] use genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding "good" compiler optimization sequences that reduced code size. Unfortunately, their technique is application-specific. That is, a genetic algorithm has to *retrain* for each program to decide the best optimization sequence for that program. The genetic algorithm builds up chromosomes pertaining to different sequences of optimizations and adapts these for each individual program. Mutations can involve adding new optimizations into the sequence or removing existing ones from the sequence. Their technique was successful at reducing code size by as much as 40%.

Alessandro De Gloria and Paolo Faraboschi suggest a method of code compaction on very long instruction word (VLIW) architectures using Boltzmann machines [27]. Code compaction in VLIW architectures is the problem of assigning instructions into a mini-

mum number of large instruction packets (each packet corresponds to a very long instruction word, hence the name VLIW). The problem of code compaction has many of the same properties as instruction scheduling. Boltzmann machines are a class of neural networks that use simulated annealing in their learning procedure [30]. The authors looked at compacting code for seven basic blocks ranging in size from 13 to 39. They show, by performing several hundred annealing iterations, that they are able to achieve performance equivalent to “hand-compiled” code. Unfortunately, like simulated annealing, this method is very computationally intensive, requiring many iterations over a training set before the learning procedure converges. Another problem with the scheduling technique introduced in the paper is that it is allowed to schedule instructions arbitrarily without regard to data dependency constraints. Therefore, along with valid schedules the technique can also produce many invalid schedules. Their solution is to check the validity of the schedule against the original data dependence DAG, but this adds costly computation. Also, a Boltzmann machine must be retrained for each different basic block that is being scheduled. To alleviate the problem of the expensive learning procedure, the authors suggest the possibility of having many Boltzmann machines implemented in hardware and scheduling different basic blocks concurrently.

CHAPTER 8

DISCUSSION

This dissertation describes a process for constructing compiler heuristics automatically that we call *LOCO*. We introduce a taxonomy of compiler heuristics and then use LOCO to construct heuristics for three different compiler problems.

Using LOCO, we construct automatically a heuristic called a *filter* that controls whether or not an optimization is performed. Specifically, we constructed filters to predict when instruction scheduling is beneficial, significantly reducing the cost of scheduling (by 75%) while still retaining most of its benefit of scheduling always.

For the second compiler problem we solved we constructed a heuristic, which we call *hybrid optimization*, that controls which algorithm to use from a set of two or more different register allocation algorithms. Our hybrid register allocator chooses between either graph coloring or linear scan for each method that gets compiled. Hybrid allocators are typically as effective (and in some cases more effective) at achieving low benchmark running times as graph coloring allocators, while significantly reducing the amount of effort spent allocating registers.

Finally, we used LOCO to construct successfully heuristics that control how to schedule instructions. For two different architectures, LOCO induced heuristics that produced code comparable to that produced by hand-tuned heuristics.

We now discuss some general principles learned over the course of this research.

8.1 Features and Heuristics: Simple and Cheap

Constructing features is a very important step in the process of applying LOCO to a compiler problem. The features used must be predictive as well as cheap-to-compute. We found that a little domain knowledge was sufficient to choose good features successfully. For the compiler problems in this dissertation, we were able to construct features that were adequate to solve the problems competently. Also, the cost of computing the features (and the heuristic) is relatively low compared to the cost of the optimization they control.

In Chapter 6 we developed heuristics for a well-known compiler problem (list scheduling) and thus there was ample previous work from which to draw features. It was therefore not completely surprising that we were able to devise a set of features that could be used to construct effective heuristics. In contrast, considering that there was no prior work for the research done in Chapter 4 and Chapter 5, we were surprised to discover that we could develop simple heuristics that solve those problems effectively. We also found that using a technique called *thresholding* allowed us to induce the best performing heuristics.

8.2 Supervised Learning: Excellent Function Inducers

A critical component of the LOCO methodology is supervised learning algorithms. Supervised learning algorithms induce heuristic functions automatically from training data. We found that supervised learning was excellent at solving the problems we tackled. In Chapter 6 we induced heuristics, automatically, using supervised learning, whose performance was comparable to hand-tuned heuristics on this well-studied problem. We also found that a variety of learning algorithms all had low classification error rates and thus performed equally well. We conclude that the compiler problems we tackled were not difficult learning problems.

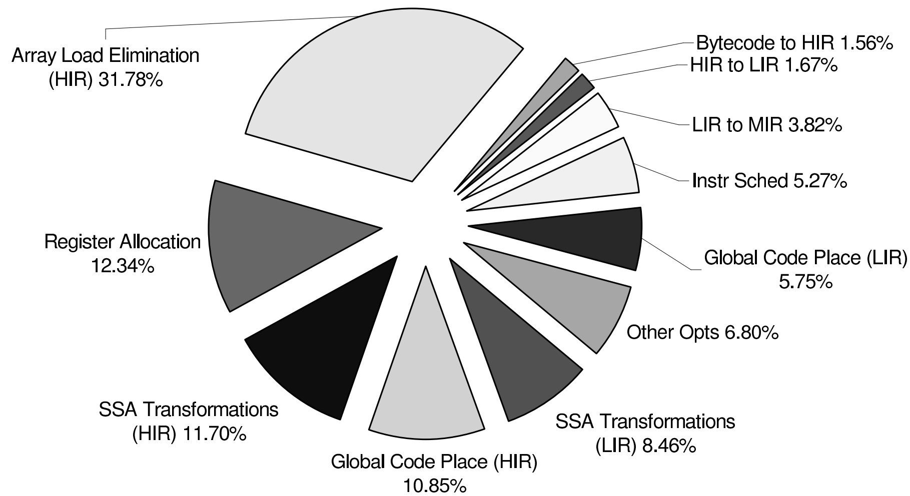


Figure 8.1. Percentage of Time Spent in Compiler Optimizations

8.3 Compiler Optimizations

As mentioned through the course of this research we noticed that there were several different classes of compiler heuristics, which we introduced in Chapter1. We believe we are the first to construct heuristics for two of these classes: *filters* and *hybrid optimizations*. These heuristics are important innovations in compiler research. The research in this dissertation clearly shows that we can use simple models to predict performance adequately on a real machine. The fact that we were able to induce effective heuristics validates this.

We construct heuristics automatically for two important compiler optimizations that, combined, account for about 25% to 30% of compile time. Figure 8.1 shows the percentage of time taken in different optimization phases for *raytrace* compiled at the highest level of optimization possible. These percentages are representative of the other Java benchmarks in this study. *SSA Transformations* pertains to various transformations performed during SSA

(static single assignment) based optimizations, including going into and out of SSA form. From the figure we can see that we might want to build filters and/or hybrid optimizations for optimizations *Array Load Elimination* and *Global Code Placement*.

8.4 Speculations

The ease with which we were able to construct effective feature sets may have something to do with the types of optimizations with which we experimented. All the optimizations we looked at in this dissertation were back-end optimizations; that is, they are applied during the final stages of compilation. The fact that these optimizations transform low-level code makes it easier to estimate the performance effects of transformations as opposed to optimizations done at a higher-level. It may be more difficult to apply LOCO successfully to optimizations closer to the front end of the compiler. In future work we intend on answering this question by applying LOCO to higher-level optimizations.

We have at least some anecdotal evidence that supervised learning is preferable to use for solving compiler problems than other machine learning techniques, such as genetic algorithms or reinforcement learning. Genetic algorithms can take days of computer time to derive a heuristic [52], whereas our supervised learning procedure completes in seconds (once we have developed the training instances). Also, these techniques can take a considerable amount of tuning to achieve the best result. McGovern *et al.* [36, 37] had some success in applying reinforcement learning to solve the same problem in Chapter 6. They achieved performance within a few percent of commercial compilers. However, they never induced a heuristic that beat the tuned heuristics produced by LOCO.

CHAPTER 9

CONCLUSIONS

This dissertation has contributed to both the understanding of compiler heuristics and how to induce compiler heuristics automatically with supervised learning. In this chapter, we recap the key contributions of the dissertation and discuss directions for future research.

9.1 Key Contributions

This dissertation describes a methodology of solving compiler problems we call LOCO. We used LOCO to induce heuristics for three different compiler problems: (1) learning whether to optimize, (2) learning which optimization algorithm to use, and (3) learning how to optimize. Our solution is the first one automatically and quickly to induce effective and efficient heuristics for these three problems. Our primary conclusion of this dissertation is that supervised learning can find, automatically, quite competent compiler heuristics.

9.1.1 Learning Whether to Optimize

Choosing *whether* to apply potentially costly compiler optimizations is an important open problem. In Chapter 4 we considered the particular case of instruction scheduling, with the possible choices being a traditional list scheduler (LS) and no scheduling (NS). In that chapter, we demonstrated that it is possible to induce a function that is competent at making this choice: we obtain almost all the benefit of LS at less than 1/4 of the cost. We also found that thresholds helped in improving *both* the efficiency and the effectiveness of our induced function.

9.1.2 Learning Which Optimization to Use

Choosing *which* optimization algorithm to apply among different optimization algorithms that differ in efficiency and effectiveness can avoid potentially costly compiler optimizations. In Chapter 5 we considered the particular case of register allocation, with the possible choices being linear scan (LS), graph coloring (GC), and hybrid allocators that choose between these two algorithms. What we demonstrated, for an aggressive optimizing compiler (optimization level O3 in Jikes RVM), is that it is possible to induce a function that is competent at making this choice: we obtain almost all the benefit of GC at less than 42% of the cost. This work also shows that thresholds can help in improving the effectiveness and efficiency of our hybrid allocators.

9.1.3 Learning How to Optimize

Choosing *how* to control an optimization algorithm is the task of heuristic functions, typically called priority functions. In Chapter 6 we considered the particular case of inducing the heuristic that controls how to schedule a block. We showed that we could construct heuristics comparable to two vendor production compilers, as shown on two standard benchmark suites.

9.2 Future Directions

We have made significant inroads into understanding and developing a methodology for using supervised learning to solve compiler problems, but substantial work remains toward the goal of achieving a comprehensive study of applying machine learning to compilers. In what follows we discuss several directions for future work.

To date there is only anecdotal evidence to prefer supervised learning over other machine learning algorithms, specifically genetic algorithms and reinforcement learning. It would be interesting to have a head-to-head comparison of these techniques at solving a similar problem. It is clear that supervised learning has the edge in efficiency, but can other

machine learning algorithms find other heuristics that outperform any heuristic induced from supervised learning algorithms?

We would also like to use LOCO to construct heuristics to control other optimizations. More complex compiler optimizations, such as redundancy elimination, almost certainly need more complex features to use in deciding if the optimization is likely to be worthwhile, but we hope that this positive experience will inspire success on harder problems.

I would also like to use techniques inspired by LOCO to induce heuristics for other software systems. For example, one might apply LOCO to construct heuristics that control which garbage collection algorithm to apply at a particular point during a program's execution. There is some preliminary evidence [48] showing that using a technique of dynamically *hot-swapping* different garbage collection algorithms at different points during a program's execution can be beneficial.

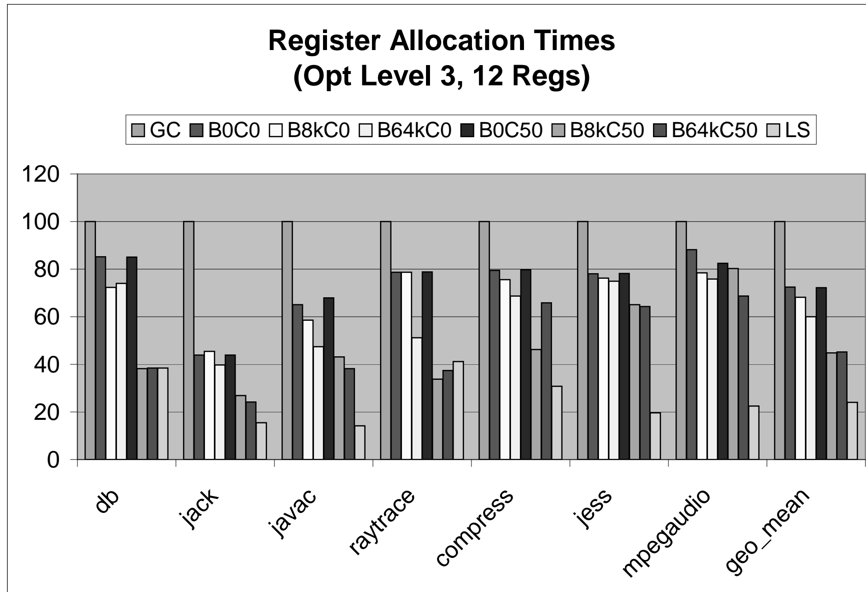
We feel that LOCO could be used for any heuristic found within an optimizing compiler. We intend on applying LOCO to a variety of optimizations, especially ones that can stress the methodology. For example, heuristics used to control compilation for dynamic environments (such as FPGAs) or for large scale multiprocessor systems (such as Grid systems).

This dissertation showed that thresholding is a key component in getting the most benefit of LOCO. Using thresholding involves a fair amount of search to find the best threshold values. We intend on constructing a technique of automating the search process for finding good threshold values.

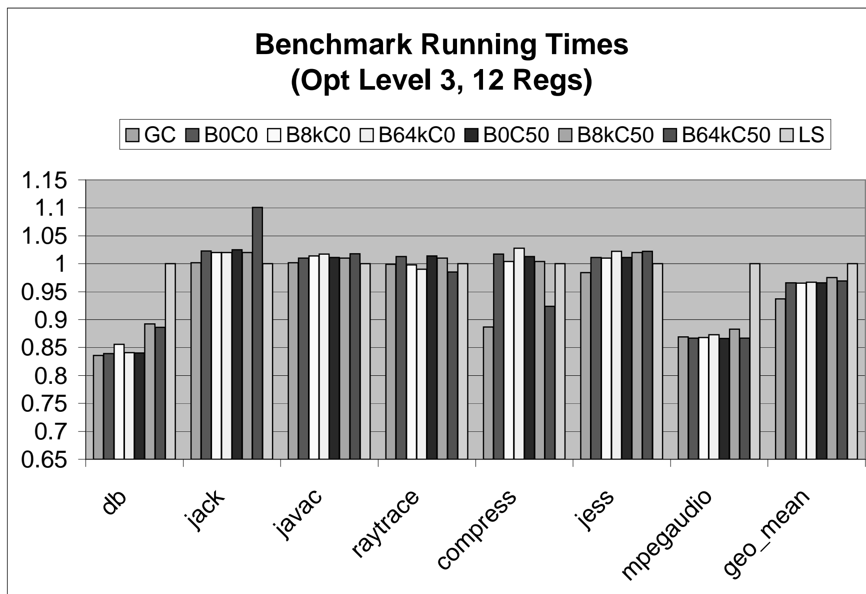
APPENDIX A

HYBRID ALLOCATOR RESULTS FOR OPTIMIZATION LEVEL O3

We present effectiveness results in the following figures (Figures A.1 through A.4) using hybrid, linear scan, and graph coloring allocators for different register set sizes for an aggressive optimization level (O3). This optimization level includes an optimization called *live-range splitting*, which is performed as a side-effect of going into and out of SSA form. This optimization splits live ranges, reducing conflicts between variables, and thus allowing register allocation algorithms to do a better job of allocating registers to variables. These graphs show that as the number of available registers increases, there is less difference in performance of the different allocators. We now highlight some interesting results from this set of graphs. For a register set size of 12, graph coloring has a significant benefit for benchmarks `db`, `compress`, and `mpegaudio`. Our hybrid allocators correctly choose when to apply graph coloring for benchmarks `db` and `compress`. However, Figure A.1 suggests that for `compress`, most of the hybrids (except $B64kC50$) miss allocating one or more important methods with graph coloring and therefore fail to achieve the same level of performance as GC. For a register set size of 16, hybrid allocators achieve the performance of GC on `db` and `mpegaudio`, although there is a large degradation in performance for `jack`. For the rest of the graphs we see similar performance between graph coloring and our hybrids. For this optimization level, we see that the hybrids are between 70% to 40% more efficient that GC.

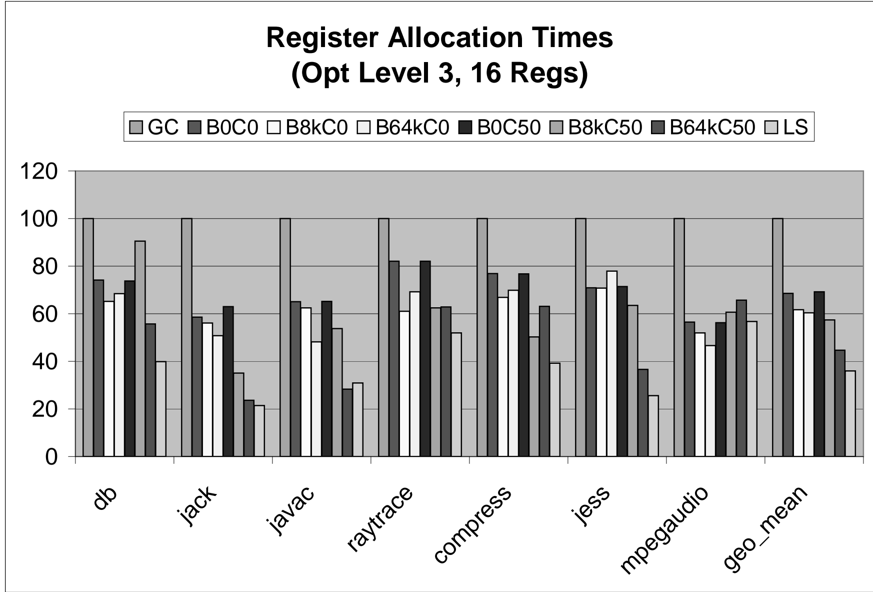


(a) Allocation Time Using Hybrids (opt level O3, 12 Regs)

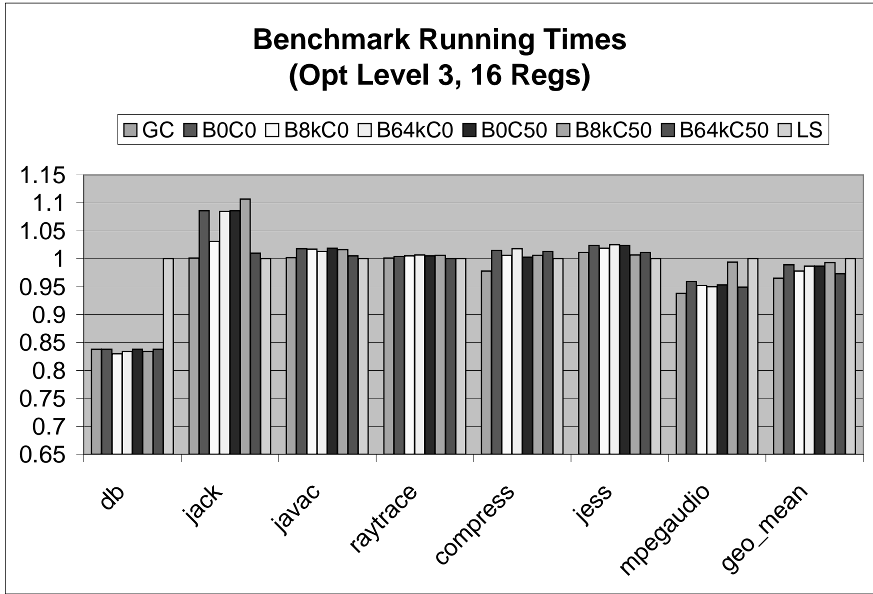


(b) Application Running Time Using Hybrids (opt level O3, 12 Regs)

Figure A.1. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (12 Regs).

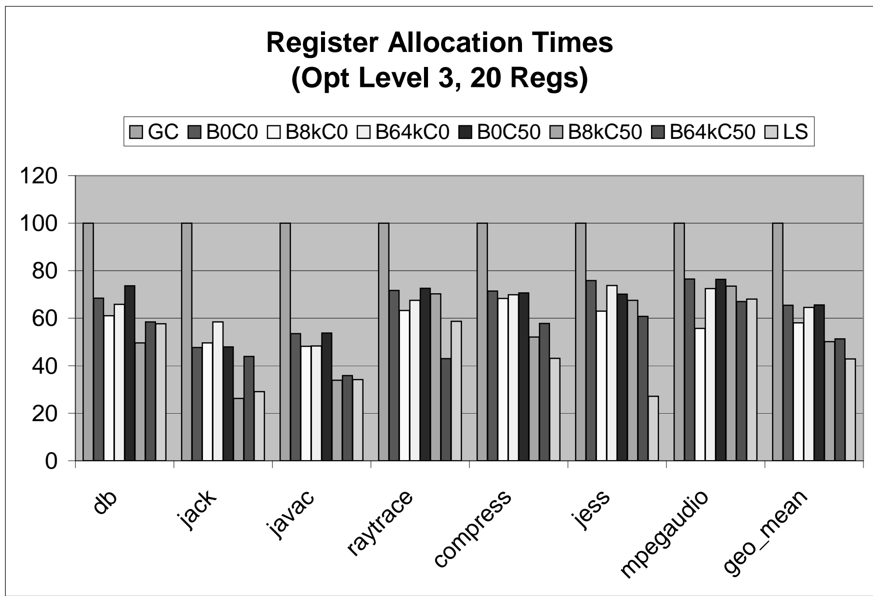


(a) Allocation Time Using Hybrids (opt level O3, 16 Regs)

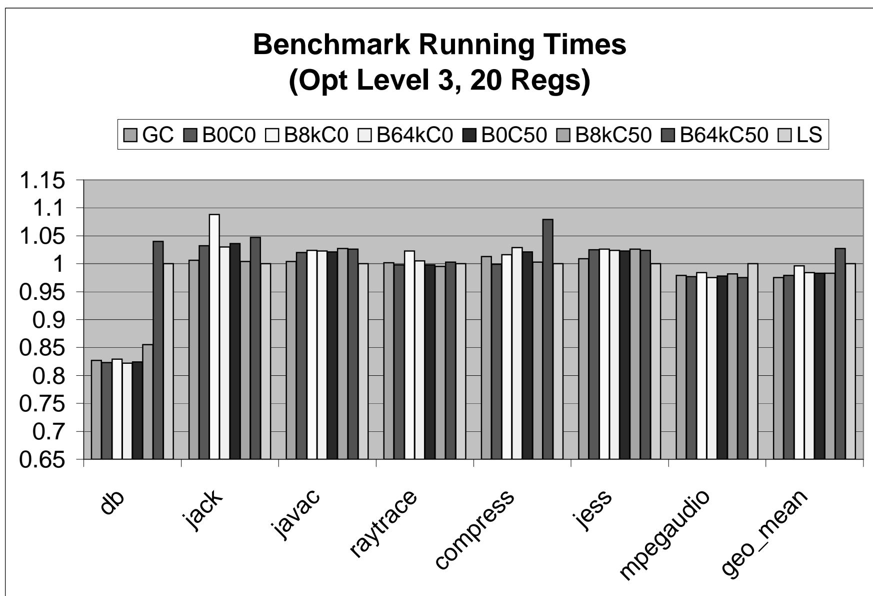


(b) Application Running Time Using Hybrids (opt level O3, 16 Regs)

Figure A.2. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (16 Regs).

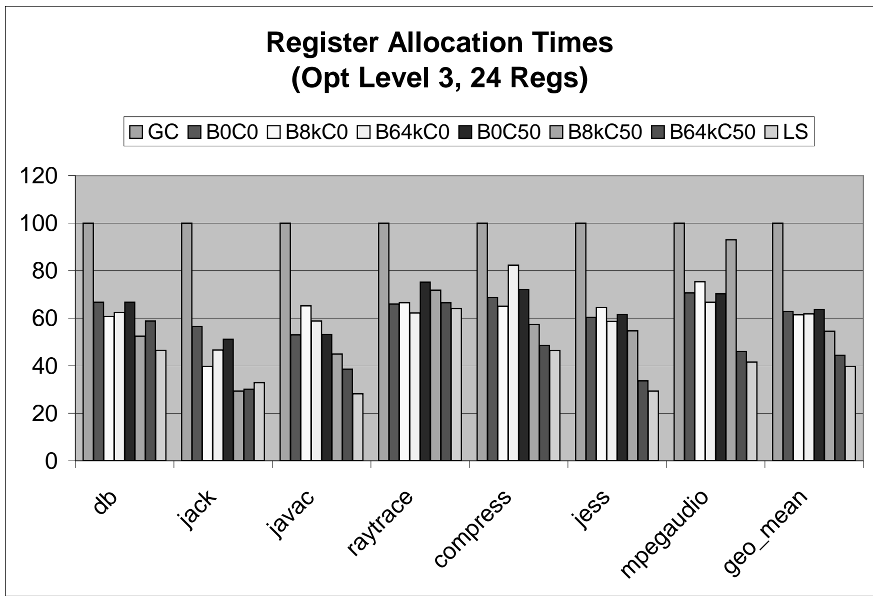


(a) Allocation Time Using Hybrids (opt level O3, 20 Regs)

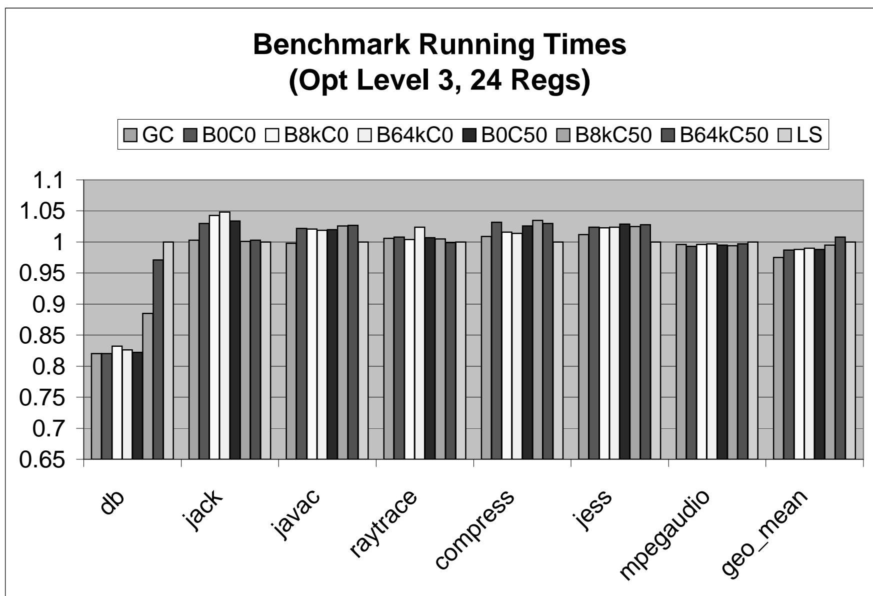


(b) Application Running Time Using Hybrids (opt level O3, 20 Regs)

Figure A.3. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (20 Regs).



(a) Allocation Time Using Hybrids (opt level O3, 24 Regs)



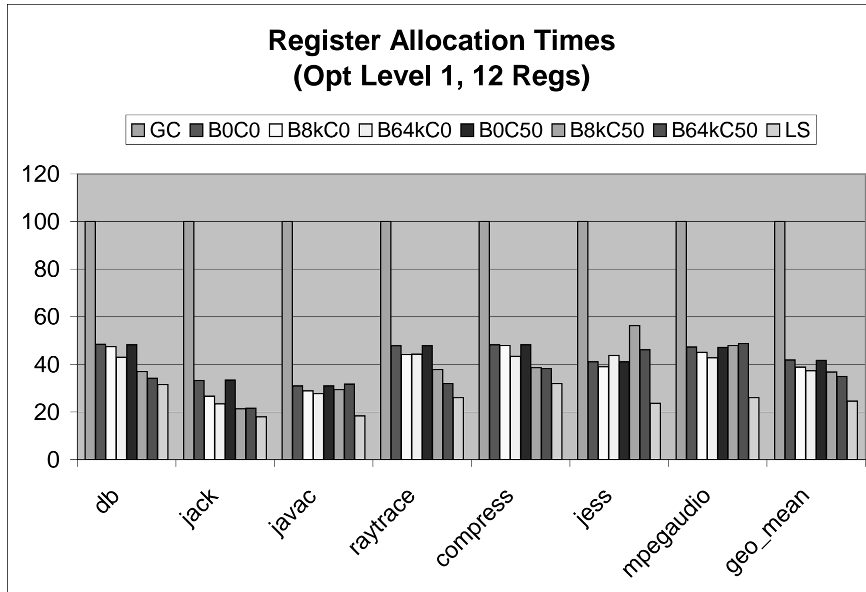
(b) Application Running Time Using Hybrids (opt level O3, 24 Regs)

Figure A.4. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O3 (24 Regs).

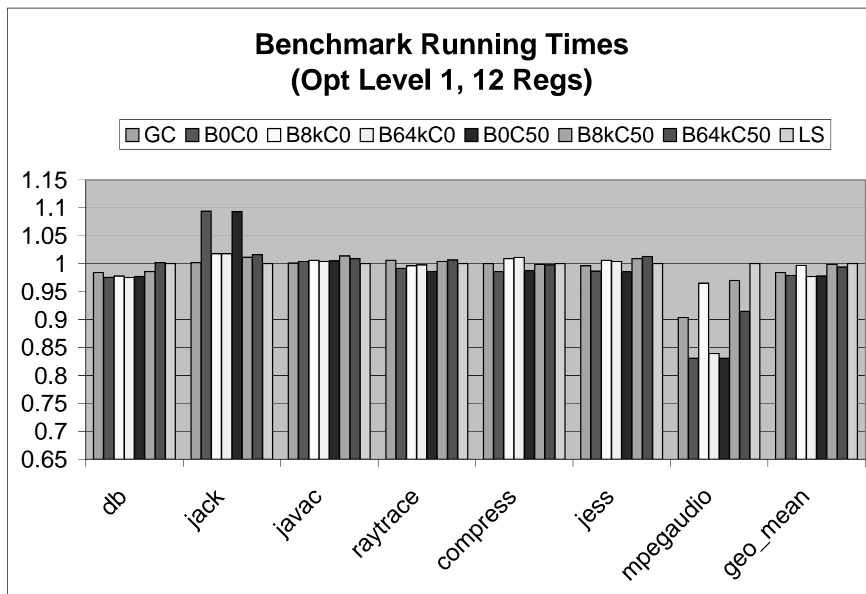
APPENDIX B

HYBRID ALLOCATOR RESULTS FOR OPTIMIZATION LEVEL O1

We present effectiveness results in the following figures (Figures B.1 through B.4) using hybrid, linear scan, and graph coloring allocators for different register set sizes for a less aggressive optimization level (O1). This optimization level includes cheap optimizations. Because this optimization does not include live-range splitting, there is not as large a difference in performance between the different allocators. Again, as the number of available registers increases, graph coloring has less benefit, therefore linear scan should be preferred more often. There are two important differences between the graphs for this optimization level and those for optimization level O3. First, there is a significant drop in register allocation effort when using hybrid for this optimization level. For the most part, the hybrid allocators take 40% of the time to allocate registers than does GC. Also, for this optimization level we see that hybrid allocators can outperform graph coloring. For benchmarks `db` and `mpegaudio` (for all register set sizes), our hybrid allocators show more benefit than always applying graph coloring.

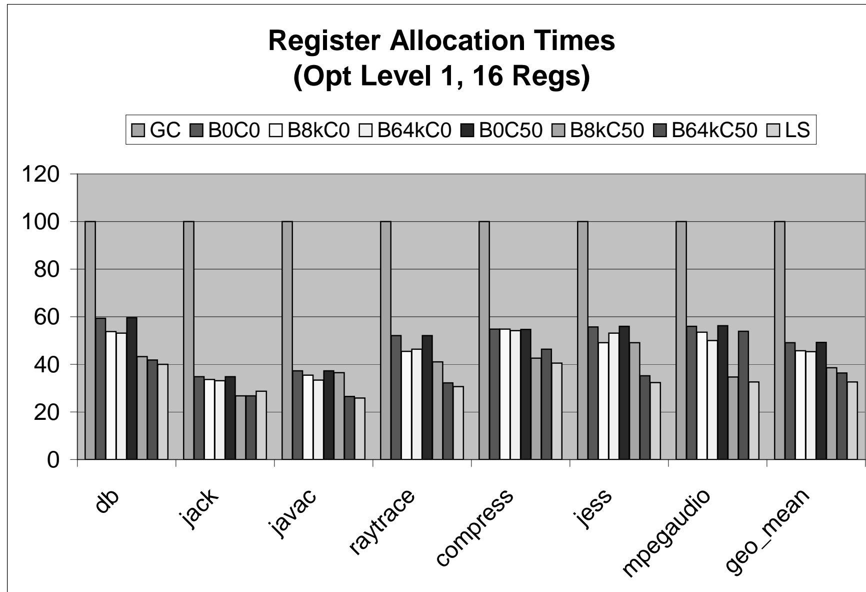


(a) Allocation Time Using Hybrids (opt level O1, 12 Regs)

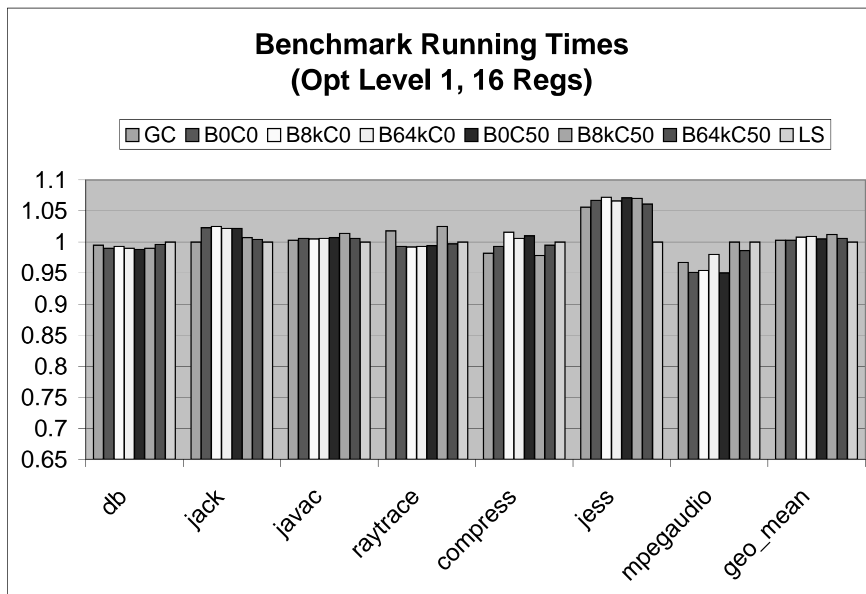


(b) Application Running Time Using Hybrids (opt level O1, 12 Regs)

Figure B.1. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (12 Regs).

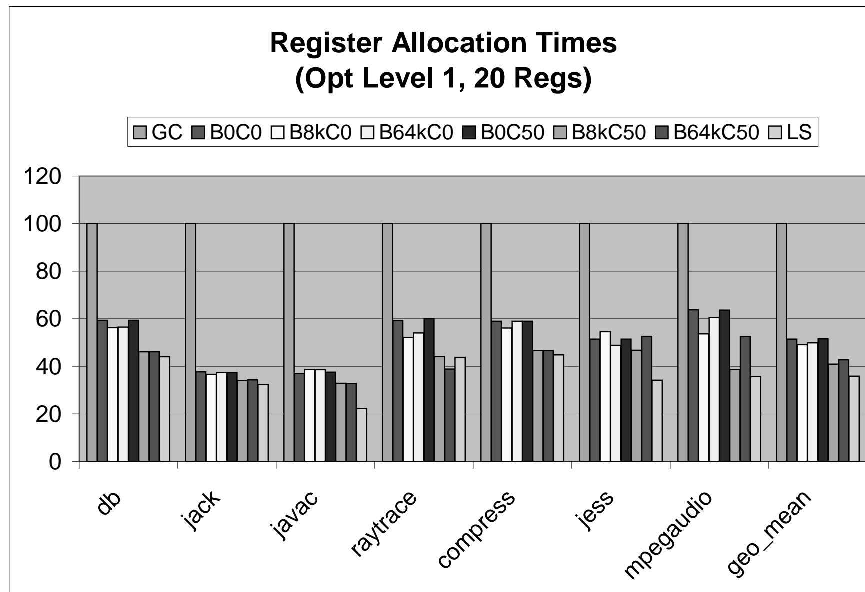


(a) Allocation Time Using Hybrids (opt level O1, 16 Regs)

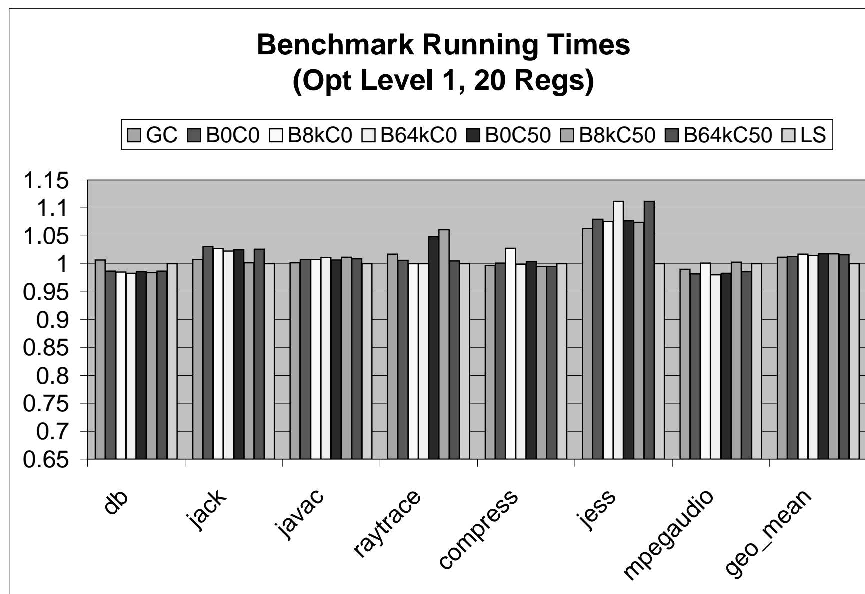


(b) Application Running Time Using Hybrids (opt level O1, 16 Regs)

Figure B.2. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (16 Regs).

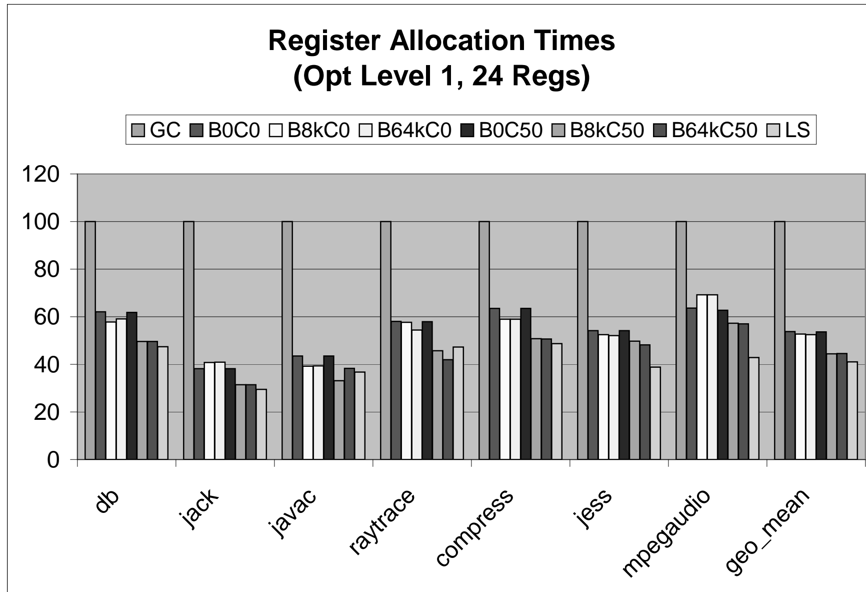


(a) Allocation Time Using Hybrids (opt level O1, 20 Regs)

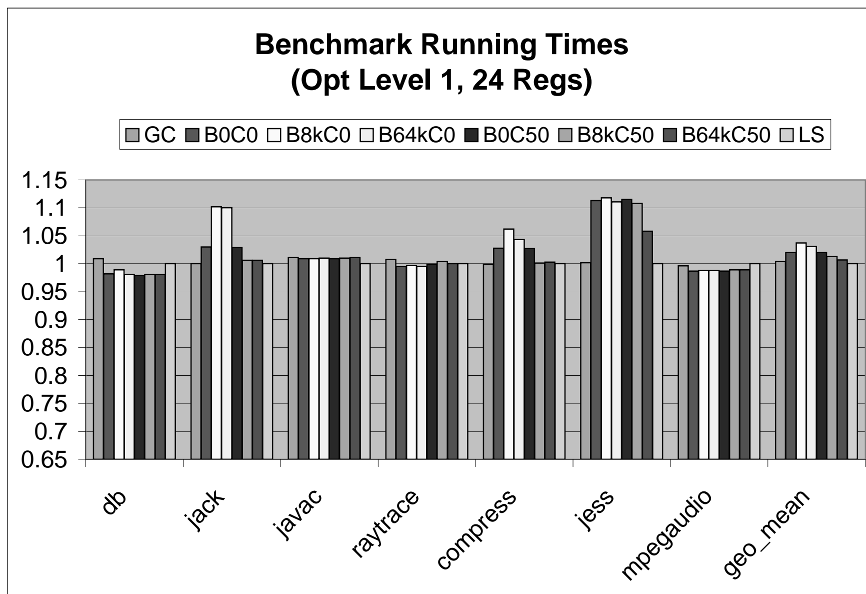


(b) Application Running Time Using Hybrids (opt level O1, 20 Regs)

Figure B.3. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (20 Regs).



(a) Allocation Time Using Hybrids (opt level O1, 24 Regs)



(b) Application Running Time Using Hybrids (opt level O1, 24 Regs)

Figure B.4. Efficiency and Effectiveness Using Hybrid Allocators with Opt Level O1 (24 Regs).

BIBLIOGRAPHY

[1] Santosh G. Abraham, Waleed Meleis, and Ivan D. Baev. Efficient backtracking instruction schedulers. In *IEEE PACT*, pages 301–308, Philadelphia, PA, October 2001. IEEE Computer Society.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 314–324, Denver, CO, November 1999. ACM Press.

[4] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño—a compiler-supported Java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 99)*, Atlanta, GA, May 1999. ACM Press.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Minneapolis, MN, October 2000. ACM Press.

[6] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, UT, June 2001. ACM Press.

[7] BEA. BEA Weblogic JRockit: Java for the Enterprise. http://www.bea.com/products/weblogic/jrockit/whitepapers.shtml.

[8] Steven J. Beaty, Scott Colcord, and Philip H. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics. In *Proceedings of the International Conference on Massively Parallel Computer Systems*, Ischia, Italy, May 1996. IEEE Computer Society.

[9] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, Portland, Oregon, June 1989. ACM Press.

[10] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. P. Grove, S. O. Hobbs, and W. B. Noyce. The GEM optimizing compiler system. Digital Equipment Corporation Technical Journal, Special Issue 1992.

[11] Daniel Brélaz. New methods to color vertices of a graph. *Communications of the ACM*, 22(4):251–256, April 1979.

[12] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 275–284. ACM Press, June 1989.

[13] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V. C. Sreedhar, and Harini Srinivasan. The Jalapeño dynamic optimizing compiler for Java. In *1999 ACM Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999. ACM Press.

[14] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[15] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Conference on Programming Language Design and Implementation*, pages 98–101, Boston, Massachusetts, June 1982. ACM Press.

[16] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio Serrano, and Harini Srinivasan. Dependence analysis for Java. In *1999 Workshop on Languages and Compilers for Parallel Computing*, pages 35–52, La Jolla, CA, August 1999. Springer.

[17] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B.R. Rau, and M.Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 58–67, Paris, France, December 1996. IEEE Computer Society.

[18] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, pages 21–31, Toulouse, France, September 1999. ACM Press.

[19] Fred C. Chow and John L. Hennessey. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[20] William W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, Lake Tahoe, CA, November 1995. Morgan Kaufmann.

[21] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, Georgia, July 1999. ACM Press.

[22] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, August 2002.

[23] Digital Equipment Corporation, Maynard, MA. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, first edition, October 1992.

[24] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale, February 1985.

[25] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[26] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings ACM SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 11–16, Palo Alto, California, June 1986. ACM Press.

[27] Alessandro De Gloria and Paolo Faraboschi. A Boltzmann machine approach to code optimization. *Parallel Computing*, 17(9):969–982, December 1991.

[28] Alessandro De Gloria, Paolo Faraboschi, and Mauro Olivieri. A non-deterministic scheduler for a software pipelining compiler. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 41–44, Portland, Oregon, December 1992. IEEE Computer Society.

[29] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen mei W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 247–255, Austin, Texas, December 1993. IEEE Computer Society.

[30] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, NY, 1994.

[31] John R. Koza. Genetic programming. In James G. Williams and Allen Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 39, pages 29–43. Marcel-Dekker, 1998.

[32] Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518, Stanford, CA, June 2000. Morgan Kaufmann.

[33] Daniel M. Lavery, Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

[34] S. Lin and B. W. Kernighan. An effective heuristic procedure for the traveling salesman problem. *Operations Research*, 21:498–516, 1971.

[35] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, Koichiro Hotta, and Hiromitsu Takagi. OpenJIT A Reflective Java JIT compiler. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, Vancouver, British Columbia, Canada, 1998. ACM Press.

[36] Amy McGovern and Eliot Moss. Scheduling straight-line code using reinforcement learning and rollouts. In Sara Solla, editor, *Advances in Neural Information Processing Systems 11*, pages 903–909, Denver, Colorado, December 1998. MIT Press.

[37] Amy McGovern, Eliot Moss, and Andrew G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning*, 49(2/3):141–160, November-December 2002.

[38] Antoine Monsifrot and F. Bodin. A machine learning approach to automatic production of compiler heuristics. In *Tenth International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA)*, pages 41–50, Varna, Bulgaria, September 2002. Springer Verlag.

[39] J. Eliot B. Moss, Paul E. Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla E. Brodley, and David T. Scheeff. Learning to schedule straight-line code. In *Proceedings of Advances in Neural Information Processing Systems 10*, pages 929–935, Denver, Colorado, December 1997. MIT Press.

[40] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[41] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[42] John Ross Quinlan. *C4.5 Programs For Machine Learning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1987.

[43] John Ross Quinlan. Simplifying decision trees. *Special Issue: Knowledge Acquisition for Knowledge-based Systems. Part 5*, 27(3):221–234, September 1987.

[44] D. E. Rumelhart, G. E. Hinton, and R. J Williams. Learning internal representations by error propagation. In Rumelhart and McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[45] Philip J. Schielke. *Stochastic Instruction Scheduling*. PhD thesis, Rice University, November 2000.

[46] Richard Sites. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, Maynard, MA, 1992.

[47] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 93–102, Albuquerque, New Mexico, November 1991. IEEE Computer Society.

[48] Sunil Soman, Chandra Krintz, and David Bacon. Adaptive, application-specific garbage collection, 2003.

[49] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994. ACM Press.

[50] Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM98 Benchmarks*, 1998.

[51] Darko Stefanovic. The character of the instruction scheduling problem. OSL+SAA Memo 1997-01-V1. Object Systems Laboratory, Department of Computer Science, University of Massachusetts, March 1997.

[52] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90, San Diego, Ca, June 2003. ACM Press.

[53] Sun. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[55] Gregory Tarsy and Michael J. Woodard. Method and apparatus for optimizing cost-based heuristic instruction schedulers. US Patent # 5,367,687, 1994. Filed 7/7/93, granted 11/22/94.

[56] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 142–151, Montreal, Canada, June 1998. ACM Press.

[57] P. E. Utgoff and D. Precup. Constructive function approximation. Technical Report 97-04, University of Massachusetts, Department of Computer Science, Amherst, MA, 1997.

[58] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, October 1997.

[59] Paul E. Utgoff and Sharad Saxena. Learning a preference predicate. In *Proceedings of the 4th International Conference on Machine Learning*, pages 115–121, Irvine, CA, June 1987. Morgan Kaufmann.

[60] Vladimir N. Vapnik. *Statistical Learning Theory*. John Wiley & sons, Hoboken, NJ, September 1998.

[61] Steven R. Vegdahl. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 150–154, Alanta, Ga, May 1999. ACM Press.

[62] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 121–133, Vancouver, BC, Canada, June 2000. ACM Press.

[63] Kamen Yotov, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 63–76, San Diego, Ca, June 2003. ACM Press.

[64] Cliff Young and Michael Smith. Better global scheduling using path profiles. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 199–206, Ann Arbor, MI, November 1995. IEEE Computer Society.

[65] Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1114–1120, Montreal, Canada, August 1995. Morgan Kaufmann.