Towards a More Principled Compiler: Register Allocation and Instruction Selection Revisited

David Ryan Koes

CMU-CS-09-157

October 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Seth Copen Goldstein, Chair
Peter Lee
Anupam Gupta
Michael D. Smith, Harvard University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2009 David Ryan Koes

This research was sponsored by the National Science Foundation under grant numbers CCF-0702640, CCR-0205523, EIA-0220214, and IIS-0117658; and Hewlett Packard under grant number 1010162.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Compilers, Register Allocation, Instruction Selection, Backend Optimization

For Mary, Andrew, and Alex
But especially for Mary

Abstract

Backend optimizations are a critical part of an optimizing compiler. This thesis develops a principled approach for understanding, evaluating, and solving backend optimization problems. Our principled approach is to develop a comprehensive and expressive model of the backend optimization problem, and design solution techniques for this model that achieve or approach optimality. We apply our principled approach to the classical backend optimizations of register allocation and instruction selection.

We develop an expressive model of register allocation based on multi-commodity network flow. This model exactly represents the complexities of the target architecture. We design progressive solution techniques for our model. Progressive solution techniques quickly find an initial solution and then improve upon the solution as more time is allotted for compilation. Our progressive allocator allows the programmer to explicitly manage the trade-off between compile-time and code quality. As more time is allowed for compilation, the resulting allocation approaches optimal, and substantial improvements in code quality are obtained.

We describe an expressive directed acyclic graph representation of the instruction selection problem and develop a near-optimal, linear-time algorithm that solves the instruction selection problem using this expressive model. Our principled approach to instruction selection results in significant improvements in code quality compared to traditional algorithms.

We evaluate our principled approaches to register allocation and instruction selection on a range of architectures and benchmarks. We achieve significant reductions in code size and increases in performance relative to previous approaches. Our results confirm that our principled approach is a major advance in the state of the art of backend optimization.

Contents

1 Introduction **1**
- 1.1 Problem Description . . . 4
  - 1.1.1 Register Allocation . . . 4
  - 1.1.2 Instruction Selection . . . 6
- 1.2 Contribution . . . 7

2 Related Work **9**
- 2.1 Register Allocation . . . 9
  - 2.1.1 Graph Coloring Register Allocation . . . 10
  - 2.1.2 SSA Register Allocators . . . 14
  - 2.1.3 Linear Scan Allocators . . . 15
  - 2.1.4 Alternative Heuristic Allocators . . . 20
  - 2.1.5 Optimal Register Allocation . . . 21
  - 2.1.6 Limitations . . . 22
  - 2.1.7 Summary . . . 24
- 2.2 Instruction Selection . . . 25

3 Global MCNF Register Allocation Model **29**
- 3.1 Multi-commodity Network Flow . . . 29
- 3.2 Local Register Allocation Model . . . 33
  - 3.2.1 Source Nodes . . . 35
  - 3.2.2 Sink Nodes . . . 36
  - 3.2.3 Allocation Class Nodes . . . 37
  - 3.2.4 Crossbar Groups . . . 38
  - 3.2.5 Instruction Groups . . . 41
  - 3.2.6 Full Model . . . 42
- 3.3 Global Register Allocation Model . . . 44
- 3.4 Persistent Memory . . . 50
- 3.5 Modeling Costs . . . 52
- 3.6 Limitations . . . 58
- 3.7 Hardness of Single Global Flow . . . 61
- 3.8 Simplifications . . . 64
- 3.9 Summary . . . 66

4 Evaluation Methodology **67**

4.1 Benchmarks 67
4.2 Code Quality Metrics 68
4.2.1 Code Size 68
4.2.2 Code Performance 69
4.3 Instruction Set Architectures 71
4.3.1 x86-32 71
4.3.2 x86-64 71
4.3.3 ARM 72
4.3.4 Thumb 72
4.4 Microarchitectures 73

5 Heuristic Register Allocation **75**
5.1 Iterative Heuristic Allocator 75
5.1.1 Algorithm 76
5.1.2 Improvements 81
5.1.3 Asymptotic Analysis 87
5.2 Simultaneous Heuristic Allocator 88
5.2.1 Algorithm 88
5.2.2 Improvements 96
5.2.3 Asymptotic Analysis 103
5.3 Boundary Constraints 105
5.3.1 Asymptotic Analysis 108
5.4 Hybrid Allocator 108
5.5 Compile Time 113
5.6 Summary 114

6 Progressive Register Allocation **115**
6.1 Relaxation Techniques 115
6.1.1 Linear Programming Relaxation 116
6.1.2 Lagrangian Relaxation 119
6.2 Subgradient Optimization 121
6.2.1 Flow Calculation 123
6.2.2 Step Update 125
6.2.3 Price Update 129
6.2.4 Price Initialization 132
6.2.5 Summary 136
6.3 Progressive Register Allocation 136
6.3.1 Code Quality: Size 138
6.3.2 Code Quality: Performance 141
6.3.3 Optimality 147
6.3.4 Compile Time 148
6.4 Summary 150

7 Near-Optimal Linear-Time Instruction Selection **151**

7.1 Problem Description and Hardness 151
7.2 NOLTIS 155
7.3 0-1 Programming Solution 161
7.4 Implementation 162
7.5 Results 163
7.5.1 Optimality 163
7.5.2 Comparison of Algorithms 166
7.5.3 Impact on Code Size 167
7.5.4 Compile Time Performance 167
7.6 Limitations and Future Work 168
7.7 Interaction with Register Allocation 169
7.8 Summary 170

8 Conclusion **171**

Bibliography **173**

List of Figures

1.1 The structure of a typical compiler. 2
1.2 Simple register allocation example. 4
1.3 An example of instruction selection on a tree-based IR. 6

2.1 The flow of a traditional graph coloring algorithm. 10
2.2 Live ranges and the corresponding interference graph. 10
2.3 An example of the simplify and select phases of a graph coloring allocator. 11
2.4 The linear ordering of basic blocks, live intervals, and lifetime holes. 16
2.5 Result of simple linear scan and second-chance binpacking linear scan. 17
2.6 Percent of functions which do not spill. 23
2.7 Decrease in code quality resulting from spill code and assignment heuristics. . . . 24
2.8 The effect of various components of register allocation. 25

3.1 A simple example of a multi-commodity network flow problem. 30
3.2 A simple example of local register allocation. 34
3.3 Source nodes of a MCNF model of register allocation. 35
3.4 Sink nodes of a MCNF model of register allocation. 37
3.5 Crossbar groups for the local register allocation problem of Figure 3.2. 38
3.6 Two possible crossbar group network structures. 39
3.7 Instruction groups for the local register allocation problem of Figure 3.2. 41
3.8 The full MCNF model of the local register allocation problem of Figure 3.2. . . . 43
3.9 A simple control flow graph. 44
3.10 The three types of flow nodes in the global MCNF model of register allocation. . 44
3.11 Entry and exit groups of a global MCNF model of register allocation. 45
3.12 A crossbar group with nodes for anti-variables. 48
3.13 A network that demonstrates value modification, load remat. and anti-variables. . 49
3.14 The accuracy of the code size global MCNF cost mode. 53
3.15 Impact of single-execution costs on dynamic memory operations. 54
3.16 Impact on performance of varying single-execution costs. 55
3.17 Decrease in code quality when coalescing is separated from an optimal allocator . 58
3.18 An example of a reduction from global MCNF to minimum graph labeling. . . . 62
3.19 Decrease in code quality when move insertion is restricted in an optimal allocator. 65

5.1 An example of the behavior of the iterative heuristic allocator. 78
5.2 A simple example of global variable usage. 81
5.3 The importance of block ordering in the iterative allocator. 84

5.4 The importance of tie breaking strategies in the iterative allocator. 85
5.5 Running time of iterative allocator for all benchmarked functions. 87
5.6 Example execution of the simultaneous heuristic allocator. 90
5.7 Example eviction decisions in the simultaneous heuristic allocator. 94
5.8 Effect of tie breaking heuristics on code quality in the simultaneous allocator . . 97
5.9 An example control flow graph decomposed into traces. 98
5.10 Effect of trace decompositions on code quality in the simultaneous allocator. . . . 99
5.11 Effect of trace update policy on code quality in the simultaneous allocator 102
5.12 Running time of the simultaneous allocator for all benchmarked functions. 104
5.13 A CFG that illustrates the subtleties of setting boundary constraints. 105
5.14 Code size improvement of heuristic allocators. 109
5.15 Code size improvement of heuristic allocators. 110
5.16 Memory operation reduction of heuristic allocators. 111
5.17 Average code quality improvement of heuristic allocators 112
5.18 Slowdown of various allocators relative to extended linear scan. 113

6.1 The percentage of functions that demonstrate an integrality gap. 117
6.2 Linear programming solution times of the global MCNF problem. 118
6.3 Convergence behavior of the basic subgradient optimization algorithm. 122
6.4 Convergence of subgradient optimization with different flow calculations. 124
6.5 Graphical depiction of five ratio step update rules. 125
6.6 Convergence of subgradient optimization with different step update rules. 126
6.7 Convergence of the subgradient optimization with Newton's method step update. 128
6.8 Example price behavior using different price update strategies. 129
6.9 Convergence of subgradient optimization with different price update strategies. . 131
6.10 Effect of price initialization on the initial lower bound. 134
6.11 Convergence of subgradient optimization with different price initializations. . . . 134
6.12 Convergence of heuristic price initialization with different initial allocations. . . . 135
6.13 The behavior of three heuristic allocators within a progressive allocator. 137
6.14 Average code size improvement of the progressive allocator. 138
6.15 Code size improvement of the progressive allocator. 139
6.16 Code size improvement of the progressive allocator. 140
6.17 Average memory operation reduction of the progressive allocator. 142
6.18 Average performance improvement of the progressive allocator. 142
6.19 Memory operation reduction of the progressive allocator. 143
6.20 Code performance improvement of the progressive allocator for x86-32. 144
6.21 Code performance improvement of the progressive allocator for x86-64. 145
6.22 Effect of block frequency estimator on code quality. 146
6.23 Code size optimality bounds of progressive allocator. 148
6.24 Code performance optimality bounds of progressive allocator. 149
6.25 Register allocation time breakdown of progressive allocator. 149

7.1 An example of instruction selection as a tiling problem. 152
7.2 Expressing Boolean satisfiability as an instruction selection problem. 154

7.3 An example of instruction selection on a DAG. 158
7.4 The application of the NOLTIS algorithm to the example from Figure 7.3. 160
7.5 Improvement in tiling cost relative to maximal munch 164
7.6 Final code size improvement relative to maximal munch 165
7.7 Cumulative total improvement relative to maximal munch 166
7.8 Average slowdown of each instruction selection algorithm relative to *cse-all* 167
7.9 Influence of constant rematerialization on final code size 168

List of Pseudocode

5.1 CONSTRUCTFEASIBLESOLUTIONITERATIVE . . . 76
5.2 MARKFEASIBLEPATH . . . 76
5.3 SHORTESTFEASIBLEPATH . . . 77
5.4 FEASIBLEEDGECOST . . . 79
5.5 CONSTRUCTFEASIBLESOLUTIONSIMULTANEOUS . . . 88
5.6 ALLOCATEVARATLAYER . . . 91
5.7 ASSIGNVARTONODE . . . 91
5.8 PROPAGATEALLOCS . . . 92
5.9 NODEVALIDFORALLOCATION . . . 92
5.10 SELECTALLOCCLASS . . . 93
5.11 EVICTIONCOST . . . 94
5.12 SETBOUNDARYCONSTRAINTS . . . 106
5.13 SETUNUSABLE . . . 107
5.14 VIOLATESBOUNDARYCONSTRAINT . . . 107
6.1 CALCULATENODEPRICEWEIGHT . . . 132
6.2 PRICEINITIALIZATION . . . 133
7.1 SELECT, BUTTOMUPDP, TOPDOWNSELECT . . . 156
7.2 IMPROVECSEDECISIONS . . . 157
7.3 GETOVERLAPCOST . . . 157
7.4 GETTILECUTCOST . . . 159

List of Tables

3.1 Reduced benchmark suite suitable for optimal allocation. . . . 52

4.1 Benchmarks used in evaluation of code quality. . . . 68
4.2 Characteristics of microarchitectures used to evaluate performance. . . . 72

6.1 Example price behavior using different price update strategies. . . . 130

Chapter 1

Introduction

Compilers are a ubiquitous and essential tool, and improvements to compiler technology have a broad impact. A compiler is responsible for translating human-understandable source code, such as C or Java, into machine-executable binary object code. The tool-flow of a typical compilation system is shown in Figure 1.1. The front-end of the compiler translates the source code of a program into a target-independent intermediate representation (IR). After target-independent optimization, the backend of the compiler translates the IR code into a target-specific assembly listing.

Backend optimizations are a critical part of an optimizing compiler. These optimizations are responsible for fully exploiting the complex and varied features of modern architectures. Since backend optimization problems are typically NP-complete, the predominant approach to solving these problems has been an amalgamation of ad-hoc heuristics. This thesis develops a principled approach for understanding, evaluating, and solving backend optimization problems. Our principled approach is to

- develop a comprehensive and *expressive model* of the backend optimization problem, and
- design solution techniques that use this model to *achieve or approach the optimal solution*.

Our principled approach is a departure from conventional approaches. Existing backend optimizations often have no explicit model of the problem. Instead, *ad hoc* heuristics are used. Alternatively, if there is an underlying model, the model design does not fully capture all the features of the optimization problem, but instead abstracts away the complexities of the prob-

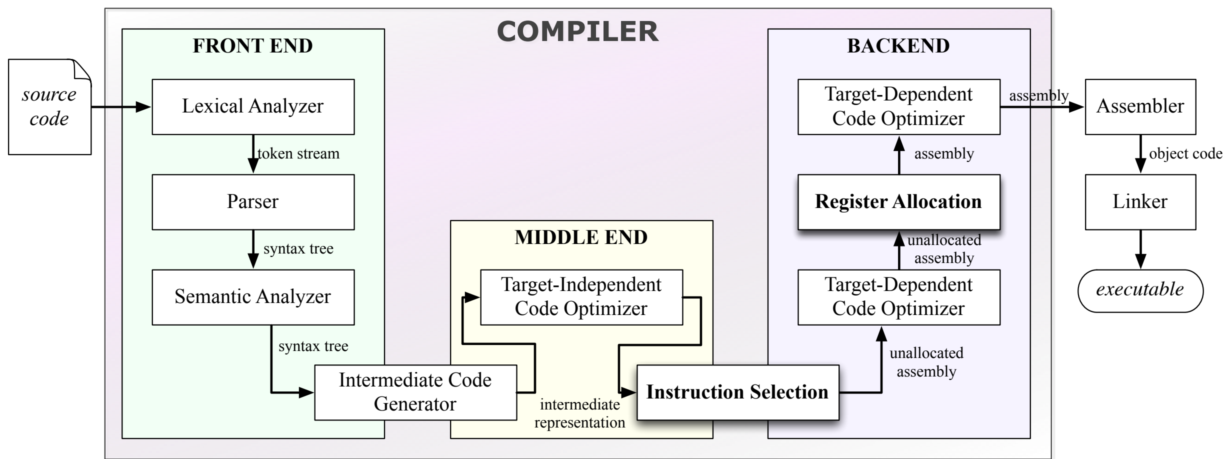


Figure 1.1: The structure of a typical compiler.

lem into a form that is more readily solved. In contrast, our approach embraces the complexity of the problem. We do not compromise the fidelity of the model to accommodate algorithmically simple solution techniques. Instead, we develop solution techniques that exploit the full expressiveness of the model. If it is not practical to obtain optimal or near-optimal solutions, we develop progressive solution techniques.

Progressive solution techniques approach the optimal solution as more time is alloted for compilation. Unlike conventional approaches, our focus when developing solution techniques is to achieve the highest quality code even at the expense of substantial increases in compilation time. Progressive compilation makes this approach practical. The programmer, not the compiler developer, decides what the appropriate trade-off is between compile time and code quality.

We apply our principled approach to the key backend optimization problems of *register allocation* and *instruction selection*. We find that our principled approach, which uses an expressive model and solution techniques that achieve or approach optimality, results in better code quality and a better compiler.

EXPRESSIVE MODEL

Many compiler optimization passes use a simplified model of the target architecture and, as a result, can actually produce less optimized code. Even optimizations that are intrinsically linked to architectural features, such as register allocation, use inappropriately simple architectural mod-

els. For example, traditional register allocators were designed for regular, RISC-like architectures with large uniform register sets. Embedded architectures, such as the 68k, ColdFire, x86, ARM Thumb, MIPS16, and NEC V800 architectures, tend to be irregular, CISC architectures. These architectures may have small register sets, restrictions on how and when registers can be used, support for memory operands within arbitrary instructions, variable sized instructions, or other features that complicate register allocation. The register allocator in a principled compiler needs to explicitly represent and optimize for these features.

The first step of our principled approach is to define an expressive model of the optimization problem that accurately represents the pertinent features of the problem. Chapter 3 discusses our novel model of register allocation using a multicommodity network flow framework that accurately represents the intricacies of register allocation for irregular architectures. Chapter 7 describes an expressive directed acyclic graph model for instruction selection that, unlike conventional tree-based representations, explicitly represents redundant expressions.

OPTIMALITY

The general optimization problem, finding a correct instruction sequence that results in the best code quality, is provably undecidable since such an optimizer could be used to solve the halting problem. Instead, we consider the *internal optimality* of an optimization pass. An optimization pass is internally optimal if it optimally performs its specific optimization goal. For example, dead-code elimination can eliminate all code that is dead in a meets over all paths static analysis. Dead-code elimination is internally optimal: given a restricted, but reasonable, definition of the problem (remove all static dead code) it finds the optimal result. Many compiler optimizations, including register allocation and instruction scheduling, are provably NP-hard for even simple representations of the problem. In these cases, it is unlikely that efficient internally optimal algorithms exist. Instead, we propose the use of *progressive compilation*.

Progressive compilation bridges the gap between fast heuristics and slow optimal algorithms. A progressive algorithm quickly finds a good solution and then progressively finds better solutions until an optimal solution is found or a preset time limit is reached. The use of progressive solution techniques fundamentally changes how compiler optimizations are enabled. Instead

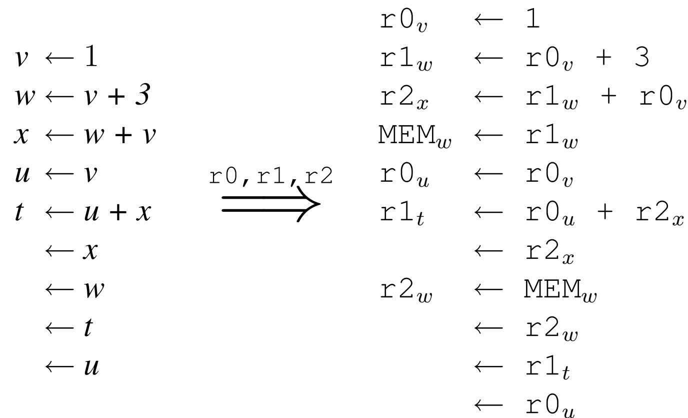


Figure 1.2: A simple example of register allocation. In this example there are only three registers. After the definition of t there are four live variables, x, w, t, and u, so it is necessary to spill a variable to memory, in this case w.

of selecting a discrete optimization level, a programmer *explicitly* trades compilation time for improved optimization.

The second step of our principled approach is to develop near-optimal or progressive algorithms for the expressive model of the optimization problem. Chapters 5 and 6 describe novel progressive algorithms for solving the global MCNF model of register allocation and Chapter 7 describes a novel near-optimal instruction selection algorithm.

1.1 Problem Description

Register allocation and instruction selection are essential passes of any compiler backend. As can be seen in Figure 1.1, together they are responsible for finalizing a compiler's intermediate representation of code into machine executable assembly. In this section we define what these passes entail and characterize their difficulty.

1.1.1 Register Allocation

The *register allocation problem* is to find a desirable assignment of program variables to memory locations and hardware registers as illustrated in Figure 1.2. Various metrics, such as execution speed, code size, or energy usage, can be used to evaluate the desirability of the allocation.

Local register allocation considers only the task of allocating a single basic block (an instruction sequence containing no control flow). *Global register allocation* finds an allocation for an entire function. Inter-procedural register allocation is typically not done; instead, calling conventions dictate the use of registers across function boundaries.

The *register sufficiency problem*, which is often confused with the register allocation problem, is to determine, for a particular function, if it is possible to find an assignment of variables to the available registers. In other words, it is not necessary to *spill* (i.e., store to memory) a variable. It is this problem that Chaitin et. al. [32] proved to be NP-hard for arbitrary control flow graphs. However, later work has shown that program structure can be exploited to more easily solve the register sufficiency problem [19]. For programs with bounded treewidth [15], including all programs written in Java and `goto`-free C [56, 124], the register sufficiency problem can be solved in linear time (but exponential in the fixed number of registers) [14, 101]. Alternatively, constant factor approximation algorithms can be used [68, 124]. For programs that are in SSA form, the register sufficiency problem is also readily solved [19, 25, 58, 59, 102], although optimally converting out of SSA form remains difficult [105].

Although the register sufficiency problem is readily solved, there is much more to the problem of register allocation than register sufficiency. Other important components of the register allocation problem are *spill code optimization*, *rematerialization*, *coalescing*, and *register preferences*. When program variables cannot be allocated solely to registers, it is necessary to generate spill code to store and load values to and from memory. Determining the minimum number of loads and stores needed is NP-hard even for local register allocation [20, 42]. In some cases the register allocator may be able to avoid spilling by rematerializing a known value. In addition, the register allocator may be able to improve code quality by allocating two variables to the same register. For example, if the two variables are joined by a move instruction it may be possible to *coalesce* the variables into the same register and eliminate the need for the move instruction. Optimal coalescing is NP-hard, even for structured programs [18]. Finally, many architectures, such as the x86 architecture, do not have uniform register sets. Instead, the operands of certain instructions prefer or require specific registers. For example, the x86 `div` instruction always writes its result to the `eax` and `edx` registers. In the presence of such constraints, even the lo-

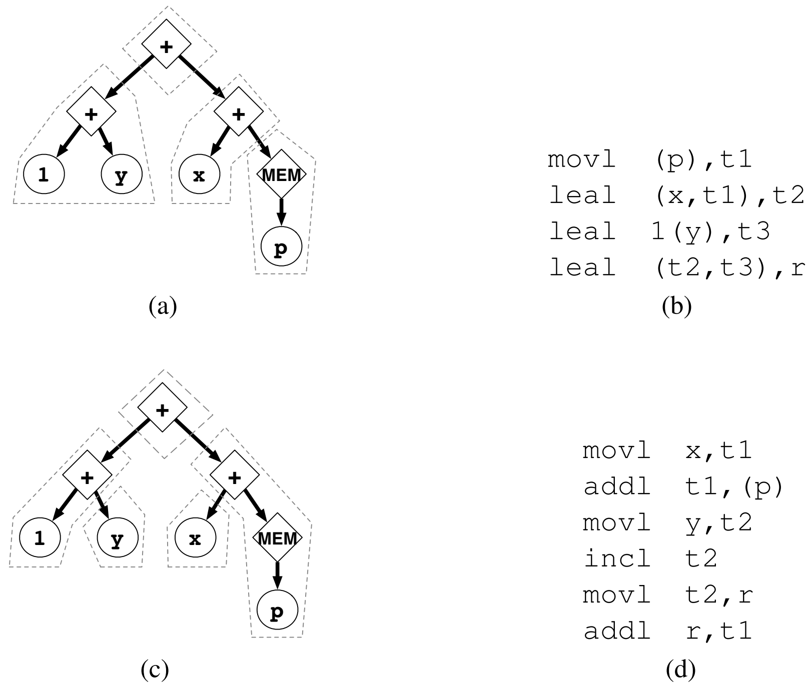


Figure 1.3: An example of instruction selection on a tree-based IR. Two possible tilings, (a) and (c), with their corresponding instruction sequences, (b) and (d), are shown.

cal register sufficiency problem is NP-hard [20, 132], although fixed parameter tractable in the number of registers [13].

The register allocation problem is an NP-hard problem consisting of several important components. In order to generate quality code, a register allocator must not only perform register assignment, but also optimize spill code, perform coalescing and rematerialization, and take register preferences into account.

1.1.2 Instruction Selection

The *instruction selection problem* is to find an efficient conversion from the compiler's target-independent intermediate representation (IR) of a program to a target-specific assembly listing. In the most general sense, instruction selection is undecidable since an optimal instruction selector could solve the halting problem (halting side-effect free code would be replaced by a `nop` and non-halting code by an empty infinite loop). Because of this, instruction selection is usually defined as finding an optimal *tiling* of the intermediate code with a predefined set of machine

instruction tiles. Each tile is a mapping from IR code to assembly code and has an associated cost. An optimal instruction selection minimizes the total cost of the tiling.

An example of instruction selection, where a tree-based IR is converted to x86 assembly, is shown in Figure 1.3. In this example, and in general, there are many possible correct instruction sequences. The difficulty of the instruction selection problem is finding the best sequence for a given cost metric, such as code performance, code size, or some other statically determined cost metric.

For a given tile set and cost metric it is possible to find an optimal tiling in polynomial time if the intermediate representation is in the form of expression trees [1]. However, the tree representation is limited in its expressiveness. A more expressive representation uses directed acyclic graphs (DAGs) to explicitly represent redundancies. As we show in Chapter 7, if a more expressive directed acyclic graph (DAG) intermediate representation is used, then the optimal tiling problem becomes NP-complete. Despite the fundamental hardness of the problem, in Chapter 7 we develop the NOLTIS (near-optimal linear-time instruction selection) algorithm which efficiently computes an optimal or near-optimal tiling of expression DAGs.

1.2 Contribution

The primary contribution of this thesis is the development of principled approaches to the challenging yet critical backend optimization problems of register allocation and instruction selection. We present expressive models that explicitly model the complexities of the target architecture. We then approach an optimal solution to the problem by using progressive or near-optimal solution techniques. After performing an extensive evaluation, we conclude that our principled approach results in better code quality and a better compiler.

Chapter 2

Related Work

Register allocation and instruction selection are critical components of the compiler backend and have been the subject of extensive study. In this chapter we describe the state-of-the-art in register allocation, instruction selection, and instruction selection–register allocation integration.

2.1 Register Allocation

Register allocation is a fundamental part of any compiler backend and has been extensively studied. The textbook [5, 8, 37, 93, 95] approach represents the register allocation problem as a graph coloring problem. Although many improvements to this technique have been proposed, the graph coloring representation is fundamentally limited, especially when compiling for highly constrained and irregular architectures. Alternative heuristic allocators such as linear scan are equally limited. These allocators lack an expressive and complete model of the full register allocation problem.

Less limited methods of register allocation which use expressive models and find optimal allocations have been proposed but are prohibitively slow. The progressive solution techniques of the thesis will bridge the gap between existing slow, but optimal, and fast, but suboptimal, allocators allowing programmers to explicitly trade compilation time for code quality.

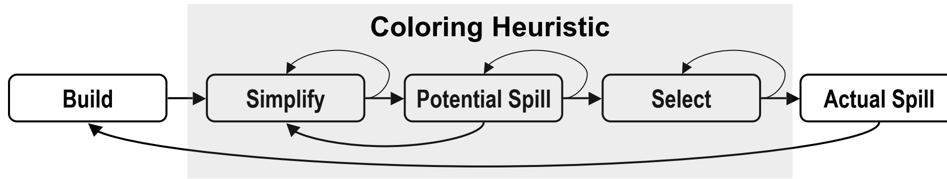


Figure 2.1: The flow of a traditional graph coloring algorithm.

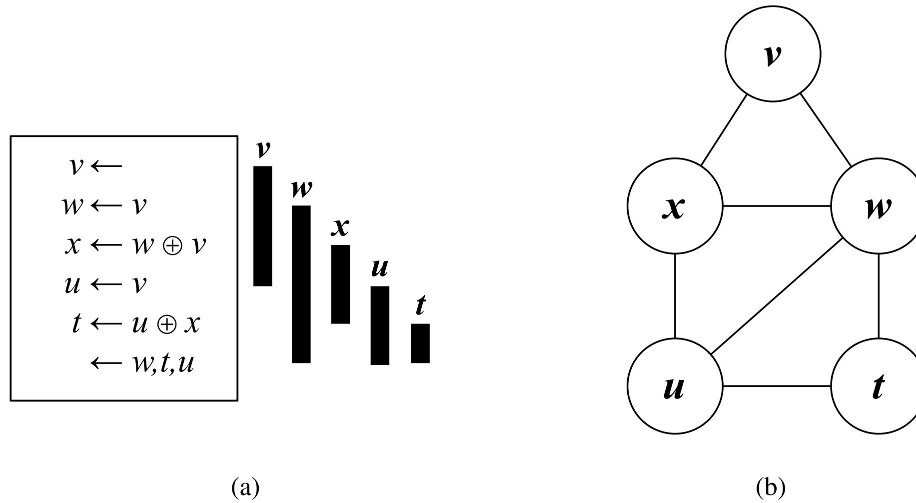


Figure 2.2: (a) Live ranges and (b) the corresponding interference graph.

2.1.1 Graph Coloring Register Allocation

A traditional graph coloring allocator constructs an interference graph which is then labeled with k "colors" representing each of k available registers. In Chapter 5 we use some of the concepts of graph coloring allocation as building blocks for our progressive allocator and so describe graph coloring allocation in detail.

The traditional optimistic graph coloring algorithm [21, 24, 31] consists of five main phases as shown in Figure 2.1:

Build An interference graph is constructed using the live ranges of variables, which are computed using data flow analysis. A node in the graph represents a variable. An edge connects two nodes if the variables represented by the nodes interfere. Two variables interfere if their live ranges overlap and the variables cannot be allocated to the same register. In the example show in Figure 2.2(a), the variables v and w have overlapping live ranges and

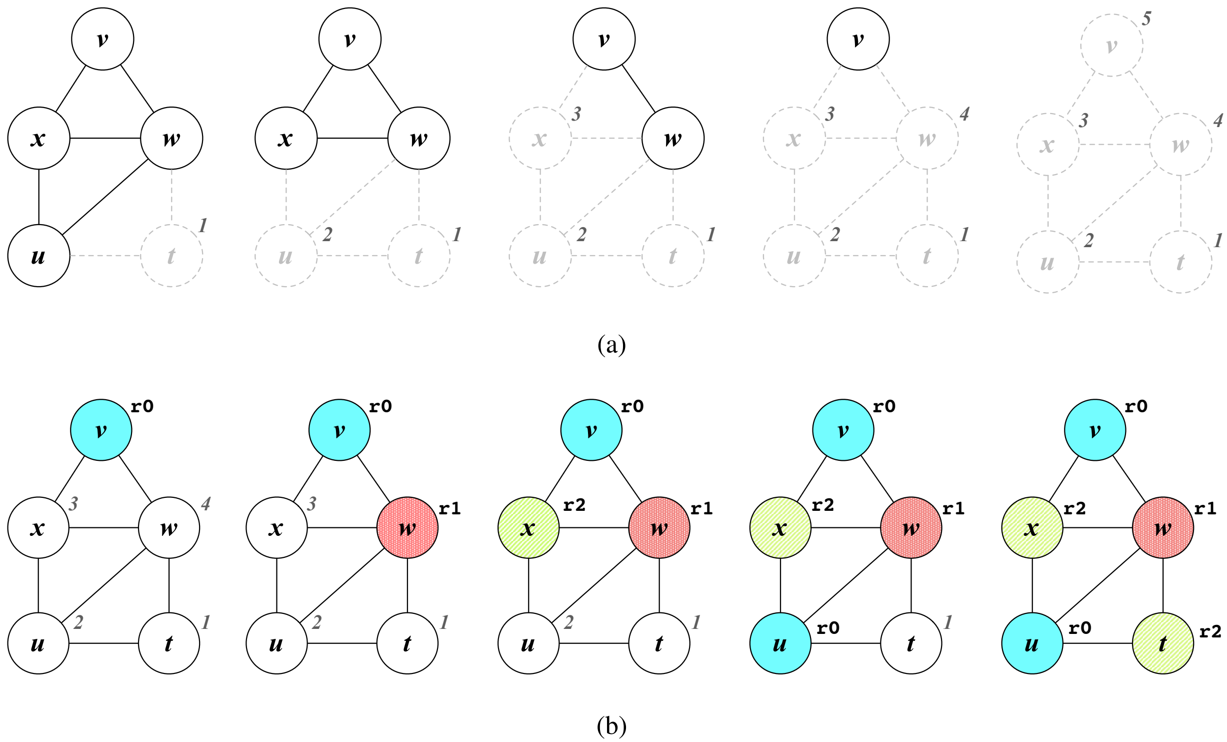


Figure 2.3: An example of the (a) Simplify and (b) Select phases of a graph coloring allocator.

there is an edge between their nodes in the interference graph in Figure 2.2(b). Restrictions on what registers a variable may be allocated to can be implemented by adding precolored nodes to the graph.

Simplify A heuristic is used to reduce the size of the graph. The most commonly used heuristic [69] removes any node with degree less than k, where k is the number of available registers, and places it on a stack. This is repeated until all nodes are removed, in which case the Select phase is executed, or no further simplification is possible. In the interference graph shown in Figure 2.3(a), for $k = 3$ there are initially only two nodes, v and t, with degree less than k. The removal of node t from the graph reduces the degree of nodes u and w and the node u becomes a candidate for simplification. In this example, simplification continues until all nodes are removed from the graph. More complicated heuristics [88, 127] can also be used to further simplify the graph.

Potential Spill If the graph is not fully simplifiable, we mark a remaining node as a potential spill node, remove it from the graph, and optimistically push it onto the stack. We repeat

this process until there exist nodes in the graph with degree less than k, at which point we return to the Simplify phase.

Select In this phase all of the nodes have been removed from the graph. Nodes are popped off the stack and assigned a color (corresponding to a register) in the reverse order they were simplified. This process is shown in Figure 2.3(b). If a node is not a potential spill node, when it was pushed onto the stack it had a degree less than k in the simplified interference graph. Therefore at most $k-1$ of the node's neighbors have already been pushed off the stack and assigned a color, and it will always be possible to find a non-conflicting color for this node. If a node is a potential spill node, then it still may be possible to assign it a color; if it is not possible to color the potential spill node, we mark it as an actual spill and leave it uncolored.

Actual Spill Spill code is generated for every node that is marked as an actual spill. We generate spill code that loads and stores the variable represented by each node into new, short lived, temporary variables everywhere the variable is used and defined. Because new variables are created, it is necessary to rebuild the interference graph and start over.

The Simplify, Potential Spill, and Select phases together form a heuristic for graph coloring. If this heuristic is successful, there will be no actual spills. Otherwise, the graph is modified so that it is easier to color by spilling variables and the entire process is repeated. This coloring heuristic is a "bottom-up" coloring [37]. A "top-down" coloring [33, 34] uses high-level program information instead of interference graph structure to determine a priority coloring order for the variables and then greedily colors the graph.

As an alternative to the iterative approach where the interference graph is rebuilt and reallocated every time variables are spilled, a single-pass allocator can be used. A single-pass allocator reserves registers for spilling. These registers are not allocated in the coloring phase and instead are used to generate spill code for all variables that did not get a register assignment.

A number of improvements to the basic graph coloring algorithm have been proposed. Five common improvements are:

Web Building [31, 67] Instead of a node in the interference graph representing all the live ranges of a variable, a node only represents the connected live ranges of a variable (called webs).

For example, if a variable i is used as a loop iteration variable in several independent loops, then each loop represents an unconnected live range of i, a web. Each web can then be allocated to a different register, even though the webs represent the same variable.

Coalescing [24, 31, 51, 103] If the live ranges of two variables are joined by a move instruction and the variables are allocated to the same register, it may be possible to coalesce (eliminate) the move instruction. Coalescing is implemented by adding move edges to the interference graph. If two nodes are connected by a move edge, they should be assigned the same color. Move edges can be removed to prevent unnecessary spilling. Coalescing techniques differ in how aggressively they coalesce nodes and when and how the decision to coalesce is finalized.

Spill Heuristic [12] A heuristic is used when determining which node to mark in the Potential Spill stage. Spill heuristics try to choose a node with a low spill cost (requiring only a small number of dynamic loads and stores to spill) or a node whose absence will make the interference graph easier to color and therefore reduce the number of future potential spill nodes.

Improved Spilling [11, 24, 36] If a variable is spilled, loads and stores to memory may not be needed at every read and write of the variable. It may be cheaper to rematerialize [22] the value of the variable (if it is a constant, for example). Alternatively, the live range of the variable can be partially spilled. In this case, the variable is only spilled to memory in regions of high interference. Live range splitting can be applied before or during register allocation [36, 78, 98].

Support for Irregular Architectures [23, 24, 66, 76, 122] The graph coloring model implicitly assumes a uniform register model and so must be further extended to target irregular architectures. These techniques heuristically extend the interference graph representation and coloring algorithm to take into account register class preferences and nonuniform usage requirements.

2.1.2 SSA Register Allocators

Recently, several researchers have discovered that if a program is in single static assignment (SSA) form [38] then the register sufficiency problem can be solved in polynomial time [19, 25, 59, 102]. Programs in SSA form will alway generate an interference graph that is perfect and chordal. The special structure of these graphs admits an optimal coloring algorithm that is linear in the number of edges in the graph. However, efficiently and optimally coloring the interference graph of a program in SSA form is not sufficient to obtain a quality register allocation since most interference graphs are not colorable: the chromatic number of the interference graph is larger than the number of registers. In these cases, spill code must be inserted to reduce the number of interfering live variables. A more subtle limitation of the SSA approach is that an optimal register assignment of the SSA form of the program does not directly map to an optimal register assignment when the SSA Φ functions are removed. In fact, optimally converting out of SSA form is an NP-complete problem [105].

Several register allocators have been developed that attempt to exploit the colorability of programs in SSA form [17, 55, 60, 131, 132]. These approaches perform spill code generation as a separate phase. A heuristic is used to reduce the number of live variables at each program point to be less than or equal to the number of available registers. When this requirement is met, the size of the maximal clique of the resulting interference graph is less than or equal to the number of registers. Since in a chordal graph the size of the maximal clique is equal to the chromatic number, it is possible to find a valid register assignment if the program is in SSA form. The problem then becomes finding an assignment that minimizes the number of move and swap instructions in the final instruction sequence after converting out of SSA form. That is, the goal is to assign the input and output operands of the Φ functions to the same register so that the move instructions that implement the Φ instruction are coalesced out of the instruction sequence. Somewhat surprisingly, these SSA-based register allocators focus mostly on effectively solving this NP-complete coalescing problem and have relied on existing simplistic heuristics to perform the spill code optimization pass.

SSA-based register allocators attempt to take advantage of the colorability of interference graphs of structured programs. Despite utilizing an optimal coloring algorithm, these approaches

generally perform no better than graph coloring allocators. This is not surprising for, as described in Section 2.1.6, other components of the register allocation problem, such as spill code optimization and register preferences, have a substantial impact on code quality.

2.1.3 Linear Scan Allocators

Linear scan allocators find a register allocation in a single sweep of the program code. They were initially designed for just-in-time compilers and sacrificed code quality for compile-time performance. Instead of constructing an interference graph, linear scan allocators linearize the control flow graph and assign a numerical linear ordering to program points. The live ranges of a variable can then be expressed as an interval between program points. For example, in Figure 2.4, the live intervals for the variables a, b, c, and d are (1,17), (4,6), (5,13), and (12,15) respectively. A simple live interval may contain one or more *lifetime holes*, a range of program points within the live interval where the variable is not actually live. For instance, in the example, variables a and c have lifetime holes.

The simplest and fastest linear scan allocator [106] does not represent lifetime holes. This allocator iterates over the list of live intervals, sorted by start point, and maintains a list of active intervals that are currently assigned a register. For each interval, the allocator first frees the register of any interval in the active list that has expired. An active interval has expired if its end point is smaller than the start point of the current interval. Then the allocator assigns an available register to the current interval. If no register is available, a simple heuristic is used to choose an interval to spill to memory. The current interval and any interval on the active list are valid candidates for spilling. Intervals that are spilled are assigned a memory location for their entirety. If values cannot be accessed directly in memory, a scratch register must be reserved so that load or store instructions can be generated at every reference of a spilled variable.

When executed on the example shown in Figure 2.4 with a register set of two registers, `r0` and `r1`, the simple linear scan allocator first allocates a to `r0` and then allocates b to `r1`. No registers are available to allocate c, so either a, b, or c must be spilled. If the allocator chooses a, then c will be allocated to `r0`. Finally, the allocator processes d's interval. Since b has expired (its endpoint, 6, is less than 12, the start point of d) the register `r0` is available and is allocated to

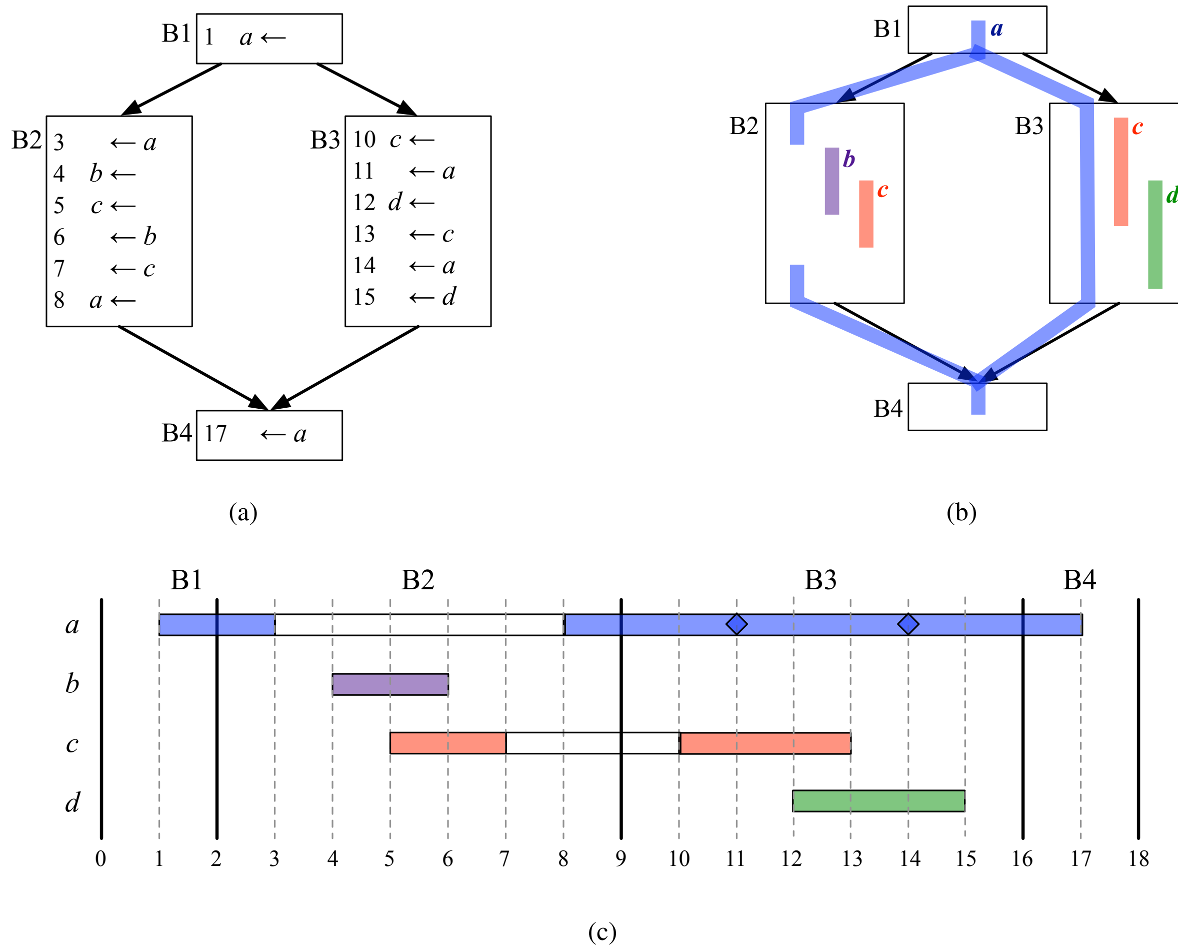


Figure 2.4: Example illustrating the linear ordering of basic blocks, live intervals, and lifetime holes. (a) Source code and (b) live intervals illustrated within the control flow graph. (c) Linear scan allocators linearize the control flow graph so that each variable has a single live interval which may have holes.

d. The final allocation is shown in Figure 2.5(a). The variable a is spilled everywhere resulting in six references to memory.

Simple linear scan allocation is unsophisticated, but fast. The assignment phase is linear in the number of variables. Generating the live intervals and rewriting the code to implement the found assignment are both linear in the number of instructions. More sophisticated linear scan allocators retain the linear asymptotic complexity of simple linear scan, but support lifetime holes, the splitting of intervals and other optimizations [115, 125, 128]. We build on the ideas of these more sophisticated linear scan allocators in Chapter 5 and so describe them in detail.

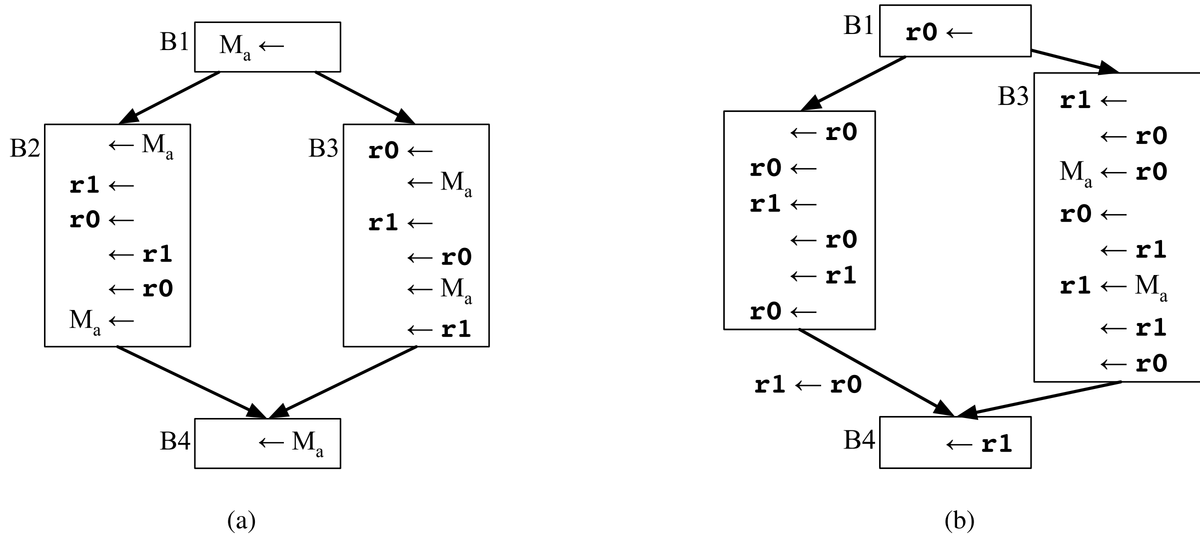


Figure 2.5: Result of (a) simple linear scan and (b) second-chance binpacking linear scan.

A *second-chance binpacking* linear scan allocator [125, 128] keeps track of lifetime holes and has a flexible spilling strategy. A variable may have several live intervals, none of which contain holes. For example, the live intervals for variables a and c in Figure 2.4 are (1,3;8,17) and (5,7;10,13) respectively. Each live interval is allocated independently. As with the simple linear scan allocator, the second-chance binpacking allocator maintains a list of active variables that are currently allocated to registers. Unlike the simple allocator, second-chance binpacking iterates over the instructions of the program and simultaneously allocates and rewrites the code. For each instruction, the allocator considers the variables accessed by the instruction. For each variable, there are three possible cases:

- The variable is **read** by the instruction and currently **active**, i.e., assigned a register. In this case the instruction is simply rewritten to access to correct register.
- The variable is **read** by the instruction and currently **inactive**, i.e., spilled to memory. Unless the instruction can directly access memory, a load instruction is inserted before the current location to load the variable into a register. If no registers are available, a register is made available by evicting a variable from the active list. The variable read by the instruction is now allocated to this register and added to the active list. This is the

variable's "second-chance" to be allocated to a register. The instruction is rewritten to use the appropriate register.

- The variable is **written** by the instruction, and this is the start of a new live interval. If a register is available, it is assigned to the variable and the variable is added to the active list. If no register is available, a register is made available by evicting a variable from the active list. This register is then assigned to the variable written by the instruction and the variable is added to the active list. The instruction is rewritten to use the appropriate register.

When a variable is evicted from the active list, a store instruction saving the variable to memory is inserted at the eviction point. If the value to be stored is known to already reside in memory (due to an earlier store) then this store can be omitted.

Unlike the simple linear scan allocator, in second-chance binpacking a variable is not assigned to a single register or memory location. This additional flexibility can result in conflicts in allocation decisions at basic block boundaries. For instance, consider the case where the two live intervals of a in Figure 2.4 are allocated to different registers, `r0` and `r1`. Although this allocation is legal in the linearized traversal of the code used by the allocator and illustrated in Figure 2.4(c), in actuality the control flow edge between blocks $B1$ and $B3$ necessitates that the allocation of a at program point 2 equals that at program point 9. To resolve these conflicts, a *resolution phase* is run after allocation that traverses the control flow edges of the program and inserts move, load, and store instructions as necessary.

We now trace the execution of the second-chance binpacking allocator on the example shown in Figure 2.4 with a register set of two registers, `r0` and `r1`.

1 a is allocated to `r0`, a is added to the active list, the instruction is rewritten to use `r0`.

ACTIVE: a[1,3]:`r0`

3 The instruction is rewritten to use `r0` for a, and a is removed from the active list.

ACTIVE:

4 b is allocated to `r0`, b is added to the active list, and the instruction is rewritten to use `r0`.

ACTIVE: b[4,6]:`r0`

5 c is be allocated to `r1`, c is added to the active list, and the instruction is rewritten to use `r1`.

ACTIVE: b[4,6]:`r0`, c[5,7]:`r1`

6 The instruction is rewritten to use `r0` for b, and b is removed from the active list.

ACTIVE: c[5,7]:`r1`

7 The instruction is rewritten to use `r1` for c, and c is removed from the active list.

ACTIVE:

8 a is allocated to `r0`, a is added to the active list, and the instruction is rewritten to use `r0`.

ACTIVE: a[8,17]:`r0`

10 c must be allocated to `r1`, c is added to the active list, and the instruction is rewritten to use `r1`. Note that the current allocation of a to `r0` is due to its most recent allocation at 8, not its allocation at the exit of block $B1$ (position 2).

ACTIVE: a[8,17]:`r0`, c[10,13]:`r1`

11 The instruction is rewritten to use `r0` for a.

ACTIVE: a[8,17]:`r0`, c[10,13]:`r1`

12 No registers are available so a variable from the active list must be evicted. a is chosen for eviction, a store from `r0` to memory is inserted, and a is removed from the active list. d is allocated to `r0`, d is added to the active list, and the instruction is rewritten to use `r0`.

ACTIVE: d[12,15]:`r0`, c[10,13]:`r1`

13 The instruction is rewritten to use `r1` for c, and c is removed from the active list.

ACTIVE: d[12,15]:`r0`

14 a is required to be in a register. The register `r1` is available so a load of a from memory to `r1` is inserted. a is added to the active list and the instruction is rewritten to use `r1`

ACTIVE: d[12,15]:`r0`, a[8,17]:`r1`

15 The instruction is rewritten to use `r0` for d, and d is removed from the active list.

ACTIVE: a[8,17]:`r1`

17 The instruction is rewritten to use `r1` for a, and a is removed from the active list.

ACTIVE:

This completes the allocation phase. The resolution phase scans through the control flow edges of the control flow graph looking for conflicts. In this example, there is a conflict for variable a on the edge between block $B2$ and block $B4$. The variable a is allocated to `r0` at the exit of $B2$ and to `r1` at the entry of block $B4$. The conflict is resolved by inserting a move instruction along this edge. The resulting allocation is shown in Figure 2.5(b). Compared to the simple linear scan allocator, second-chance binpacking generates four fewer memory accesses and one additional move instruction. In general, second-chance binpacking produces code with fewer spill instructions and has the same linear asymptotic complexity as simple linear scan.

The extended linear scan algorithm [115] first performs spill code minimization as a separate phase, and then uses second-chance binpacking techniques to find a spill-free register assignment. The spill code minimization phase identifies program points where the register need exceeds the register availability and heuristically chooses live intervals to spill until register need is less than or equal to register availability at every program point. Once this requirement is met, second-chance binpacking can always find a register assignment without further spilling as long as move and swap instructions can be inserted at basic block boundaries. This is the approach used in the LLVM 2.4 [87] compiler framework.

2.1.4 Alternative Heuristic Allocators

Several other approaches to register allocation have been studied and implemented in production compilers. Several allocators, including the one used by the GNU `gcc` compiler version 4.4 [47], separate the register allocation problem into global allocation and local allocation problems, each of which is done separately. Other allocators attempt to exploit program structure.

Although allocators that perform local and global register allocation separately may perform global allocation first [93], typically local allocation is performed first in order to take advantage of effective linear-time local register allocation algorithms [42, 63, 86]. In probabilistic register allocation [111] and demand-driven allocation [112], the results of local allocation are used by the global allocator to determine which variables get registers. In the `gcc` allocator, the local

allocator performs a simple priority-based allocation. The global allocator then performs its own single-pass priority-based allocation. A final reload phase generates and optimizes the necessary spills for any variables that remain unallocated. When compilation time is at a premium, the global pass, which must calculate a full interference graph, can be skipped.

Allocators that exploit program structure break the control flow graph into regions or tiles. In hierarchical register allocation [28, 35] a tile tree corresponding to the control-flow hierarchy is constructed. A partial allocation is computed in a bottom-up pass of the tile tree, and then the final register assignment is calculated with a second top-down pass. A similar technique can also be used with regions derived from program dependence graphs [100]. Hierarchical allocation results in a more control-flow aware allocation (for example, less spill code in loops), but decisions made when fixing the allocation of a tile may have globally poor results. A graph fusion allocator [89] avoids fixing an allocation at tile boundaries. Instead, tiles are “fused” together until the entire control flow graph is covered by one fused tile. Each fusion operation maintains the invariant that the interference graph of a fused tile is simplifiable (easily colored) by splitting live ranges and spilling variables as necessary. Register assignment is then performed on the final interference graph. Hierarchical allocators typical exhibit mixed results, with an average case improvement over graph-coloring allocators. When these allocators perform poorly, it is usually because the built-in heuristics fail and excessive spill and shuffle code is generated at tile boundaries.

2.1.5 Optimal Register Allocation

The NP-hard nature of register allocation makes it unlikely that a practical optimal register allocation algorithm exists. However, several optimal or partially optimal approaches have been investigated. Although these algorithms do not demonstrate practical running times, they provide insight into what is achievable and, in some cases, suggest improvements to heuristic solutions.

The local register allocation problem has been solved optimally using a dynamic programming algorithm that requires exponential space and time [63]. This algorithm has been extended to handle loops and irregular architectures [74] and multi-issue machines [92]. Essentially, this algorithm performs a pruned exhaustive search of all possible register allocations. The exponential part of the algorithm can be replaced by a heuristic to get an efficient local allocator

that outperforms other local allocators on average and is generally close to optimal. Local spill code optimization for uniform register sets can also be solved using integer linear programming techniques [86].

The global register sufficiency problem has been solved optimally [14, 101] or approximately [124] by exploiting the bounded treewidth property of structured programs. The asymptotic running time of the optimal approaches includes a constant factor that is exponential in the number of registers. While the ability of these algorithms to exploit program structure is insightful, they do not actually solve the complete register allocation problem.

The complete register allocation problem for both regular [48, 49, 54] and irregular [52, 75, 96, 97] architectures has been solved by expressing the problem as an integer linear program (ILP) which is then solved using powerful commercial solvers. Although these techniques demonstrate the significant reduction in spill code possible using optimal allocators, their compile-time performance does not scale well as the size of the input grows. In particular, the ILP solver is unable to find even a feasible solution for most functions with more than 1000 instructions [49].

As an alternative to ILP formulations, a simplified version of the register allocation problem has been modeled as a partitioned boolean quadratic optimization problem (PBQP) [62, 116]. This formulation can then either be solved optimally, but exponentially slowly, or with an efficient polynomial-time heuristic which is competitive with graph coloring allocators.

2.1.6 Limitations

Existing register allocators have several fundamental limitations. Several approaches, such as graph coloring and SSA-based allocators, focus primarily on solving the register assignment problem. Simply solving the register assignment problem is not enough to obtain quality code. As shown in Figure 2.6, architectures with limited registers sets, such as the Intel x86 architecture, frequently do not have sufficient registers to avoid spilling. Since almost half of all the functions in Figure 2.6 had to generate spill code, it is clearly important that the compiler explicitly optimize spill code. Effectively optimizing spill code is especially important when optimizing for performance. As shown in Figure 2.7, the use of a heuristic spill code optimizer

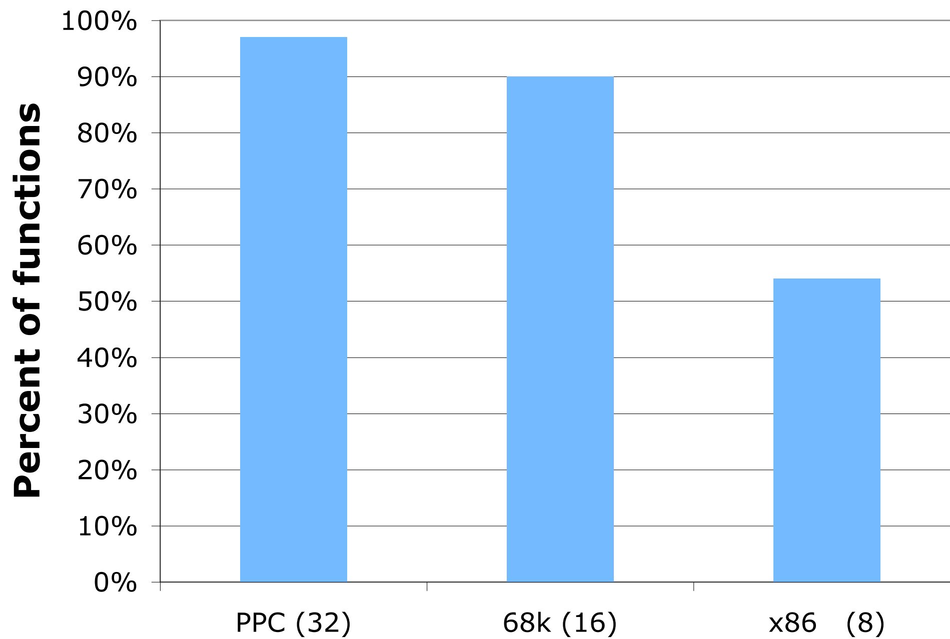


Figure 2.6: The percent of over 10,000 functions assembled from various benchmark suites for which no spilling is performed by a conventional graph coloring allocator. All functions are treated equally; no attempt is made to weight functions by execution frequency or size. Although these results are for a heuristic allocator, the heuristic used fails to find a spill-free allocation when one actually exists in only a handful of cases [72].

with optimal register assignment greatly decreases code quality relative to an optimal allocator, especially when optimizing for performance. In contrast, a heuristic register assignment algorithm does not result in as large decreases of code quality when paired with an optimal spill code optimizer.

The importance of other components of register allocation are further demonstrated in Figure 2.8, which shows the effect of replacing the heuristic coloring algorithm in a traditional graph coloring allocator with an optimal coloring algorithm as described in [72]. The use of an optimal coloring algorithm substantially degrades code quality unless additional components of register allocation are incorporated into the objective function of the optimal allocator. Since this is done in an *ad hoc* manner, i.e., no explicit cost model is used, the results are mixed with the optimal-coloring based allocator performing more poorly on average than a purely heuristic based allocator. These results strongly suggest that developing a register allocator around the register sufficiency problem, as with the graph coloring and SSA-based allocators, and then

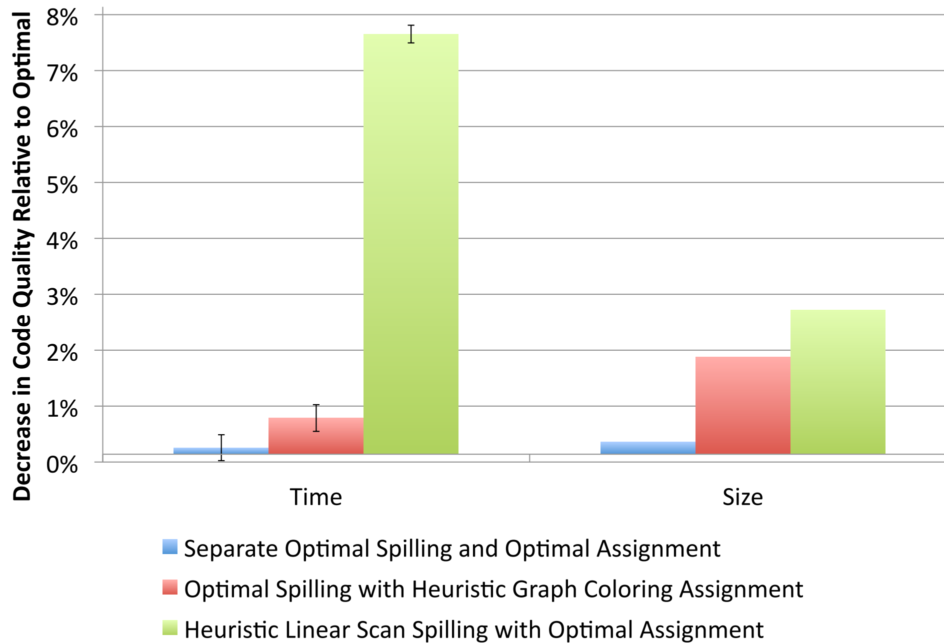


Figure 2.7: Increases in execution time and code size relative to an optimal allocator when spill code optimization and assignment are performed separately and heuristically. The optimal allocator and benchmark suite used are described in [73] and are evaluated on the Intel x86-32 architecture. Using heuristics to perform register assignment results in an allocation that is closer to optimal than when heuristics are used to perform spill code optimization.

heuristically extending it to incorporate the additional components of register allocation is not the the best approach when targeting constrained and irregular architectures.

2.1.7 Summary

Register allocation is a critical component of the compiler backend. Although it has been extensively studied, there remains substantial opportunity for improvement. Existing allocators do not effectively represent all the pertinent features of register allocation. In particular, most allocators focus on the register satisfiability problem. This thesis presents a principled approach to register allocation that uses a comprehensive and expressive model coupled with progressive solution techniques to bridge the gap between fast, suboptimal, heuristic register allocation algorithms and slow, but optimal, algorithms.

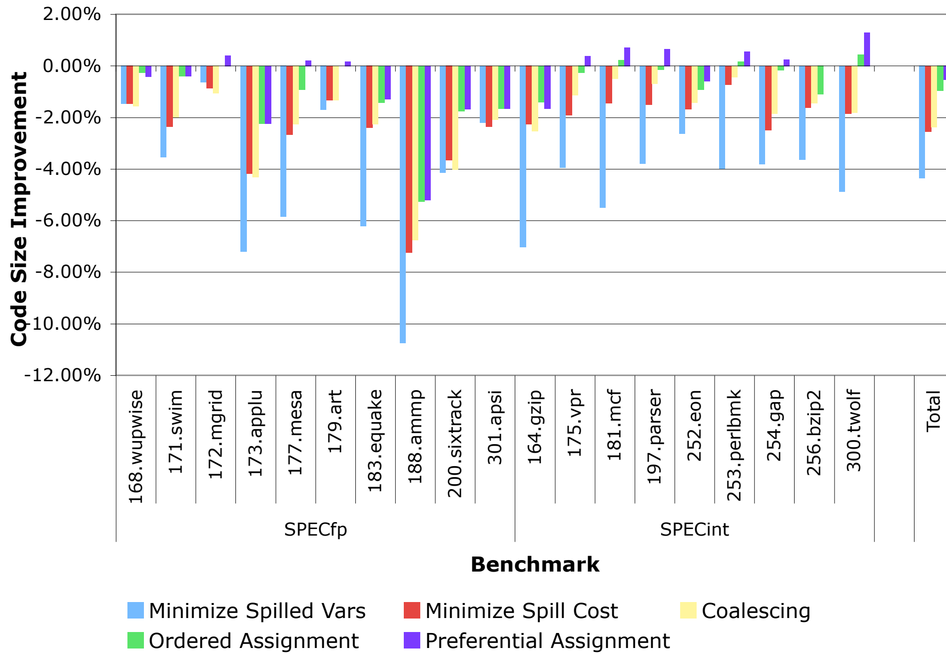


Figure 2.8: The effect of incorporating various components of register allocation into the coloring algorithm. The coloring heuristic of a traditional graph allocator is replaced with an optimal coloring algorithm. Results are shown for an algorithm that optimally minimizes the number of spilled variables, that minimizes the total heuristic cost of spilled variables, and that minimizes total spill cost while preferring allocations that are biased towards coalescing and register preferences.

2.2 Instruction Selection

Instruction selection, or code generation, converts the compiler's intermediate representation (IR) into a target-specific assembly listing. Like register allocation, instruction selection has been extensively studied. The textbook [3, 5, 8, 37, 95] approach is to represent instruction selection as a tree tiling problem. Effective and efficient algorithms for tree tiling exist, but this representation is fundamentally limited in expressiveness. An alternative approach tiles a directed acyclic graph (DAG) representation. DAGs are more expressive than trees since they explicitly encode redundant operations, but the problem of tiling a DAG optimally is provably NP-complete.

Instruction selection is commonly performed by tiling expression trees. This was initially done using dynamic programming [1, 118] for a variety of machine models including stack machines, multi-register machines, infinite register machines, and superscalar machines [16].

These techniques have been further developed to yield code-generator generators [30, 53] which take a declarative specification of an architecture and, at compiler-compile time, generate an instruction selector. These code-generator generators either perform the dynamic programming at compile time [4, 40, 45] or use BURS (bottom-up rewrite system) tree parsing theory [104, 109] to move the dynamic programming to compiler-compile time [46, 110].

Directed acyclic graphs are an alternative representation to sequences of trees. Expression DAGs are more expressive than expression trees as they explicitly model redundant expressions. Tiling expression DAGs is significantly more difficult than tiling expression trees. DAG tiling has been shown to be NP-complete for one-register machines [26] and for two-address, infinite register machine models [2]. DAG tiling remains difficult on a three-address, infinite register machine if the exterior tile nodes have labels that must match [108]. These labels may correspond to value storage locations (e.g. register classes or memory) or to value types. Such labels are unnecessary if instruction selection is separated from register allocation and if the IR has already fully determined the value types of edges in the expression DAG. Chapter 7 includes a proof that the problem remains NP-complete even without labels.

Although DAG tiling is NP-complete in general, for some tile sets it can be solved in polynomial time [41]. If a tree tiling algorithm is adapted to tile a DAG and a *DAG optimal* tile set is used to perform the tiling, the result is an optimal tiling of the DAG. Although the tile sets for several architectures are DAG optimal [41], these tile sets use a simple cost model and the DAG optimality of the tile set is not preserved if a more complex cost model, such as code size, is used. A simplified version of the DAG tiling problem can be solved within a constant approximation ratio using heuristics [2].

Traditionally, DAG tiling is performed by using a heuristic to break up the DAG into a forest of expression trees [3]. More heavyweight solutions, which solve the problem optimally, use binate covering [84, 85], constraint logic programming [83], integer linear programming [97] or exhaustive search [70]. In addition, a 0-1 integer programming representation of the problem is described in Chapter 7. These techniques all exhibit worst-case exponential behavior.

An alternative, non-tiling, method of instruction selection, which is better suited for linear, as opposed to structural, IRs, is to incorporate instruction selection into peephole optimization [37,

39, 43, 44, 71]. In peephole optimization [91], pattern matching transformations are performed over a small window of instructions, the “peephole.” This window may be either a physical window, where the instructions considered are only those scheduled next to each other in the current instruction list, or a logical window where the instructions considered are just those that are data or control related to the instruction currently being scanned. When performing peephole-based instruction selection, the peepholer simply converts a window of IR operations into target-specific instructions. If a logical window is being used, then this technique can be considered a heuristic method for tiling a DAG.

The tiling representation of instruction selection requires that the DAG or tree nodes that make up an instruction tile be connected. That is, there must be a data dependency between the nodes. Some instructions, such as SIMD instructions, perform inherently parallel and independent operations and so cannot be accurately represented by a traditional instruction tile. Although some effort has been made to incorporate such parallel instructions into a tiling based framework [81, 82], these approaches have worst-case exponential behavior and do not scale well in practice. In general, vectorization and parallelization are considered separate problems from instruction selection and are beyond the scope of this thesis.

When represented as a tiling problem over expression trees, instruction selection is a solved problem. In contrast, scalable algorithms for tiling expression DAGs have not been developed. This thesis describes the Near-Optimal Linear-Time Instruction Selection (NOLTIS) algorithm that finds empirically near-optimal instruction tilings of expression DAGs in worst-case linear-time.

Chapter 3

Global MCNF Register Allocation Model

Existing register allocators do not effectively represent or optimize for all the pertinent features of register allocation. A principled approach to register allocation requires an expressive and complete model of the problem and effective progressive solution techniques. In this chapter we describe an expressive and complete model of register allocation based on multi-commodity network flow (MCNF).

We begin by describing the classical MCNF problem. We use MCNF to create an expressive model of register allocation for straight-line code that explicitly and exactly represents the pertinent components of the problem. We then extend this MCNF model to handle control flow and describe how we model the persistence of values in memory. We describe the implementation of two code quality metrics, code size and code performance, within the global MCNF model. Finally, we discuss limitations and potential simplifications of the model.

3.1 Multi-commodity Network Flow

The multi-commodity network flow (MCNF) problem is to find a minimum cost flow of commodities through a constrained network. The classical use of MCNF is to model transportation problems where the commodities are physical goods that need to be transported from warehouses to customers as cheaply as possible. For example, Figure 3.1(a) is a model of a transportation problem where two warehouses, nodes 1 and 2, each stock a unit of a commodity. This is repre-

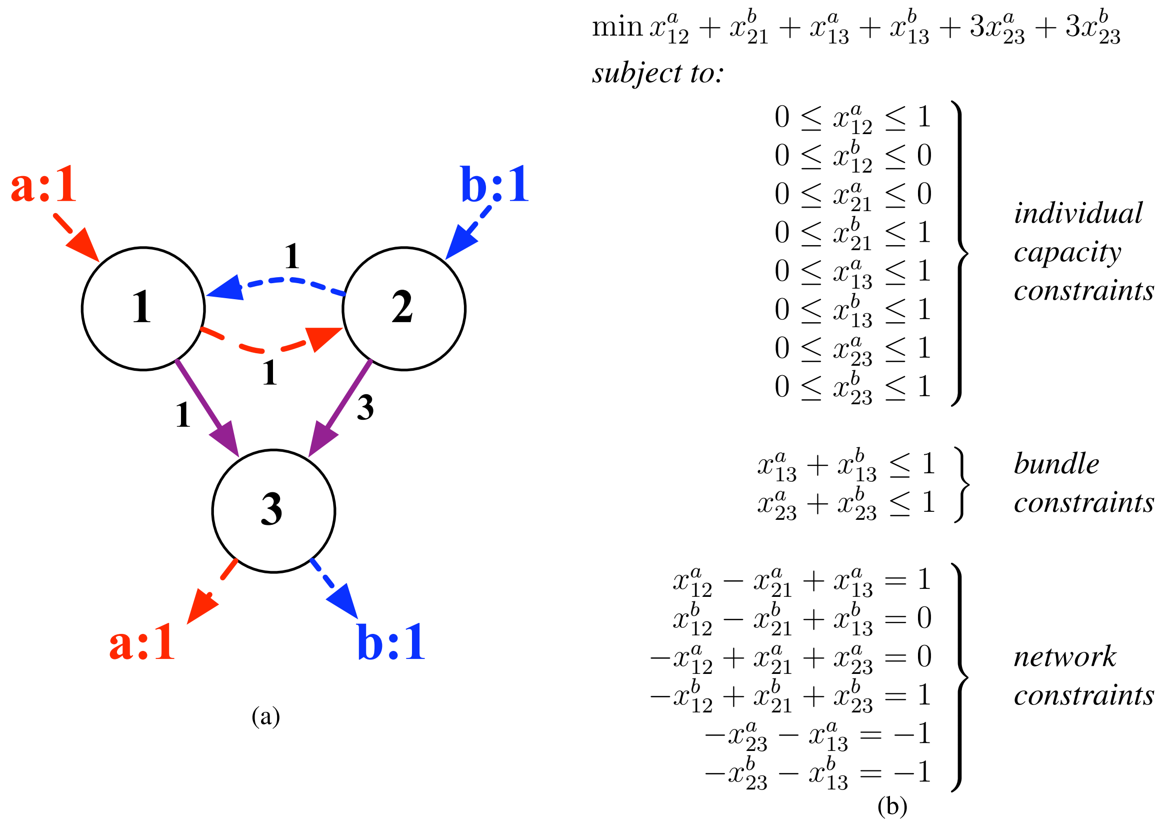


Figure 3.1: A simple example of a multi-commodity network flow problem. (a) A graphical representation of the problem. The two commodities, a and b, have nodes 1 and 2 respectively as source nodes and share node 3 as a sink node. All edges have a total capacity of one, with edges $(1, 2)$ and $(2, 1)$ restricted to commodities a and b respectively. (b) The complete problem in equation form.

sented in the figure by an incoming edge labeled with the commodity and its amount (one unit of a for node 1 and one unit of b for node 2). These warehouse nodes are *source nodes* because they create a supply of a commodity. In this example there is a single customer represented by node 3 that has a demand for both one unit of a and one unit of b. Nodes that create a demand for a quantity are called *sink nodes*.

The edges connecting nodes in an MCNF problem have costs and capacities. In a transportation problem, the cost might represent a monetary cost associated with moving a commodity. Edge costs can be specific to each commodity or, as is the case in Figure 3.1(a), all commodities can incur the same cost along an edge. In the example, both a and b incur a cost of 1 along the

edge $(1, 3)$. In a transportation problem the capacity of an edge represents a physical constraint, such as how many commodities can be loaded onto a truck. *Individual capacity constraints* are edge capacities that are specific to each commodity. In the example, as indicated the edge style, the individual capacity of edge $(1, 2)$ for commodity a is 1, but for b is zero. In a transportation problem this might correspond to a commodity requiring a special kind of truck that isn't available between two destinations. An edge can also have a *bundle constraint* that constrains the total capacity of the edge. In the example edges $(1, 3)$ and $(2, 3)$ both have a bundle constraint of one. In a transportation problem the bundle constraints correspond to the payload capacity of a truck. In the example a truck can transport only a single commodity.

In addition to the individual capacity constraints and bundle constraints, a network flow model also has *network constraints*. These constraints enforce the natural requirement that the amount of flow into a node equals the amount of flow out of the node. Source nodes are initialized with a positive amount of incoming flow while sink nodes have a positive amount of outgoing flow. The amount of flow available at the source nodes of a commodity must match the amount of flow available at the sink nodes. The MCNF problem is to minimize the total cost of all the commodity flows through the network while respecting all the constraints. Finding the minimal cost flow of a single commodity is solvable in polynomial time even if all flows must be integer. In Figure 3.1(a), the minimal flow for commodity a is $1 \rightarrow 3$ and for b is $2 \rightarrow 1 \rightarrow 3$. However, taken together these two flows violate the bundle constraint of edge $(1, 3)$. The optimal solution to the MCNF problem requires that commodity b flow over the more expensive edge $(2, 3)$. As shown by the example, the bundle constraints increase the complexity of the MCNF problem. In fact, finding an optimal integer solution to the MCNF problem is NP-complete [6].

We now define the MCNF problem formally as an optimization function over the commoditized edges of a network. Let x_{ij}^q represent the flow of a commodity q over an edge (i, j), then

the MCNF problem is:

$$\min \sum_{i,j,q} c_{ij}^q x_{ij}^q \quad \textit{cost function} \tag{3.1}$$

subject to the constraints:

$$0 \le x_{ij}^q \le v_{ij}^q \quad \textit{individual capacity constraints} \tag{3.2}$$

$$\sum_q x_{ij}^q \le u_{ij} \quad \textit{bundle constraints} \tag{3.3}$$

$$\mathcal{N} x^q = b^q \quad \textit{network constraints} \tag{3.4}$$

where c_{ij}^q is the cost incurred by commodity q along edge (i, j), v_{ij}^q is the maximum amount of flow of a single commodity q allowed along edge (i, j), and u_{ij} is maximum amount of flow of all commodities allowed along edge (i, j). For example, in Figure 3.1(a), edge $(1, 2)$ has an individual capacity constraint of zero for commodity b resulting in a value of zero for v_{12}^b and the constraint

$$0 \le x_{12}^b \le 0$$

and the edge $(1, 3)$ has a bundle constraint of one resulting in a value of one for u_{13} and the constraint

$$x_{13}^a + x_{13}^b \le 1$$

The matrix $\mathcal{N}$ represents the network constraints and is a matrix representation of the network topology. The $\mathcal{N}$ matrix for Figure 3.1(a) is

$$\mathcal{N} = \begin{bmatrix} 1 & -1 & 0 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 \end{bmatrix}$$

Each row represents a node and each column an edge. In this example, the columns correspond to the edges $(x_{12}, x_{21}, x_{23}, x_{13})$ and the rows to the nodes $(1, 2, 3)$. A value in the matrix is positive if the corresponding edge is leaving the corresponding node and negative if the edge is entering the node. In the example, edge x_{12} flows out of node 1 so the value in the first column and first row of $\mathcal{N}$ is positive one. The vector b^q contains the source and sink information. Each value corresponds to a node and is positive if the node is a source for commodity q, negative if the node

is a sink for commodity q, and zero otherwise. The vectors b^a and b^b for Figure 3.1(a) are

$$b^a = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \qquad b^b = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

The rows correspond to the nodes $(1,2,3)$. Node 1 is a source of one unit of commodity a so the first value of b^a is a positive one. The network constraints are derived by multiplying the matrix $\mathcal{N}$ by the flow vector x^q of a commodity and setting the result equal to b^q. For example, we construct the network constraints for commodity a in Figure 3.1(a) as follows:

$$\mathcal{N}x^a = b^a \Rightarrow$$

$$\begin{bmatrix} 1 & -1 & 0 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} x^a_{12} \\ x^a_{21} \\ x^a_{23} \\ x^a_{13} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \Rightarrow$$

$$x^a_{12} - x^a_{21} + x^a_{13} = 1$$
$$x^b_{12} - x^b_{21} + x^b_{13} = 0$$
$$-x^a_{12} + x^a_{21} + x^a_{23} = 0$$

A complete mathematical representation of the MCNF problem of Figure 3.1(a) is shown in Figure 3.1(b).

3.2 Local Register Allocation Model

We formulate an expressive model of the local register allocation problem using multi-commodity network flow. Local register allocation considers only the problem of allocating the straight-line code of a single basic block. Consider the simple example of local register allocation shown in Figure 3.2. As this register allocation problem contains several features that an expressive model must be able to accurately represent, we use it as a running example in our description of our formulation of an MCNF model of local register allocation. Using a simplification of the ISA restrictions of the x86 architecture and optimizing for code size we derive the following constraints and costs for each of the unallocated variables:

```
int example(int a, int b)
{
    int d = 1;
    int c = a - b;
    return c+d;
}
```

(a) Source code

```
MOVE 1   →  d   5 bytes
SUB a,b  →  c   3 bytes
ADD c,d  →  c   3 bytes
MOVE c   →  r0  2 bytes
```

Total Size: 13 bytes

(b) Unallocated assembly

Figure 3.2: A simple example of local register allocation.

a The parameter a is passed into the function on the stack. The register allocator may either incur the cost of a load (3 bytes) or leave the variable in memory and pay the cost of directly accessing memory in the SUB instruction (1 byte). The SUB instruction supports accessing at most one unique memory location. If a is left in memory, the values of b and c must be available in a register. The SUB instruction is the last instruction to use a. a may be allocated to any integer register.

b The parameter b is passed into the function on the stack. The register allocator may either incur the cost of a load (3 bytes) or leave the variable in memory and pay the cost of directly accessing memory in the SUB instruction (1 byte). The SUB instruction supports accessing at most one unique memory location. If b is left in memory, the values of a and c must be available in a register. The SUB instruction is the last instruction to use b. b may be allocated to any integer register.

c The variable c is defined by the SUB instruction. It may be defined into any integer register or directly into memory at a cost of 1 byte.[1] Since the SUB instruction supports accessing at most one operand in memory, if c is defined into memory, both a and b must be available in registers. The ADD instruction redefines the value of c. The ADD instruction can directly access at most one memory location for a cost of 1 byte. The final MOVE instruction is the last instruction to use c. Since this is a move into a hardware register, `r0`, if c is allocated to `r0` the MOVE instruction can be eliminated for a savings of 2 bytes.

[1]For purposes of exposition we are allowing the SUB instruction to have a three-operand form where each operand can be separately allocated. In the full x86 ISA a two-operand form is required. The ramifications of such a requirement are discussed in Section 3.6.

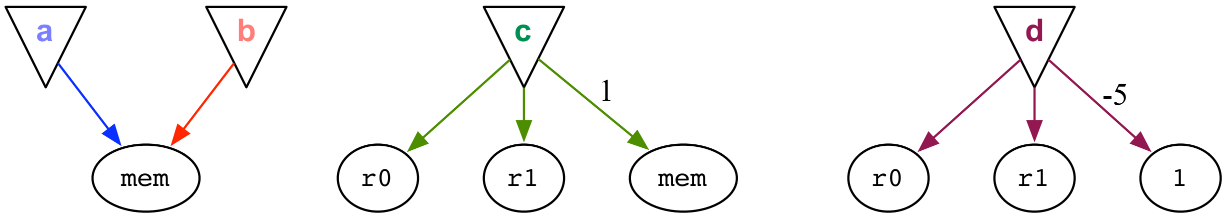


Figure 3.3: Source nodes of a MCNF model of register allocation.

Alternatively, the MOVE instruction may directly access c in memory (becoming a load instruction) for a cost of 1 byte.

d The initial MOVE instruction defines d to have the constant value 1. The MOVE is not permitted to define a constant value directly into memory. Instead of defining d into an integer register at this point, the allocator has the option of eliminating the MOVE instruction for a savings of 5 bytes and rematerializing the constant value later. The ADD instruction uses d. The ADD instruction can directly access at most one operand in memory for a cost of 1 byte. Alternatively, instead of accessing the value of d in a register, the constant value 1 of d can be directly used and a 2 byte smaller INC instruction can be used instead. The ADD instruction is the last instruction to use d.

Our MCNF model of local register allocation can exactly represent all of these constraints and costs. The network is constructed such that each commodity represents a variable and the flow of the commodity exactly maps to a detailed allocation of that variable. The nodes of the network consist of *source nodes*, *sink nodes*, and *allocation class nodes*. Allocation class nodes represent a specific choice of allocation at a specific location in the program and are grouped into *instruction groups* and *crossbar groups* based on the location they represent. We now build up a local MCNF model for the straight-line code of Figure 3.2.

3.2.1 Source Nodes

Source nodes in the MCNF model correspond directly to source nodes in the classical MCNF problem. There is at least one source node for every variable and every source node is associated with exactly one variable. A source node contributes exactly one unit of flow to the network.

There is a source node for every defining location of a variable that is not a redefinition of the variable. For example, in the instruction sequence of Figure 3.2, the SUB instruction defines the variable c, and the ADD instruction redefines the variable c. Since there is only one defining location that is not a redefinition, there is only a single source node for variable c corresponding to the defining SUB instruction.

The source node of a variable connects to allocation class nodes that correspond to the location of the definition of the variable. Only allocation class nodes representing legal allocation choices for the definition of the variable are connected. If defining a variable into a specific allocation class incurs a cost, the edge connecting the source node to the corresponding allocation class node incurs the same cost.

The source nodes of our running example are shown in Figure 3.3. The parameters a and b are connected to the same memory node because they are defined at the same location (the entry of the function) and are constrained to be initially allocated to memory (on the stack frame). The source node of variable c is connected to register nodes and to a memory node since c is defined by the SUB instruction. The edge to the memory node has a cost that exactly matches the increase in code size of accessing a memory operand within the SUB instruction. The source node of variable d is connected to register nodes and to a constant allocation class node. The defining instruction of d does not support defining d directly into memory, and so there is no connection to a memory node. However, if d is defined into a constant allocation class for later rematerialization, the defining instruction can be eliminated resulting in a code size savings of 5 bytes as indicated by the -5 cost on the edge to the constant allocation class node. Note that c and d are defined at different locations and so connect to distinct sets of allocation class nodes.

3.2.2 Sink Nodes

Sink nodes in the MCNF model correspond directly to sink nodes in the classical MCNF problem. There is at least one sink node for every variable and every sink node is associated with exactly one variable. A sink node creates a demand of exactly one unit of flow for a variable. There is a sink node for every location where a variable ceases to be live, i.e., the location of the

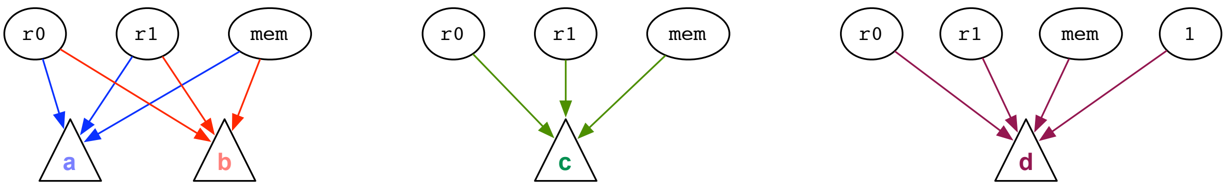


Figure 3.4: Sink nodes of a MCNF model of register allocation.

last use of the variable. For example, for the problem of Figure 3.2, both a and b are last used by the SUB instruction.

The sink node of a variable is connected to allocation class nodes that correspond to the location of the last use of the variable. Only allocation class nodes representing legal allocation choices for the variable at this point are connected. The sink nodes of our running example are shown in Figure 3.4. Note that a and b are connected to the same set of allocation class nodes. This is because they both cease to be live at the same location: the SUB instruction. All four variables are last used by an instruction that can access them either in registers or memory and so are connected to the corresponding allocation class nodes. Additionally, the last use of d is the ADD instruction which can directly use the constant value of d and so the sink of d is connected to an additional constant allocation class node.

3.2.3 Allocation Class Nodes

Allocation class nodes represent a specific *allocation class*, a storage location that a variable can be allocated to. An allocation class can be a register, a class of constants, or a memory space. Although a register allocation class typically represents exactly one register, constant and memory allocation classes may correspond to a class of constants or memory locations. Constants or memory locations are grouped into a class if they are all accessed similarly. For example, there may be separate constant allocation classes for 8-bit integer constants, floating point constants, symbolic constants, constant address locations, and read-only hardware registers (such as the stack pointer). The number and kind of constant classes depends upon the instruction set architecture. For example, if all integer constants are accessed identically in an ISA, then there is no need for separate 8-bit and 32-bit constant classes. In our running example, we define

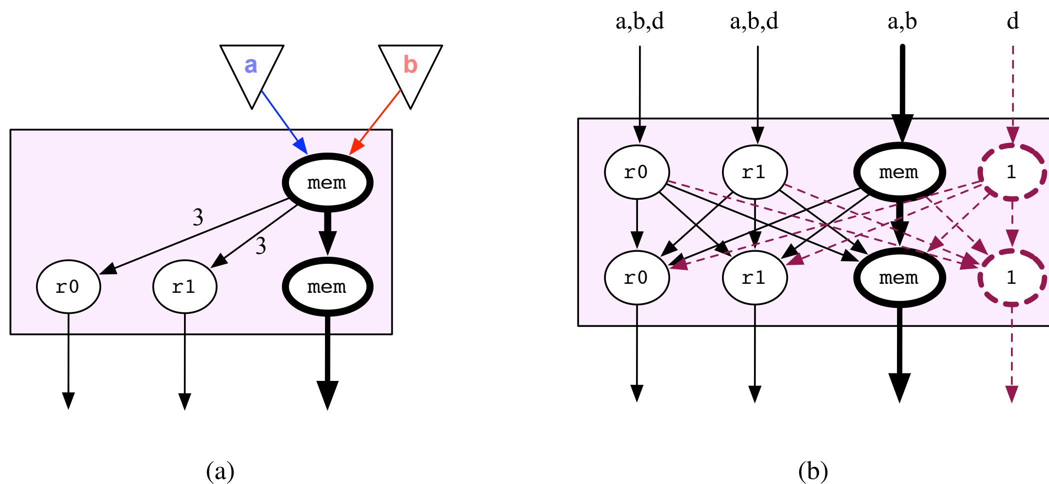


Figure 3.5: Crossbar groups for the local register allocation problem of Figure 3.2. The memory nodes are uncapacitated. (a) is the crossbar before the initial MOVE instruction. (b) is the crossbar after the initial MOVE instruction. The dashed edges can only be traversed by the flow for d. Cost labels are omitted for clarity. The memory nodes and constant nodes are uncapacitated.

a constant class for the constant 1 since the ADD instruction treats the constant 1 differently than other constants (it converts to an INC instruction). Typically, only a single memory allocation class representing an area on the stack frame is necessary, but in theory other memory allocation classes, such as an SRAM buffer or heap memory, could be considered as well.

Allocation class nodes are organized in layers where each layer corresponds to a program point or instruction and contains nodes representing each allocation class. Nodes corresponding to program points (locations in between instructions) belong to *crossbar groups* while nodes corresponding to instructions belong to *instruction groups*.

3.2.4 Crossbar Groups

A crossbar group contains multiple layers of allocation class nodes. Each layer consists of at most one representative of every possible allocation class (e.g. there are not two `r0` nodes in a layer). A layer represents a specific program point. Crossbar groups are inserted between every instruction group. A crossbar group is also inserted before the first instruction if there are variables live into the basic block. The defining point for these variables is considered to be the top of the crossbar. In our running example, the parameters a and b are live into the basic

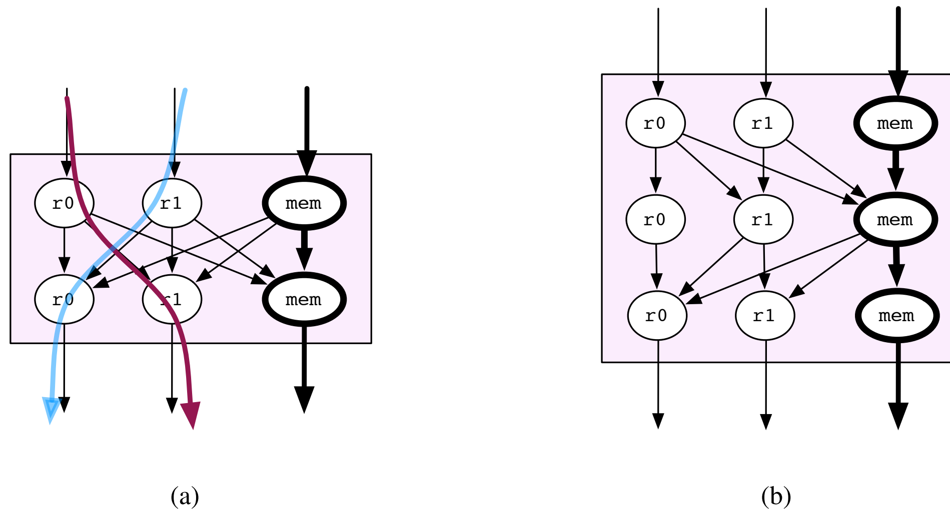


Figure 3.6: Two possible crossbar group network structures. (a) A full crossbar and (b) a zig-zag crossbar.

block and are defined into the top of the first crossbar as shown in Figure 3.5(a). Similarly, if any variables are live out of the block, a crossbar is inserted after the last instruction and the sink nodes of these variables are connected to the bottom of the crossbar.

The purpose of the crossbar group is to allow the allocation class of a variable to change. For instance, a variable might be stored from a register into memory, loaded from memory into a register, moved from one register to another, or rematerialized as a constant into a register. All of these transformations can be modeled by an edge from one layer of the crossbar group to another. For example, in the initial crossbar of Figure 3.5(a) the edges from the memory allocation class to the register classes have a cost of 3 since the size of a load instruction is 3 bytes.

Not all variables may be allowed to flow over all edges of the crossbar. Individual capacity constraints are used to prohibit variables from flowing through invalid allocation classes. An allocation class is invalid for a variable if it cannot be legally allocated to that class. For instance, a floating-point variable might not be able to be accurately or efficiently stored in an integer register. Constant allocation classes are restricted to variables that have a known constant value at the corresponding program point. In our example the variable d is known to have a constant value throughout the network and the constant can be rematerialized (loaded into a register) or used directly at any point. Figure 3.5(b) shows the crossbar inserted after the first instruction of

our running example. The edges to and from the constant allocation nodes are restricted to the flow of d.

The crossbar nodes are responsible for implementing the capacity constraints of the represented allocation classes. In our example, since only a single variable can be allocated to a single register at any program point, an `r0` register class node permits only a single unit of flow through it. In contrast, stack space is assumed to be essentially unlimited so the memory allocation class node does not limit the amount of flow through it. We implement these constraints by applying bundle constraints to the nodes of the crossbar. This is different from the classical representation of MCNF (Figure 3.1) where capacities are applied to edges. A node with a bundle constraint can be implemented within the framework of the classical MCNF formulation by converting the node to an additional capacitated edge.

We consider two possible crossbar network structures. *Full crossbars*, as shown in Figure 3.5 and Figure 3.6(a), consist of two layers of nodes. There is an edge from every node in the top layer to every node in the bottom layer. In a full crossbar, as shown in Figure 3.6(a), there are flows where the corresponding changes in allocation class cannot be implemented solely with move instructions. Instead, swap instructions are necessary. However, since the costs of the edges in the crossbar correspond to the cost of move instructions, this means that the cost model for the full crossbar may be inaccurate. For example, the single swap instruction that is needed in Figure 3.6(a) may be less costly than two move instructions.

A *zig-zag crossbar*, Figure 3.6(b), exactly represents the costs and constraints of a target architecture without support for swap instructions. In the zig-zag crossbar an additional middle layer of nodes is needed. From the top layer to the middle layer, data movement is only allowed in one direction (a zig to the right in Figure 3.6(b)) and then from the middle to the bottom layer, data movement is only allowed in the alternate direction (a zag to the left). This construct eliminates any need for swap instructions since flows will never cross over each other.

A full crossbar is more compact than a zig-zag crossbar. This results in lower memory requirements for the model and ultimately faster register allocation performance. The full crossbar is more flexible in that it supports the generation of swap instructions, but does not accurately model the costs of these swap instructions. In addition, not all architectures support arbitrary

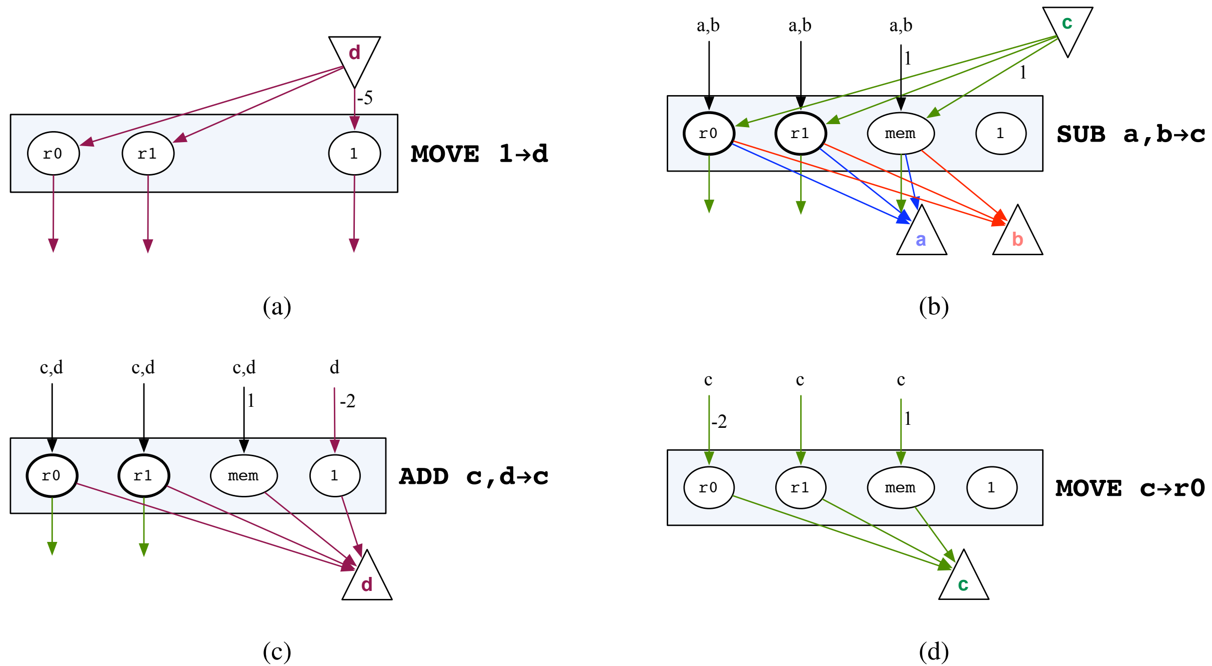


Figure 3.7: Instruction groups for the local register allocation problem of Figure 3.2. The register nodes are uncapacitated.

swap instructions and even when an architecture supports such an instruction, many compilers do not generate or support the instruction. As a result, the zig-zag crossbar is likely the more practical choice for most compilers and architectures.

3.2.5 Instruction Groups

An instruction group represents a specific instruction in the program and contains a single node for every allocation class that may be used by the instruction. The instruction groups of the running example are shown in Figure 3.7. Instruction groups represent the allocation constraints and preferences of the corresponding instruction.

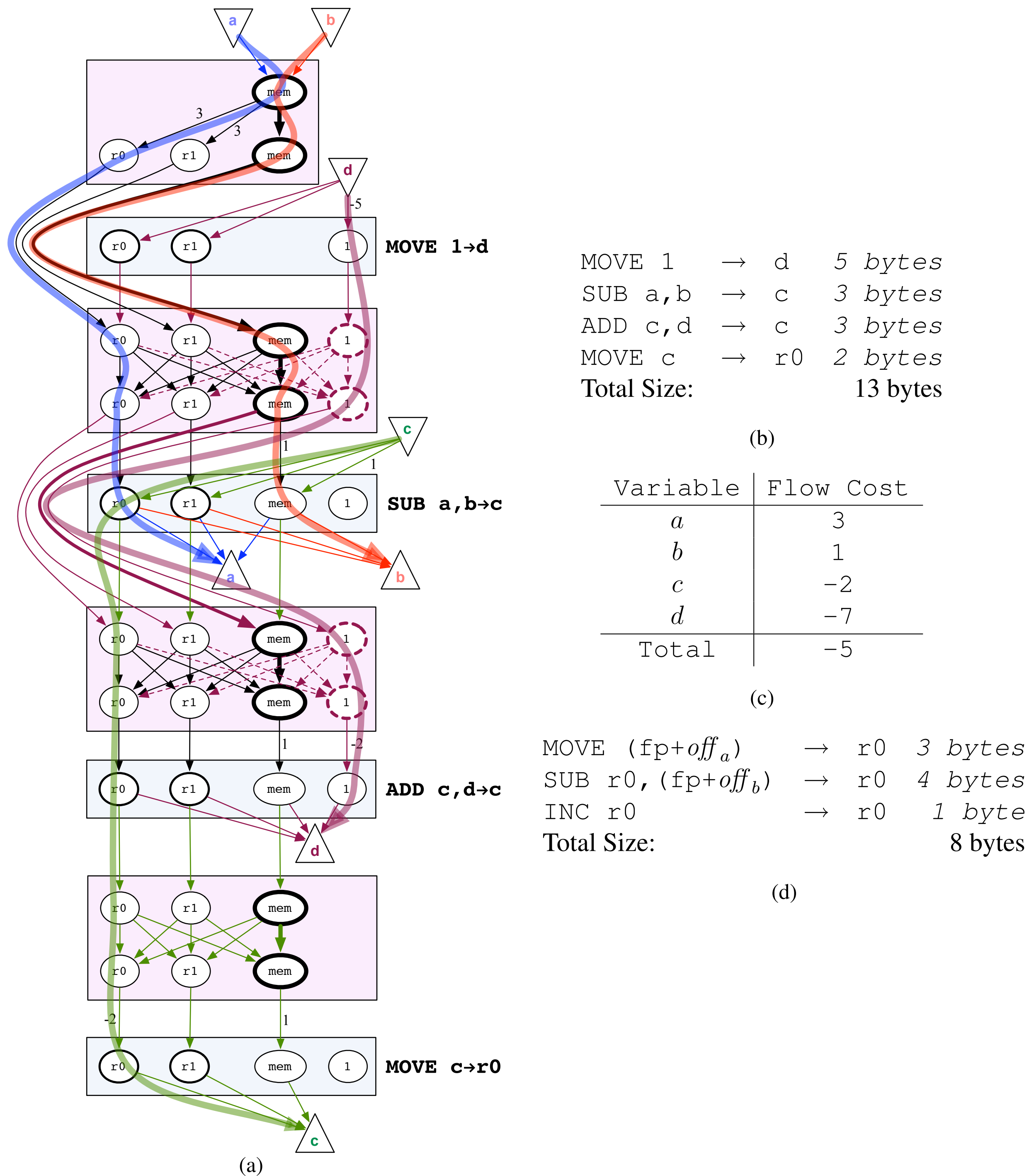
The nodes in an instruction group constrain what allocation classes are legal for the variables used by that instruction. In our running example, the first MOVE instruction cannot contain a memory operand so the corresponding instruction group, Figure 3.7(a), has no memory node. Instruction group nodes are also responsible for limiting the usage of allocation classes within an instruction. For example, in Figures 3.7(b), 3.7(c), and 3.7(d), only a single memory operand is

allowed in the corresponding instructions. As a result, unlike in a crossbar group where a memory node is uncapacitated, a bundle constraint is applied to the memory node in the instruction group. However, most instruction group nodes do not have a capacity constraint. The lack of a capacity constraint allows a variable that is defined by an instruction to be allocated to the same allocation class as a variable that is not live out of the instruction. For instance, in the SUB instruction of Figure 3.7(b), both a and c can be legally allocated to the register `r0`.

The costs on the edges into the instruction group nodes represent the allocation class preferences of the corresponding instruction. For example, in Figures 3.7(b), 3.7(c), and 3.7(d), any edge into the memory instruction group node has a cost of 1 representing the one byte increase in code size when an instruction contains a memory operand. In the instruction group for the first MOVE instruction, shown in Figure 3.7(a), the edge into the constant class node has a cost of -5 since the instruction can be eliminated if d is not loaded into a register at that point. The final MOVE instruction, shown in Figure 3.7(d), has a cost of -2 on the edge into the `r0` node since the instruction is rendered unnecessary if c is allocated to `r0` at that point.

3.2.6 Full Model

Source nodes, sink nodes, crossbar groups, and instruction groups are combined to generate an expressive model of local register allocation. The full model for the running example is shown in Figure 3.8(a). Instruction group nodes are directly connected to the corresponding allocation class nodes of the surrounding crossbar group nodes. If a variable is used within an instruction, it must flow through the nodes of the corresponding instruction group. Variables not used by an instruction bypass the instruction into the next crossbar group. In Figure 3.8(a) variables a and b bypass the first MOVE instruction, but flow through the nodes of the instruction group of the SUB instruction. The MCNF model exactly represents all the pertinent features of the local register allocation problem of Figure 3.2. As a result, the optimal solution to the MCNF model shown in Figure 3.8(a) corresponds directly to the optimal allocation of Figure 3.8(d).




```
MOVE 1      →  d     5 bytes
SUB a,b     →  c     3 bytes
ADD c,d     →  c     3 bytes
MOVE c      →  r0    2 bytes
Total Size:         13 bytes
```

(b)

Variable	Flow Cost
a	3
b	1
c	-2
d	-7
Total	-5

(c)

MOVE (fp+off_a)	→	r0	3 bytes
SUB r0,(fp+off_b)	→	r0	4 bytes
INC r0	→	r0	1 byte
Total Size:			8 bytes

(d)

Figure 3.8: (a) The full MCNF model of the local register allocation problem of Figure 3.2. The thick lines demonstrate a set of flows that solve the MCNF model. (b) The original unallocated assembly code, (c) the costs incurred by each variable in the shown MCNF solution, and (d) the allocated assembly code corresponding to this solution.

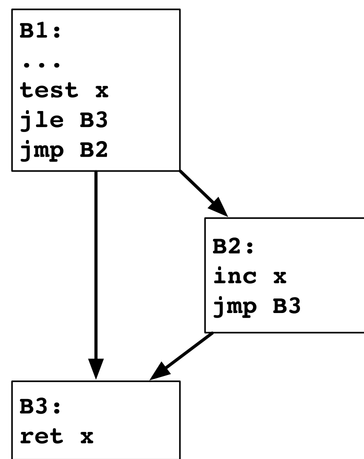


Figure 3.9: A simple control flow graph.

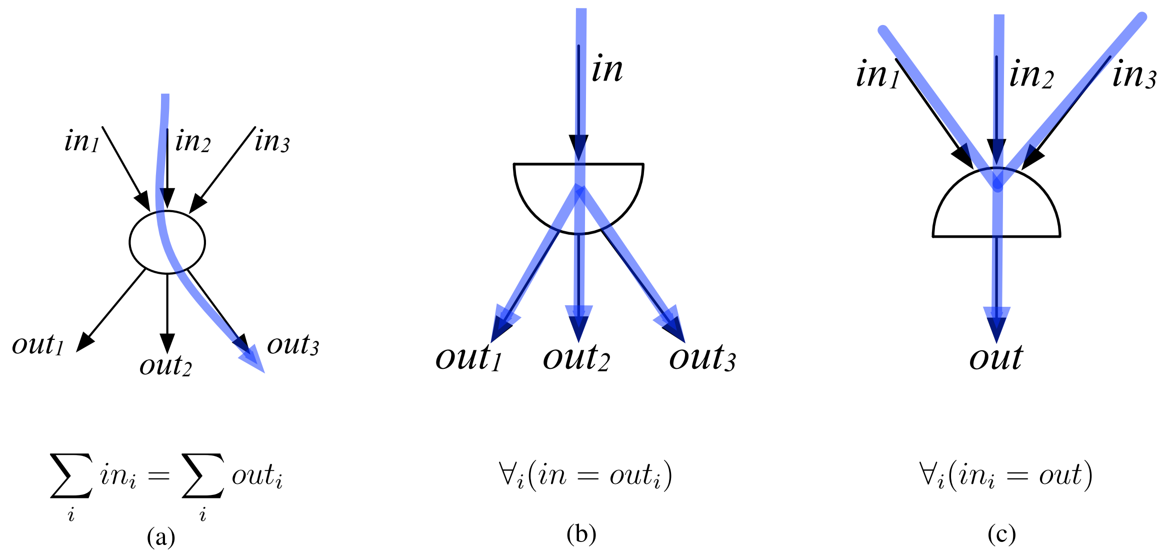


Figure 3.10: The three types of flow nodes in the global MCNF model of register allocation. A (a) normal network node, (b) an exit node with split flow constraints, and an (c) entry node with merge flow constraints.

3.3 Global Register Allocation Model

The MCNF model of the local register allocation problem can exactly express the costs and constraints of the most pertinent features of local register allocation. However, it cannot represent control flow. Consider the control flow graph of Figure 3.9. Although a local MCNF model can be constructed for each block, it is not possible to connect these models into a larger model that

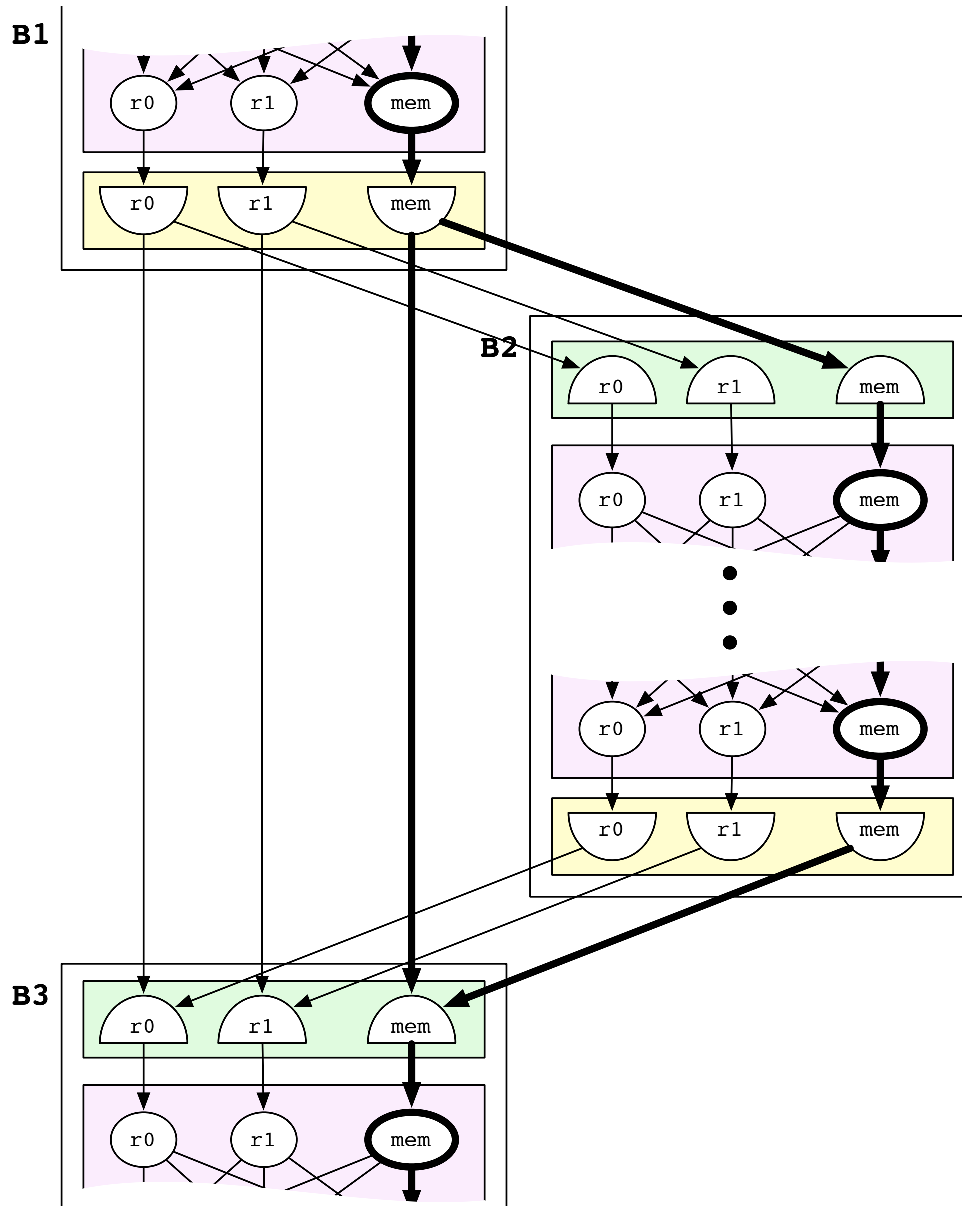


Figure 3.11: Entry and exit groups of a global MCNF model of register allocation.

accurately represents the global register allocation problem without extending the MCNF model to include additional constraints.

In the local MCNF model, a program point has exactly one successor program point and at each program point a variable is only allocated to a single allocation class. This invariant is maintained in the MCNF model through the node network constraints, Figure 3.10(a), which require that the amount of flow into a node equals the amount of flow out of the node. Since source nodes provide a single unit of flow, at each layer in the network a node either has a single unit of flow or no flow. When control flow is permitted, a program point may have multiple successors. For example, in Figure 3.9 the program point at the exit of block $B1$ has two successors: the entry of block $B2$ and the entry of block $B3$. Both block $B2$ and block $B3$ expect a full unit of flow for every live-in variable, but due to the node network constraints, the nodes at the exit of block $B1$ can only provide a total of one unit of flow. In order to represent control-flow, new flow constraints that support splitting flow (for branch points in the control flow graph) and merging flow (for merge points in the control flow graph) are needed. These node flow constraints are an extension of the classical MCNF problem and are represented using *exit nodes* for splitting flow and *entry nodes* for merging flow.

Exit Nodes Exit nodes do not have traditional flow constraints. Instead, the flow equations of an exit node, Figure 3.10(b), split the flow of the predecessor into multiple equivalent flows to each of the successor nodes. If the incoming flow of a variable is one, all the outgoing flows will be one. If the incoming flow of a variable is zero, all the outgoing flows will be zero.

A single layer of exit nodes, an *exit group*, is constructed at the end of every basic block. In order to model Figure 3.9, an exit group needs to be created for blocks $B1$ and $B2$ as shown in Figure 3.11. An exit node, as shown in Figure 3.10(b), has a single predecessor and multiple successors. Typically, the predecessor and successor nodes of the exit node all represent the same allocation class since data movement instructions cannot be inserted on control flow edges. The predecessor of an exit node is the bottom node in the final crossbar group of the basic block. The successors of an exit node are the entry nodes belonging to the successor blocks of the block containing the exit node. Only variables that are live out of the block flow out of an exit node.

In Figure 3.11, the `r0` exit node of block $B1$ has the bottommost `r0` crossbar group node as a predecessor and the `r0` entry nodes of blocks $B2$ and $B3$ as successors. If the variable x, which is live out of block $B1$, is allocated to `r0` at the exit of block $B1$, it is required to be allocated to `r0` at the entry of blocks $B2$ and $B3$.

Entry Nodes Entry nodes do not have traditional flow constraints. Instead, the flow equations of an entry node, Figure 3.10(c), merge all the flows of the predecessors into a single reduced flow that is equal in value to each of the incoming flows. That is, if all the incoming flows are one, the outgoing flow will be one. If all the incoming flows are zero, the outgoing flow will be zero. If the incoming flows are different values, the flow constraint of the entry node is not satisfied.

A single layer of entry nodes, an *entry group*, is constructed at the start of every basic block. An entry node has several predecessors, but only a single successor. Typically, all the predecessors and the successor of the entry node represent the same allocation class since data movement instructions cannot be inserted on control flow edges. The successor of an entry node is the top node in the initial crossbar group of the basic block. The predecessors of a entry node are exactly the exit nodes belonging to the predecessor blocks of the block containing the entry node. Only variables that are live into the block flow through a entry node. In Figure 3.11, the `r0` entry node of block $B3$ has the topmost `r0` crossbar group node as a successor and the `r0` exit nodes of blocks $B1$ and $B2$ as predecessors.

The global MCNF model exactly represents the control flow of the input control flow graph. In some cases, a higher quality register allocation may be obtained if the control graph is modified. For example, if critical edges are split, then spill and shuffle code can always be inserted along a control flow edge. If the input control flow graph contains critical edges, then the cost of inserting a new basic block along this edge can be approximated in the global MCNF model. Any data movement edges in crossbars after a branch instruction include the cost of creating a new basic block.

We now define the global MCNF problem formally as an optimization function over the commoditized edges of a network. Let x^q_{ij} represent the flow of a commodity q over an edge

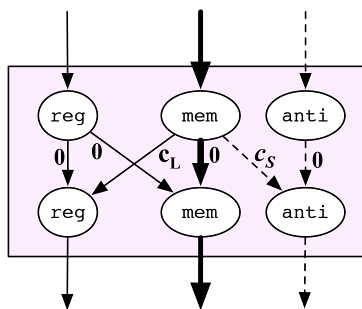


Figure 3.12: A crossbar group with nodes for anti-variables. Anti-variables pay the cost of a store, c_S, to move from a memory node to an antionly *node. Regular variables incur no cost to move to a memory node, but a variable and the corresponding anti-variable cannot be allocated to the same memory node. The cost of a load, c_L, is always incurred when moving from memory to a register.*

(i, j) and let (ext, ent) represent a pair of connected exit and entry nodes where $pred_{ext}$ is the predecessor of ext and $succ_{ent}$ is the successor of $entry$, then the global MCNF problem is:

$$\min \sum_{i,j,q} c^q_{ij} x^q_{ij} \qquad \textit{cost function} \tag{3.5}$$

subject to the constraints:

$$0 \leq x^q_{ij} \leq v^q_{ij} \qquad \textit{individual capacity constraints} \tag{3.6}$$

$$\sum_q x^q_{ij} \leq u_{ij} \qquad \textit{bundle constraints} \tag{3.7}$$

$$\mathcal{N} x^q = b^q \qquad \textit{network constraints} \tag{3.8}$$

$$x^q_{pred_{ext},ext} = x^q_{ext,ent} \qquad \textit{exit boundary constraints} \tag{3.9}$$

$$x^q_{ext,ent} = x^q_{ent,succ_{ent}} \qquad \textit{entry boundary constraints} \tag{3.10}$$

$$x^q_{i,j} \in \{0, 1\} \qquad \textit{integrality constraints} \tag{3.11}$$

This definition is identical to the MCNF problem formulation given in Section 3.1 with the addition of the exit and entry *boundary constraints*, which implement the flow constraints of exit and entry nodes, and the *integrality constraints*, which ensure that a variable cannot be simultaneously fractionally allocated.

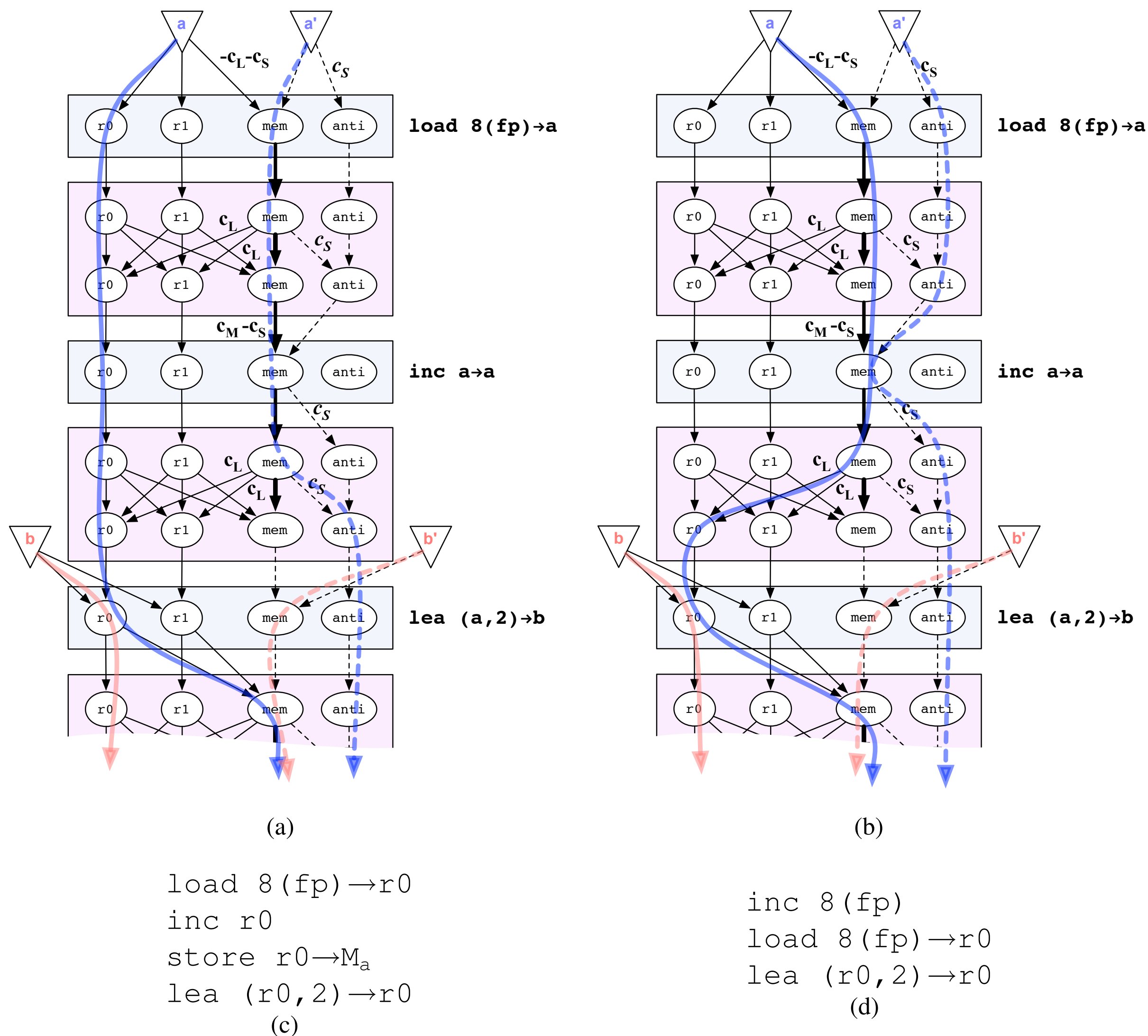


Figure 3.13: A network that demonstrates value modification, load rematerialization and anti-variables. The first instruction defines the variable a as the result of a rematerializable load from a known stack location. The second instruction increments a. This instruction is capable of modifying the value directly in memory. The third instruction uses a and defines b and requires that both operands be allocated to a register. We consider two possible allocations, both of which, for the sake of exposition, seek to allocate a and b to `r0` *at the* `lea` *instruction. This requires that a be evicted immediately after the* `lea` *instruction. In allocation (a) the variable a is first allocated to* `r0` *and then, prior to the* `lea` *instruction, is stored to memory to support the eviction. The resulting allocation has a total cost of c_S and is shown in (c). In contrast, allocation (b) leaves the variable a in memory, increments it directly in memory, and then loads the value into* `r0`*. The resulting allocation has a total cost of $-c_L - c_S + c_M - c_S + c_L + c_S + c_S = c_M$ and is shown in (d).*

3.4 Persistent Memory

The cost of an operation, such as a load or move, can usually be represented by a cost on the edge that represents a move between allocation classes. However, this does not accurately reflect the cost of storing to memory. If a variable has already been stored to memory and its value has not changed, it is not necessary to pay the cost of an additional store. Values in memory are persistent, unlike those in registers which are assumed to be overwritten.

In a model that represents the persistence of values in memory, the cost of a store should only be incurred the first time a variable is stored to memory. This cost cannot be implemented as a simple variable edge cost. An edge cost is incurred regardless if the transition to memory is the first transition, which requires a store, or the second transition, which is free since the value is already available in memory. Instead of modeling the cost of a store at the point where a variable transitions into memory, we use an additional commodity, an *anti-variable*, to denote whether a value is available in memory or not. If the value of a variable is not available in memory at a location (a store is needed in order to transition the variable into memory), then the anti-variable occupies the memory node for that location at no cost. However, if the variable's value is available in memory, the anti-variable must be allocated to an *antionly* allocation class for the cost of a store.

Anti-variables can only flow through memory nodes or *antionly* nodes. Only anti-variables can flow through *antionly* nodes and the allocation of an anti-variable to an *antionly* node indicates that the variable's value is available in memory. Both a regular variable and its corresponding anti-variable can flow through a memory node, but they are mutually exclusive. A variable cannot coexist with the corresponding anti-variable at a memory node.

As shown in the crossbar of Figure 3.12, an anti-variable can flow from a memory node to an *antionly* node. This transition indicates that the variable's value is now available in memory and so always incurs the cost of a store, c_S, in the crossbar. As shown in Figure 3.12, a regular variable incurs no cost when flowing into a memory node. However, because a variable and its corresponding anti-variable cannot coexist in the same memory node, in order for a regular variable to be evicted to memory, the cost of a store must be incurred in order to transition the anti-variable into an *antionly* node. Once this store cost is incurred, the anti-variable stays in the

antionly allocation class indicating that the variable's value remains available in memory. The regular variable can now move from registers to memory multiple times and yet only pay the cost of a single store (of course, every transition from memory to a register pays the cost of a load, c_L, in Figure 3.12).

The transition of an anti-variable from a memory node to an *antionly* node not only represents the cost of storing the variable, it also indicates the location of the required store instruction. As a result, edges from memory nodes to *antionly* nodes should only be constructed where the corresponding store instruction would be legal. In contrast, a regular variable may legally move into memory at any point including at basic block boundaries and immediately after an instruction group. This later case is illustrated by the `lea` instruction in Figure 3.13 where both the source operand, a, and destination operand, b, of a `lea` instruction are allocated to the same register `r0`. This results in the value of a in $r0$ being overwritten, but since the value of a is available in memory (the anti-variable a' is allocated to an *antionly* node at this point) a can be immediately evicted to memory at no cost additional cost (the cost of the store was already paid by the anti-variable).

An anti-variable can remain allocated to the *antionly* allocation class as long as the value of the corresponding variable does not change. If the variable is redefined then the anti-variable is forced back into a memory node at the redefining instruction. In Figure 3.13(b), the value of a is modified by the `inc` instruction forcing the anti-variable a' back into the memory node of this instruction group. Thus an additional store will be necessary if the new value needs to be evicted to memory. If the modifying instruction supports redefining the variable directly in memory, then, as with the `inc` instruction in Figure 3.13(b), the cost of a store, c_S, is subtracted from the memory access cost, c_M, of the instruction. Since in order for the regular variable to stay in memory the anti-variable must immediately transition back to an *antionly* node and incur the cost of a store, the combined cost of the variable and anti-variable flows is simply the memory access cost.

A similar application of a negative store cost is necessary when modeling load rematerialization. If the value of a variable is already available in memory, perhaps as part of a stack allocated object, then it may not be necessary to immediately generate an explicit load instruction. This

Benchmark Suite	Name	Lines of Code
Mibench[57]	dijkstra	133
	patricia	292
	stringsearch	3070
	qsort	202
Mediabench[79]	g721 encode	901
	g721 decode	903
SPEC2006[123]	429.mcf	1574
	470.lbm	904

Table 3.1: Reduced benchmark suite suitable for optimal allocation.

is modeled by defining the variable directly into memory. Future instructions that access the variable can either directly access the original memory location or generate a load instruction to move the variable into a register. If the original memory location is read-only, the network structure is constructed accordingly. As shown by the load instruction in Figure 3.13, when a variable is defined directly into memory as part of a rematerializeable load, the cost is equal to negative the cost of a load (c_L) and a store (c_S). The negative store cost cancels out the cost of transitioning the anti-variable into an *antionly* node while the negative load cost represents the benefit of eliminating the defining load instruction. Note that the use of negative store costs to cancel out the store costs incurred by anti-variables assumes a uniform cost model where the cost of a store is the same at every program location. This is not the case when optimizing for performance, since stores inside loops are generally considered more expensive then stores outside loops. However, when optimizing for performance a memory modification operation is expected to incur the cost of both reading and writing to memory, hence no negative cost is needed.

3.5 Modeling Costs

The global MCNF model of register allocation can support highly precise and detailed cost models. A unique cost can be associated for every transformation of every variable at every program

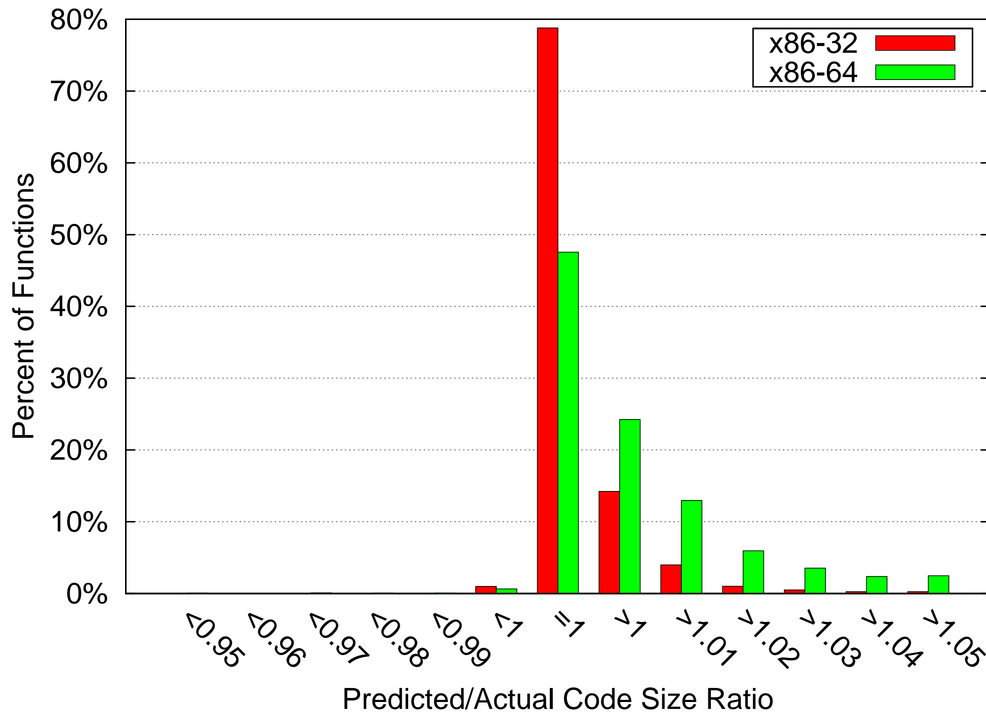
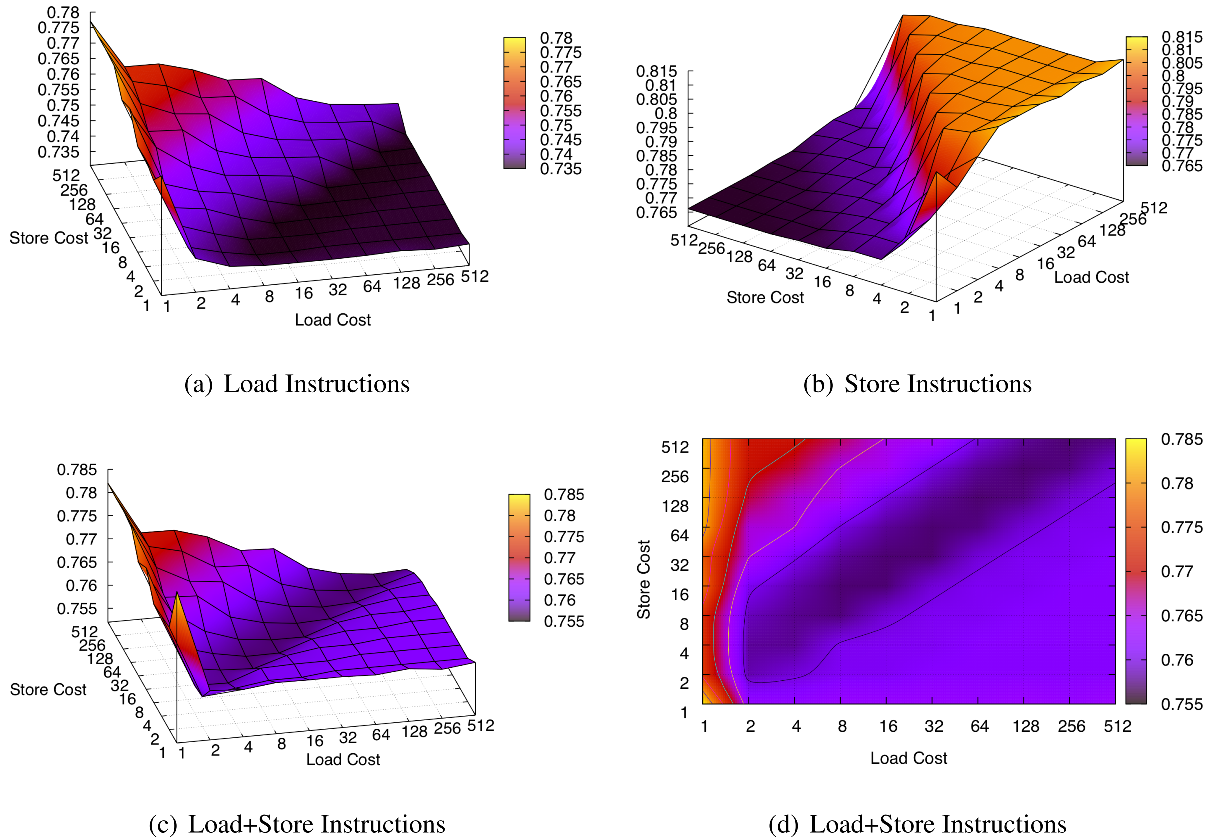


Figure 3.14: The accuracy of the code size global MCNF cost mode. The histogram bins functions based on the accuracy ratio: $predicted/actual$.

point. However, in practice, cost models with some degree of uniformity are used. We consider how two code quality metrics, code size and performance, affect the cost model.

The global MCNF model can exactly and precisely model code size, modulo the limitations of the model (Section 3.6). The code size metric is highly uniform. Costs depend exclusively on the instruction set architecture, not the program, and can be perfectly determined at compile-time. As such, an optimal solution to the global MCNF model should directly correspond to an optimal register allocation. We validate the fidelity of our code size cost model by comparing the code size predicted by the model with the actual post-register allocation code size (as measured internally in the compiler) when the hybrid allocator of Chapter 5 is used to register allocate the benchmarks described in Chapter 4.

A histogram of the ratio of the predicted value relative to the actual value is shown in Figure 3.14 for two instruction set architectures, x86-32 and x86-64. For the x86-32 ISA, nearly 80% of the functions are exactly predicted and more than 93% of the functions are predicted to be within 1% of the actual result. Nearly all of the mispredicted functions are over-predictions. In



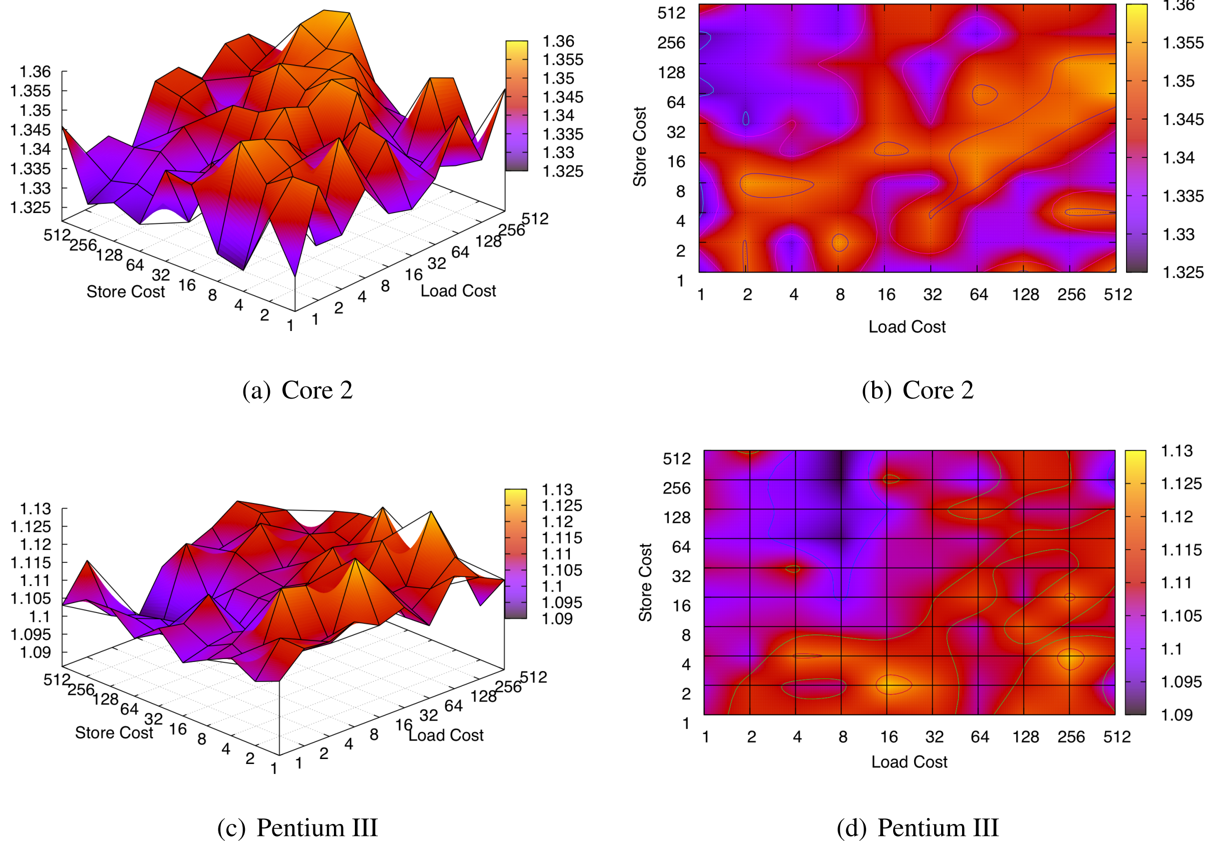
(a) Load Instructions

(b) Store Instructions

(c) Load+Store Instructions

(d) Load+Store Instructions

Figure 3.15: The relative change in the number of dynamic (a) load and (b) store instructions in an optimal register allocation as the single-execution costs of memory operations are changed. (c) and (d) show two different views of the same data. Load and store costs are relative to the cost of a executing a move instruction. The extend linear-scan heuristic allocator of the compiler is the baseline. Regardless of the cost configuration, the optimal allocation achieves at least a 20% reduction in executed memory operations compared to the default allocator. Results are from a reduced benchmark set (Table 3.1) where an optimal register allocation can be found for each configuration in a matter of hours.



(a) Core 2

(b) Core 2

(c) Pentium III

(d) Pentium III

Figure 3.16: The impact on performance of varying single-execution costs in an optimal register allocator when targeting the x86-32 instruction set architecture. Two views, (a) and (b), of the performance of an Intel Core 2 Quad (Q6600) processor and two views, (c) and (d), of the performance of an Intel Pentium III (Coppermine) processor are shown. Load and store costs are all relative to the cost of a executing a move instruction. The default extended linear-scan heuristic allocator of the compiler is the baseline. Larger ("hotter") values indicate improved performance. Results are from a reduced benchmark set (Table 3.1) where an optimal register allocation can be found for each configuration in a matter of hours.

these cases, the register allocation enables an optimization that is not captured within the model, such as move coalescing. The model is not as accurate when predicting register allocation outcomes on the x86-64 ISA. Only 71% of the functions are predicted to be within 1% of the actual result and most mispredictions are over-predictions. This is due to the increased relevance of the limitations discussed in Section 3.6. On the x86-64 architecture the model must approximate the cost of certain allocation decisions and these approximations are chosen to conservatively overestimate the final cost of allocation.

In contrast to code size, a fully accurate cost model of performance is unattainable. Performance depends on the execution behavior of the program, which is mostly unknown at compile-time. Even if a good approximation of run-time behavior can be derived, the performance characteristics of modern architectures are difficult to impossible to quantify at compile-time due to complex micro-architectural features such as caches, branch and value predictors, and reorder buffers.

In light of these difficulties, instead of attempting to design a cost model that directly models processor performance, we define a *weighted execution costs* metric of code quality that is closely related to performance. The weighted execution costs metric attempts to minimize the total dynamic cost of memory operations and instructions. In this cost model a specific kind of operation, such as a load, incurs the same cost every time it is executed regardless of the context of execution. For example, a load instruction might always incur a cost of 3. In actuality, the latency of the load instruction will depend on its execution context, such as the contents of the cache and load store queue, but given the opacity of these architectural features to the compiler, a uniform cost approximation is reasonable. Although costs are uniform across execution contexts, they a specific to each type of operation. There can be a different cost for a load than for a store, for example. We refer to these costs as the *single-execution cost* of an operation.

In order to optimize for the run-time behavior of a program, the single-execution cost of a specific operation needs to be multiplied by the number of times that operation is expected to be executed. Since all instructions in a basic block are executed the same number of times, what is needed is an approximation of run-time behavior that provides execution frequencies of basic blocks. Ideally, high quality profile data that accurately represents the future run-time

behavior of a program can provide accurate execution frequencies. If high quality profile data is not available, more sophisticated static analyses [10, 27, 130] can be used to predict basic block execution frequencies. This cost model is easily applied to the global MCNF model. The costs of memory operations and data movement instructions are represented by edge costs. An edge cost is simply set to the product of the execution frequency of the basic block and the single-execution cost of the corresponding operation.

To achieve the best possible performance, the single-execution costs need to be set appropriately. When optimizing for performance, the three most important costs are loads, stores, and moves. Using an optimal allocator targeting the x86-32 architecture and a reduced benchmark set, shown in Table 3.1, for which optimal allocations can be found relatively quickly we consider a large configuration space of possible load/store/move costs. In computing our optimal allocation we use perfect profiling data to derive basic block execution frequencies. Since the ratio between costs is what determines the quality of the optimal allocation, we fix the cost of a move instruction to 1 and vary the costs of loads and stores exponentially from 1 to 512. That is, we consider single-execution costs of 2^i for i from 0 to 9 for loads and stores resulting in 100 data points.

The effect of varying load/store cost ratios on the dynamic execution count of load operations is shown in Figure 3.15(a). As expected, as the cost of a load relative to a store and a move increases, the number of dynamic load instructions decreases. This is the bottom right point in Figure 3.15(a). On the opposing side, where store and move instructions are expensive relative to a load instruction, many more load instructions are generated. Unsurprisingly, the results for store operations, Figure 3.15(b), show the opposite trend. If the goal is to minimize the total number of memory operations, the sum of load and store operations, then Figures 3.15(c) and 3.15(d) suggest a simple strategy for setting the single-execution costs. The trough shape of this graph suggests that load and store operations have an identical cost. The value of this cost does not appear to be very important as long as it is more than twice the cost of a move instruction.

The weighted execution costs metric only imperfectly models the code performance metric. We consider the effect of varying load/store cost ratios on performance for two Intel microarchitectures in Figure 3.16. Unlike Figure 3.15, no clear patterns emerge. The noise in the data is

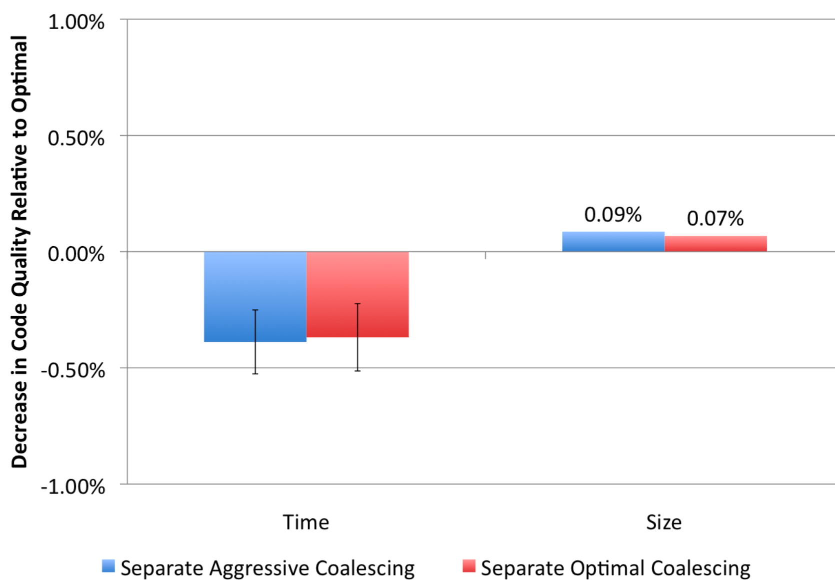


Figure 3.17: Increases in execution time and code size relative to an optimal allocator when coalescing is performed as a separate pass prior to register allocation. Results are shown both for a heuristic aggressive coalescer and an optimal coalescer. The optimal allocator and benchmark suite used are described in [73] and are evaluated on the Intel x86-32 architecture.

not due to measurement error, but to legitimate differences in performance that are not correlated with the number of memory operations. For instance, register allocation can affect branch prediction performance by incidentally changing the values of branch targets. Despite the noisiness of the data, it does appear that the best performing configurations are clustered around the same diagonal that defined the trough of Figure 3.15(c). This suggests a similar metric, where load costs and store costs are identical but more than twice move costs, is effective at optimizing for performance.

3.6 Limitations

The global MCNF model can explicitly model instruction usage constraints and preferences, spill code generation, move insertion, constant rematerialization, and load rematerialization. In addition, our model can model a limited amount of register-allocation driven instruction selection. For example, in Figure 3.8(a) the model explicitly encodes the fact that if an operand of the `ADD` instruction is the constant 1, a more efficient `INC` instruction can be used. There are

some fundamental limitations of the model. Inter-variable allocation class usage preferences and constraints cannot be represented so move coalescing, two-op conversion of commutative operations, consecutive register requirements, coupled register preferences, and source-dependent store costs cannot be precisely modeled. Additionally, the model does not support value cloning.

The global MCNF model cannot represent inter-variable allocation class usage preferences or constraints. That is, the model cannot represent a statement such as, "if a is allocated to X and b is allocated to Y in this instruction, then a 2 byte smaller instruction can be used." For example, on the x86 architecture a sign extension from a 16-bit variable a to a 32-bit variable b is normally implemented with a 3-byte `movsxw` instruction, but if both a and b are allocated to the register `eax` then a 1-byte `cwde` instruction may be used with the same effect. This code size savings cannot be exactly represented in our model because edge costs only apply to the flow of a single variable. If the instruction stream is modified so that a move from a to b is performed before the sign extension and the sign extension has b as its only operand, then the model is capable of exactly representing the cost savings of allocating b to `eax` with the caveat of requiring a more constrained and possibly less efficient instruction stream as input.

Inter-variable register usage constraints are required in order to exactly model the costs associated with move coalescing. Since our model cannot explicitly represent the benefits of move coalescing, moves are aggressively coalesced before register allocation. The model explicitly represents the benefit of inserting a move so there is no harm in removing as many move instructions as possible. Furthermore, there is little difference in code quality between aggressively coalescing and then finding an optimal, coalescing-agnostic, allocation and finding a fully optimal allocation. This is shown empirically in Figure 3.17. When optimizing for performance, a fully optimal allocator is slightly outperformed by sub-optimal allocators indicating that any performance benefit from fully integrating coalescing into an optimal framework is dominated by the unavoidable noise when optimizing for performance. In contrast, when optimizing for code size, an increase in code size is observed with the less optimal allocators. However, the difference is less than a tenth of a percent. It is also worth noting that a heuristic aggressive coalescing algorithm performs nearly as well as an optimal coalescing algorithm further justifying our approach.

Another example where inter-variable register usage preferences are useful is in modeling the conversion of a three operand representation of a commutative instruction into a two operand representation. Internally, a compiler might represent addition as $c = a + b$ even though the target architecture requires that one of the source operands be allocated to the same register as the destination operand. Ideally, the model would be able to exactly represent the constraint that one of the source operands, a or b, be allocated identically with c. Converting *non*-commutative instructions into two operand form does not pose a problem for our model as these instructions can be put into standard form without affecting the quality of register allocation.

On some architectures inter-variable register usage constraints might exist that require a double-width value to be placed into two consecutive registers. The SPARC architecture, for example, requires that 64-bit floating point values be allocated to an even numbered 32-bit floating point register and its immediate successor. Our MCNF model is not capable of representing such a constraint. This limitation does not effect our ability to handle SIMD instruction that manipulate several values at once. The register allocator treats these several values as a single variable that requires a SIMD register allocation class.

A *coupled register preference* is when an instruction prefers that two or more of its operands be allocated from the same set of registers. For example, the x86-64 ISA has 16 integer registers, 8 of which are legacy registers from the x86-32 ISA (`eax`, `edx`, etc.). If an instruction manipulates values with bit-widths less than or equal to 32, an additional byte is needed in the instruction word if one or more of the non-legacy registers are used. Since only a single additional byte is needed regardless of how many operands are allocated to non-legacy registers, this preference cannot be exactly modeled using costs on individual flows. Instead, when optimizing for code size, we model an approximation where each non-legacy register allocation contributes a byte. If two operands are allocated to a non-legacy register, then a cost of two bytes is incurred.

Source-dependent store costs are another feature of the x86-64 ISA that require inter-variable allocation class preferences, in this case between a variable and the corresponding anti-variable. An ISA has source-dependent store costs if the cost of the store depends on the source register. On x86-64, a 32-bit or smaller store from a legacy register is one byte smaller than a store from a non-legacy register. In our model, the cost of the store is paid by the anti-variable and

is independent of the register the corresponding variable is allocated to at the location of the store. This problem only arises if a variable can be legally allocated to two source registers with different store costs at the same program point. Also, load instructions, where the cost is paid by the variable itself, can still be correctly modeled. The over-estimation of costs in the x86-64 model due to coupled register preferences and source-dependent store costs is clearly seen in the histogram skew of Figure 3.14.

An additional limitation of our model is that it assumes that it is never beneficial to allocate the same variable to multiple registers at the same program point. This limitation arises because there is a direct correspondence between the flow of a variable through the network and the allocation of the variable at each program point. The assumption that it will not be beneficial to allocate a variable to multiple registers at the same program point seems reasonable for architectures with few registers where register pressure is a prime concern. However, if an architecture has highly non-orthogonal register preferences, it might be advantageous to clone the value of a variable to meet these preferences without introducing extra data movement instructions.

3.7 Hardness of Single Global Flow

In this section we show that the global MCNF problem is harder than the local MCNF representation of register allocation. Although both problems are NP-complete (both local and global register allocation are NP-hard problems), the optimal flow for a single variable in the local MCNF model can be found with a simple shortest path computation. In contrast, we now prove that in global MCNF the optimal flow problem for a single variable is NP-complete. We first define the *minimal graph labeling* problem, then show that global MCNF single optimal flow is reducible to minimal graph labeling, which we show to be NP-complete via a reduction from graph coloring. Finally, we show that minimal graph labeling is reducible to global MCNF single optimal flow, completing the NP-completeness proof.

First, we simplify our consideration of the problem by computing shortest paths in every block in the model and using the results to reduce each block to a full crossbar between the possible allocation classes. The costs on each edge of the crossbar are the shortest path costs

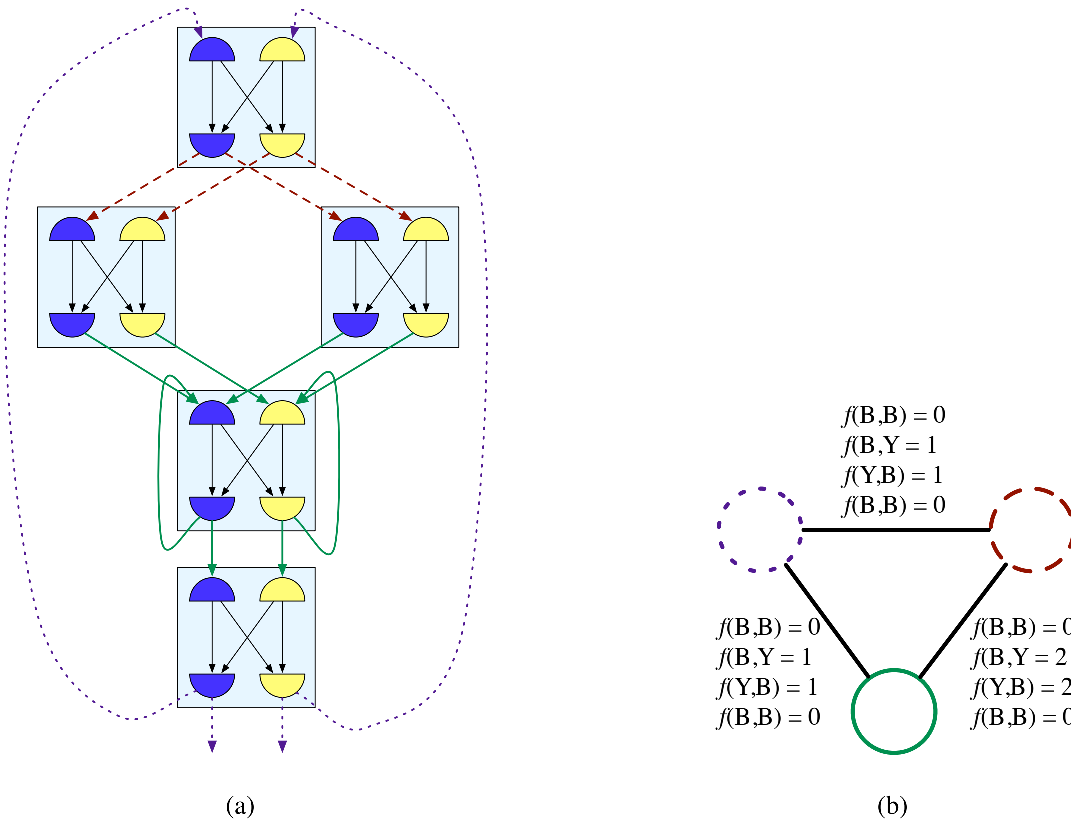


Figure 3.18: An example of a reduction from global MCNF to minimum graph labeling. In the global MCNF problem (a) there are three fully connected sets of cross-block edges. These edge sets map directly to nodes in a minimal graph labeling problem (b). The two allocation classes map directly to the labels B and Y. The edge cost functions are determined from the costs of the cross bar edges in the global MCNF problem.

from the entry node of the top allocation class to the exit node of the bottom allocation class. Furthermore, we recognize that due to the boundary constraints, any set of fully connected cross-block edges must all carry the same flow. That is, each set of fully connected cross-block edges represents a single allocation choice.

Using these insights, we view the global single optimal flow problem as a minimal graph labeling problem. Given a graph $G = (E, V)$ and a set of $k \in K$ labels where every edge $e \in E$ has a cost function $f_e : K \times K \rightarrow \mathbb{Z}$ the minimal graph labeling problem is to find a labeling $l(v) : V \rightarrow K$ such that $\sum_{e \in E} f_e(l(e_{src}), l(e_{dst}))$ is minimized.

The global MCNF single optimal flow problem is reducible to the minimal graph labeling problem. Let $G = (E, V)$ be constructed such that every $v \in V$ corresponds to a set of fully

connected cross-block edges in the global MCNF problem and an edge $(v_1, v_2) \in E$ exists if the sets of fully connected cross-block edges corresponding to v_1 and v_2 are adjacent: share a block as a common endpoint. Let K correspond to the allocation classes in the global MCNF problem. Define $f_e(k_1, k_2)$ to be equal to the cost of transitioning allocation classes between cross-block edge sets. If the edge sets are separated by a single block then this cost is just the result of a single shortest path computation. Otherwise it is a sum of shortest path computations. An example of such a reduction is shown in Figure 3.18.

It is easy to see that graph coloring is also reducible to the minimal graph labeling problem. Simply let K correspond to the set of colors and define f_e such that:

$$f_e(k_1, k_2) = \begin{cases} 0 & k_1 \neq k_2 \\ 1 & k_1 = k_2 \end{cases}$$

If the minimal labeling has a cost of zero, the graph G is colorable. Since the decision problem for minimal graph labeling is clearly in NP and graph coloring is reducible to minimal graph labeling, minimal graph labeling is NP-complete.

We now show that the minimal graph labeling problem is reducible to the general global MCNF single optimal flow problem. Given an instance of the minimal graph labeling problem, we create a global MCNF problem. Let K correspond to the allocation classes of the global MCNF problem. Let $G' = (V', E')$ be the line graph of G. That is, for every edge in G create a node in G' and connect two nodes in G' if the corresponding edges in G are adjacent (share a common endpoint). Now let G' represent the control flow graph of a global MCNF problem. Every block in the global MCNF problem corresponds to a node in G' that corresponds to an edge in G with a cost function f_e. For each block, construct a full crossbar such that the cost of an edge from allocation class k_1 to allocation class k_2 is equal to $f_e(k_1, k_2)$. Clearly, an optimal single flow through the resulting global MCNF problem directly corresponds to an optimal minimal labeling. Since minimal graph labeling is NP-complete and minimal graph labeling is reducible to a single optimal flow global MCNF problem, the global MCNF single optimal flow problem is NP-complete.

Although the general form of the problem is NP-complete, it is worth noting that real global MCNF problems are not necessarily as challenging. First, the proof relies on the ability to cre-

ate arbitrary control flow graphs. Although arbitrary control graphs can always be constructed using language constructs such as `goto`, real programs are generally limited in their structure [56, 124]. Additionally, the proof relies on the ability to create arbitrary crossbars. In particular, a graph-coloring derived global MCNF problem would need to consistently penalize maintaining an allocation in the same allocation class across a block while rewarding changing allocation classes. Such a cost structure could be created through the highly contrived use of register preferences and constraints, but real programs would be expected to exhibit the opposite behavior. It is therefore reasonable to expect that effective solution techniques for tackling this NP-complete problem can be developed.

3.8 Simplifications

The global MCNF model of register allocation is a highly detailed and expressive representation of the register allocation problem. However, in some cases the model is overly expressive. Model simplifications that reduce the number of nodes or edges not only reduce the memory requirements of the model, but also make the model easier to solve.

The global MCNF model is uniformly expressive. However, in several instances the expressiveness of the model can be reduced with little or no impact on the optimal value of the model. One easy simplification is to not model instructions, such as unconditional jumps, that do not influence register allocation. These instructions make up about 4-5% of an average instruction stream.

Another relatively straightforward simplification is to only model loads of a variable immediately before the instructions that use the variable. Similarly, stores of a variable need only be modeled immediately after the instructions that define the variable. Crossbar groups at the entry and exit of all basic blocks are left unsimplified so that inter-block allocation decisions can be unified. A variable can always be loaded from memory or stored to memory at the beginning and end of each block. This simplification does not affect the value of the optimal solution as long as the cost of a memory operation is independent of the position of the operation within a block.

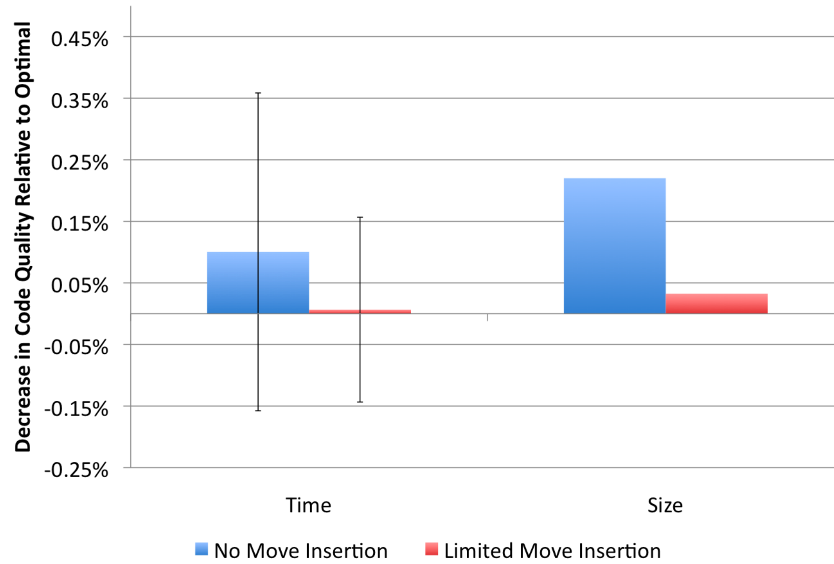


Figure 3.19: Increases in execution time and code size relative to an optimal allocator when move insertions are restricted. The optimal allocator and benchmark suite used are described in [73] and are evaluated on the Intel x86-32 architecture. In the no move insertion configuration the register allocator cannot insert any register to register moves. In the limited move insertion configuration the register allocator can only insert register to register moves at basic block boundaries. The difference in quality is small for the no move insertion configuration, but miniscule for the limited move insertion configuration.

A more sophisticated simplification identifies regions with no register pressure. In the absence of register preferences, there is no incentive for variables to change allocations within such a region since there is no competition for registers. If register preferences are present but only affect a single allocation class per a variable, then there remains no incentive for variables to change allocations within the region. As a result, such regions can be greatly simplified by removing all the crossbar groups and directly connecting instruction groups. Any load or store edges can be shifted prior to the simplified region (loads) or after the simplified region (stores). Empirically, when targeting the x86-32 architecture, an average function has about two-thirds of the crossbars simplified out. This fraction is expected to be larger for architectures with larger register sets.

A simplification that can substantially reduce the number of edges in the network, but that may alter the optimal value of the problem, reduces the number of edges in every crossbar group by prohibiting or restricting the movement of variables between register allocation classes. The

justification for this simplification comes from the observation that register to register move instructions rarely need to be inserted to obtain a near-optimal allocation. Indeed, as shown in Figure 3.19, restricting register to register move insertions to basic block boundaries has almost no affect on code quality in an optimal register allocator.

Finally, we consider a simplification that dramatically reduces the size of the model, but also fundamentally changes the nature of the problem. Instead of treating each register as a single register allocation class with a unit capacity, we can merge similar registers into a single register allocation class with a capacity equal to the number of registers. This approach is especially effective on architectures with many registers. However, the resulting global MCNF problem does not model the full register allocation problem. Instead, it represents the spill code optimization problem. A solution to this problem could replace heuristic spill code generators in register allocators that separate the spill code and assignment problems (Section 2.1.2) or it could be used as a starting point for finding a solution to the full global MCNF problem.

3.9 Summary

In this chapter we described our global MCNF model of register allocation. Our global MCNF model explicitly and exactly represents the pertinent components of the register allocation problem. As we shall see in the next chapters, the structure of the model naturally lends itself both to heuristic solution techniques and more principled solution techniques that can approach an optimal solution. These properties make global MCNF an ideal model for register allocation.

Chapter 4

Evaluation Methodology

In this chapter we describe the methodology used to evaluate the effectiveness of our backend compiler optimizations. We implement our optimization algorithms within version 2.4 of the state-of-the-art LLVM compiler infrastructure [87]. In the LLVM compiler, the code is converted out of SSA-form and move instructions are aggressively coalesced prior to register allocation. We evaluate the quality of register allocation on an expansive benchmark suite using two code quality metrics, four instruction set architectures, and two microarchitectures. Unless stated otherwise, all our results are expressed relative to the extended linear-scan allocator of the LLVM compiler. This allocator has been shown to outperform traditional graph coloring allocators [115].

4.1 Benchmarks

We perform our evaluations using a subset of the C benchmarks of the SPEC2006 [123] benchmark suite. This is an industry standard suite of real-world benchmarks. A descriptive list of the benchmarks is shown in Table 4.1. These benchmarks consist of approximately 100,000 lines of code. A subset is used so that we can perform an extensive evaluation in a limited amount of time.

Suite	Benchmark	Lang.	LoC	Brief Description
SPECint	401.bzip2	C	5,731	File compression. bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
	429.mcf	C	1,574	Vehicle scheduling. Uses a network simplex algorithm to schedule public transport.
	456.hmmer	C	20,658	Protein sequence analysis using profile hidden Markov models.
	458.sjeng	C	10,544	A highly-ranked chess program that also plays several chess variants.
	462.libquantum	C	2,605	Simulates a quantum computer running Shor's polynomial-time factorization algorithm.
	464.h264ref	C	36,098	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets.
SPECfp	433.milc	C	9,575	A gauge field generating program for quantum chromodynamics.
	470.lbm	C	904	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D.
	482.sphinx3	C	13,128	A widely-known speech recognition system from Carnegie Mellon University

Table 4.1: Benchmarks used in evaluation of code quality.

4.2 Code Quality Metrics

We consider two code quality metrics: code size and code performance. Both metrics are common measures of code quality and most compilers support explicitly optimizing for these metrics.

4.2.1 Code Size

Code size is the size of the binary application after compilation and linking. Code size as a code quality metric is of primary interest to the embedded community, where applications often face tight memory constraints. Reductions in code size can directly translate to higher profit margins if a smaller amount of ROM can be used.

Since code size depends upon the instruction set architecture and the static program code, it can be almost perfectly predicted at compile-time (on some architectures the final size of branch instructions must be determined at link-time). As a result, we can evaluate both the *post-regalloc code size*, the code size immediately after register allocation, and the *final code size*. We measure the post-regalloc code size internally in the compiler by summing the size of all the instructions

in all the compiled functions. We measure final code size as the size of the `.text` section of the final executable. The two measurements differ because optimizations after register allocation may impact the final code size and the size of the `.text` section includes alignment padding between functions. Unless stated otherwise, we report code size results in terms of final code size.

Unless stated otherwise, when reporting code size results we report a code size improvement relative to the default LLVM allocator:

$$\text{code size improvement} = \frac{size_{LLVM} - size}{size_{LLVM}}$$

Note that there is a maximum code size improvement of 100% corresponding in a complete elimination of all code. A -100% code size improvement corresponds to a doubling in code size. When reporting an average code size improvement of a suite of benchmarks we report the geometric mean of the improvements of the component benchmarks. The cumulative code size improvement of a suite of benchmarks is computed by treating the entire benchmark suite as a single benchmark. The sum of the sizes of the individual benchmarks is used to calculate the code size improvement.

Large improvements in code size are difficult to obtain. Improving code size by as little as 1% in a high quality compiler is considered a feat worthy of a full year of development [7]. It is worth keeping in mind that our code size improvements are due exclusively to improvements in one pass of the compiler, the register allocator. Unlike performance improvements, code size improvements are indicative of how well an allocator can optimize the entire code of the program, not just a few frequently executed regions. Also, since the global MCNF model can exactly represent the code size metric, code size results provide a relatively noise-free assessment of how the different global MCNF solution techniques discussed in this thesis compare.

4.2.2 Code Performance

Code performance is the execution time of an application on a representative input. As discussed in Section 3.5, it is not possible to incorporate an exact model of performance into the global MCNF model. Instead, a weighted execution costs metric with a load:store:move cost ratio of

8:8:1 is used. We consider both *exact execution frequencies* and *simple static execution frequencies*. Exact execution frequencies of basic blocks are determined using perfect profiling data, i.e., the input used to benchmark is the same as the input used to profile. Simple static execution frequencies use the loop depth of a basic block to calculate an estimation of its frequency:

$$\text{BB execution frequency} = 10^{loopdepth(BB)}$$

Except when stated otherwise, we report results using exact execution frequencies.

In addition to reporting performance, we report the change in total memory operations. The total number of memory operations is the sum of the dynamic count of load and store instructions. We measure the dynamic count of these instructions during execution using high-resolution performance counters [77]. We report the reduction in total memory operations relative to the default LLVM allocator:

$$\text{memory operations reduction} = \frac{memops_{LLVM} - memops}{memops_{LLVM}}$$

The memory operation reduction metric indicates how well an allocator is optimizing for the weighted execution costs metric.

We measure the execution time of benchmarks in cycles using high-resolution performance counters. The resulting execution times are more precise and more accurate than more conventional wall clock measurements. Benchmarks are run under the Ubuntu 9.0.4 Linux operating system. Virtual address space randomization is turned off in the kernel to improve the repeatability of the results. Performance results are reported as an improvement relative to the LLVM default allocator:

$$\text{performance improvement} = \frac{time_{LLVM} - time}{time}$$

Note that a 100% performance improvement means the benchmark was twice as fast, i.e., took half as long. A -50% performance improvement means the benchmark took twice as long. When evaluating performance we execute the benchmark 3 times and report the mean and the standard error (shown as error bars). When reporting a performance improvement of a suite of benchmarks we report the geometric mean of the improvements of the component benchmarks.

4.3 Instruction Set Architectures

We consider two CISC-like instruction set architectures, x86-32 and x86-64, and two RISC-like instruction set architectures, ARM and ARM Thumb. The Intel x86 architectures dominate the desktop, workstation, and server markets. In the embedded space, there is no predominant architecture, but ARM is the market leader and the CISC design principles of the x86 are commonplace.

4.3.1 x86-32

The venerable Intel x86 32-bit instruction set [65] has variable length instructions, supports direct access to memory in most instructions, and has a limited register set of 8 integer and 8 floating point registers. Two of the integer registers, `ebp` and `esp`, are reserved for use as the frame and stack pointers. Floating point registers are implemented as a register stack; however, in LLVM the register allocator allocates floating point values to a fictional 7-register flat register file, and a later pass uses this allocation to generate register stack operations. As a result, the result of register allocation does not map exactly to the final output for floating-point values.

4.3.2 x86-64

The Intel x86 64-bit instruction set [65] also has variable length instructions and support for memory operands, but has an extended register set of 16 integer and 16 floating point registers. Two of the integer registers, `rbp` and `rsp`, are reserved for use as the frame and stack pointer. Although the legacy 8-register floating point stack is still available, LLVM instead uses the 16-register flat floating-point `xmm` register file to allocate floating point values.

The x86-64 ISA is an extension of the x86-32 ISA that is implemented in large part through the use of prefix bytes that modify the meaning of x86-32 instructions. As a result, the same instruction can have different sizes depending upon what register modes are used within the instruction. These additional code size costs are approximated as register preferences within the global MCNF model as described in Section 3.6.

	Atom 330	Core 2 Quad
Cores	2	4
Clock Speed	1.6Ghz	2.4Ghz
L1 Cache	24KB/32KB	32KB/32KB
L2 Cache	2x512KB	2x4MB
Reorder Buffer Size	-	96
Reservation Station	-	32
Execution Width	2	4
FSB Speed	533Mhz	1066Mhz
Transistors	47 million	582 million

Table 4.2: Characteristics of microarchitectures used to evaluate performance.

4.3.3 ARM

The ARM instruction set [117] has four-byte RISC-like instructions and 15 general purpose integer registers that are uniformly accessed. Two registers are reserved for use as a stack pointer and a link register. We target an ARMv6 core with software floating point.

4.3.4 Thumb

The Thumb instruction set [117] is an alternative instruction set for ARM processors optimized for code size. It has two-byte RISC-like instructions and can only efficiently access 8 integer registers. The stack and link registers are distinct from these 8 general purpose registers and are accessed implicitly. For our investigation we restrict the allocator to only allocate to these 8 efficiently accessed registers so that our Thumb target is representative of architectures with limited register sets. We target an ARMv6 core with software floating point and do not generate Thumb-2 instructions.

4.4 Microarchitectures

When evaluating performance we utilize two x86-based microarchitectures: an Intel Atom 330 and an Intel "Kentsfield" Core 2 Quad (Q6600). The differences between these microarchitectures are shown in Table 4.2. The Core 2 Quad is a modern out-of-order superscalar architecture with large caches, plentiful memory bandwidth, and ample resources to support parallelism. The Atom 330 is a modern low-power nettop processor. Unlike the Core 2, the Atom is an in-order processor and does not speculatively execute instructions. As a result, we expect that reducing the amount of memory accesses will have a larger impact on performance for the Atom than the Core 2 processor. We evaluate both architectures using the same x86-32 ISA in order to illuminate the effect of the interaction between the microarchitecture and the quality of register allocation on performance. Resource constraints prevented us from evaluating the x86-64 ISA on the Atom processor. Also because of resource constraints, we do not evaluate performance on the ARM or Thumb architectures.

Chapter 5

Heuristic Register Allocation

In this chapter we consider heuristic solution techniques for solving the NP-complete global MCNF model of register allocation described in Chapter 3. The heuristic solvers described in this chapter are used both to find an initial solution and as a component of the progressive solver described in Chapter 6.

We describe two heuristic solvers for the global MCNF problem that are capable of quickly finding an allocation competitive with existing allocators. The *iterative heuristic allocator* greedily builds up a solution by iterating over all the variables and adding the best possible path for each variable to the solution. Each path corresponds to a program-point specific allocation of a variable. This allocator behaves similarly to a single-pass graph coloring allocator. In contrast, the *simultaneous heuristic allocator* traverses the network a layer at a time simultaneously maintaining an allocation of all live variables. This allocator behaves similarly to a second-chance binpacking linear-scan allocator.

5.1 Iterative Heuristic Allocator

The iterative heuristic allocator allocates variables one at a time using shortest path computations. For each variable, a flow through the global MCNF model is found that corresponds to a complete allocation for that variable. As proven in Section 3.7, the minimum optimal path problem is NP-

```
1: procedure ConstructFeasibleSolutionIterative(GlobalMCNF f)
2:     for v ∈ allocSort(variables(f)) do                    ▷ iterate over variables
3:         for BB ∈ blockOrder(basicBlocks(f)) do
4:             for source ∈ definingNodes(BB, v) do   ▷ entry nodes and source nodes
5:                 path ← shortestFeasiblePath(source, v, BB)
6:                 markFeasiblePath(path, v)
```

Listing 5.1: CONSTRUCTFEASIBLESOLUTIONITERATIVE *Greedily construct a feasible solution for the global MCNF problem, f, one variable at a time.*

```
1: procedure markFeasiblePath(path path, variable v)
2:     for n ∈ path do
3:         n.flow++
4:         n.flowVars.insert(v)
5:         if type(n) = Entry ∨ type(n) = Exit then
6:             setBoundaryConstraints(n, v)
```

Listing 5.2: MARKFEASIBLEPATH *Given the shortest $path$ for variable v, reserve the flow along the path. Constrains the entry and exit nodes of connected blocks as necessary.*

complete in the global MCNF model. Due to the complexity of the problem, a sub-optimal algorithm is used to construct the flow for each variable.

5.1.1 Algorithm

The high-level pseudocode for the iterative heuristic allocator is shown in Listing 5.1. Variables are allocated in an order determined by the heuristic *allocSort* function in line 2. To allocate a variable, basic blocks are traversed in some heuristic order (line 3), and a shortest path computation is performed starting at every defining node of the block (line 5). Nodes that define a variable are either source nodes or, if a variable is live into the block, an entry node. The *shortestFeasiblePath* function finds and returns the shortest cost feasible path for the variable through the local network of the block. A feasible path respects the flows of already allocated variables, obeys the boundary constraints,[1] and does not prevent the allocation of unallocated variables. Given a path, the procedure *markFeasiblePath* records the found path as allocated.

[1] If the entry and exit constraints of a block must match, e.g., if the block is a loop, and a variable is live into and out of the block, then the path must have matching allocations at the entry and exit of the block. If this does not hold for the shortest path, an alternative legal path is chosen. This detail is not shown in the pseudocode.

```
1:  function SHORTESTFEASIBLEPATH(node source, variable v, BasicBlock BB)
2:      for n ∈ nodes(BB) do
3:          n.cost ← ∞
4:          n.prev ← ∅
5:      last.cost ← ∞
6:      Q.push(n)
7:      while ¬Q.empty() do
8:          n = Q.pop()
9:          for d ∈ successors(n) do
10:             cost ← n.cost + feasibleEdgeCost(n, d, v)
11:             if cost < d.cost ∨ (cost = d.cost ∧ tieBreak(n, d)) then
12:                 d.prev ← n
13:                 d.cost ← cost
14:                 if type(d) = Exit ∨ type(d) = Sink then
15:                     if d.cost < last.cost ∨ (d.cost = last.cost ∧ tieBreakLast(d, last)) then
16:                         last ← d
17:                 else if ¬Q.contains(d) then
18:                     Q.push(d)
19:     return last
```

Listing 5.3: SHORTESTFEASIBLEPATH *Find the shortest feasible path for variable* v *from node* $source$ *in basic block* BB *and return the last node of the path.*

As it traverses the shortest path, the procedure *markFeasiblePath*, shown in Listing 5.2, processes each node by: incrementing the amount of flow through the node (line 3), recording that the variable v is flowing through the node (line 4), and, if the node is an exit or entry node, constraining the connected exit and entry nodes to respect the allocation (line 6). The procedure *setBoundaryConstraints* fixes the boundary allocation of the variable v in connected, still-to-be-allocated blocks and also prevents infeasible boundary allocations of still-to-be-allocated variables. The subtleties of the implementation of $setBoundaryConstraints$, which is also used by the simultaneous heuristic allocator, are discussed in Section 5.3.

Once a path is marked in a block, the corresponding allocation of the variable is fixed and does not change. Therefore, it is essential that the function *shortestFeasiblePath*, shown in Listing 5.3, is always able to find a path. This function is a standard shortest path algorithm [6] with two modifications. First, due to the layered and acyclic structure of the network, a simple queue, as opposed to a priority queue, can be used to store the nodes on the frontier of the search. Second, the costs of edges are computed using the function $feasibleEdgeCost$ of Listing 5.4.

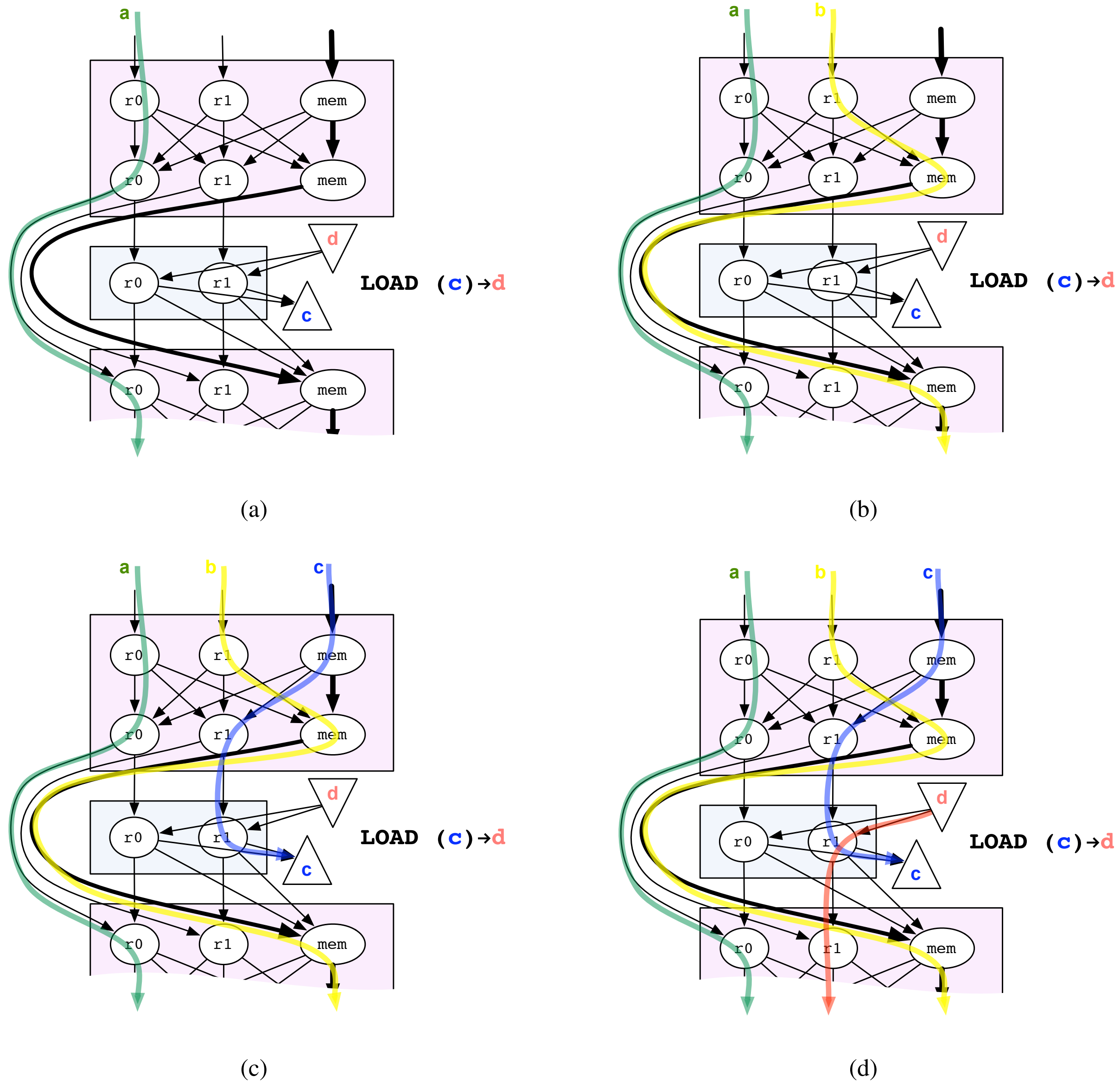


Figure 5.1: An example of the behavior of the iterative heuristic allocator. Antinonly *allocation class nodes are omitted for clarity. The operands of the LOAD instruction, c and d, must be allocated to a register.*

This function will return a cost of ∞ if an edge is blocked by a prior allocation (lines 2–9) or if allocating the variable v to the edge prevents the allocation of yet-to-be-allocated variables (lines 10–28).

The *feasibleEdgeCost* function must guarantee that the unallocated operands of an instruction remain allocable. If an edge enters (line 10), exits (line 11), or crosses an instruction–from the

```
1: function FEASIBLEEDGECOST(node s, node d, variable v)
2:     if (s.flow = s.capacity) ∨ (d.flow = d.capacity) then
3:         return ∞                                              ▷ already allocated
4:     if (anti(v) ∈ s.flowVars) ∨ (anti(v) ∈ d.flowVars) then
5:         return ∞        ▷ a variable and its anti-variable cannot flow through the same node
6:     if (type(s) = Entry) ∧ violatesBoundaryConstraint(s, v) then
7:         return ∞      ▷ boundary constraints require v be allocated elsewhere at block entry
8:     if (type(d) = Exit) ∧ violatesBoundaryConstraint(d, v) then
9:         return ∞       ▷ boundary constraints require v be allocated elsewhere at block exit
    ▷ a variable flowing into, out of or around an instruction, can not block future allocations
10:    if (type(d) = Insn) ∨
11:                        (type(s) = Insn) ∨
12:                                       (isBottomXBar(s) ∧ isTopXBar(d)) then
13:        ac ← allocClass(s)                      ▷ assert: allocClass(s) = allocClass(d)
14:        i ← crossedInsn(s, d)
15:        for n ∈ instructionGroupNodes(i) do
16:            n_in ← xbarPredecessor(n)
17:            n_out ← xbarSuccessor(n)
18:            availIn[n.allocClass] ← n_in.capacity − n_in.flow
19:            availOut[n.allocClass] ← n_out.capacity − n_out.flow
20:        if v ∈ liveIn(i) then availIn[ac]−−
21:        if v ∈ liveOut(i) then availOut[ac]−−
22:        for u ∈ (unallocedVarsofInsn(i) − {v}) do
23:            if hasAvailableClass(u, i, availIn, availOut) then
24:                ac = chooseAvailableClass(u, i, availIn, availOut)
25:                if u ∈ liveIn(i) then availIn[ac]−−
26:                if u ∈ liveOut(i) then availOut[ac]−−
27:            else
28:                return ∞                        ▷ potentially inhibits future allocations
29:    return edgeCost(s, d)
```

Listing 5.4: FEASIBLEEDGECOST *Return the cost of variable v traversing the edge from node s to node d. Returns ∞ if allocating v to this edge would conflict with already allocated variables or would prevent the allocation of currently unallocated variables.*

bottom of the crossbar before the instruction group to the top of the crossbar after the instruction group–(line 12) allocating the flow of variable v to the edge may potentially render the operands of the instruction unallocable. For instance, if an unallocated operand of an instruction must be allocated to a register and an edge crossing the instruction represents the last unallocated register at that point, then no other variable should be allowed to use that edge. This is exactly the case

shown in Figure 5.1 where the variable a has been allocated to `r0` around the LOAD instruction in Figure 5.1(a). When the shortest path for the variable b is computed, the *shortestFeasiblePath* function will call *feasibleEdgeCost* on the edge that spans the LOAD instruction and connects the `r1` crossbar nodes. This edge does not conflict with existing allocations (lines 2–9) and does interfere with an instruction (line 10). The function then considers the impact of allocating b to `r1` on the allocability of the LOAD instruction.

The impact of allocating a class to a variable is assessed in lines 10–28 of *feasibleEdgeCost* (Listing 5.4). First the availability of all allocation classes potentially used by the instruction is computed. An allocation class is available if the amount of already allocated flow is less than the capacity of the class. Availability is computed both immediately prior to the instruction (line 18) and immediately after the instruction (line 19). The variable a fully occupies the allocation class `r0` both before and after the LOAD instruction so $availIn[\texttt{r0}] = availOut[\texttt{r0}] = 0$. In contrast, there is no flow allocated to the `r1` crossbar nodes and the capacity of the `r1` class is one, so $availIn[\texttt{r1}] = availOut[\texttt{r1}] = 1$. Next, the availability is modified to represent the impact of allocating b to `r1` (lines 20–21) resulting in $availIn[\texttt{r1}] = availOut[\texttt{r1}] = 0$. An attempt is then made to allocate the unallocated operands of the instruction, c and d. However, since no register class is available (line 23), ∞ is returned (line 28) indicating that b cannot flow across this edge in a feasible solution. Instead, as shown in Figure 5.1(b), the variable b is allocated to memory across the LOAD instruction.

The importance of distinguishing between the availability of allocation classes both before and after an instruction is demonstrated in the allocation of c and d, shown in Figures 5.1(c) and 5.1(d). During the shortest path computation for variable c, the edge into the `r1` instruction group node is considered by *feasibleEdgeCost*. Since c is not live out of the instruction, only the incoming availability of `r1` is reduced (line 20). It is then still possible to choose the `r1` allocation class for d (line 24) since d, which is not live into the instruction, only needs its allocation class to be available out of the instruction. Note that after `r1` is chosen for d the availability of `r1` is decremented in line 26 preventing any other unallocated operands of the instruction from being allocated to the same class.

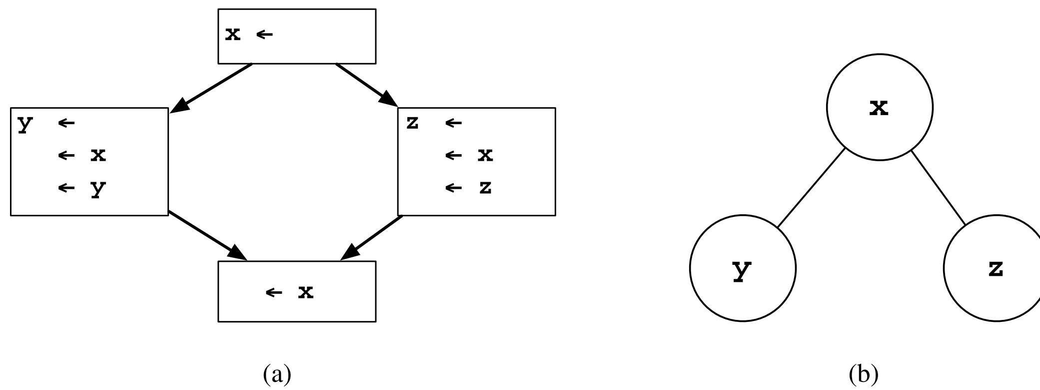


Figure 5.2: A (a) simple example of global variable usage and (b) its interference graph.

5.1.2 Improvements

The iterative heuristic allocator performs no back-tracking. Once a variable is allocated, the allocation is fixed. During the allocation of a variable, once an allocation within a basic block is selected, that local allocation is fixed and determines the initial allocation of all successor blocks. The lack of back-tracking means the choices made by the allocator can have a significant effect on the quality of allocation. In particular, we consider the *allocation order* of the variables, the *block order* used in the traversal of the control flow graph, and the *tie breaking* method for choosing between equal cost shortest paths. We discuss how each effects the resulting allocation and describe the approach taken in our implementation.

ALLOCATION ORDER

The order in which variables are allocated is determined by the $allocSort$ function on line 2 of Listing 5.1. The earlier a variable is allocated, the more likely it is to receive a favorable allocation. Variables that are allocated later must work around the allocations of already allocated variables. For example, consider Figure 5.2 and an architecture with only two registers, `r0` and `r1`. If y is allocated to `r0` and z is allocated to `r1`, then x will be spilled to memory to work around these allocations.

Both the iterative heuristic allocator and a single-pass "top-down" graph coloring allocator (Section 2.1.1) greedily allocate variables. Just as with a single-pass graph coloring allocator, our iterative allocator uses a priority function that calculates a score for each variable. Variables

with larger scores are allocated first. The priority function provides larger scores for frequently accessed variables, but also reduces the score for variables with long live ranges. A variable with a long live range will conflict with many other variables; allocating such a variable early will potentially adversely effect the allocations of all the conflicting variables. The priority function we use for regular variables is:

$$priority_v = \frac{\sum\limits_{u \in uses(v)} 10^{loopdepth(u)} + \sum\limits_{d \in defs(v)} 10^{loopdepth(d)}}{\#\textit{instructions spanned by live ranges of v}}$$

This is the same priority function used within the default register allocator of the LLVM compiler infrastructure [87]. Anti-variables are always allocated after the corresponding regular variable so that the regular variable will have the full use of the memory network when performing its shortest path computations.

BLOCK ORDER

When allocating a variable, the iterative allocator traverses the control flow graph and, for each block, computes a shortest feasible path for the variable. The order in which blocks are processed is determined by the $blockOrder$ function in line 3 of Listing5.1. We consider to orderings: *depth first order* and *loop ordering*. When blocks are processed in depth first order, the connectedness of the ordering means that if a variable is live into a block, the entry condition will always be set. In loop ordering, blocks with innermost loops are processed first. Blocks at the same loop depth in the same loop are processed in depth first order. Since loop blocks are processed before the blocks connecting the loops, a variable may be allocated differently in different loops and data movement instructions may have to be inserted in between the blocks. If optimizing performance, a loop ordering will process the most important regions of the control flow graph first.

The effect of these two block ordering strategies on code size is shown in Figure 5.3(a). Somewhat surprisingly, the results are mixed. The disconnectedness of the loop ordering only results in worse average code size for the x86-64 architecture and results in slight improvements for the other three architectures. Since the degradation for x86-64 is significantly greater than

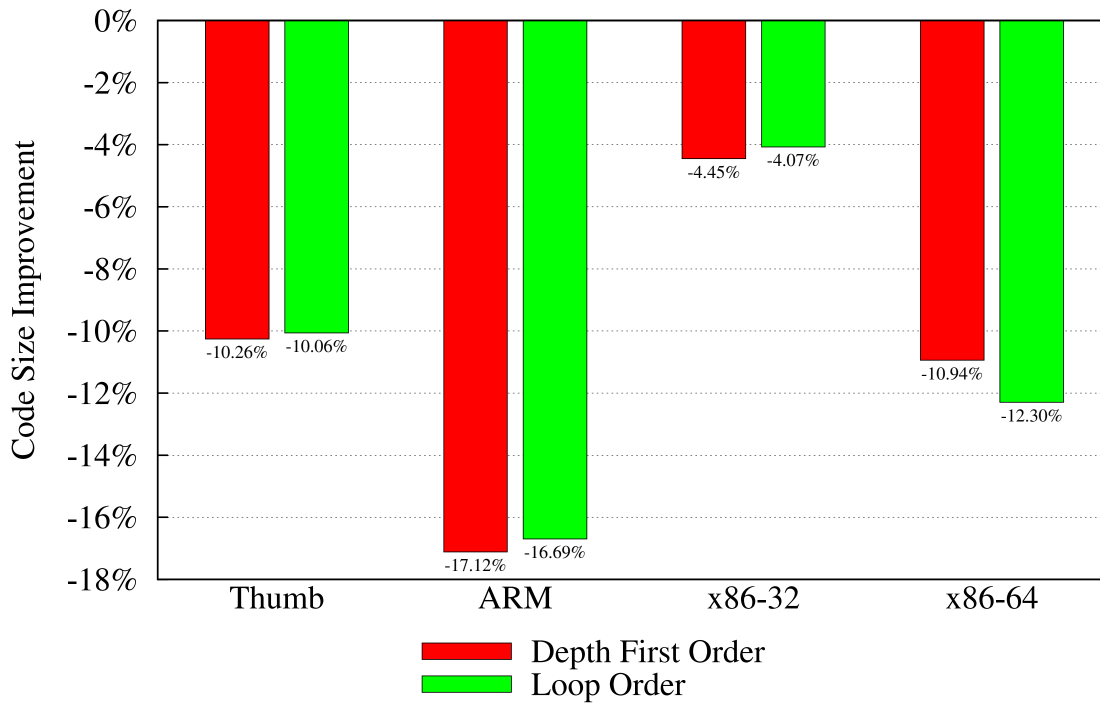
the improvements exhibited by the other architectures, when optimizing for code size, we use the depth first block ordering within the default implementation of our iterative allocator.

We consider the effect of the two block ordering strategies when optimizing for performance in Figure 5.3(b). We evaluate the reduction in memory operations executed at run-time as a measurable proxy for the weighted execution metric we explicitly optimize for. As expected, the loop ordering strategy significantly outperforms the depth first ordering strategy. We use the loop block ordering with the default implementation of the iterative allocator when optimizing for performance.

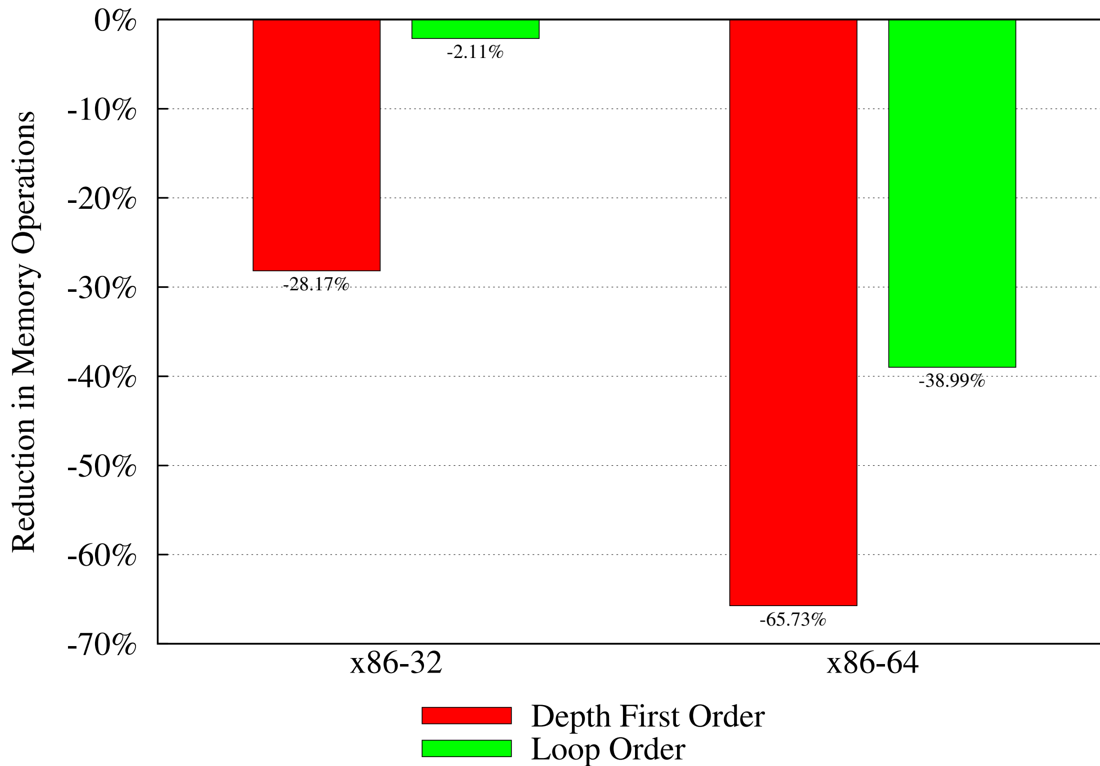
TIE BREAKING

When there is more than one shortest path to a node, the $tieBreak$ function in line 11 of Listing 5.3 chooses which path is used. The $tieBreakLast$ function in line 15 chooses between equal cost shortest paths at the end of the block. There are frequently several possible paths from which to choose. For example, in most cases, variables do not have strong preferences for particular registers and there is a minimum cost shortest path coinciding with each available register. In this case, the tie breaking functions determine what register is assigned to the variable.

We use tie breaking functions to inject a global view of variable interactions into the shortest path decision. For instance, consider Figure 5.2 and an architecture with only two registers, `r0` and `r1`. Imagine that variable y has been allocated to `r0` and the iterative allocator is processing the first block with the definition of x. Since y is not live anywhere within this block to interfere with the flow of x, there are three shortest paths, each with value zero at the end of the block: a path ending in `r0`, a path ending in `r1`, and a path ending in a memory node (recall that regular variables transition to memory for zero cost; the store cost is paid by the anti-variable). A simple rule in the tie breaking function prefers register classes to memory classes, since register classes will not incur an anti-variable cost. In order to break the tie between the register classes, global information about the current allocation state and the interference graph, shown in Figure 5.2(b), are used. When a variable, such as y, is allocated, we record what allocation classes are used. When the tie breaking function chooses between two allocation classes, it seeks to minimize the number of conflicts with already allocated variables. For example, if y is allocated to `r0`, then

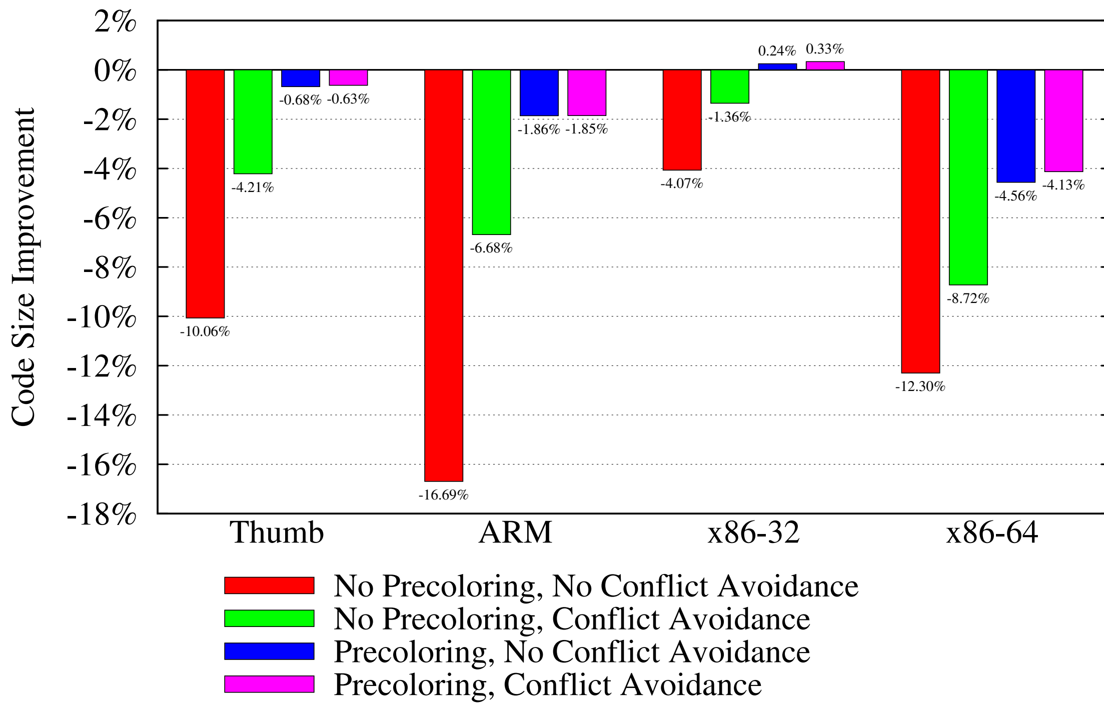


(a)

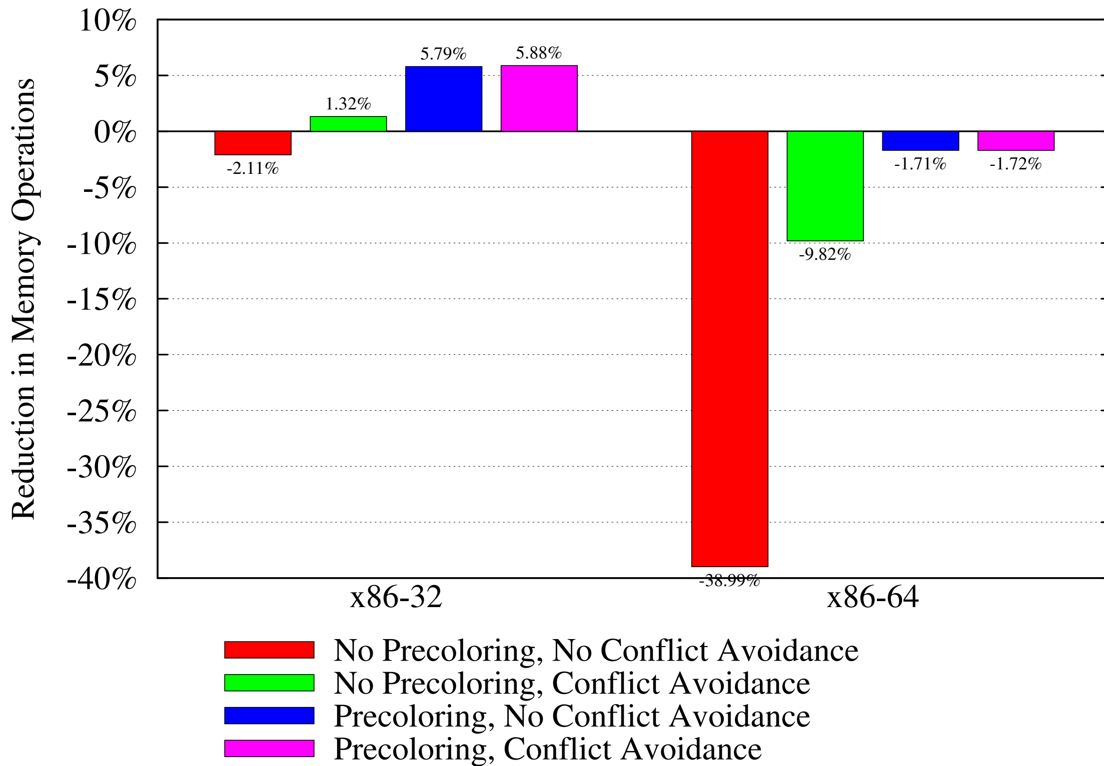


(b)

Figure 5.3: The effect on code quality of two different block orderings in the iterative allocator.



(a)



(b)

Figure 5.4: The effect on code quality of tie breaking strategies in the iterative allocator.

there would be one conflict with choosing the `r0` class for x and zero conflicts with choosing the `r1` class. This is only an approximation of potential conflicts since variables may be allocated to different allocation classes at different locations.

Using the interference graph and current allocation information is effective at avoiding unnecessary conflicts with already allocated variables, but does not bias the tie breaking choices against allocation decisions that adversely effect currently unallocated variables. We address this issue to some extent by pre-coloring the interference graph using a single pass of the "bottom-up" graph simplification coloring algorithm (Section 2.1.1). The pre-coloring provides a partial allocation (or a full allocation if the graph is colorable). If a variable is successfully pre-colored, the tie breaking functions will prefer allocating the variable to the corresponding allocation class.

Effective tie breaking heuristics with a global view of the allocation are essential in order for the iterative allocator to produce quality code. The effects of the two heuristics, conflict avoidance and pre-coloring, are shown in Figure 5.4. Without these tie breaking strategies, the iterative allocator produces code that is substantially larger than the default LLVM allocator for all architectures (Figure 5.4(a)). Conflict avoidance substantially improves the code size but pre-coloring has an even more dramatic impact. The same trends are apparent when optimizing for performance (Figure 5.4(b)). The improvements due to pre-coloring appear to essentially subsume any improvements derived from conflict avoidance. The code quality improvements achieved when combining the two tie breaking strategies are nearly identical to that of using only pre-coloring.

We use the pre-coloring tie breaking heuristic within the default implementation of the iterative allocator. With the inclusion of this tie breaking strategy, the iterative allocator is very similar to a single-pass graph coloring allocator where the assignment heuristic uses the results of a graph simplification coloring algorithm. However, the iterative allocator is more flexible. In a single-pass graph coloring allocator, a variable is either allocated to a single register or is spilled everywhere and a scratch register must be reserved to load and store spilled values. In contrast, the iterative allocator only spills as needed and supports allocating a variable to multiple register classes. The iterative allocator also benefits from the expressive global MCNF model. Allocation class preferences and costs are modeled in the flow costs.

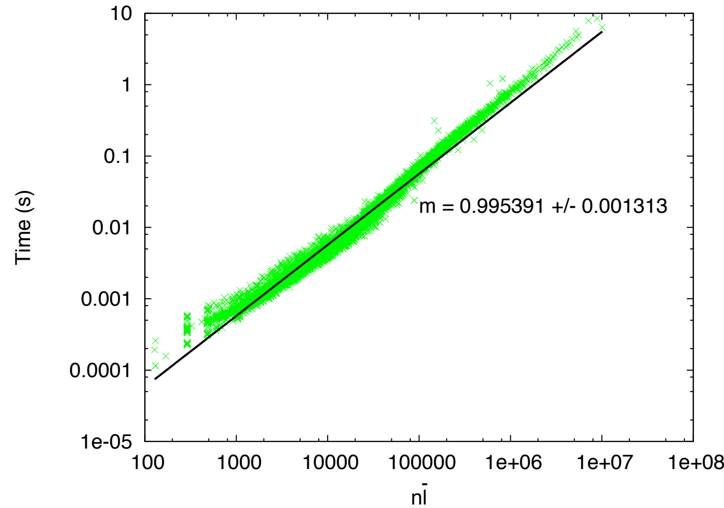


Figure 5.5: Running time of iterative allocator for all benchmarked functions related to $n\bar{l}$ on a log-log scale.

5.1.3 Asymptotic Analysis

Let n be the number of instructions and v the number of variables in a program. The number of allocation classes, c, is a constant determined by the target architecture. We assume that $v < n$ since typically at most one variable is defined per instruction. The number of nodes in the global MCNF model is $O(nc)$ since there are a fixed number of layers of c nodes for each instruction. The number of edges is $O(nc^2)$ since each node connects with at most $O(c)$ other nodes.

The iterative heuristic allocator performs a shortest path computation for every variable v in every block. This shortest path computation is linear in the number of nodes and edges because the network of a block is a directed acyclic graph. Since c is a constant, the worst-case asymptotic running time of the iterative heuristic allocator is $O(nv)$. However, the cost of computing the shortest path is not $O(n)$ for every variable. It is equal to the length of the path. That is, it is proportional to the number of instructions where the variable is live. Therefore, a tighter bound on the worst case complexity of the iterative allocator is $O(n\bar{l})$ where $\bar{l}$ is the average number of variables live at any point.

The empirical running times of all benchmarked functions are shown in a log-log plot relative to to $n\bar{l}$ in Figure 5.5. The slope of a line in a log-log plot equals the exponent k of a function

```
1:  variableAlloc: variable → allocClass                ▷ map from variable to allocation class
2:  classAlloc: allocClass → set of variables           ▷ map from allocation class to variables
3:  procedure CONSTRUCTFEASIBLESOLUTIONITERATIVE(GlobalMCNF f)
4:      for BB ∈ blockOrder(basicBlocks(f)) do
5:          computeAllShortestPaths(BB)                 ▷ respects boundary constraints
6:          insnLayer ← ∅          ▷ a layer is an array of nodes indexed by allocation class
7:          currLayer ← entryNodes(BB)                  ▷ get entry nodes of block
8:          for v ∈ liveIn(BB) do
9:              allocateVarAtLayer(v, insnLayer, currLayer)
10:         while ¬isExitLayer(currLayer) do
    ▷currLayer is a fully allocated crossbar layer
    ▷nextLayer is the next crossbar layer and needs to be allocated
    ▷insnLayer is either ∅ or is the currently unallocated instruction group between these layers
11:             (insnLayer, nextLayer) ← getNextLayer(currLayer)
12:             propagateAllocs(currLayer, insnLayer, nextLayer)
13:             for v ∈ definedVars(insnLayer) do
14:                 allocateVarAtLayer(v, insnLayer, nextLayer)
15:             currLayer ← nextLayer
16:         for v ∈ liveOut(BB) do
17:             setBoundaryConstraints(currLayer[variableAlloc[v]], v)
```

Listing 5.5: CONSTRUCTFEASIBLESOLUTIONSIMULTANEOUS *Construct a feasible solution for the global MCNF problem f layer by layer.*

x^k. The tight clustering of running times around a best-fit line with a slope of .995 empirically confirms the $O(n\bar{l})$ asymptotic complexity of the iterative allocator.

5.2 Simultaneous Heuristic Allocator

As an alternative to the iterative allocator, we describe a simultaneous allocator which functions similarly to a second-chance binpacking linear scan allocator (Section 2.1.3) but uses the explicit prices of the global MCNF model to guide allocation decisions.

5.2.1 Algorithm

The high-level pseudocode for the simultaneous allocator is shown in Listing 5.5. The algorithm traverses the control flow graph in some heuristic order (line 4) and allocates a block at a time. In

each block, the allocator traverses the MCNF model a layer at a time, maintaining an allocation of the live variables, evicting and modifying variable flows as necessary.

The $computeAllShortestPaths$ function, called in line 5 of Listing 5.5, computes the shortest path through the local MCNF model for every variable in the block. Shortest paths originate either at a source node or at an entry node and terminate either at a sink node or at an exit node. If a variable is live into a block, it may be constrained by the allocation of previous blocks to use a specific entry node. Each node records the value and direction of the shortest paths to and from the node for each variable. The sum of the costs of the incoming and outgoing shortest paths of a variable at a node is a lower bound on the cost of allocating a variable to the corresponding allocation class at the program point represented by the node.

After computing the shortest paths, the algorithm iterates over the layers of the MCNF network, updating the flows of variables, and maintaining the current allocation of every live variable in two maps: $variableAlloc$ (line 1) and $classAlloc$ (line 2). $variableAlloc$ stores the current allocation class for each variable. $classAlloc$ stores the set of variables that are allocated to a specific allocation class at the current point. The allocation maps are initialized by allocating all the variables live into the block using the function $allocateVarAtLayer$ in line 9 of Listing 5.5.

The function $allocateVarAtLayer$, shown in Listing 5.6, allocates variables that are being defined into the network at the program points represented by the passed layers. At the entry of the block, $nextLayer$ corresponds to the entry group layer and $insnLayer$ is empty. The function $assignVarToNode$, shown in Listing 5.7, performs the actual allocation of flow (lines 2–3) and updates the allocation maps (lines 4–5).

An example of the execution of the simultaneous allocator on a basic block is shown in Figure 5.6. The allocator state as it enters the main loop (line 10 of Listing 5.5) is shown in Figure 5.6(a). The two variables live into the block, a and b, and their anti-variables, a' and b', have been allocated to the entry nodes of the block based on existing boundary constraints. The results of the shortest path computation are shown as the thin lines.

The function $propagateAllocs$, shown in Listing 5.8, is called as the allocator iterates over the layers of the network to propagate the current allocations from one layer to the next using shortest path information (line 6 of Listing 5.8). Instruction group nodes are allocated at the

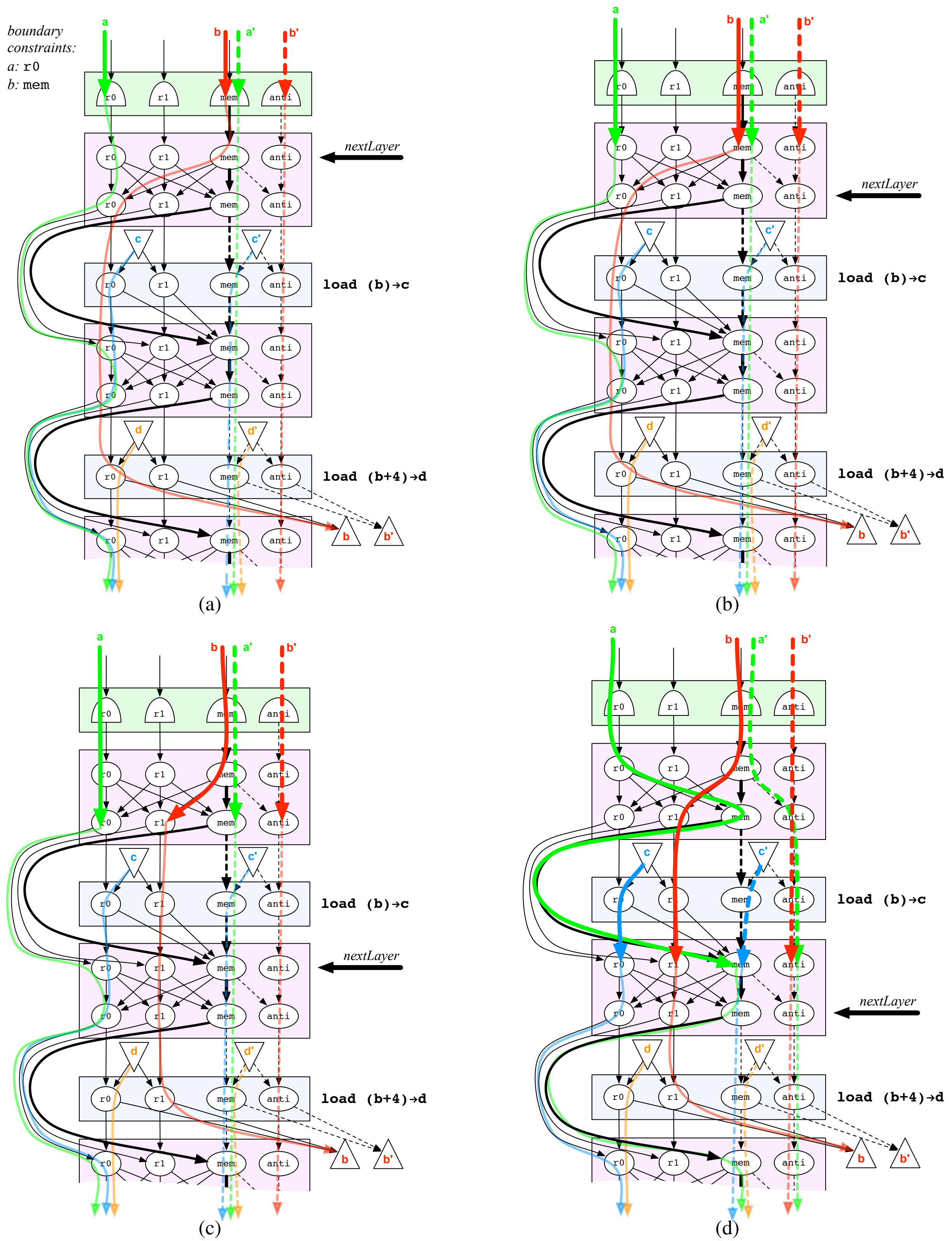


Figure 5.6: An example of the execution of the simultaneous heuristic allocator.

```
1: procedure ALLOCATEVARATLAYER(variable v, nodes insnLayer, nodes nextLayer)
2:     (ac, evictInfo) ← selectAllocClass(v, insnLayer, nextLayer)
3:     if evictInfo ≠ ∅ then
4:         evictVar(evictInfo, nextLayer)
5:     if v ∈ insnLayer[ac].variables then                ▷ v used in instruction
6:         assignVarToNode(v, insnLayer[ac])
7:     assignVarToNode(v, nextLayer[ac])
```

Listing 5.6: ALLOCATEVARATLAYER *Given a variable* v *that is being defined, allocate* v *into the current layer. Other variables may already be allocated to* $insnLayer/nextLayer$*. Potentially evicts allocated variables into different allocation classes. Updates* $variableAlloc$ *and* $classAlloc$*.*

```
1: procedure ASSIGNVARTONODE(variable v, node n)
2:     n.flow++
3:     n.flowVars.insert(v)
4:     variableAlloc[v] ← node.allocClass
5:     classAlloc[node.allocClass].add(v)
```

Listing 5.7: ASSIGNVARTONODE *Adds a unit of flow for* v *to* $node$*. Updates* $variableAlloc$ *and* $classAlloc$ *to reflect the allocation of* v *to* c*.*

same time as the successor layer (lines 8–10) since only a subset of variables may be used by the instruction. In order to ensure that the allocation of a variable within an instruction is valid, it must be propagated to the next layer (the top of a crossbar) where all variables live out of the instruction are represented. If propagating an allocation along the shortest path results in a valid allocation, the variable is allocated to the next node in the shortest path (line 12–14); otherwise, it is saved as a problem variable (line 16). Problem variables are allocated using $allocateVarAtLayer$ after all the non-problematic variables have been allocated (line 18).

In the most common case, allocations of variables do not change, as is the case from Figure 5.6(a) to Figure 5.6(b). However, since allocations are propagated along shortest paths, it is possible for allocations to change. In Figure 5.6(b) the shortest path for variable b transitions from memory to `r0`. This is because b must be accessed as a register in the upcoming load instruction. However, the shortest path for a also flows into `r0`, resulting in a conflict. If the variable a is allocated first in $propagateAllocs$, then b will be considered problematic and allocated using $allocateVarAtLayer$. This function, shown in Listing 5.6, selects a new allocation class for b using $selectAllocClass$ (line 2), performs any necessary eviction to support this allocation (line 4), and then allocates the variable to the correct node (line 6).

```
1: procedure PROPAGATEALLOCS(nodes currLayer, nodes insnLayer, nodes nextLayer)
2:     classAlloc.clear()                                    ▷ will regenerate this map
3:     for v ∈ liveAtLayer(currLayer) do      ▷ propagate the allocation of every live variable
4:         currAC ← variableAlloc[v]
5:         node ← currLayer[currAC]
6:         next ← nextNodeInShortestPath(node, v)
7:         inode ← ∅
8:         if next.type = Insn then
9:             inode ← next
10:            next ← nextNodeInShortestPath(node, v) ▷ allocate through instruction group

11:        if nodeValidForAllocation(next, v) then                          ▷ allocate v
12:            if inode ≠ ∅ then
13:                assignVarToNode(v, inode)
14:            assignVarToNode(v, next)
15:        else                                  ▷ could not allocate along shortest path
16:            problems.add(v)
17:    for v ∈ problems do
18:        allocateVarAtLayer(v, insnLayer, nextLayer)              ▷ will evict if necessary
```

Listing 5.8: PROPAGATEALLOCS *Finds a flow for all live and currently allocated variables from* $currLayer$ *to* $insnLayer/nextLayer$. *Potentially evicts allocated variables into different allocation classes. Updates* $variableAlloc$ *and* $classAlloc$.

```
1: function NODEVALIDFORALLOCATION(node node, variable v)
2:     if node.flow = node.capacity then
3:         return false
4:     if anti(v) ∈ s.flowVars ∨ anti(v) ∈ d.flowVars then
5:         return false      ▷ a variable and its anti-variable cannot flow through the same node
6:     if nodeBlockedByInsn(node, v) then   ▷ check successor capacitated instruction nodes
7:         return false
8:     return true
```

Listing 5.9: NODEVALIDFORALLOCATION *Return true if* v *can be allocated to* $node$ *without requiring an eviction.*

The $nodeValidForAllocation$ function, shown in Listing 5.9, determines if it is safe to propagate a variable's allocation to a node, or if such an allocation would be problematic. This function checks to make sure a node isn't fully allocated (line 2–5). It also checks to make sure that the allocation class is available for use by the variable in the next instruction (line 6). This check is necessary to support capacitated nodes within instructions. For example, if an instruction can only access a single memory operand and a variable used by the instruction is already allocated to

```
 1: function SELECTALLOCCLASS(variable v, nodes insnLayer, nodes nextLayer)
 2:     for c ∈ allocClasses do            ▷ compute cost of allocating v to each allocation class
 3:         node ← nextLayer[c]
 4:         cost ← costOfAlloc(v, node) ▷ sum of current path to and shortest path from node
 5:         if classAlloc[c].isFull then                                    ▷ eviction required
 6:             minEvictcost ← ∞
 7:             for u ∈ classAlloc[c] do
 8:                 (evictCost, evictInfo) ← evictionCost(u, node)
 9:                 if evictCost < minEvictCost then
10:                     minEvictCost ← evictCost
11:                     evictInfos[c] ← evictInfo
12:             cost ← cost + minEvictCost
13:         costs[c] ← cost
14:     minc ← 0
15:     for c ∈ allocClasses do
16:         if (costs[c] < costs[minc]) ∨ (costs[c] = costs[minc] ∧ tieBreak(v, c, minc)) then
17:             minc ← c
        return (minc, evictInfos[minc])
```

Listing 5.10: SELECTALLOCCLASS *Find the best allocation class for variable v in $nextLayer$. Returns the allocation class and, if it is necessary to evict an already allocated variable, the eviction information.*

memory, no other variables used by the instruction can be allocated to memory; they are blocked by the current allocation.

The *selectAllocClass* function, shown in Listing 5.10, finds the best allocation class for a variable at a layer. It is called when a variable is defined into a layer and when, as with the variable b in our example, there is a problem propagating the allocation of a variable along its shortest path. Every allocation class is considered (line 2) and the cost of allocating the variable to that class is computed. This cost has two components: the independent cost of allocating the variable (line 4) to the allocation class and the cost of evicting any variable that is preventing this variable from using the allocation class (lines 5–11). The independent cost of allocating the variable is the sum of the current cost of allocation, any cost incurred when transitioning from the current allocation to the new allocation class, and the value of the shortest path out of the node. For example, in Figure 5.6(b), the variable b must incur the cost of a load to transition to either class `r0` or class `r1`, but can transition to the memory class for no cost. However, there is no shortest path for b from the memory node in the bottom of the first crossbar since b must

```
 1: function EVICTIONCOST(variable v, nodes layer)
 2:     minCost ← ∞
 3:     for c ∈ allocClasses do
 4:         node ← layer[c]
 5:         if nodeValidForAllocation(node, v) then
 6:             info ← shortestAvailablePathToMarkedNode(node, v)
 7:             cost ← info.cost + shortPathFromNodeCost(node, v)
 8:             if cost < minCost ∨ (cost = minCost ∧ tieBreak2(bestInfo, info)) then
 9:                 minCost ← cost
10:                 bestInfo ← info
```

Listing 5.11: EVICTIONCOST *Return the cost of evicting v from its current allocation in $layer$. Also provides the information necessary to perform the eviction.*

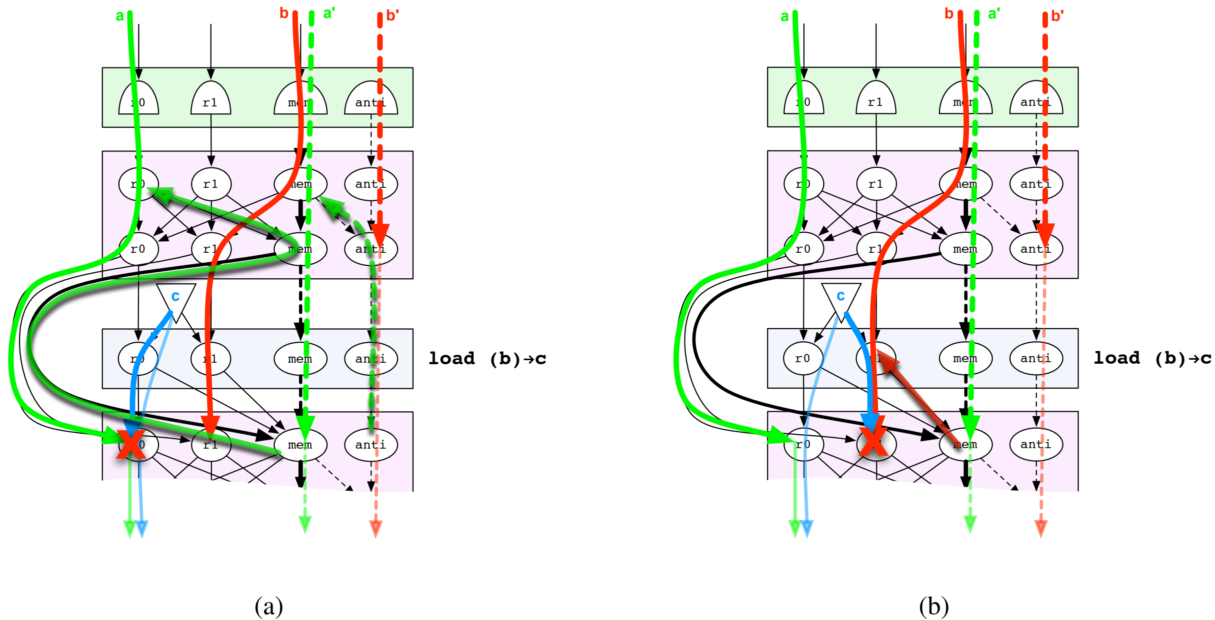


Figure 5.7: Example eviction decisions in the simultaneous heuristic allocator.

be accessed in a register in the LOAD instruction. Hence, the cost of the shortest path from the memory node is infinite. The cost of any eviction is added to the independent cost (line 12) and the lowest cost class is chosen. If multiple classes have the same cost, a $tieBreak$ heuristic function is used. In our example, the `r0` class is already fully allocated to the variable a and so an eviction is necessary in order to allocate b to `r0`. Since this would result in an additional cost, the `r1` class is selected for b, as shown in Figure 5.6(c).

The $evictionCost$ function, shown in Listing 5.11, determines the cost of evicting a variable from its current allocation. It also returns the information necessary to implement the eviction. Consider the example in Figure 5.6(c). The allocations of a and b are successfully propagated to remain in their current allocation classes, `r0` and `r1`, and then, at line 14 of Listing 5.5, the variable c, which is defined into the instruction group prior to $nextLayer$, is allocated. There are only two valid allocation classes for c, `r0` and `r1`, that are considered by the $selectAllocClass$ function. In order to allocate c to `r0`, the variable a must be evicted. To compute the cost of evicting a from `r0`, the $evictionCost$ function considers every possible allocation class (line 3) that a can be validly allocated to (line 5) given the current allocation state. Evictions are not recursive; the eviction of one variable will not result in the eviction of another unrelated variable. In this case, the only valid allocation class for a is the memory allocation class. The shortest cost path from this memory node to the currently allocated flow of a is then computed (line 6) as shown in Figure 5.7(a). Only nodes with available flow are used by the shortest path computation. This path corresponds to the rerouting of the flow of a needed to evict it from its current allocation in `r0` to an allocation in memory. If an anti-variable prevents the computation of this eviction path, as is this case in Figure 5.7(a), then a second eviction path for the anti-variable is simultaneously computed and its cost is incorporated into the cost of the eviction. In the example, the cost of evicting a is the cost of a store, which is paid by the eviction of the anti-variable.

When selecting an allocation class for a variable being defined, as with c in Figure 5.6(c), every possible allocation class is considered. In this case, in addition to `r0`, which requires the eviction of a, the class `r1` is considered as shown in Figure 5.7(b). The `r1` node is fully allocated to b, so the eviction cost of b is computed. The only class that b can be evicted to is the memory class. The shortest cost path through available nodes is the single edge shown in Figure 5.7(b). The anti-variable of b, b', is already in the *antionly* class and so no anti-variable eviction path is necessary. The found eviction path has zero cost. However, since the second load instruction of the example requires that b reside in a register, if it is allocated to memory at this point, the cost of a load would be incurred later. This is reflected in the cost of the shortest path from the memory node and so the total cost of the eviction will be equal to the cost of a load (line 7). When selecting the allocation class for c, both possible allocation classes, `r0` and `r1`, have an eviction

cost. In our example, the class chosen by $selectAllocClass$ for the variable c is `r0` resulting in the allocation shown in Figure 5.6(d).

The simultaneous allocator continues to traverse the MCNF model a layer at a time until the exit layer is reached. The boundary constraints for all the variables live out of the block are then set and allocation continues in another block.

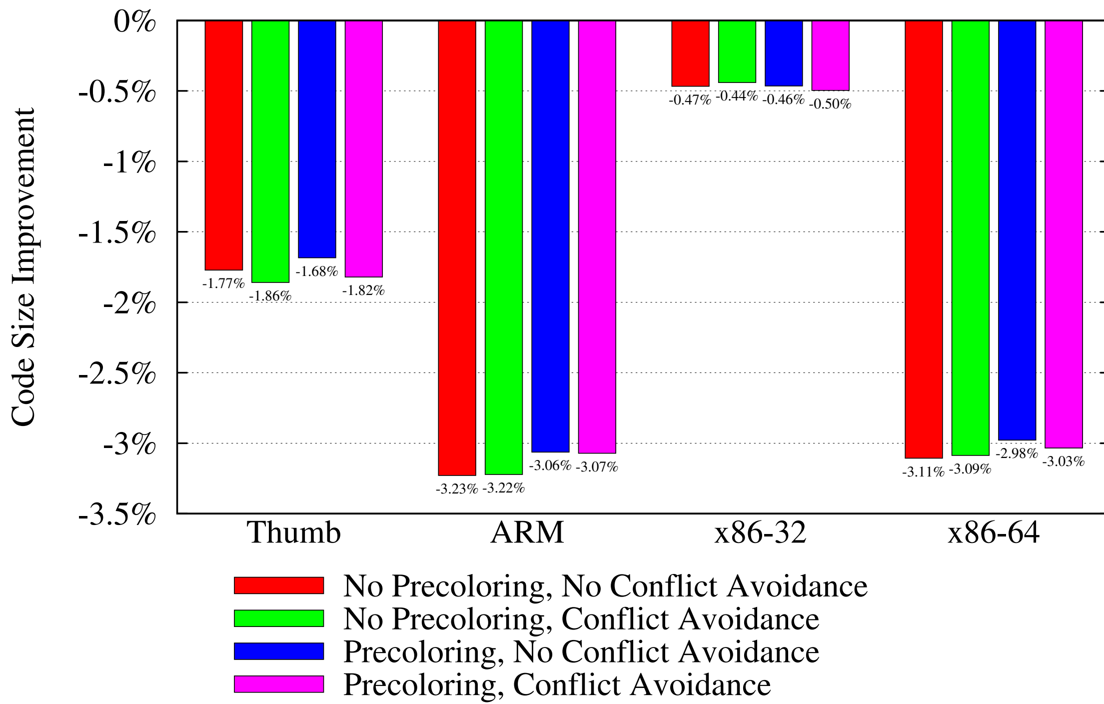
5.2.2 Improvements

Unlike the iterative allocator, the simultaneous allocator can undo allocation decisions through the eviction mechanism. This ability to back-track on allocation decisions is limited since evictions are constrained by the current allocation (variables are not recursively evicted) and allocations are fixed a basic block boundaries. We consider two improvements that seek to moderate these limitations: *tie breaking heuristics* and *trace formation*.

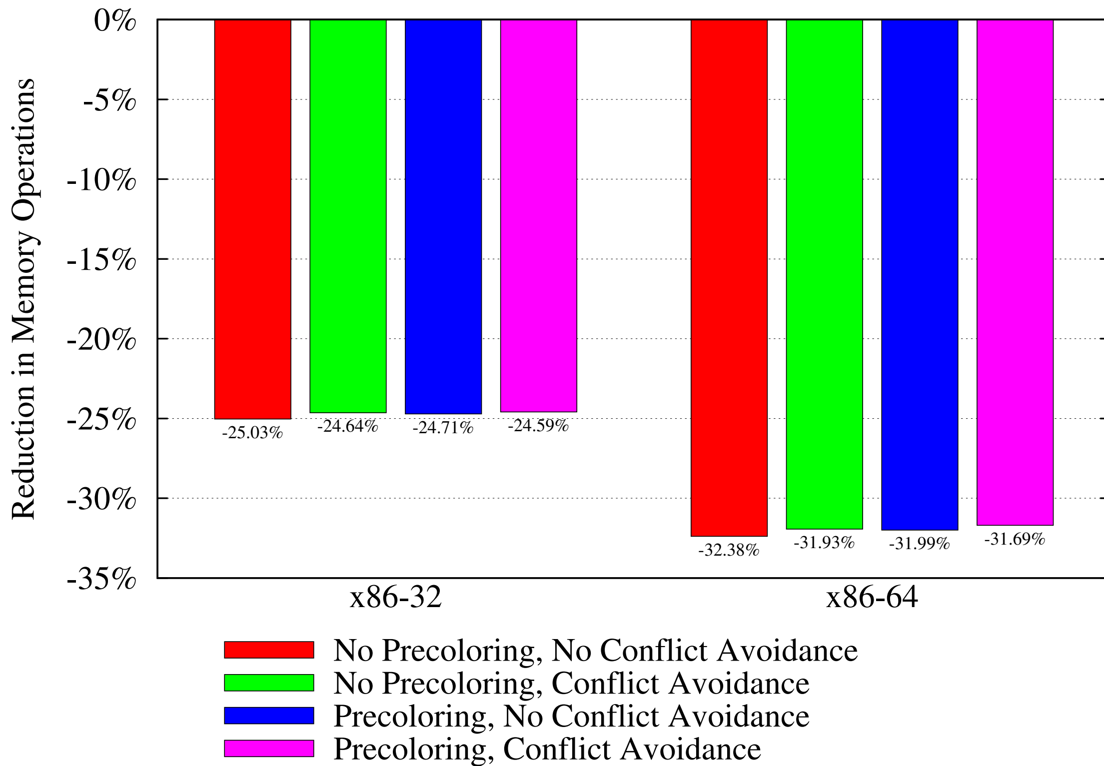
TIE BREAKING

The simultaneous allocator is guided by the explicit costs of the global MCNF model when making allocation decisions. Frequently, the cost of two potential allocations are equal. In this case some method for breaking the tie is necessary. Tie breaking is performed when selecting an allocation class for a variable (line 16 of Listing 5.10) and when choosing what allocation class to evict a variable to (line 8 of Listing 5.11). Just as with the iterative allocator, these heuristic decisions can be guided by pre-calculated global interference information. Allocation decisions that increase the number of conflicts in the interference graph are biased against, while allocations that match a pre-coloring of the interference graph are preferred in the tie breaking decisions.

Interestingly, the tie breaking strategies have very little impact on code quality, as shown in Figure 5.8. There is a minor improvement in code size (Figure 5.8(a)) when pre-coloring is used for most architecture. Similarly, when optimizing for performance, the use of pre-coloring results in a slight reduction in memory operations (Figure 5.8(b)). This demonstrates the flexibility of the simultaneous allocator. Allocation decisions can be revisited through the eviction mechanism, making the allocator less sensitive to the quality of the heuristics used to make the initial



(a)



(b)

Figure 5.8: Effect of tie breaking heuristics on code quality in the simultaneous allocator

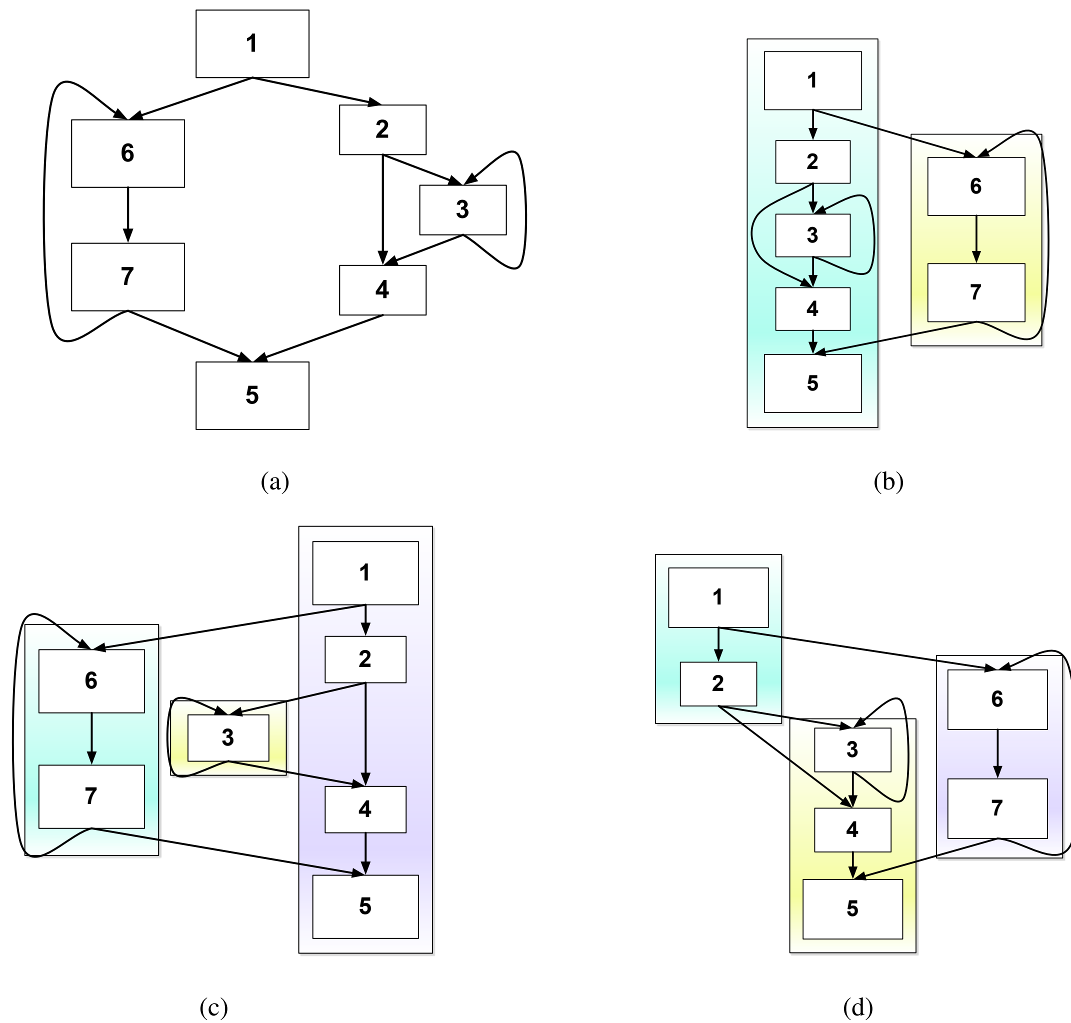
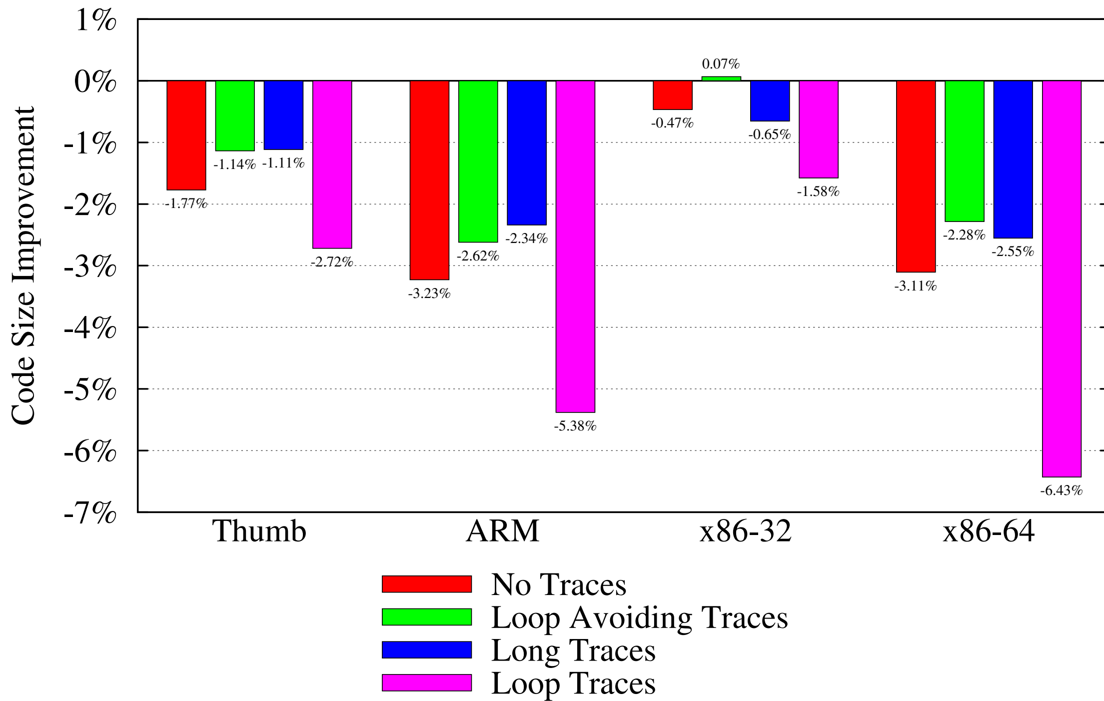


Figure 5.9: (a) A control flow graph and several possible trace decompositions: (b) a longest trace decomposition, (c) a loop trace decomposition, and (d) a loop avoiding trace decomposition.

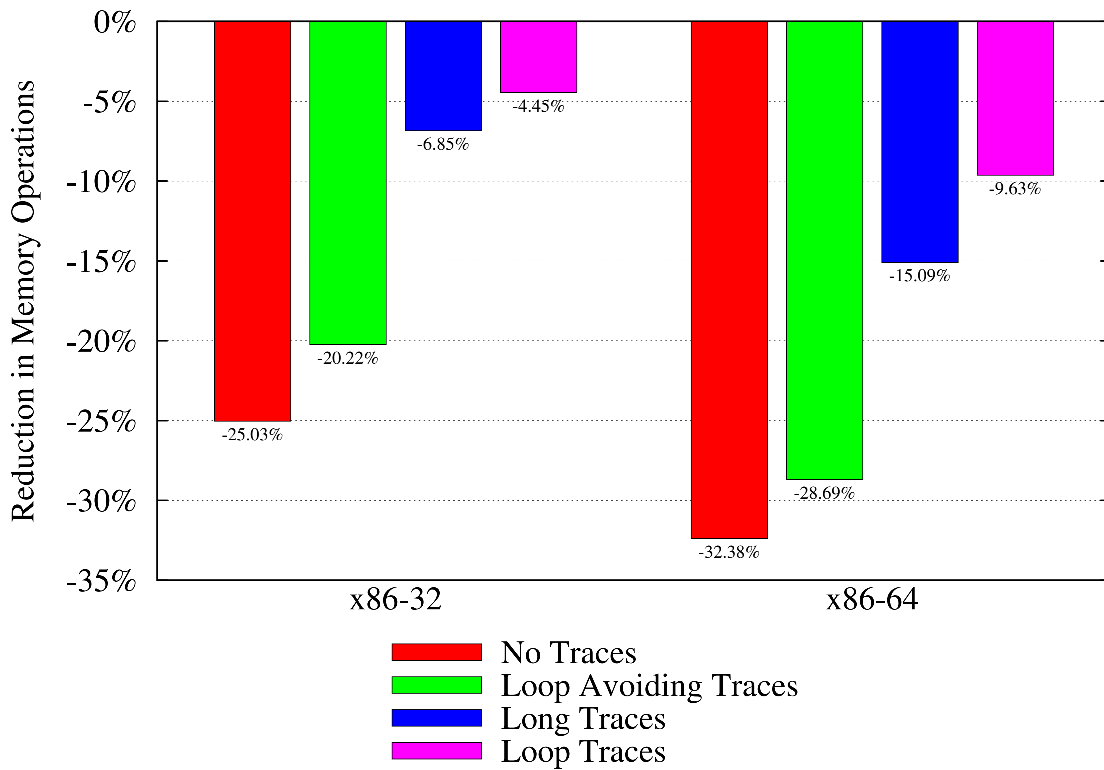
allocation decisions. Since the interference graph tie breaking strategies do not significantly influence code quality and introduce more overhead into the allocator, our default implementation of the simultaneous allocator does not use them.

TRACE FORMATION

The flexibility of the simultaneous allocator is limited by the fixing of allocation decisions at basic block boundaries. A more flexible approach is to group blocks into *traces*, sequences of connected basic blocks, and allocate each trace similarly to how a single block is allocated.



(a)



(b)

Figure 5.10: Effect of trace decomposition strategy on code quality in the simultaneous allocator

Shortest paths are computed across the length of the entire trace and, in the absence of intra-trace constraints, eviction paths can cross basic block boundaries.

We decompose the control flow graph into linear traces of basic blocks, where the predecessor of a block in a trace is also a predecessor of the block in the control flow graph. Traces may contain both internal and external control flow. We consider three possible trace decompositions:

LONG TRACES Loopless depth first search is used to compute the trace with the most blocks. This trace is selected as the first trace for allocation. The computation is then repeated on the remaining control flow graph until all blocks have been added to a trace. An example of a long traces decomposition is shown in Figure 5.9(b).

LOOP TRACES Traces are computed within each loop nesting level. Each trace contains blocks with the same loop nesting level and more deeply nested blocks are allocated first. An example of loop traces is shown in Figure 5.9(c).

LOOP AVOIDING TRACES Similar to long traces, but basic blocks at the head of a loop always start a new trace. An example of a loop avoiding trace is shown in Figure 5.9(d).

The impact on code quality of each trace decomposition strategy is shown in Figure 5.10. Both long traces and loop avoiding traces result in code size improvements (Figure 5.10(a)). Loop traces result in an increase in code size. This is because loop traces, unlike the other cases, are disconnected, and each inner loop is allocated before the surrounding code. This disconnectedness may require the insertion of shuffle code to resolve differences in allocation between traces representing inner loops. The loop avoiding trace decomposition results in slightly better code size improvements than the long trace strategy. Loop avoiding traces only contain a loop if the loop begins the traces. This means the loop entry and exit allocations are already set before the trace is allocated resulting in fewer complications with intra-trace control flow. We use the loop avoiding trace decomposition strategy within our default implementation of the simultaneous allocator when optimizing for code size. When optimizing for performance, the long trace and loop trace decomposition strategies have the best improvements (Figure 5.10(b)) with the loop trace strategy demonstrating the greatest reduction in memory operations. The long trace strategy allows for some evictions across loop boundaries, resulting in fewer memory accesses inside loops. The loop strategy allocates inner loops first and so reduces memory operations

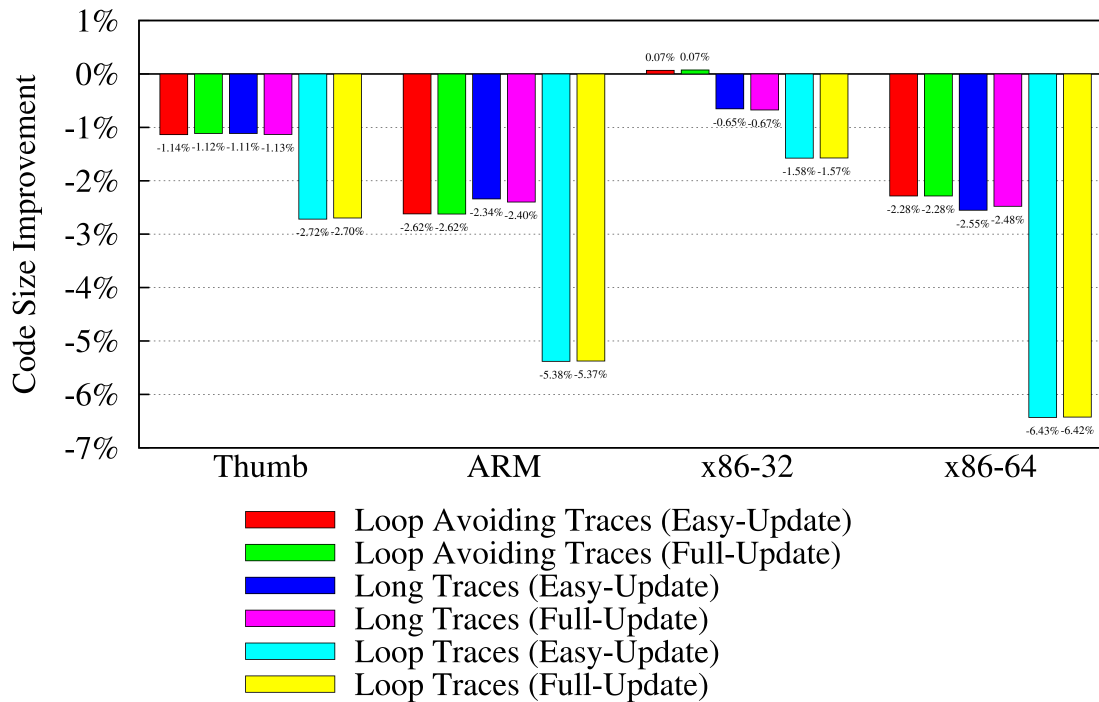
inside loops at the expense of additional shuffle code in between loops. We use the loop trace decomposition strategy within our default implementation of the simultaneous allocator when optimizing for performance.

The presence of control flow within each trace creates complications. When an allocation decision is made at a block boundary, that decision must be propagated to all connected blocks within the trace. For example, the exit allocation of block 1 in Figure 5.9(b) fixes the starting allocation of block 5 and the exit allocation of block 4 in the same trace. This is due to the control flow edges between blocks 1, 6, 7, and 5. Similarly, the exit allocation of block 2 fixes the exit allocation of block 3 within the trace.

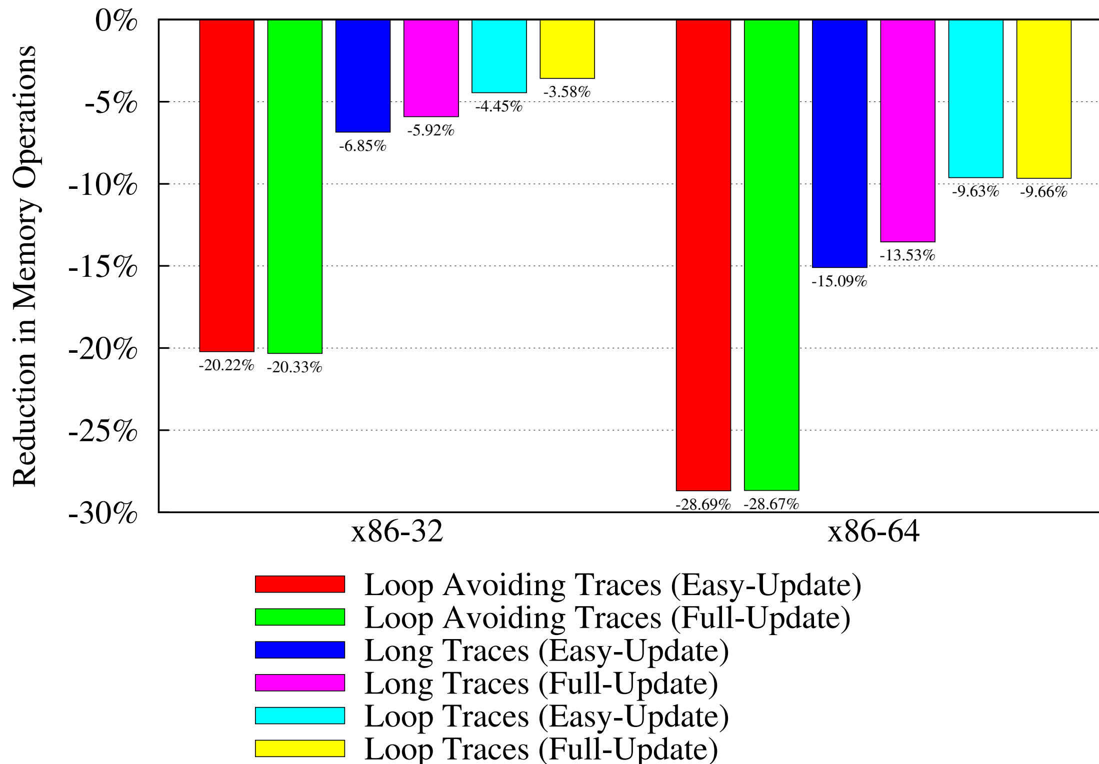
Intra-trace control flow inhibits the ability of the allocator to revise allocation decisions at basic block boundaries when implementing variable evictions. An allocation is allowed to change at a basic block boundary within a trace only if the change in allocation will only affect unallocated blocks. Unallocated blocks may be in other traces or be internal to the current trace. For example, in Figure 5.9(b), when allocating a layer in block 2, it is legal to evict a variable across the exit of block 1 (change the exit allocation of this variable in block 1). However, it is not legal to evict a variable across the exit of block 1 when allocating a layer in block 4 since modifying the exit allocations of block 1 will change the exit allocations of block 4 while it is being allocated.

An additional complication of intra-trace control flow is the effect it has on the values of the shortest cost paths through the trace. In the initial computation of shortest paths, variables are not constrained to specific allocations at basic block boundaries within the trace (unless the allocation of a previous trace has imposed such constraints). As blocks in the trace are allocated, the allocation of variables within later blocks in the trace may be constrained. For example, in Figure 5.9(b), after block 1 is allocated, the flow of variables between blocks 4 and 5 is constrained to match the current exit allocation of block 1. These additional constraints may change the value and direction of the shortest paths through the trace.

We consider two methods for updating shortest path information within a trace as blocks are allocated. The first, *easy-update*, does the minimal amount of recomputation necessary for correctness. Only blocks directly effected by the boundary allocation have their shortest paths



(a)



(b)

Figure 5.11: Effect of trace update policy on code quality in the simultaneous allocator

recomputed, and only when the recomputation is needed. For example, in Figure 5.9(b), after allocating block 1, block 4 has to be recomputed as its exit allocations have changed. However, this recomputation can be postponed until the allocator starts allocating layers within block 4. Although these recomputations result in extra work compared to the original simultaneous allocator, they are necessary to ensure that the shortest paths within the block end in the appropriate allocation class. With easy-update, shortest paths will only be computed in a block at most twice. Once during the first computation over the entire trace and possibly a second time immediately prior to the allocation of the block.

The second method, *full-update*, recomputes shortest paths in all blocks of the trace whenever a boundary allocation is constrained in the trace. The full-update technique is computationally more expensive (potentially quadratically more updates) but provides more up-to-date information for the simultaneous allocator in blocks not immediately affected by the boundary allocation. For example, in Figure 5.9(b), if a variable were to spill to memory and then be loaded back into a register in block 2, it is likely best for the variable to be loaded into the same register it was allocated to at the exit of block 1 (to avoid a move into that register before the exit of block 4). With full-update the allocator is aware of this cost since block 2, 3, and 4 are recomputed after the allocation of block 1.

The effect of the two trace update policies on code quality is shown in Figure 5.11. Surprisingly, the full-update policy results in little or negative code size improvement (Figure 5.11(a)). When optimizing for performance, the full-update policy does have some effect (Figure 5.11(b)), but only significantly for the long trace decomposition strategy. Since the minor impact on code quality does not justify the increased algorithmic complexity of the full-update policy, we use the easy-update policy within the default implementation of the simultaneous allocator.

5.2.3 Asymptotic Analysis

The simultaneous heuristic allocator, like the iterative algorithm, must compute shortest paths for every variable v in every block. Unlike the iterative algorithm, the simultaneous allocator does not need to compute each path successively and instead can compute all paths in the same pass. However, the asymptotic running time of the shortest path computation remains $O(n\bar{l})$,

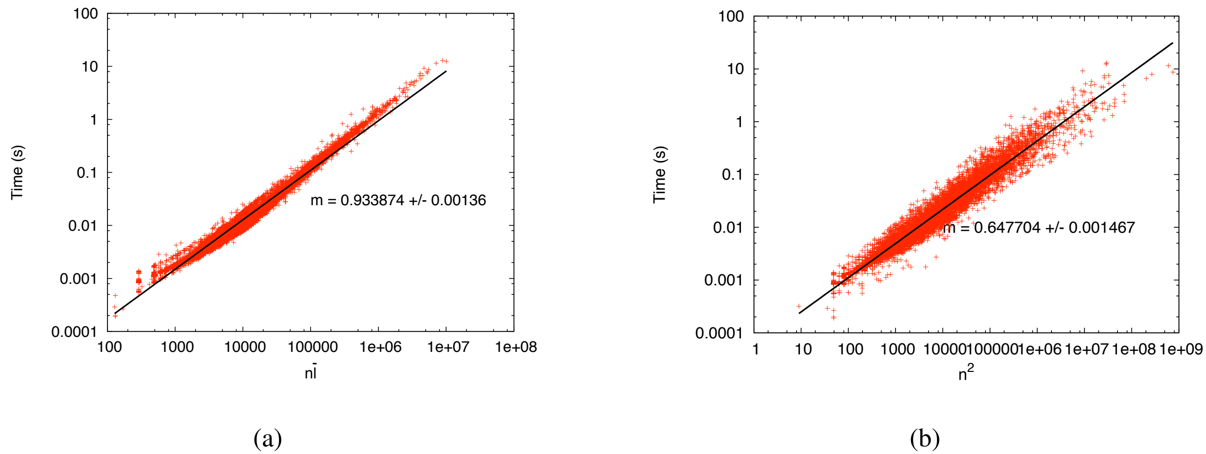


Figure 5.12: Running time of simultaneous allocator for all benchmarked functions related to (a) $n\bar{l}$ and (b) n^2 on a log-log scale.

where n is the number of instructions and $\bar{l}$ the average number of variables live at any point. In the worst case, the allocator has to compute the eviction cost of $O(v)$ variables at every layer. Computing the eviction cost involves performing an $O(n)$ shortest path computation. Since there are $O(n)$ layers in the network, this results in a worst case asymptotic running time of $O(n^2v)$. This analysis assumes an unrealistic number of evictions. In actuality, since the number of variables defined and used by an instruction is bound by a constant, the number of expected evictions at each layer is similarly bound since only those variables accessed by an instruction should necessitate an eviction. In this case there will be only $O(n)$ evictions and the worst case asymptotic running time of the simultaneous allocator is $O(n^2)$.

The empirical running times of all benchmarked functions are shown in Figure 5.12. The running times are plotted on a log-log scale both relative to $n\bar{l}$ and n^2. There is a tighter relation to $n\bar{l}$. The slope of the best-fit line relative to $n\bar{l}$ is .93 compared to .65 relative to n^2. This implies that although the worst-case asymptotic behavior of the simultaneous allocator is $O(n^2)$, in practice evictions do not dominate the performance of the allocator and the expected behavior is $O(n\bar{l})$.

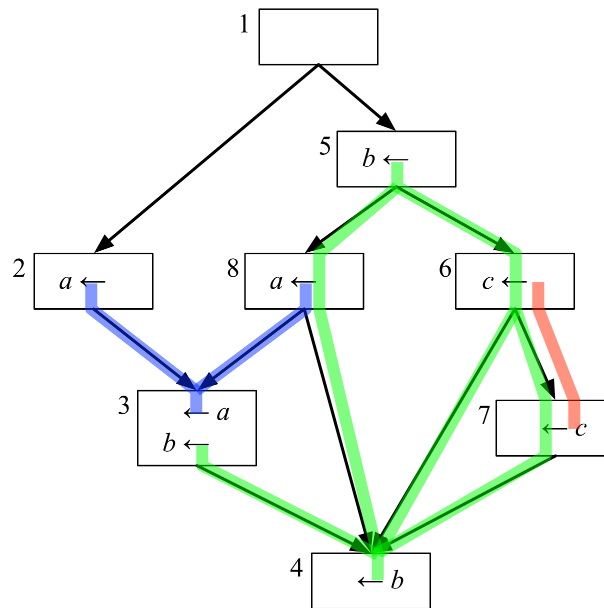


Figure 5.13: A control flow graph that illustrates the subtleties of setting boundary constraints.

5.3 Boundary Constraints

Both the iterative and simultaneous allocators must fix allocations at basic block boundaries to ensure consistency across control flow edges. This is in contrast to second-chance binpacking which runs a separate resolution phase after allocation. By fixing basic blocks as they are allocated, the allocators can make better allocation decisions within the affected blocks and avoid generating needless shuffle code.

The function $setBoundaryConstraints$, shown in Listing 5.12, is responsible for fixing allocations at basic block boundaries and ensuring that no cross-block conflicts arise. When a variable is allocated at a block boundary, the allocation is recorded and propagated to any connected blocks where the variable is live. For example, consider the case where blocks 1 and 2 in Figure 5.13 have been allocated, and the variable a is allocated to `r0` at the exit of block 2. The exit allocation of a in block 2 is recorded by $setBoundaryConstraints$ in lines 24–26. The function is then called recursively on the successor block, block 3 (lines 27–28). The entry allocation of a in block 3 is set to `r0` (lines 13–15) and, since this is the entry of block 3, a recursive call is made on all the predecessors of block 3 resulting in the setting of the exit allocation of block 8. Since these three blocks are the only blocks where a is live, every location where a must

```
    ▷ store allocation maps for each block entry and exit
 1: blockEntryVarsSet: block → set of variables
 2: blockEntryVarAlloc: block → variable → allocClass
 3: blockEntryClassAlloc: block → allocClass → set of variables
 4: blockExitVarsSet: block → set of variables
 5: blockExitVarAlloc: block → variable → allocClass
 6: blockExitClassAlloc: block → allocClass → set of variables
 7: visited: set of nodes

 8: procedure SETBOUNDARYCONSTRAINTS(node n, variable v)
 9:     ac ← n.allocClass
10:     if (n.type = Entry) ∧ (v ∈ n.variables) then            ▷ v live into this block
11:         if v ∈ blockEntryVarsSet[n.block] then           ▷ already processed, halt recursion
12:             return
13:         blockEntryVarsSet[n.block].add(v)
14:         blockEntryVarAlloc[n.block][v] ← ac
15:         blockEntryClassAlloc[n.block][ac].add(v)
16:         for p ∈ n.predecessors do
17:             setBoundaryConstraints(p, v)          ▷ process exit nodes of predecessor blocks
18:         if blockEntryClassAlloc[n.block][ac].isFull then
19:             invalidVars ← n.variables − blockEntryClassAlloc[n.block][ac]
20:             setUnusable(n, v, invalidVars)
21:     else if (n.type = Exit) ∧ (v ∈ n.variables) then          ▷ v live out of this block
22:         if v ∈ blockExitVarsSet[n.block] then            ▷ already processed, halt recursion
23:             return
24:         blockExitVarsSet[n.block].add(v)
25:         blockExitVarAlloc[n.block][v] ← ac
26:         blockExitClassAlloc[n.block][ac].add(v)
27:         for s ∈ n.successors do
28:             setBoundaryConstraints(s, v)          ▷ process entry nodes of successor blocks
29:         if blockExitClassAlloc[n.block][ac].isFull then
30:             invalidVars ← n.variables − blockExitClassAlloc[n.block][ac]
31:             setUnusable(n, invalidVars)
```

Listing 5.12: SETBOUNDARYCONSTRAINTS *Given an exit or entry node n and variable v, set the boundary constraints of the connected basic blocks appropriately.*

be allocated to `r0` has been properly constrained. However, this is not sufficient to preventing cross-block conflicts with the allocation of a.

Consider the allocation of variable b at the exit of block 8. Since a is allocated to `r0`, b cannot be allocated to `r0` at the exit of block 8. The allocation of b at the exit of block 8

```
1: procedure SETUNUSABLE(node n, set of variables invalid)
2:     invalid = invalid ∩ n.variables                ▷ consider only variables live at n
3:     if invalid ⊂ n.unusableVars then return        ▷ halt recursion
4:     n.unusableVars = n.unusableVars ∪ invalid
5:     if node.type = Entry then
6:         for p ∈ n.predecessors do
7:             setUnusable(p, invalid)
8:     else if node.type = Exit then
9:         for s ∈ n.successors do
10:            setUnusable(s, invalid)
```

Listing 5.13: SETUNUSABLE *Marks the variables in the set* $invalid$ *as unusable in a boundary node* n*. Any connected nodes where a variable in* $invalid$ *is still live is recursively marked.*

```
1: function VIOLATESBOUNDARYCONSTRAINT(node n, variable v)
2:     if n.type = Entry then
3:         if v ∈ blockEntryVarsSet[n.block] then
4:             if blockEntryVarAlloc[n.block][v] = n.allocClass then
5:                 return false        ▷ correct allocation, does not violate constraint
6:             else
7:                 return true
8:     else if n.type = Exit then
9:         if v ∈ blockExitVarsSet[n.block] then
10:            if blockExitVarAlloc[n.block][v] = n.allocClass then
11:                return false        ▷ correct allocation, does not violate constraint
12:            else
13:                return true
14:    return v ∈ n.unusableVars
```

Listing 5.14: VIOLATESBOUNDARYCONSTRAINT *Return true if allocating variable* v *to exit or entry node* n *would violate the existing boundary constraints.*

must equal its allocation at the entry of blocks 4 and 7 and the exits of blocks 3, 6, and 7. Therefore, if a is allocated to `r0` at the exit of block 2, b cannot be allocated to `r0` at any of these points. One approach that prevents this conflict is to block the allocation of any variable to `r0` at all the block boundaries that are connected to the exit of block 2. However, this approach is needlessly conservative and would prevent the allocation of c to `r0` in blocks 6 and 7. Instead, it is more precise to only block the allocation to `r0` of variables that conflict with a at some point. This is exactly what is accomplished by the calls to $setUnusable$ in lines 18–20 and 29–31 in Listing 5.12.

The $setUnusable$ function, shown in Listing 5.13, is called on a node with a set of variables that cannot be allocated to that node. Initially, this is all the variables that are live at the node, but not allocated to the already fully allocated node. These variables are marked as unusable by the node (line 4) and then $setUnusable$ is called recursively on all connected nodes. At each node, the set of invalid variables is reduced to contain only variables live at the node (line 2). The invalid set will therefore only contain variables that at one point conflicted with the original variable. In our example, the invalid set will consist of only b at the exit of block 8. The entry `r0` nodes of blocks 4 and 7 and exit `r0` nodes of blocks 3, 6, 7, and 8 will be marked as invalid for b. However, at no point will the variable c be added to the invalid set.

The function $violatesBoundaryConstraint$, shown in Listing 5.14, checks if it is legal to allocate a variable to a boundary node. If the variable has been set to a particular allocation class (lines 3,9), then the allocation class of the node must match the set allocation class (lines 4,10). Otherwise, the allocation is valid as long as the variable is not marked as unusable for this node (line 14).

5.3.1 Asymptotic Analysis

Let b be the number of blocks and v the number of variables in a program. Since entry and exit allocations are only set once for each variable, $setBoundaryConstraints$ will only process a node $O(bv)$ times. Similarly, $setUnusable$ will only process a node if the invalid set contains previously unprocessed variables so at most $O(bv)$ nodes are processed. The additional overhead of setting boundary constraints does not affect the asymptotic running time of either allocator.

5.4 Hybrid Allocator

We have described two allocators that heuristically solve the global MCNF model to quickly find a good solution. The two allocators function differently and when evaluated on each benchmark show clear distinctions as shown in the mixed results of Figures 5.14, 5.15, and 5.16. Our allocators do best when optimizing for the register-limited x86-32 architectures. This is likely because the majority of the development effort designing and tweaking these heuristics was spent target-

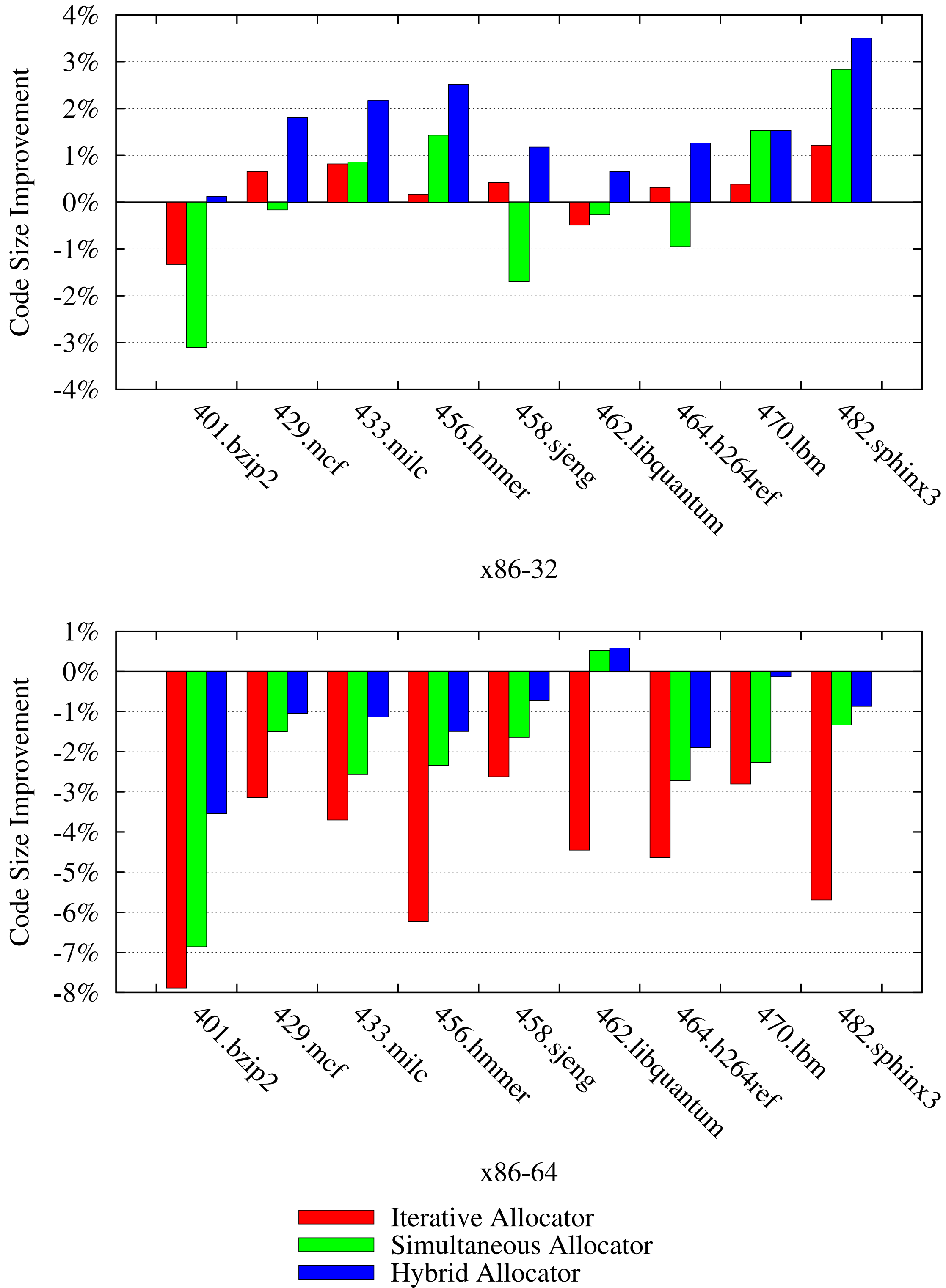


Figure 5.14: Code size improvement of heuristic allocators.

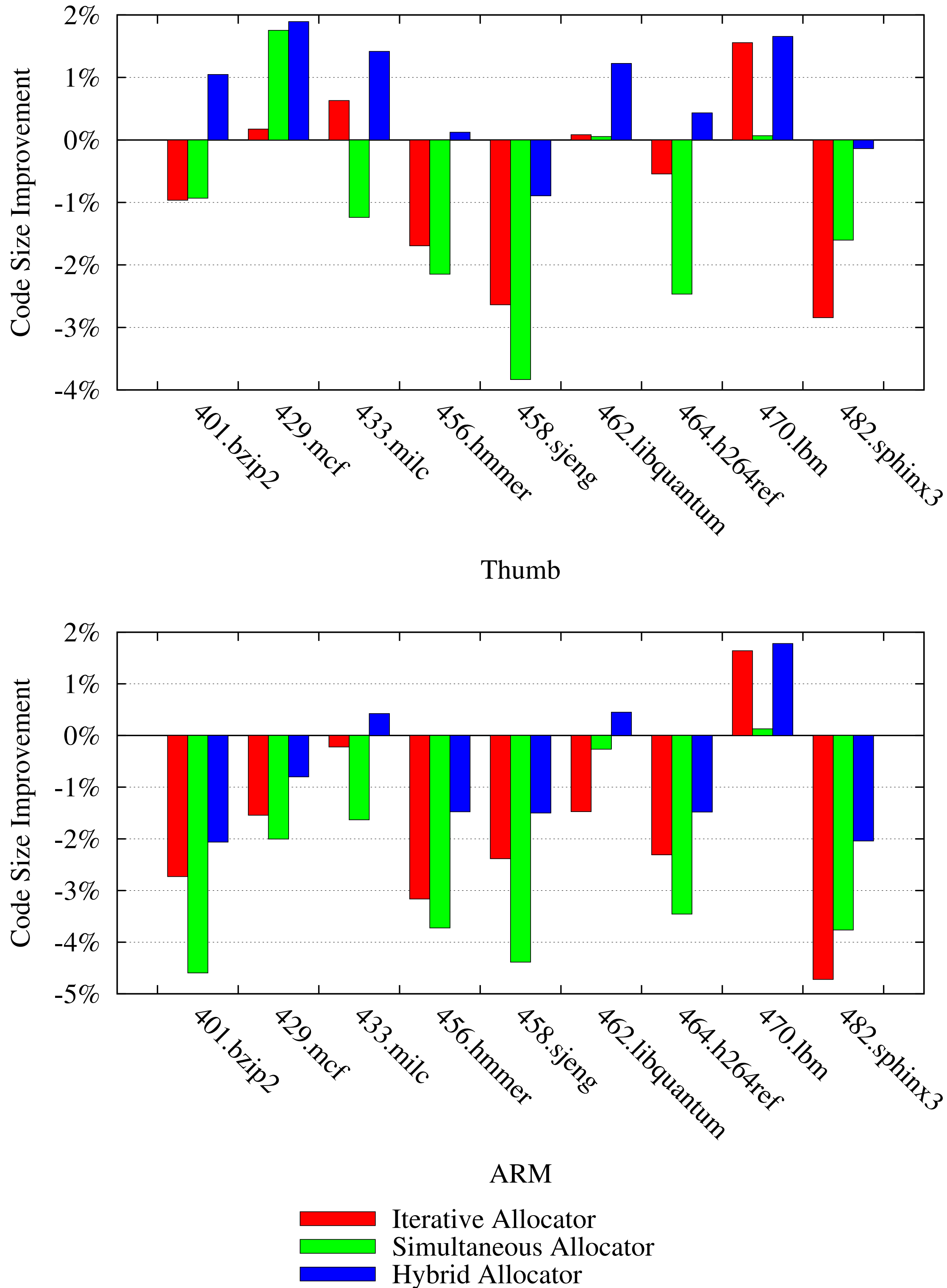


Figure 5.15: Code size improvement of heuristic allocators.

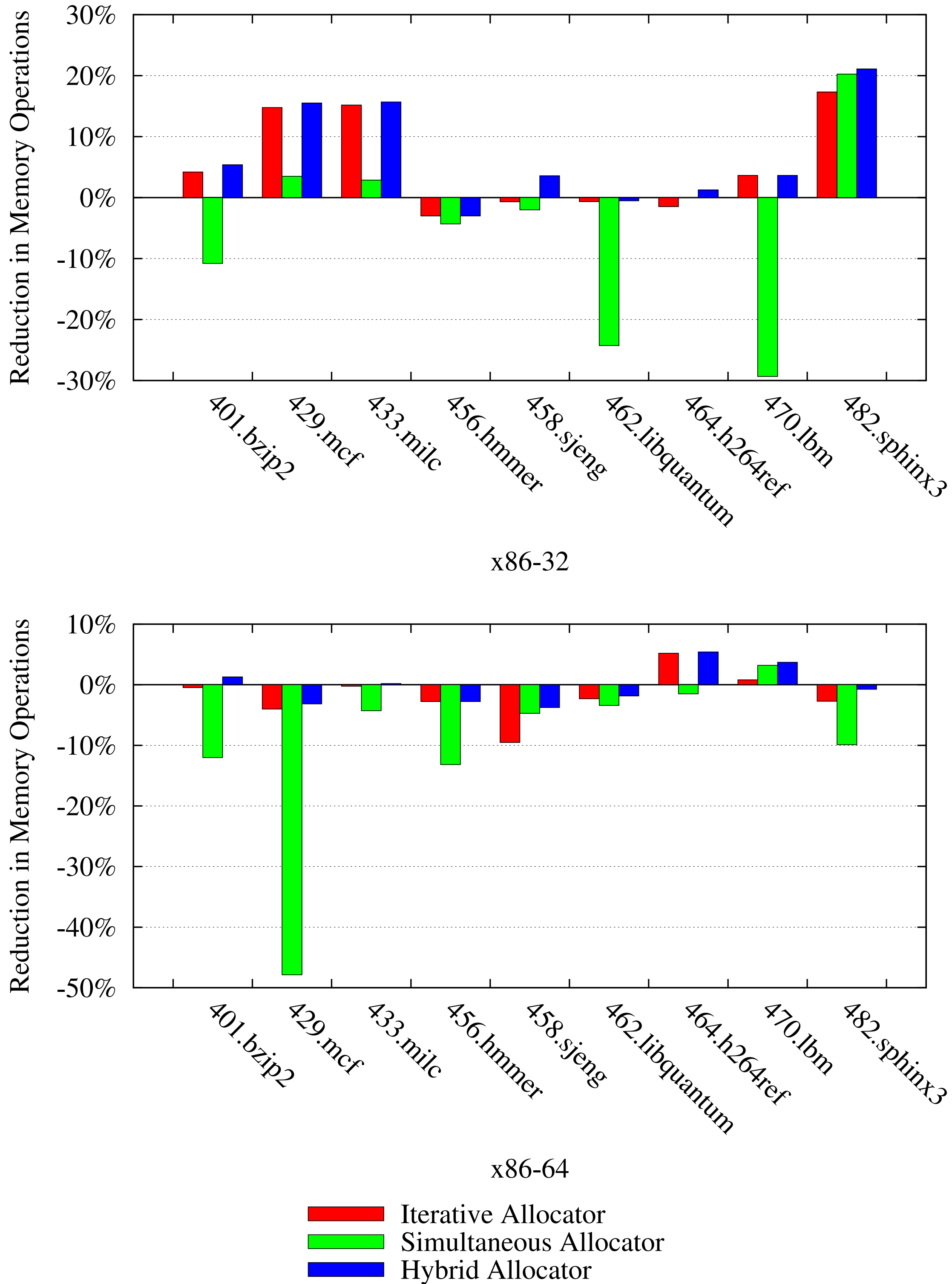
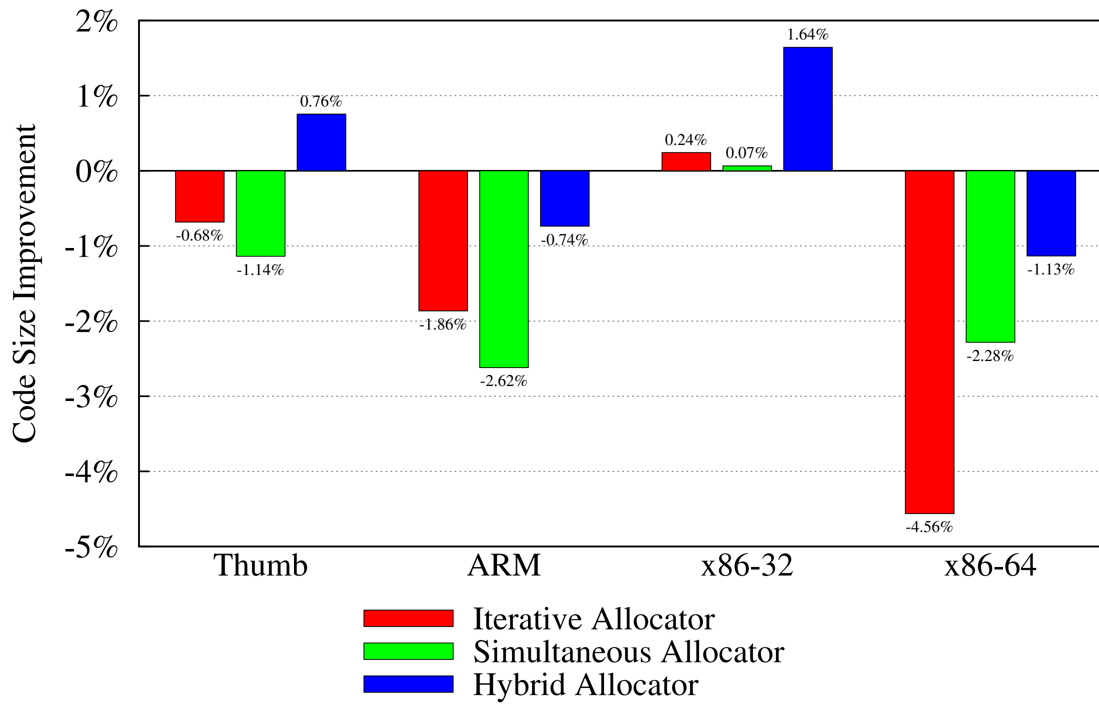
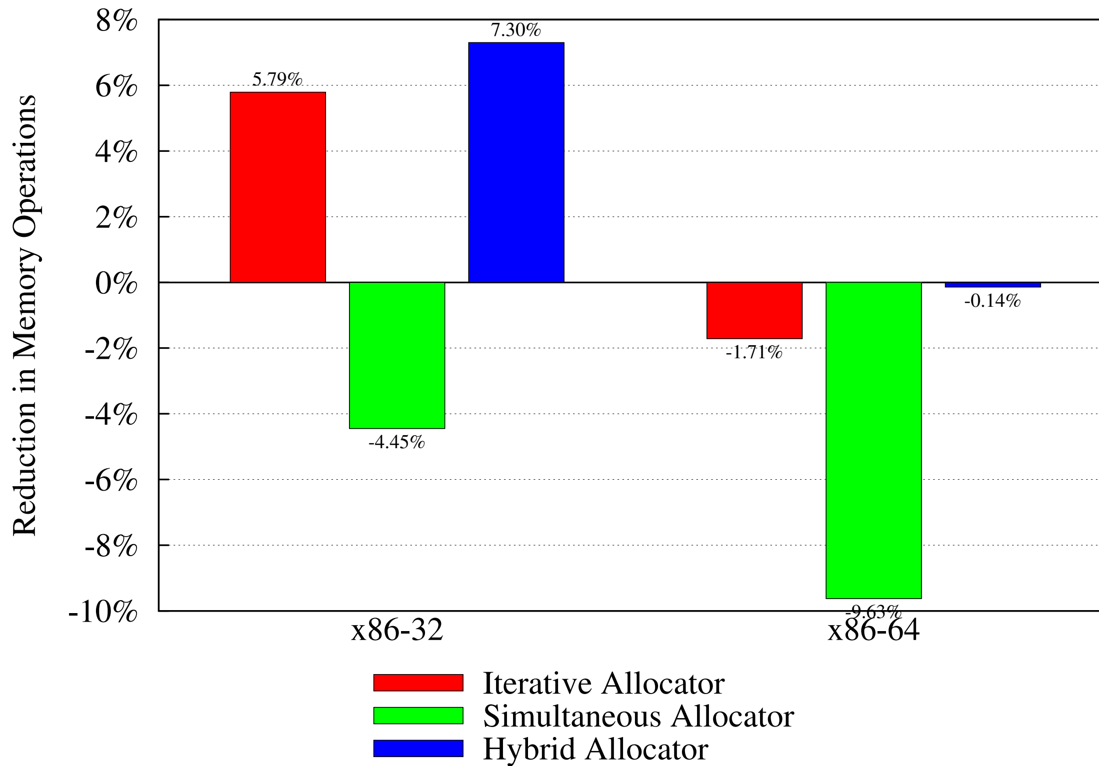


Figure 5.16: Memory operation reduction of heuristic allocators.



(a)



(b)

Figure 5.17: Average code quality improvement of heuristic allocators

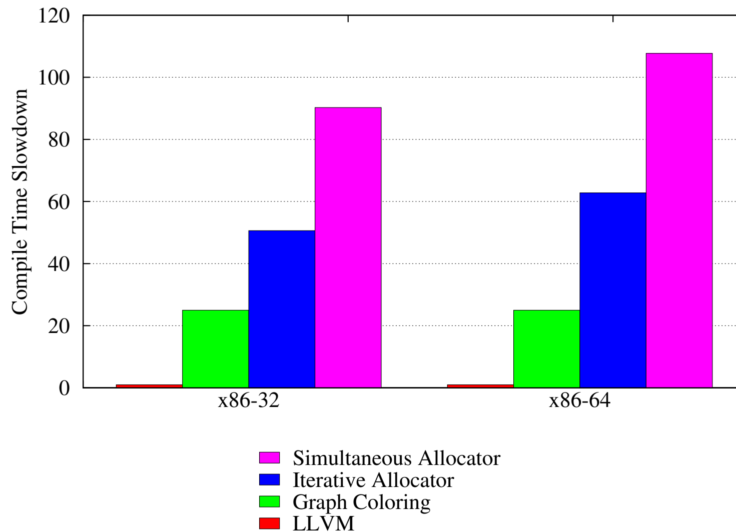


Figure 5.18: Average slowdown of various allocators relative to the extended linear scan allocator used in the LLVM compiler.

ing x86-32. However, the allocators also do well when optimizing for the Thumb architecture, suggesting that there may be innate value in our expressive model approach when optimizing for resource constrained architectures.

The explicit code model of the global MCNF model can be used to create a *hybrid allocator*. The hybrid allocator runs both the simultaneous allocator and the iterative allocator on the model and chooses the result with the better cost. The hybrid allocator substantially improves upon the iterative and simultaneous allocators both when optimizing for code size (Figure 5.17(a)) and when optimizing for performance (Figure 5.17(b)).

5.5 Compile Time

The iterative, simultaneous, and hybrid allocators generate code competitive with the default extended linear scan allocator, but are also asymptotically more complex. The slowdowns of the allocators are shown in Figure 5.18. We compare only allocation times. Our heuristic allocators are close to two orders of magnitude slower than the extended linear scan allocator. There are three reasons for this significant disparity:

- The extended linear scan algorithm is designed to be fast. It is on average 25 times faster than traditional graph coloring [115]. Relative to graph coloring allocation, our algorithms are within an order of magnitude.
- Our algorithms are implemented in a research framework where performance is sacrificed for flexibility.
- Our allocation algorithms are asymptotically slower.

Although our allocation algorithms are substantially slower than the default LLVM algorithm, the overall compile time is still tractable for real-world use. For instance, although the iterative heuristic allocator is ~40x slower on average than the extended linear scan algorithm, the average total compile time of the benchmarks is only ~4x slower.

5.6 Summary

The register allocation algorithms developed in this chapter use an expressive model to find decent solutions. We have seen that by improving the heuristics guiding these algorithms, the resulting code quality can be improved. The guiding heuristics of these algorithms can be further tweaked and additional incremental code quality improvements can be achieved. In particular very little development time has been spent on tuning the heuristics to optimize well for performance. In a principled compiler, code quality is not dependent solely on the ability of the compiler developer to cleverly tune heuristic algorithms. Instead, an expressive model is coupled with progressive solution techniques that approach the optimal solution. We describe a progressive register allocator in the next chapter that uses the heuristic allocators described in this chapter as building blocks.

Chapter 6

Progressive Register Allocation

In this chapter we describe progressive solution techniques for solving the expressive global MCNF model. Progressive solution techniques quickly find a good solution and then progressively improve upon this solution as more time is allotted for compilation. Our progressive allocator has the additional advantage that it provides a meaningful upper bound on the optimality of the solution. These features enable a new form of interaction between the programmer and compiler as the programmer can now explicitly manage the trade-off between compile-time and code quality.

First, we describe the relaxation techniques we use to derive optimality bounds and the subgradient optimization algorithm we use to find progressively better lower bounds. We then combine the subgradient optimization algorithm with the heuristic solvers of the previous chapter to create a progressive register allocator. Our progressive allocator generates substantial reductions in code size and increases in performance compared to the default LLVM allocator.

6.1 Relaxation Techniques

In the context of mathematical optimization, a *relaxation technique* is a method for relaxing a problem constraint, either by substitution or elimination, in order to derive an approximation of the original problem. Recall the definition of the global MCNF problem from Chapter 3:

$$\min \sum_{i,j,q} c^q_{ij} x^q_{ij} \qquad \textit{cost function} \tag{6.1}$$

subject to the constraints:

$$0 \leq x^q_{ij} \leq v^q_{ij} \qquad \textit{individual capacity constraints} \tag{6.2}$$

$$\sum_q x^q_{ij} \leq u_{ij} \qquad \textit{bundle constraints} \tag{6.3}$$

$$\mathcal{N} x^q = b^q \qquad \textit{network constraints} \tag{6.4}$$

$$x^q_{pred_{ext},ext} = x^q_{ext,ent} \qquad \textit{exit boundary constraints} \tag{6.5}$$

$$x^q_{ext,ent} = x^q_{ent,succ_{ent}} \qquad \textit{entry boundary constraints} \tag{6.6}$$

$$x^q_{i,j} \in \{0,1\} \qquad \textit{integrality constraints} \tag{6.7}$$

x^q_{ij} represents the flow of a commodity q over an edge (i,j) and (ext, ent) represents a pair of connected exit and entry nodes.

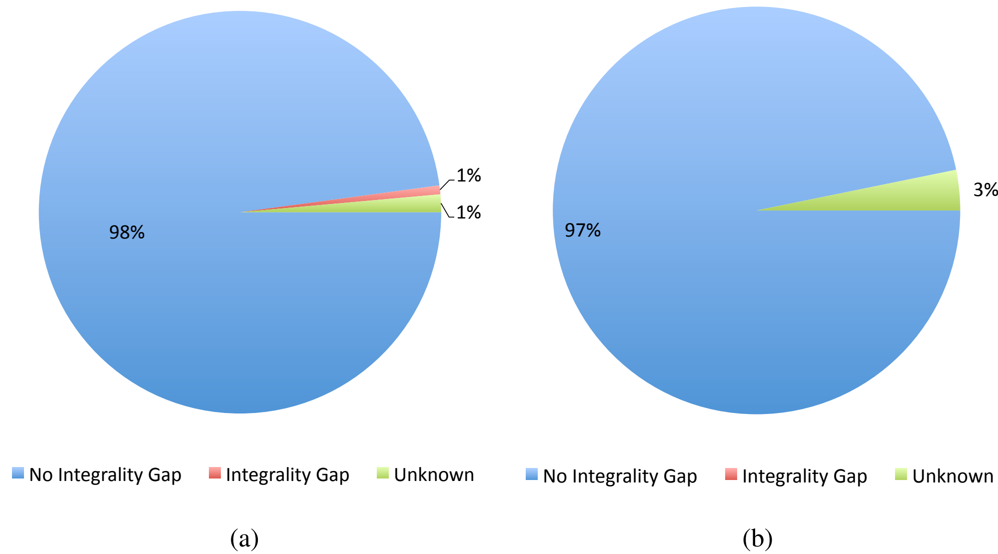
We utilize two relaxation techniques: *linear programming relaxation* and *Lagrangian relaxation*.

6.1.1 Linear Programming Relaxation

The linear programming relaxation of the global MCNF problem replaces the integrality constraints (6.7) with a pair of linear constraints:

$$x^q_{i,j} \geq 0 \qquad\qquad x^q_{i,j} \leq 1$$

The resulting relaxation is a linear program. The solution space of the original global MCNF problem is a subset of the solution space of the linear program; every solution of the global MCNF problem is a solution of the linear programming relaxation, but not vice versa. As a result, the optimal solution of the linear programming relaxation is a lower bound on the optimal solution of the global MCNF problem. The optimal solution of a linear program can be found in polynomial time [6, 126], but there are three fundamental limitations that prevent us from directly using the relaxation to find an optimal solution of the global MCNF problem: *integrality gaps*, *fractional solutions*, and *impractical solution times*.



(a) (b)

Figure 6.1: The percentage of functions within a reduced benchmark set that demonstrate an integrality gap when optimizing for (a) code size and for (b) performance targeting the x86-32 architecture.

An *integrality gap* exists if the optimal value of the linear relaxation is smaller than that of the global MCNF problem. When this is the case, the solution of the linear relaxation is guaranteed to violate the integrality constraints of the global MCNF problem. However, linear relaxations of real-world global MCNF problems rarely have an integrality gap. We demonstrate this empirically by compiling a subset of the MediaBench [79] benchmark suite consisting of the benchmarks dijkstra, g721, mpeg2, patricia, qsort, sha, and stringsearch. We use ILOG CPLEX 10.0 [64] to compute the optimal solutions to both the linear relaxation and full global MCNF model when optimizing both for code size and performance. As shown in Figure 6.1, at least 97% of the functions compiled demonstrate no integrality gap. The remaining functions either have a definite integrality gap or an optimal solution is not found within the solver time limit of 600 seconds and the existence of an integrality gap is unknown.

Even in the absence of an integrality gap, the linear relaxation cannot be used directly to solve the global MCNF problem since most of the optimal solutions of the linear relaxation are fractional. Fractional solutions violate the integrality constraints of the global MCNF problem. Standard linear programming solution techniques [6, 126] are not designed to prefer integer valued solutions and empirically almost always generate fractional solutions.

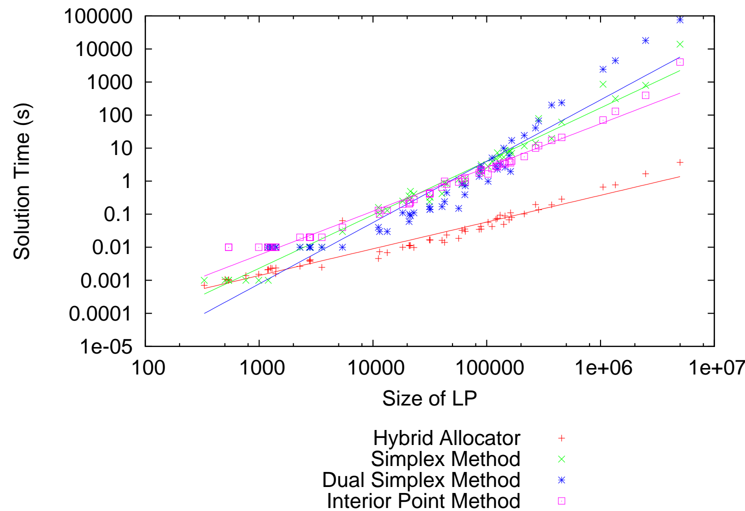


Figure 6.2: Linear programming solution times of the global MCNF problem when optimizing the functions of the 401.bzip2 benchmark for size targeting the x86-32 architecture. Times are displayed on a log-log scale relative to the number of nonzeroes in the linear program.

Although linear programs can be solved in polynomial time, solution times do not scale well with problem size. It is not practical for a full linear programming solver to be a standard part of a compiler tool flow. We illustrate this in Figure 6.2 where we graph the solution times of the linear programming relaxation of all the functions of the 401.bzip2 benchmark on a log-log scale. This benchmark can be compiled to optimality and has a good range of function sizes. We evaluate three popular linear programming solution algorithms as implemented by ILOG CPLEX [64]. The slopes of the lines fit to the data in Figure 6.2 range from 1.3 to 1.8 for the linear programming solution techniques, indicating super-linear asymptotic running times, while the hybrid allocator of Section 5.4 has a slope of 0.8, indicating a sub-linear running time relative to the size of the linear program. In addition, the hybrid allocator is several orders of magnitude faster.

It is not practical to use conventional, unmodified, linear programming techniques to solve the global MCNF problem. However, due to the rarity of integrality gaps, the linear programming relaxation is very useful in establishing an optimality bound. The value of the linear relaxation is a lower bound on the value of the original global MCNF problem. A lower bound, val_{LB}, can be used to compute an upper bound on the optimality of a solution:

$$\text{optimality bound} = \frac{val - val_{LB}}{val_{LB}}$$

6.1.2 Lagrangian Relaxation

Lagrangian relaxation [6, 80] is a general relaxation technique that removes one or more constraints from a problem and integrates them into the objective function using Lagrangian multipliers. Given an optimization problem:

$$\min f(x)$$

subject to:

$$\begin{aligned} g(x) &\leq 0 \\ g'(x) &\leq 0 \\ h(x) &= 0 \\ h'(x) &= 0 \end{aligned}$$

where $g'(x)$ and $h'(x)$ are complicating constraints (i.e., the problem is easy to solve in their absence) the Lagrangian relaxation, L, is:

$$\min L(x, w, v) = f(x) + wg'(x) + vh'(x)$$

subject to:

$$\begin{aligned} g(x) &\leq 0 \\ h(x) &= 0 \\ w &\geq 0 \end{aligned}$$

where w and v are Lagrangian multipliers.

In the Lagrangian relaxation of the global MCNF problem, we relax the bundle constraints (6.3) and boundary constraints (6.5 and 6.6).

The bundle constraints are removed by incorporating the term

$$\sum_{i,j} w_{ij} \left(\sum_{q} x_{ij}^{q} - u_{ij} \right)$$

into the objective function. The w_{ij} term is the Lagrangian multiplier, or price, and must be greater than or equal to zero.

To remove the boundary constraints we first simplify the redundant exit (6.5) and entry (6.6) constraints into a single boundary constraint and put it into standard form:

$$x^q_{pred_{ext},ext} = x^q_{ent,succ_{ent}}$$

$$x^q_{pred_{ext},ext} - x^q_{ent,succ_{ent}} = 0$$

We then include the term

$$v_{ext,ent}\left(x^q_{pred_{ext},ext} - x^q_{ent,succ_{ent}}\right)$$

where $v_{ext,ent}$ is the Lagrangian multiplier and is unrestricted in sign, into the objective function. The result is the Lagrangian relaxation:

$$L(x,w,v) = \min \sum_{i,j,q} c^q_{ij} x^q_{ij} + \sum_{i,j} w_{ij}\left(\sum_q x^q_{ij} - u_{ij}\right) + \sum_{(ext,ent)} v_{ext,ent}\left(x^q_{pred_{ext},ext} - x^q_{ent,succ_{ent}}\right)$$

subject to:

$$0 \leq x^q_{ij} \leq v^q_{ij}$$

$$\mathcal{N}x^q = b^q$$

If we define

$$v_{ij} = \begin{cases} v_{ent,ext} & j = ext, \\ -v_{ent,ext} & i = ent, \\ 0 & \text{otherwise} \end{cases}$$

then the relaxation can be simplified to:

$$L(w) = \min \sum_{i,j,q} \left(c^q_{ij} + w_{ij} + v_{ij}\right) x^q_{ij} - \sum_{i,j} w_{ij} u_{ij}$$

subject to:

$$0 \leq x^q_{ij} \leq v^q_{ij}$$

$$\mathcal{N}x^q = b^q$$

The relaxation has decomposed the original global MCNF problem into a set of independent shortest flow problems where the cost of an edge, x^q_{ij}, is the sum of the original cost, c^q_{ij}, the

Lagrangian multiplier w_{ij}, and v_{ij} which is equal to the Lagrangian multiplier $v_{ent,ext}$ if node j is an exit node and is equal to $-v_{ent,ext}$ if node i is an entry node.

A Lagrangian relaxation has the property that for any values of w and v, $L(x, w, v) \leq z^*$, where z^* is the optimal value of the original unrelaxed problem. An optimal solution of $L(x, w, v)$ is a lower bound on the optimal solution of the original problem. The best lower bound of $L(x, w, v)$ for z^* is therefore:

$$L^* = \max_{w,v} \min_{x} L(x, w, v)$$

In fact, when the original problem is a linear program, the lower bound L^* is exact; it equals the optimum value of the original problem [6]. Furthermore, if a solution, x, to L^* is feasible in the original problem and it satisfies the complementary slackness condition, it is optimal. The complementary slackness condition simply requires that any edge with a non-zero price is used to its full capacity in the solution: $w_{ij}\left(\sum_q x_{ij}^q - u_{ij}\right) = 0$. The solution of L^* is not only useful in establishing a tight lower bound on the value of the optimal solution, but may also yield an optimal solution if a flow vector x can be found that obeys the complementary slackness condition.

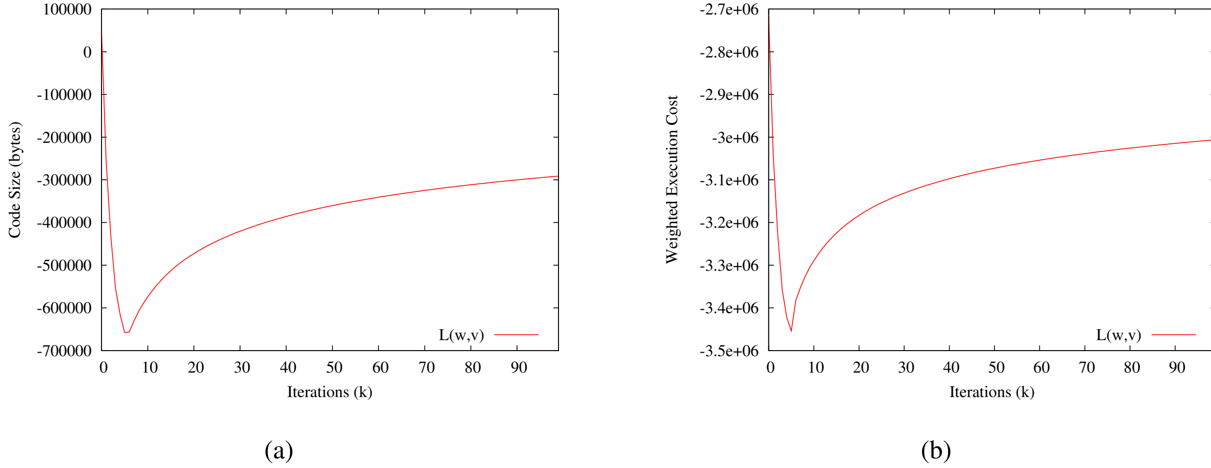
6.2 Subgradient Optimization

Solving for L^*, that is finding values of w and v that maximize $L(x, w, v)$, is known as the Lagrangian dual problem and is solved using an iterative *subgradient optimization* algorithm [6, 61, 121]. The subgradient optimization algorithm is a generalization of gradient descent algorithms. The algorithm converges to optimal values of w and v by taking successively smaller steps in the direction of the subgradient, an approximation of the gradient. In general terms, subgradient optimization computes a sequence of Lagrangian prices, $\lambda_k, k \geq 1$ using:

$$\lambda_{k+1} = \lambda_k + \theta_k g_k$$

where θ_k is the step size and g_k is the subgradient.

When applied to our Lagrangian relaxation of global MCNF, an iteration of the subgradient optimization algorithm starts with price vectors, (w^k, v^k), and finds a flow vector, y^k, that min-



(a) (b)

Figure 6.3: Convergence behavior of the basic subgradient optimization algorithm when optimizing for (a) code size and (b) performance.

imizes $L(y^k, w^k, v^k)$. This is done in linear time by computing all shortest paths in each block. Given the resulting flow vector, y^k, the standard update rules for (w, v) are:

$$\begin{aligned} w_{ij}^{k+1} &= \max\left(w_{ij}^k + \theta_k\left(\sum_q y_{ij}^q - u_{ij}\right), 0\right) \\ {v_{ent,ext}^q}^{k+1} &= {v_{ent,ext}^q}^k + \theta_k\left(y^q_{pred_{ext},ext} - y^q_{ent,succ_{ent}}\right) \end{aligned} \tag{6.8}$$

where θ_k is the current step size. Intuitively, these rules will increase the price of edges that are over-allocated and decrease the price of edges that are under-utilized. The algorithm is guaranteed to converge if θ_k satisfies the conditions:

$$\lim_{k\to\infty} \theta_k = 0$$
$$\lim_{k\to\infty} \sum_{i=1}^{k} \theta_i = \infty$$

An example of a simple method for calculating a step size that satisfies these conditions is the ratio method, $\theta_k = 1/k$.

The convergence behavior of the standard subgradient optimization algorithm with a simple ratio step update rule and zero initial prices is shown in Figure 6.3. When displaying convergence results, unless stated otherwise, we target the x86-32 instruction set architecture. We consider optimizing both for code size, which has a uniform cost metric, and for performance, where

the cost metric is nonuniform and costs vary dramatically between blocks. At each iteration of the algorithm, we compute $L(w, v) = \min_x L(x, w, v)$ using a standard shortest paths algorithm. Unless stated otherwise, we display an aggregrate $L(w, v)$ value that is the sum at each iteration of $L(w, v)$ for every function in the benchmark set. The value of $L(w, v)$ is not monotonic; the best lower bound is given by the maximum value of all iterations. Since $L(w, v)$ is a lower bound, values can be negative even if negative values are nonsensical, such as with code size, and the values increase as the prices converge. In displaying convergence results, we choose the x-axis range that provides the clearest picture of the convergence behavior.

The unmodified subgradient optimization algorithm shown in Figure 6.3 requires substantial improvement before it can provide a useful lower bound. The shown algorithm does not improve upon the initial lower bound after 100 iterations. The convergence behavior of the subgradient optimization algorithm can be improved by adjusting the *flow calculation* of y^k, the *step update rule*, the *price update rule*, and the *initial prices*.

6.2.1 Flow Calculation

Each iteration of the subgradient optimization algorithm must solve $L(x^k, w^k, v^k)$ for x^k to obtain a flow vector y^k. In general, there is not a unique solution for y^k. Since the values of y^k are part of the price update rule (6.8), the algorithm used to compute y^k affects the convergence of the optimization. We consider two algorithms for computing the flow vector y^k: *standard shortest paths* and *balanced shortest paths*.

The standard shortest path flow calculation calculates a single shortest path for each variable. The contribution of the flow of a variable to the flow vector will be either zero or one. If there are multiple possible shortest paths, a path is arbitrarily, but deterministically, chosen. In the common case where variables have no preference for a particular register, this flow calculation will result in an allocation that sends all variables through the exact same register class. In the next iteration of the subgradient optimization algorithm, the prices along this over-allocated path will increase. The subsequent shortest path calculation will then over-allocate the same set of variables to a different register class. Eventually, as the prices converge, the flows of the different

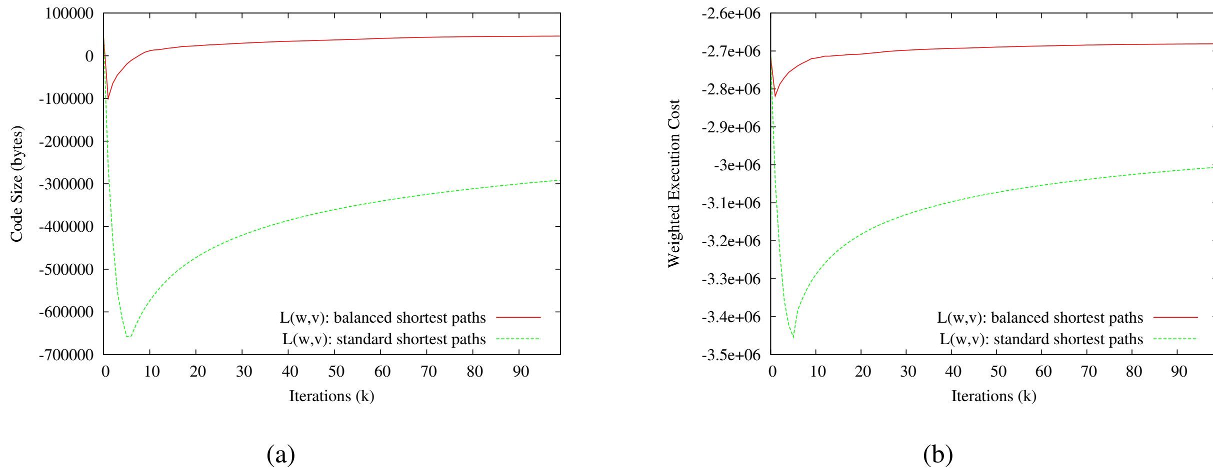


Figure 6.4: Convergence behavior of the subgradient optimization algorithm using different flow calculations when optimizing for (a) code size and (b) performance.

variables will diverge, but several iterations are required to properly "warm-up" the prices of register classes.

The balanced shortest path calculation allocates the flow of a variable equally among all possible shortest paths. If there are i shortest paths, the flow along each shortest path is incremented by $1/i$. Although in a network with n instruction groups there are $O(2^n)$ shortest paths, this calculation can be done using two linear passes over the network. The first pass computes the shortest paths. At each node, in addition to the value of the shortest path, the number of shortest paths to that node and the set of previous nodes along these paths is recorded. Since the number of incoming edges of a node in the network is $O(1)$, this set can be efficiently implemented with a bitmask. The second pass traverses the network backwards, against the flow, and allocates the flow in proportion to the number of paths traversing each node. Unlike the standard shortest path flow calculation, the resulting flow vector may include fractional flows.

The effect these two flow calculation algorithms have on the convergence of prices in the subgradient optimization algorithm is shown in Figure 6.4. As expected, the balanced shortest path approach substantially outperforms the standard shortest path calculation. We utilize this method of flow calculation in our implementation of subgradient optimization.

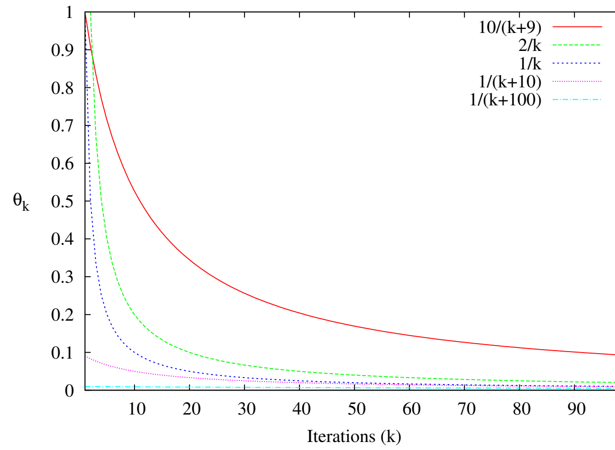


Figure 6.5: Graphical depiction of five ratio step update rules.

6.2.2 Step Update

The step update rule determines how large a step, θ_k, should be taken in the direction of the gradient when updating the values of the prices, λ, at iteration k of the algorithm:

$$\lambda_{k+1} = \lambda_k + \theta_k g_k$$

No definitive step update rule exists and the choice of step update rule depends heavily upon the problem instance [99]. The two most common step update rule frameworks that guarantee convergence are the *ratio rule* and an application of *Newton's method*:

$$\theta_{k+1} = \frac{a}{k+b} \qquad \text{(Ratio)}$$

$$\theta_{k+1} = \frac{\beta_k \left[L_{UB} - L(w,v)\right]}{||g_k||^2} \qquad \text{(``Newton's Method'')}$$

The value $||g_k||$ is the Euclidean norm of the subgradient:

$$||g_k|| = \sqrt{\sum_{ij}\left(\sum_{q} y_{ij}^q - u_{ij}\right)^2 + \sum_{ent,ext,q}\left(y^q_{pred_{ext},ext} - y^q_{ent,succ_{ent}}\right)^2}$$

The ratio rule is a simple yet effective step update rule. The rate that the step size decreases and the initial step size value can be adjusted with a multiplicative factor a in the numerator and

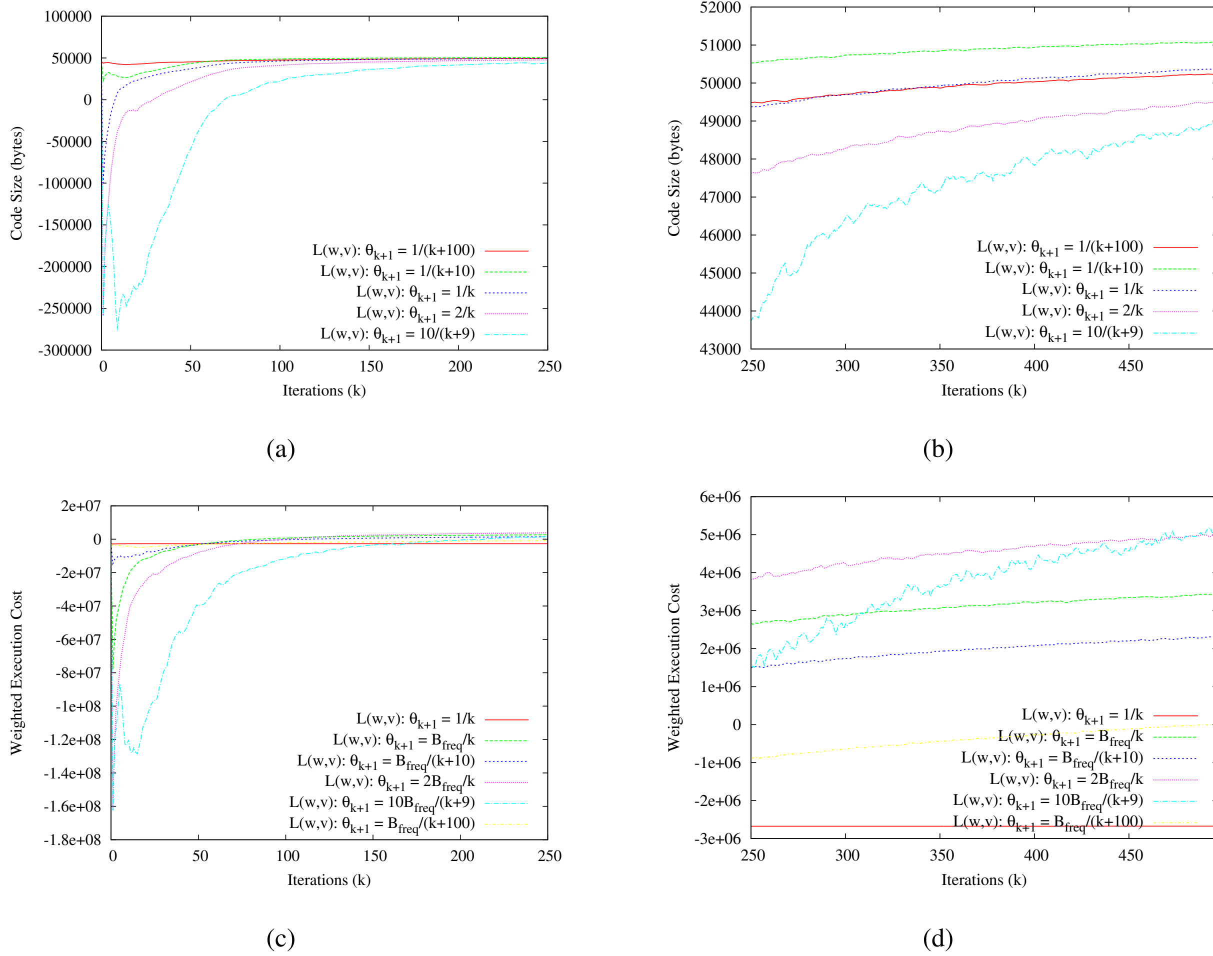


Figure 6.6: Convergence behavior of the subgradient optimization algorithm using different step update rules when optimizing for (a) (b) code size and (c) (d) performance. We show the initial and successive behavior separately to magnify the behavior of the later.

an additive term b in the denominator. We consider five possible adjustments:

$$\theta_{k+1} = \frac{1}{k} \qquad \theta_{k+1} = \frac{1}{k+10} \qquad \theta_{k+1} = \frac{1}{k+100}$$

$$\theta_{k+1} = \frac{2}{k} \qquad \theta_{k+1} = \frac{10}{k+9}$$

The first three rules decrease at the same rate but start with different initial step sizes. The last two rules decrease at slower rates. The behavior of θ_k for each of these five rules is depicted in Figure 6.5.

The effect of the various rules on the convergence of the subgradient optimization algorithm when optimizing for size is shown in Figures 6.6(a) and 6.6(b). The larger the initial step size,

the greater the initial drop in value. A large initial step size results in large price increases and dramatic changes in the flow calculation results. Smaller step sizes have a more incremental impact on the flow calculations. However, despite different initial behavior, the first three, $1/(k+a)$, rules are all close to the same value after 100 iterations. The last two rules, which reduce the step size at a slower rate, do not converge as quickly. Although the $1/(k+100)$ rule has better initial behavior, the slightly larger step sizes of the $1/(k+10)$ rule allow it to make more progress towards convergence at later iterations. We consider the $1/(k+10)$ rule to have the best overall convergence behavior.

Unlike the code size metric, the speed metric is not uniform. Since costs are weighted by execution frequencies, the cost of an operation varies substantially depending upon what block the operation is executed in. Costs within innermost loops are large and flows within these frequently executed blocks will not be affected by minor changes in prices. As a result, large step sizes are necessary in order to achieve good convergence. We consider an alternative step update rule

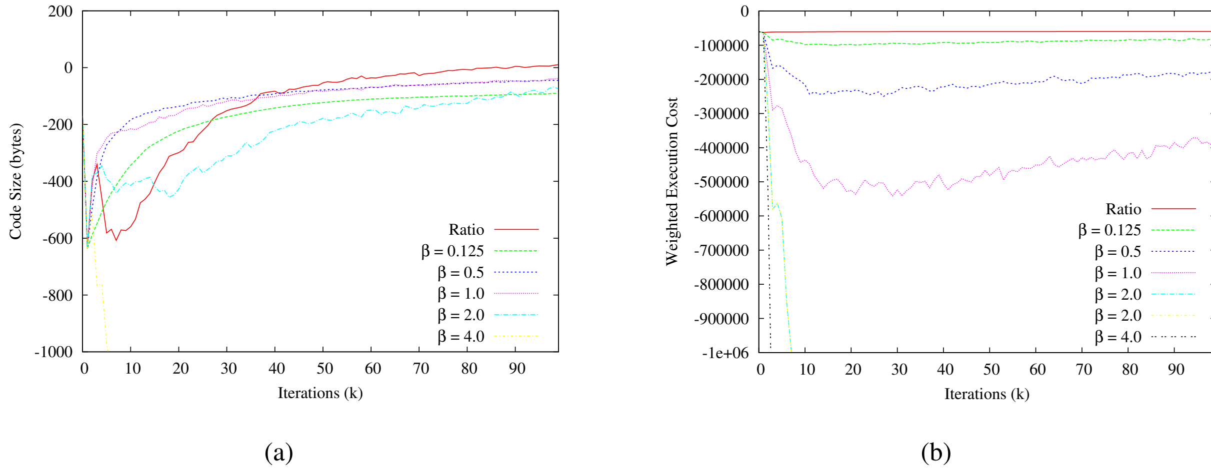
$$\theta_{k+1} = \frac{B_{freq}a}{k+b}$$

where B_{freq} is the execution frequency of a block under consideration. With this rule, the price modifications are proportional to the magnitude of the costs in the same block. As can be seen from Figures 6.6(c) and 6.6(d), this modification is necessary to obtain reasonable lower bounds when optimizing for performance. Just as with the code size case, a larger initial step size results in a greater initial drop in value. However, step update rules with larger initial prices perform better compared to the code size metric. Since the costs of the weighted execution costs performance metric are larger than those of the code size metric, prices must be correspondingly larger to have the same effect. We consider the B_{freq}/k rule to have the best overall convergence behavior.

Newton's method for solving systems of nonlinear equations can be adapted to define a step update rule:

$$\theta_{k+1} = \frac{\beta_k \left[L_{UB} - L(w,v)\right]}{||g_k||^2}$$

where $||g_k||$ is the Euclidean norm of the gradient, L_{UB} is an upper bound on the value of the Lagrangian relaxation, and β_k is a constant. If L_{UB} is the optimal value, L^*, and β_k is between 0

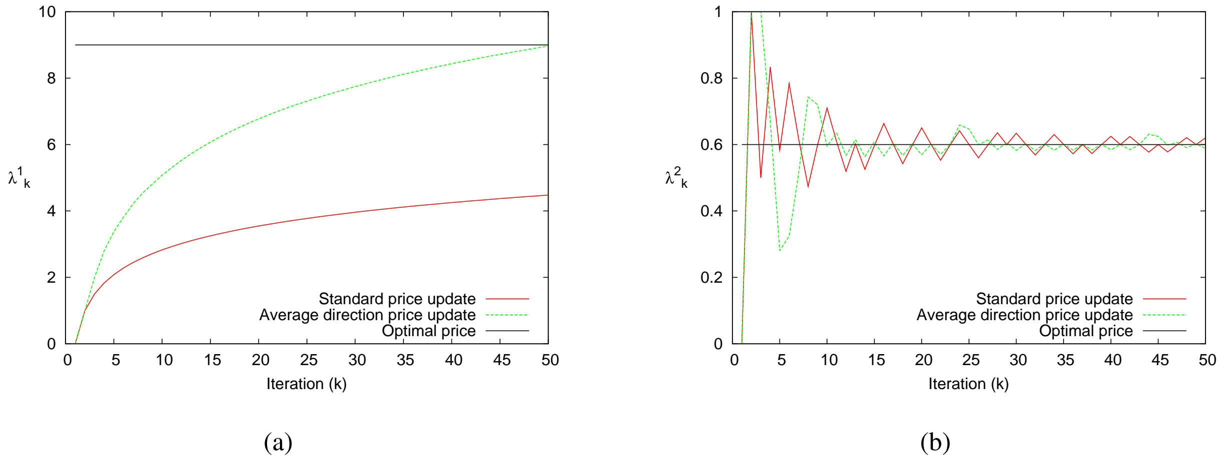


(a) (b)

Figure 6.7: Convergence of the subgradient optimization with the Newton's method step update rule when optimizing for (a) code size and (b) performance. Results are shown for a single function, quicksort. The optimal value is computed and used within the step update rule as the upper bound.

and 2, then this step update rule is guaranteed to converge [94]. In practice, L_{UB} does not equal L^* and the value of β_k is reduced until the algorithm appears to be converging. As with the ratio rule, when optimizing performance, we multiply the step size by the execution frequency of the current basic block.

We explore the behavior of the Newton's method step update rule applied to the Lagrangian relaxation of the global MCNF problem in Figure 6.7. We consider a single function, quicksort, for which we calculate the optimal values of the global MCNF problem when optimizing for code size and performance. We use these optimal values within the step update rule in the place of L_{UB}. As expected, values of β greater than 2 result in divergent behavior. This means that if $\beta_k > \frac{2L^*}{L_{UB}}$ then the algorithm will fail to converge and generate exceedingly poor lower bounds. However, if β_k is too small, the algorithm will converge very slowly, as demonstrated by the $\beta = 0.125$ case in Figure 6.7. In fact, the convergence behavior for all choices of β for the Newton's method rule is inferior to that of the simple ratio rule. For both step update methods the step size can be improved by performing a line search in the direction of the subgradient to find an optimal step size. However, line search techniques are computationally expensive, and several search-less iterations can be performed in the same time it takes to perform one iteration with a search. In practice, avoiding line search techniques has been shown to be the most effective



(a) (b)

Figure 6.8: Example price behavior using different price update strategies. The average direction strategy (a) accelerates price movement and (b) dampens oscillating price behavior.

strategy [90]. Given the generally better behavior of the ratio rule and the difficulty in selecting appropriate values for β and L_{UB}, we default to using the ratio rule when performing subgradient optimization.

6.2.3 Price Update

The *standard price update strategy* functions similarly to gradient descent methods and modifies the prices in the direction of the subgradient:

$$\lambda_{k+1} = \lambda_k + \theta_k g_k$$

Alternative price update strategies [9, 29, 90, 119, 120] are inspired by conjugate gradient descent methods [107] and use both the previous direction, d_{k-1} and current subgradient, g_k, to update the prices:

$$d_k = g_k + \Psi_k d_{k-1}$$
$$\lambda_{k+1} = \lambda_k + \theta_k d_k$$

The behavior of these strategies is determined by the deflection parameter, Ψ_k. We consider the computationally simple and problem-instance independent *average direction price update strategy* [120] where

$$\Psi_k = \frac{||g_k||}{||d_{k-1}||}$$

		Standard price update				Average direction price update						
k	θ_k	λ_k^1	λ_k^2	g_k^1	g_k^2	λ_k^1	λ_k^2	g_k^1	g_k^2	Ψ_k	d_k^1	d_k^2
1	1.000	0.000	0.000	1.0	1.0	0.000	0.000	1.0	1.0	0.000	1.000	1.000
2	0.500	1.000	1.000	1.0	-1.0	1.000	1.000	1.0	-1.0	1.000	2.000	0.000
3	0.333	1.500	0.500	1.0	1.0	2.000	1.000	1.0	-1.0	0.707	2.414	-1.000
4	0.250	1.833	0.833	1.0	-1.0	2.805	0.667	1.0	-1.0	0.541	2.307	-1.541
5	0.200	2.083	0.583	1.0	1.0	3.381	0.281	1.0	1.0	0.510	2.176	0.214
6	0.167	2.283	0.783	1.0	-1.0	3.817	0.324	1.0	1.0	0.647	2.407	1.139
7	0.143	2.450	0.617	1.0	-1.0	4.218	0.514	1.0	1.0	0.531	2.278	1.605
8	0.125	2.593	0.474	1.0	1.0	4.543	0.743	1.0	-1.0	0.507	2.156	-0.186
9	0.111	2.718	0.599	1.0	1.0	4.813	0.720	1.0	-1.0	0.653	2.409	-1.121
10	0.100	2.829	0.710	1.0	-1.0	5.080	0.595	1.0	1.0	0.532	2.282	0.403

Table 6.1: Example price behavior using different price update strategies.

With this choice of Ψ_k, the resulting direction, d_k, bisects the current gradient, g_k, and the previous direction d_{k-1}, and so can be considered an "average" direction.

The average direction strategy has the dual advantages of accelerating movement in a consistent direction and smoothing out oscillations in direction. These two properties are illustrated in Table 6.1 where we consider a simplified problem with two prices, λ^1 and λ^2. In this example, the gradient is taken to be 1 if the current price is less than the optimal value and -1 if it is greater. The optimal values of λ^1 and λ^2 are set to 9 and 0.6, respectively. Both prices are initialized to zero and the ratio rule is used to calculate the step size. Starting at zero, λ^1 consistently moves in a positive direction until it reaches its optimal value of 9. As shown in Table 6.1 and Figure 6.8(a), the average direction strategy acts as an accelerator since the combination of the previous direction and the current gradient is larger than the current gradient. In constrast, λ^2 is initialized close to its optimal value and the prices found by subgradient optimization oscillate around the optimal value of 0.6. As shown in Table 6.1 and Figure 6.8(b), the average direction strategy dampens these oscillations and generates values of λ_k^2 that are closer to the optimal value on average.

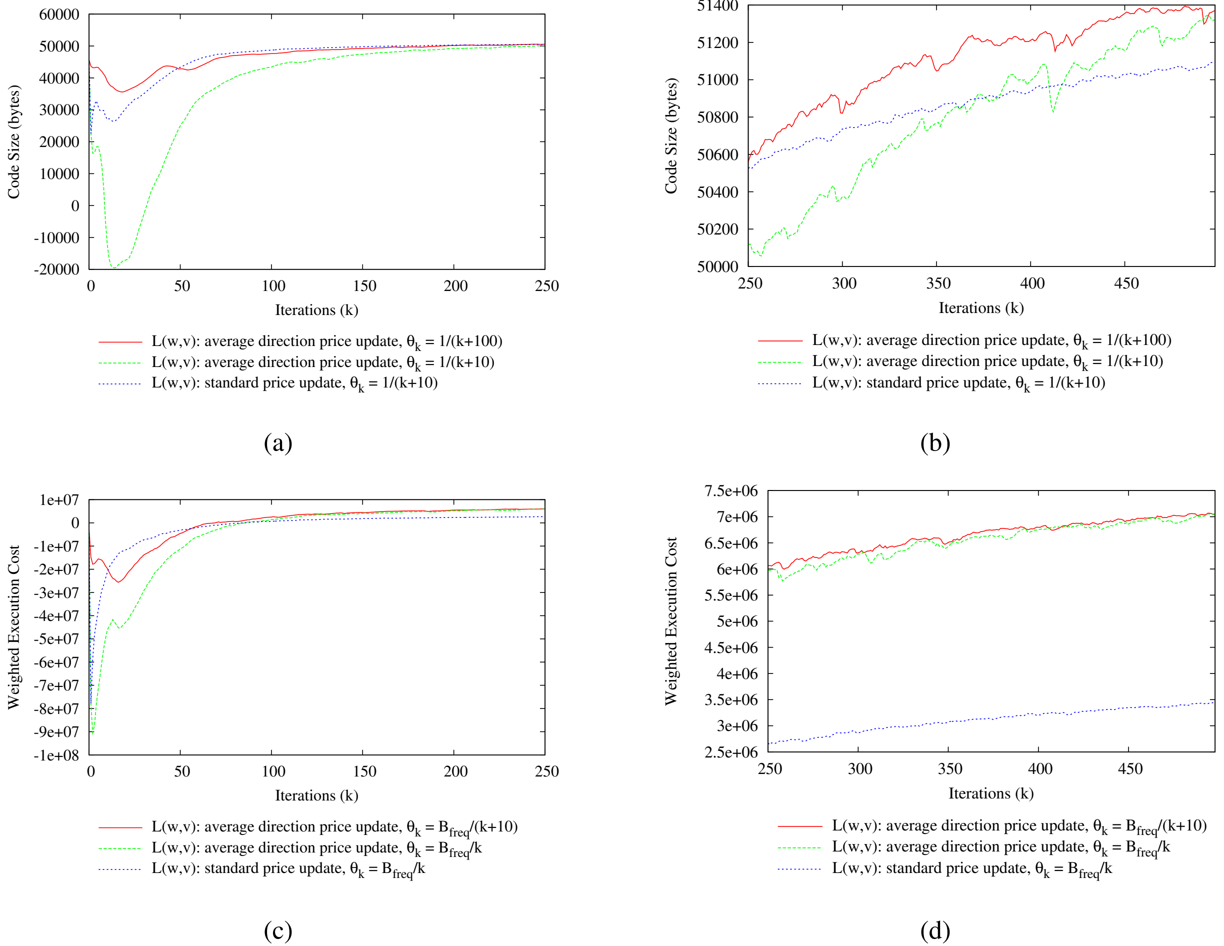


Figure 6.9: Convergence of subgradient optimization with different price update strategies when optimizing for (a)(b) code size and (c)(d) performance. We show the initial and successive behavior separately to magnify the behavior of the later.

The behavior of the average direction price update strategy is compared with the standard update strategy in Figure 6.9. The average direction strategy results in a significant initial price drop. The accelerating effect of using the average direction is counter-productive when the prices are first warming up. However, using a ratio rule that generates smaller initial step sizes counters this negative effect. Using a smaller initial step size, the average direction strategy is competitive initially with the standard update strategy (Figures 6.9(a) and 6.9(c)). Both small and large initial step sizes eventually generate larger values as the algorithm converges (Figures 6.9(b) and 6.9(d)). We use the average direction strategy with a smaller initial step size in our default implementation of subgradient optimization.

```
1: function CALCULATENODEPRICEWEIGHT( nodes layer, node n)
2:     allowedWeight ← ∞  ▷ largest price that won't change flow currently allocated to node
3:     neededWeight ← 0   ▷ smallest price that will push shortest paths to current allocation
4:     for v ∈ node.variables do
5:         ncost ← costOfShortest(v, n)    ▷ sum of shortest path to and shortest path from n
6:         mindiff ← ∞                     ▷ min distance between ncost and next highest cost node
7:         for m ∈ layer do
8:             mcost ← costOfShortest(v, m)
9:             if v ∈ m.flowVars then                  ▷ v is allocated to m in current allocation
10:                allocCost ← mcost
11:            else if mcost > ncost then
12:                if mcost − ncost < mindiff then
13:                    mindiff ← mcost − ncost
14:        if allocCost = ncost then                   ▷ current allocation has this cost
15:            if mindiff < allowedWeight then
16:                allowedWeight ← mindiff     ▷ disallow price that would change allocation
17:        else if allocCost > ncost then    ▷ shortest path for v is less than current allocation
18:            if allocCost − ncost > neededWeight then
19:                neededWeight ← allocCost − ncost  ▷ want to increase price on node for v
20:    if neededWeight > allowedWeight then
21:        return allowedWeight
22:    else
23:        return neededWeight
```

Listing 6.1: CALCULATENODEPRICEWEIGHT *Given a node* n *within a given* $layer$ *of an allocated network, compute a price that will push flows towards the current solution.*

6.2.4 Price Initialization

The initial lower bound found in the first iteration of subgradient optimization is determined exclusively by the initial set of prices. We consider using an existing allocation to set the initial prices. Ideally, given a set of feasible flows, x, we would be able to find a set of prices, (w, v), such that x is a set of shortest paths in the priced network. Since this property implies that $L(x, w, v) = L^*$, it is not possible to generate such a set of prices for an arbitrary flow vector. However, given a flow vector, we can generate prices that bias flows in the priced network towards the given flow vector.

The function $calculateNodePriceWeight$, shown in Listing 6.1, calculates a price for a node, n, that biases shortest path calculations toward the current allocation. The algorithm computes the cost of the flow of each variable through the node n (the sum of the shortest path to and

```
1: procedure PRICEINITIALIZATION( GlobalMCNF f)
2:     allocate(f)                                         ▷ heuristic initial prices
3:     computeShortestPaths(f)
4:     for BB ∈ basicBlocks(f) do
5:         for layer ∈ BB do
6:             allocateLayer(layer)                        ▷ assignment initial prices
7:             for n ∈ layer do
8:                 n.price ← n.price + calculateNodePriceWeight(layer, n)
9:             updateNextLayerPaths(layer)
```

Listing 6.2: PRICEINITIALIZATION *Initialize prices based on some heuristic allocation.*

shortest path from the node). If this cost is less than the cost of flow through the node the variable is currently allocated to, the current allocation of the variable is not along a shortest path. If the price of node n is increased sufficiently, the shortest path of the variable will be redirected to a different node. Ideally, this node is the node the variable is allocated to in the current allocation. The amount that the price of node n has to be increased in order to redirect shortest paths to the nodes of the current allocation is the $neededWeight$ value calculated by $calculateNodePriceWeight$ (line 19).

If the $neededWeight$ value is too large, variables allocated to node n in the current allocation will be redirected. In order to prevent this, $calculateNodePriceWeight$ also computes the value $allowedWeight$ (line 16). If a variable is allocated to node n, or to a node with the exact same flow cost as node n, then the price of n should not be increased to the point that the flow of the variable is redirected. The most the price can be incremented is the difference between the flow cost of the variable through n and the next most expensive node. The final weight returned by $calculateNodePriceWeight$ is the minimum of the $allowedWeight$ and $neededWeight$ values.

The general price initialization algorithm, $priceInitialization$, is shown in Listing 6.2. The algorithm iterates over each layer of the network. For every node in a layer, a price weight is computed and applied (line 8). After each layer is processed, the shortest paths into the next layer are updated using the newly computed prices. This algorithm only updates capacity constraint prices, w. Boundary constraint prices, v, remain initialized to zero.

We consider two algorithms for generating initial prices. *Heuristic initial prices* are computed using the result of the hybrid heuristic allocator (Section 5.4) as the allocation used by the

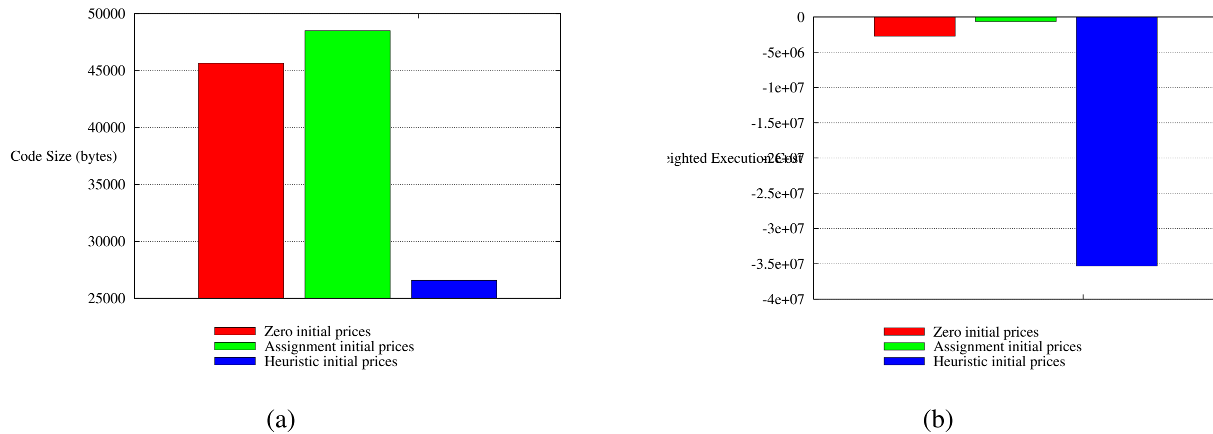


Figure 6.10: Effect of price initialization on the initial lower bound when optimizing for (a) code size and (b) performance. Larger lower bounds are better.

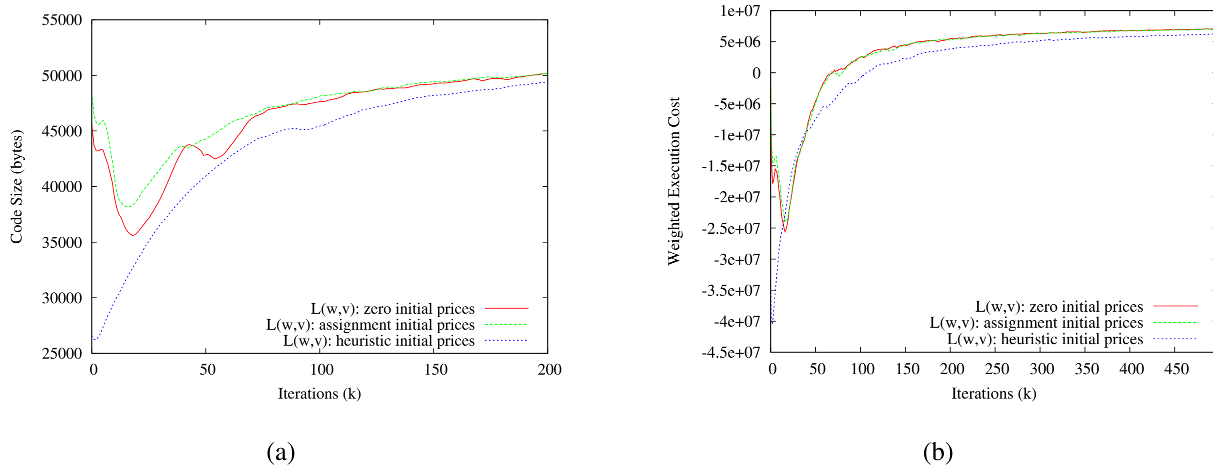


Figure 6.11: Convergence of subgradient optimization with different price initialization techniques when optimizing for (a) code size and (b) performance.

$priceInitialization$ algorithm (line 2). *Assignment initial prices* are generated from an allocation of variables to nodes that ignores network constraints. The assignment allocation algorithm ignores the network constraints of the problem and treats each layer of the network as a classical assignment problem [6] which can be solved optimally in polynomial time. The classical assignment problem finds the minimum cost matching between two sets. In our case, we match variables to nodes and the cost of a match is determined by the cost of the flow of a variable through a node. Since the costs of the assignment problem depend on the shortest path values,

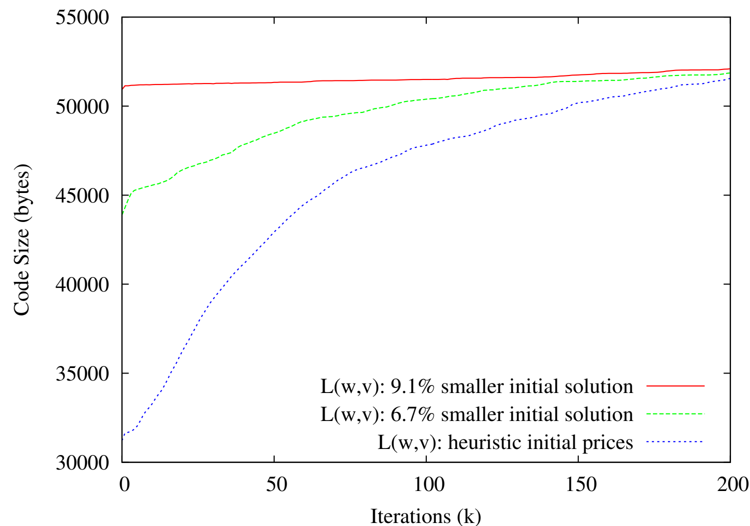


Figure 6.12: Convergence of heuristic price initialization with different initial allocations. The lower bound for the 401.bzip2 benchmark is shown when optimizing for code size. A better initial allocation results in substantially better convergence.

the allocation of each layer is computed as the $priceInitialization$ algorithm iterates over the layers (line 6) and the prices are updated (line 9).

The heuristic initial pricing algorithm biases the initial prices toward a specific valid register allocation that may be far from optimal. The assignment initial pricing algorithm does not bias the prices toward a legal allocation, but does initialize the prices to reflect the register pressure at each program point and does not depend on the quality of a heuristic allocation.

As shown in Figure 6.10, assignment initial prices generate a significantly better initial lower bound than both heuristic initial prices and zero initial prices. Heuristic initial prices generate poor lower bounds, especially when optimizing for performance. This is likely due to the lower quality solutions found by the heuristic allocators when optimizing for performance. We illustrate the dependence of the heuristic initial pricing algorithm on the quality of the initial allocation in Figure 6.12. A high quality initial allocation results in substantially better convergence behavior.

Interestingly, as shown in Figure 6.11, despite generating substantially better initial lower bounds, price initialization does not eliminate the initial price warmup phase in the subgradient optimization algorithm. However, since assignment initial prices generate significantly better initial lower bounds we use this price initialization strategy by default.

6.2.5 Summary

We implement several improvements to the standard subgradient optimization algorithm. When updating prices we use balanced shortest paths to calculate the flow vector. This algorithm substantially outperforms a standard shortest path algorithm. We find that the ratio rule for calculating the step size, θ_k, is more effective than the Newton's method rule. When optimizing for performance, we use a step size that is weighted by the execution frequency of the current basic block. We consider both the standard subgradient update strategy and the average direction price update strategy with a reduced initial step size. We implement the average direction strategy by default since it converges faster in the limit. We initialize prices by computing prices at each layer in the network that bias the shortest path solutions towards a solution to the classical assignment problem at each layer.

6.3 Progressive Register Allocation

We combine Lagrangian relaxation and subgradient optimization with our heuristic solution techniques to create a progressive register allocator. Our progressive solver first finds an initial solution in the unpriced network using a heuristic allocator. Then, in each iteration of the subgradient optimization algorithm, the current prices are used to find another solution. We modify the heuristic allocators to compute shortest paths using edge and boundary prices in addition to edge costs. Global information, such as the interference graph, is only used to break ties between identically priced paths. Otherwise, the heuristic allocators rely on the influence of the prices in the network to account for the global effect of allocation decisions.

The heuristic allocators attempt to build a minimum cost feasible solution to the global MCNF problem. If the algorithm finds a solution equal in cost to the relaxed solution and this solution obeys the complementary slackness condition, then the solution is provably optimal. When selecting among similarly priced allocation decisions, we increase the likelihood that the solution will satisfy the complementary slackness condition by favoring allocations with the lowest unpriced cost.

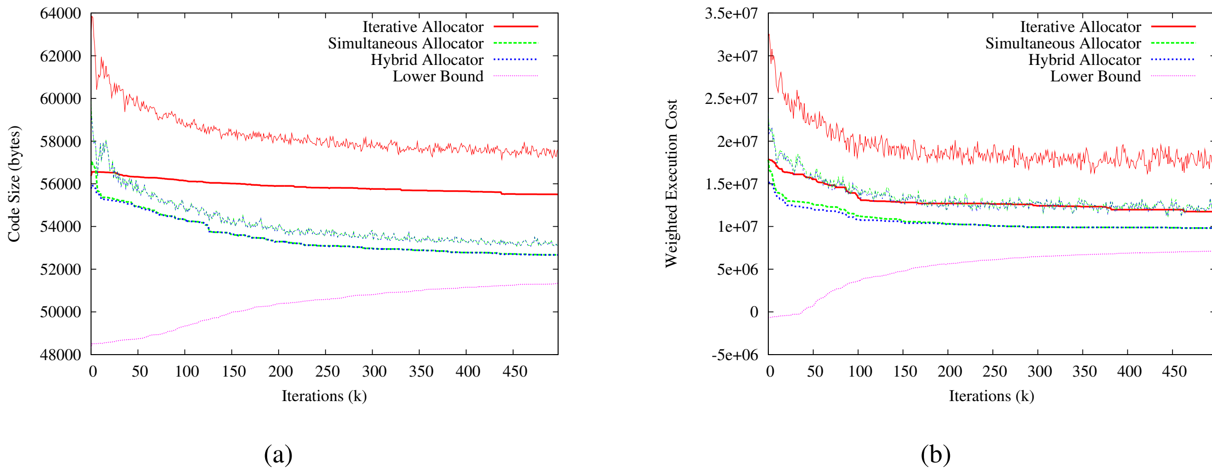


Figure 6.13: The behavior of three heuristic allocators when incorporated in a progressive register allocation framework when optimizing for (a) code size and (b) performance. Both the cumulative value of the current solutions found at an iteration (top line) and the cumulative value of the best found solutions at an iteration (bottom, monotonic line) are shown.

The behavior of each of our three heuristic allocators when incorporated in this progressive register allocation framework is shown in Figure 6.13. For each allocator both the aggregate cost of the solutions found at an iteration is shown (top line) as well as the aggregate best known solution (bottom monotonic decreasing line). The cost of the allocation is the post-regalloc code size. All three allocators progressively improve upon the quality of allocation as the prices converge.

The iterative allocator performs substantially worse than the simultaneous allocator. The iterative allocator greedily allocates with no backtracking and the quality of allocation it generates is largely dependent upon the heuristics used by the allocator when making allocation decisions. The introduction of prices into the network reduces the number of tie breaking decisions, lessening the importance of tie breaking heuristics. As a result, the quality of the allocation depends mostly on the ability of the prices to push the shortest path computations towards potentially optimal allocations. Since the prices are not fully converged, in many cases the shortest paths will not correspond to potentially optimal allocations. Since the iterative allocator greedily allocates, it cannot undo a poor allocation decision caused by inexact prices. The lack of flexibility and reliance on heuristics mean the iterative allocator does make effective use of the prices.

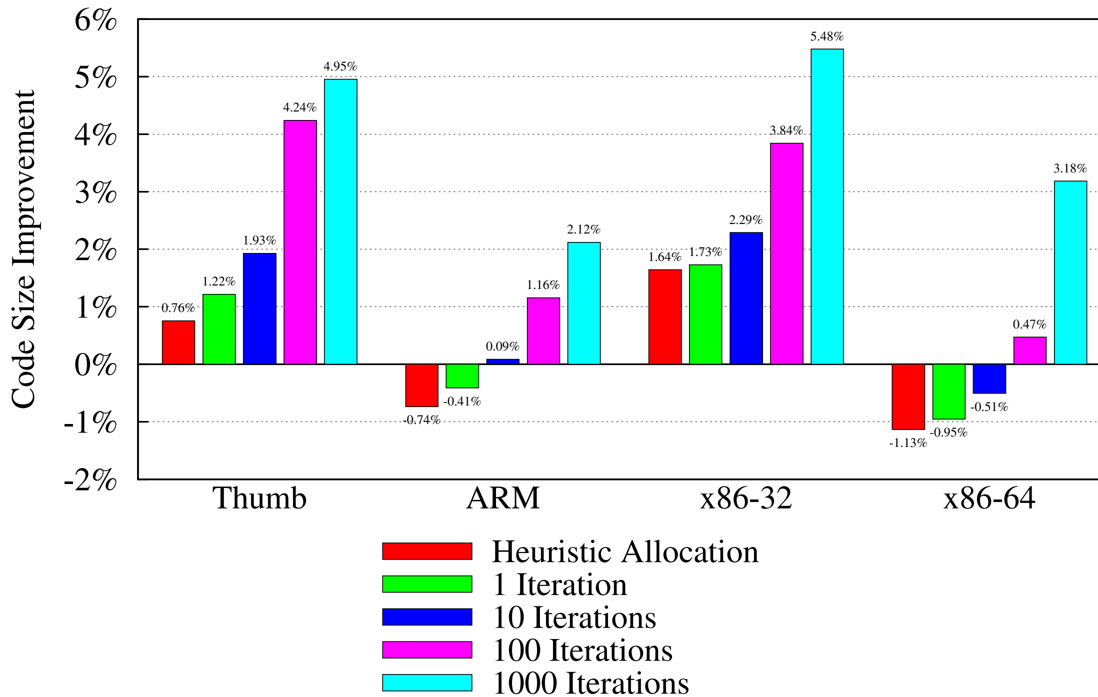


Figure 6.14: Average code size improvement of the progressive allocator.

In contrast, the simultaneous allocator is more flexible and can undo poor allocation decisions by performing evictions. The simultaneous allocator is also less dependent on tie breaking heuristics. As shown in Figure 6.13, the simultaneous allocator both finds better solutions than the iterative allocator and progressively improves the quality of allocation at a faster rate than the iterative allocator. In fact, as the prices converge, the behavior of the simultaneous allocator is almost strictly better than that of the iterative allocator. This is seen by the lack of improvement of the hybrid allocator compared to the simultaneous allocator. As a result, although we use the hybrid allocator to find the initial solution, we only use the simultaneous allocator for subsequent iterations of our progressive register allocation framework.

6.3.1 Code Quality: Size

Our expressive global MCNF model is well suited for the code size metric since it accurately represents the complexities of the target ISA. The average improvement in code size is shown in Figure 6.14, and the improvements for individual benchmarks are shown in Figures 6.15 (x86

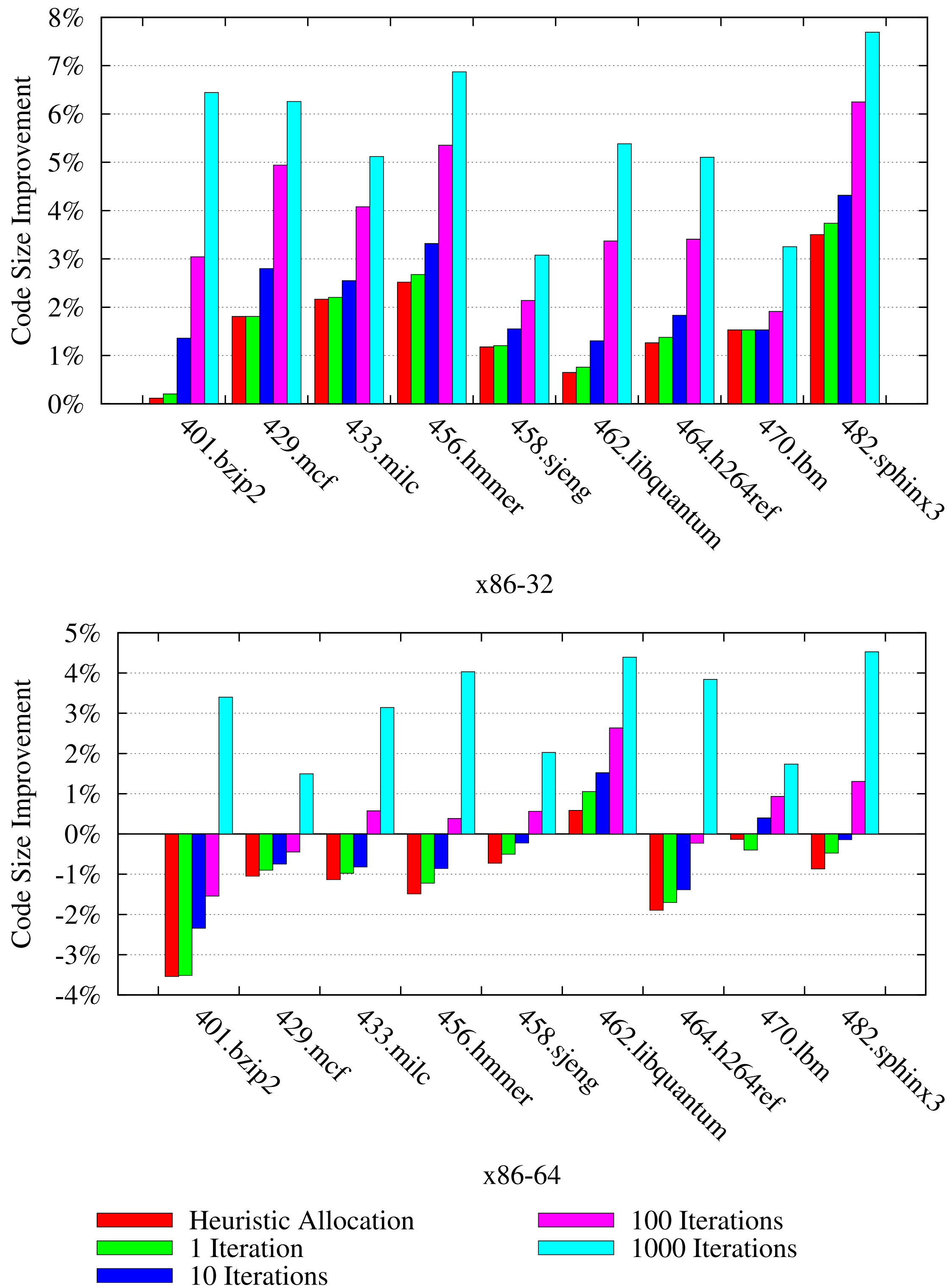


Figure 6.15: Code size improvement of the progressive allocator.

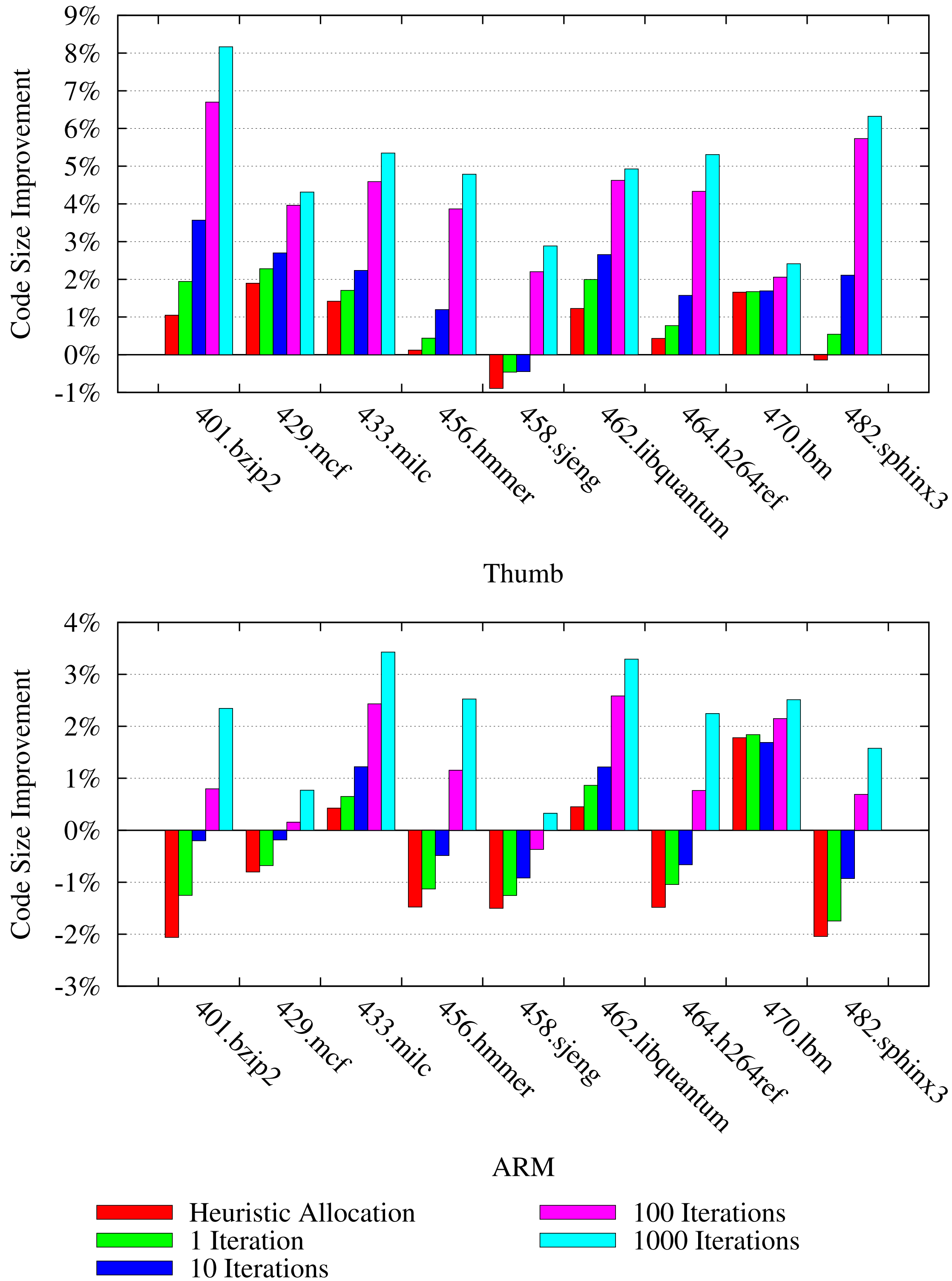


Figure 6.16: Code size improvement of the progressive allocator.

family) and 6.16 (ARM family). In all cases, as more time is allotted for compilation, the average improvement in code size increases. Our progressive allocator performs best on the register-limited x86-32 and Thumb instruction set architectures. For x86-32, we achieve an initial average code size improvement of 1.64% and, after 1000 iterations, an average improvement of 5.48%. The initial improvement for Thumb is 0.76% with an improvement of 4.95% after 1000 iterations. Individual benchmarks improve by as much as 7%.

The code size improvements are more moderate when targeting the x86-64 and ARM instruction set architectures. Architectures with more registers benefit less from the program point precision of our detailed model. The presence of more registers reduces the need for spill code. As less spill code is needed, the difference between a simple spill-everywhere approach and optimal spill code generation shrinks. As a result, traditional heuristic allocators are reasonably effective. Our initial heuristic allocator does not produce an average code size improvement for either x86-64 or ARM. A positive average code size improvement is not achieved for ARM until 10 iterations while a positive improvement is not achieved for x86-64 until 100 iterations. The extra difficulty when optimizing for x86-64 likely stems from the complexity and imprecision of our model of this architecture (Section 3.6). However, all benchmarks demonstrate a code size improvement within 1000 iterations for both architectures. As expected, given the greater number of registers, the improvements are not as great as with the register-limited architectures. After 1000 iterations, there is an average improvement of 3.18% for x86-64 and 2.12% for ARM. It is likely that our initial allocation would improve substantially if we used a heuristic designed for an architecture with plentiful registers. For instance, simply by projecting the LLVM allocation onto our model we could improve our initial result.

6.3.2 Code Quality: Performance

Our progressive allocator can substantially reduce the number of memory operations, as shown in Figure 6.17. On the register-limited x86-32 architecture, we reduce the total number of executed memory operations by 7.3% initially and by more than 20% after 1000 iterations. On individual benchmarks, shown in Figure 6.19, a reduction of nearly 40% is achieved on one benchmark, with all benchmarks exhibiting significant reductions and progressive improvement. In contrast,

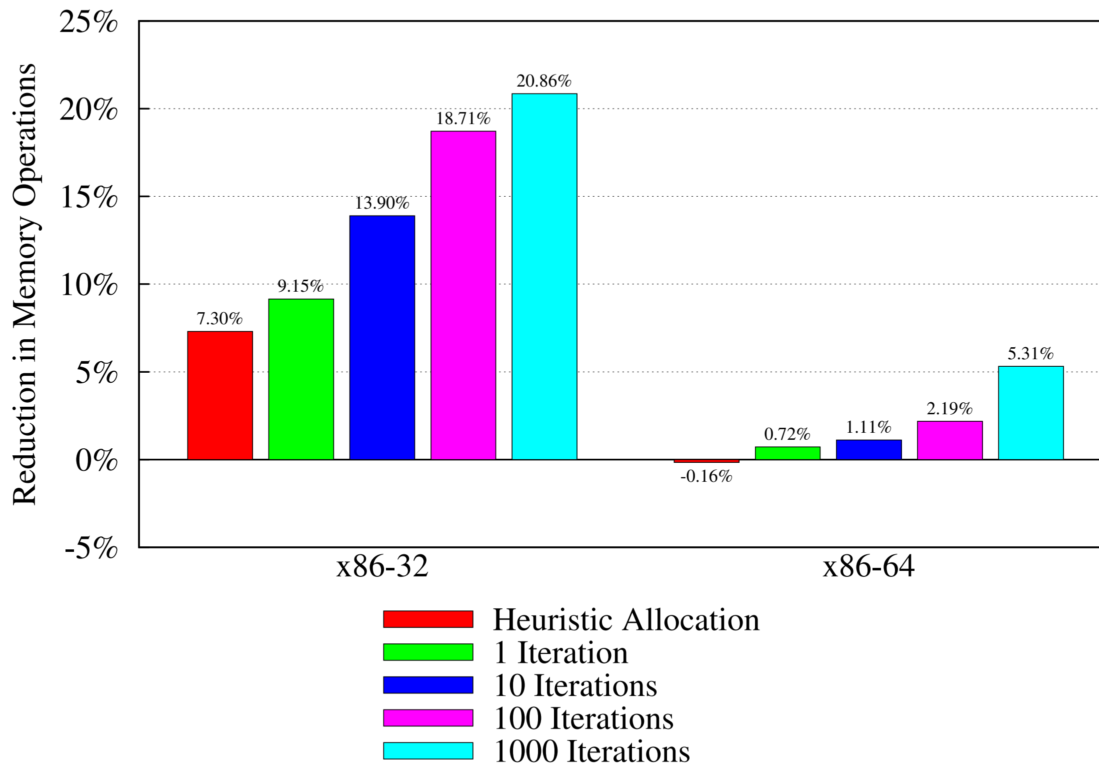


Figure 6.17: Average memory operation reduction of the progressive allocator.

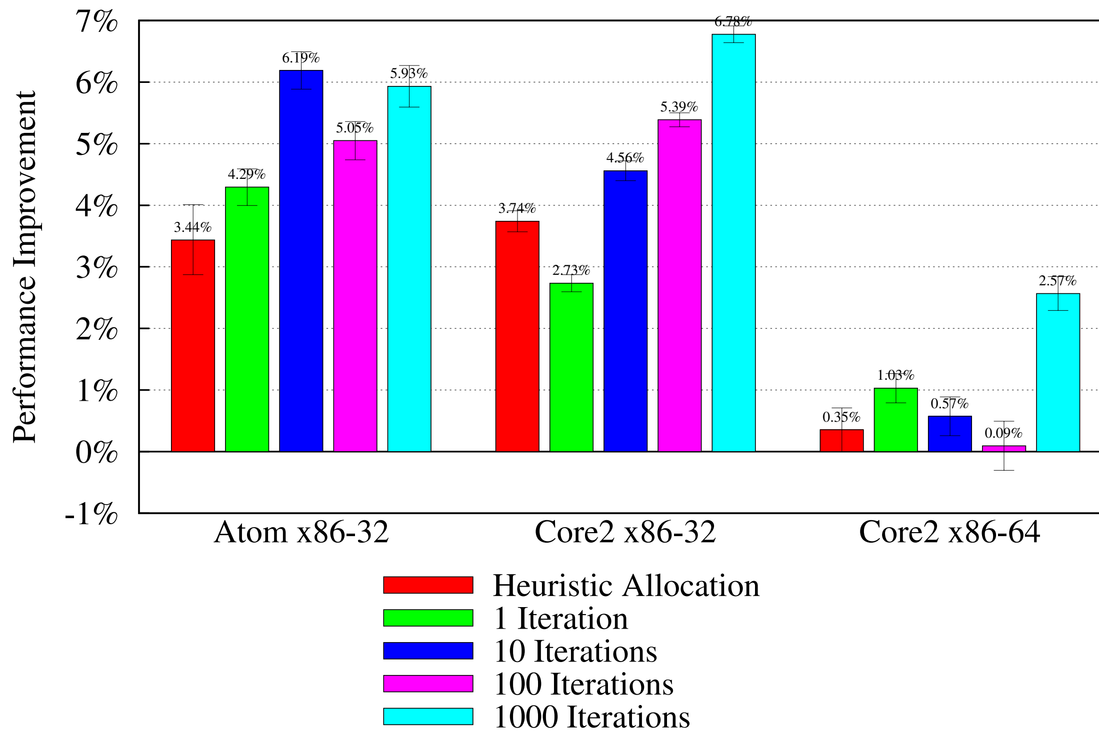


Figure 6.18: Average performance improvement of the progressive allocator.

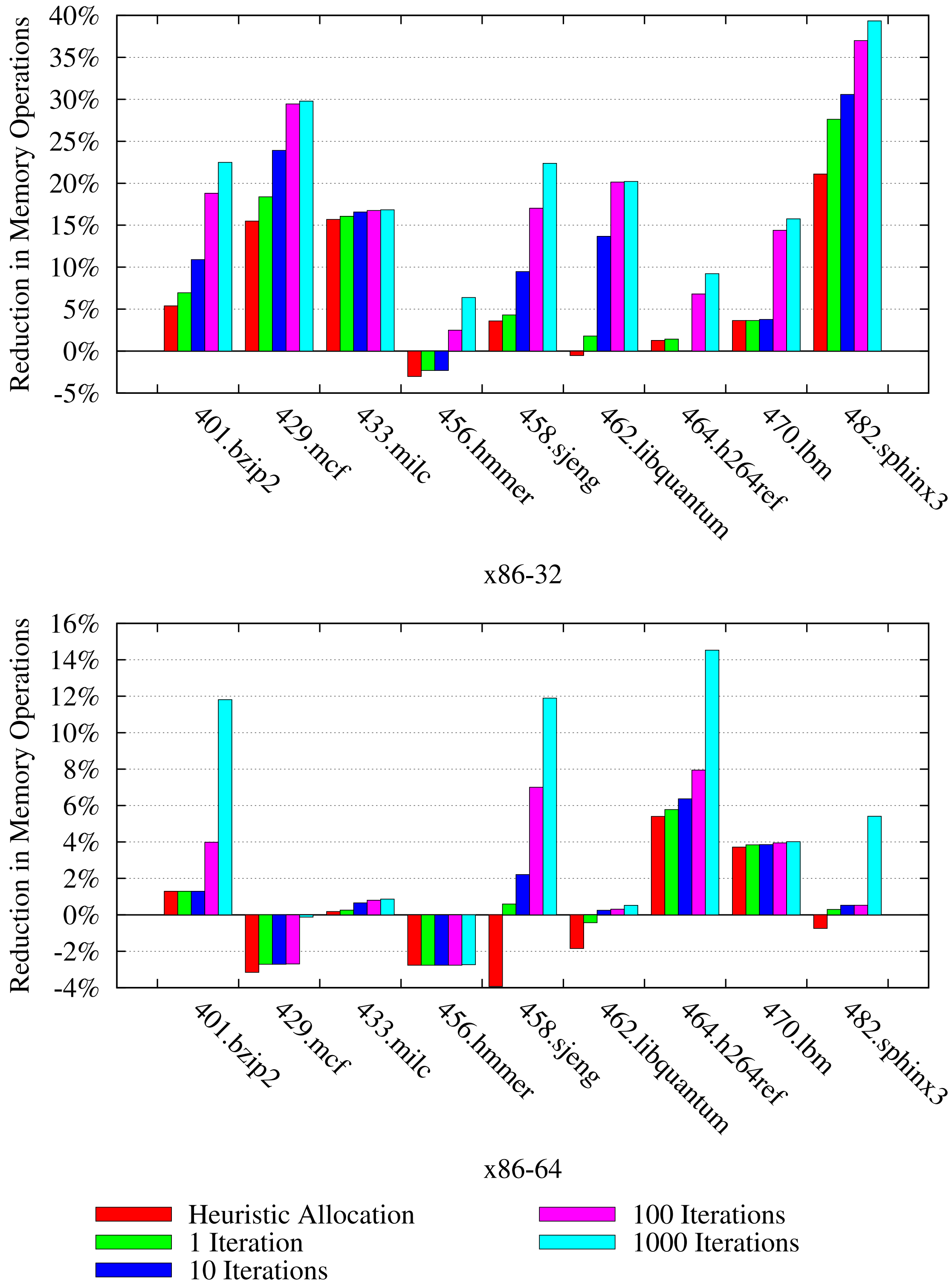


Figure 6.19: Memory operation reduction of the progressive allocator.

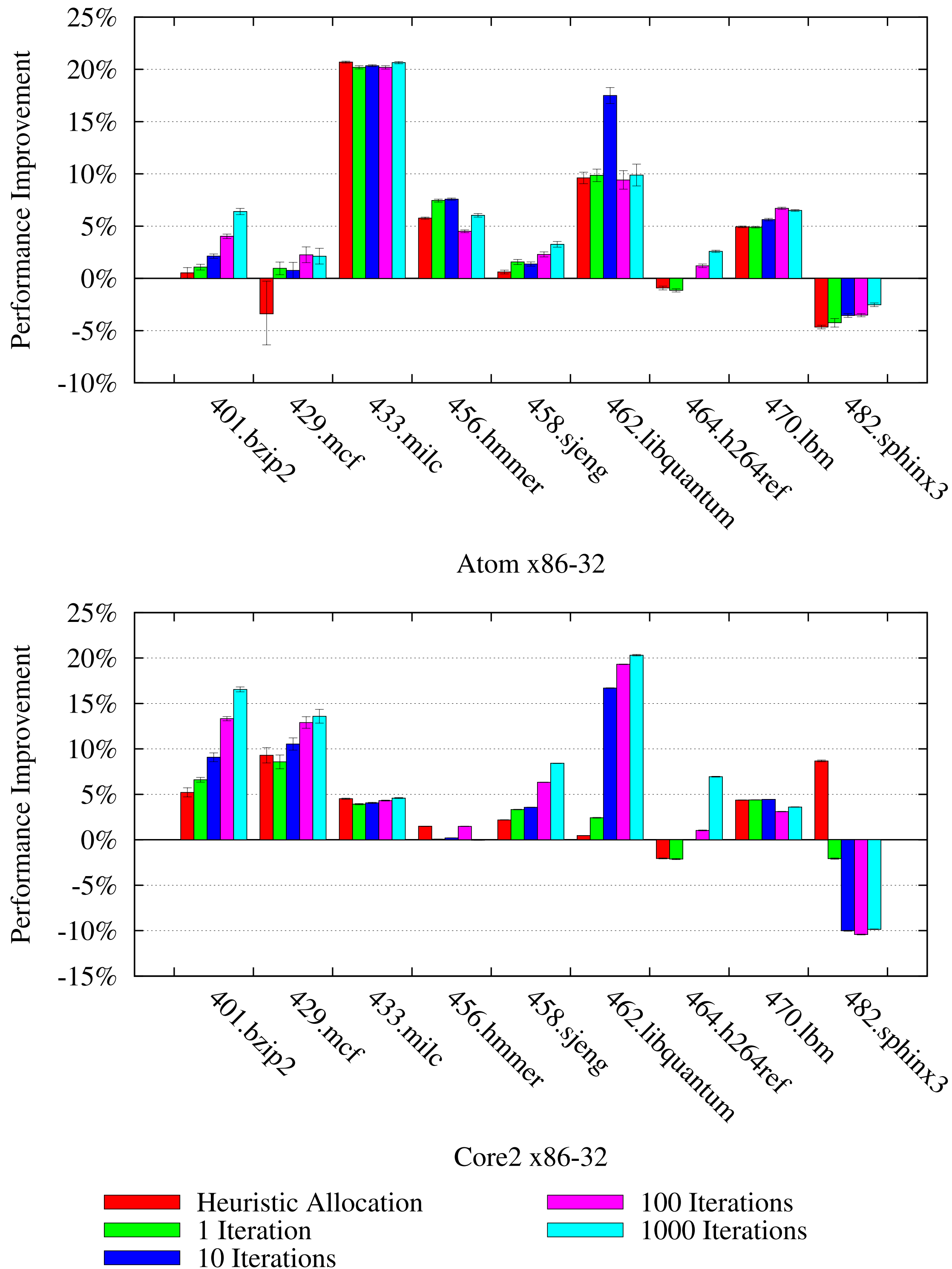


Figure 6.20: Code performance improvement of the progressive allocator for the x86-32 ISA on the Atom and Core 2 microarchitectures.

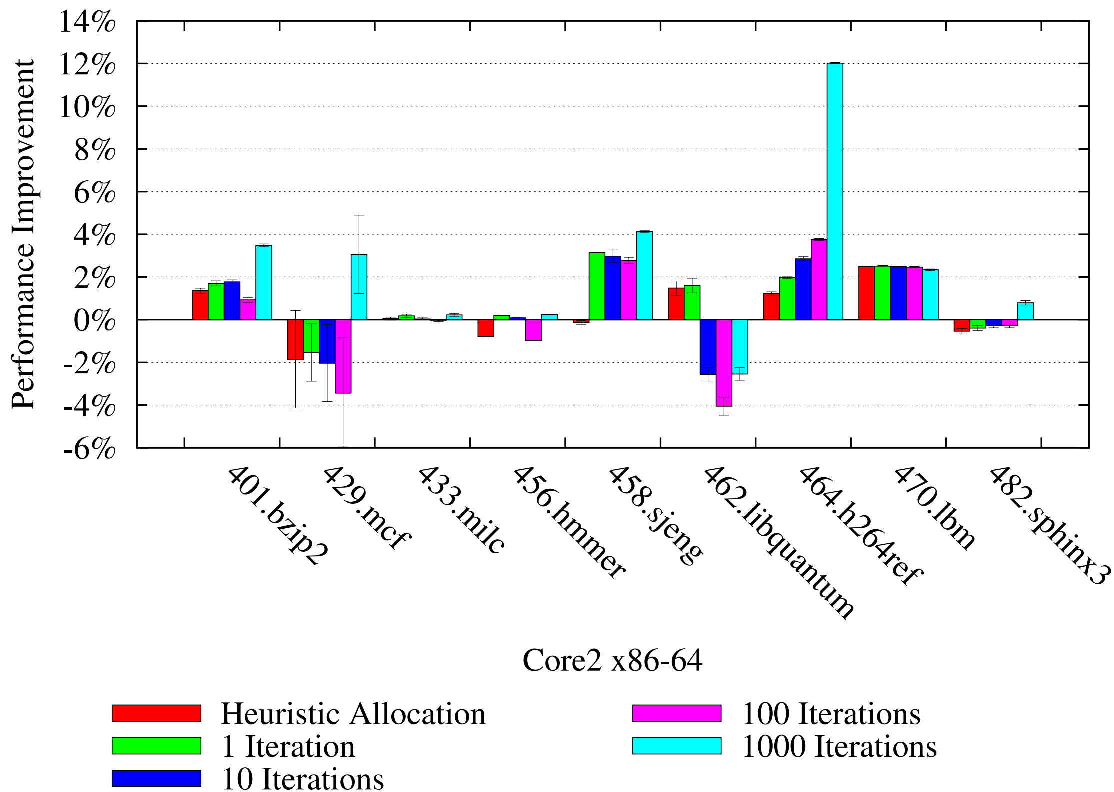


Figure 6.21: Code performance improvement of the progressive allocator for the x86-64 ISA on the Core 2 microarchitecture.

when targeting x86-64, our initial allocation results in an average reduction of -0.16% which improves to 5.31% after 1000 iterations. The larger register set of the x86-64 instruction set architecture reduces the benefit of spill code optimization. This effect is clear in the individual benchmark results (Figure 6.19), where several benchmarks exhibit little or no progressive improvement.

The average performance improvements of three microarchitectures are shown in Figure 6.18. The improvements are not strictly progressive, illustrating the lack of a tight correlation between the reduction in memory operations and performance. This is made further illustrated by the individual benchmark results shown in Figures 6.20 and 6.21. The benchmark with the greatest reduction in memory operations for x86-32, 482.sphinx3, actually exhibits a performance decrease both on the Atom and Core 2. However, in most cases, there is a positive correlation between memory operation reduction and performance, although the strength of this correlation varies across the benchmarks. Performance for the register-limited x86-32 instruction set

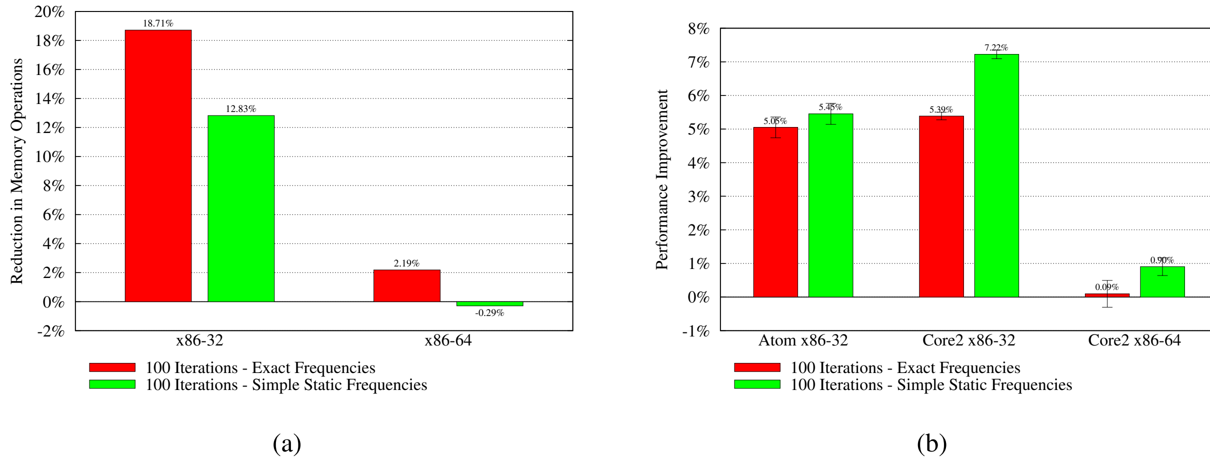


Figure 6.22: The effect on code quality of using exact block frequencies derived from profile data versus a simple static estimation of block frequencies.

architecture improves on average from 3% to 7%, while some individual benchmarks improve by as much as 20%. Although when targeting x86-64, many benchmarks exhibit no significant change in performance and the average performance improvements range from 0% to 2.5%, one benchmark does exhibit a 12% improvement.

Despite the in-order nature of the Atom, there is not a significant difference between the average performance improvements on the Atom compared to the Core 2. The individual benchmark results, shown in Figure 6.20, demonstrate substantial differences, even though the same executable is evaluated on each architecture. These differences underscore the importance of microarchitectural features in determining performance. They also highlight the difficulty of constructing a precise performance metric when only the instruction set architecture is fully exposed to the compiler.

Execution Frequencies

The weighted execution costs metric used in our progressive allocator when targeting performance relies on estimations of basic block frequencies. In order to generate a highly accurate model, we use exact execution frequencies derived from profile data. An alternative, used by most register allocators including the baseline LLVM allocator, is to use simple static execution frequencies as described in Section 4.2.2. We show the effect of using these alternative execu-

tion frequencies in Figure 6.22 when executing 100 iterations of our progressive allocator. As expected, when the less accurate simple static costs are used in the model, fewer memory operations are removed (Figure 6.22(a)). However, despite executing more memory operations, the impact on performance is positive (Figure 6.22(b)) further illustrating the looseness of the correlation between the number memory operations executed and actual performance. This result strongly implies that exact execution frequencies and profile data are not necessary to achieve high quality code.

6.3.3 Optimality

Our progressive allocator keeps track of both the value of the best solution, val_{best}, and the best lower bound, $\max L(w,v)$. These values are used to compute an upper bound on the optimality of the allocation:

$$optimality\ bound = \frac{val_{best}}{val_{before_{RA}} + \max_k L(w^k, v^k)} - 1$$

The optimality bound provides compile-time feedback to the programmer about the quality of the allocation. For example, if the optimality bound is 1% when optimizing for code size, then the current allocation is at most 1% larger than the best possible solution. This allows the programmer to better evaluate the trade-off between compile-time and code quality.

The optimality bound calculation requires some estimate of the code quality prior to allocation. A pre-allocation code size estimate is straightforward to compute. The size of each instruction is computed as if all operands of the instruction are inexpensive registers. When targeting performance, the weighted execution cost metric is applied to the entire function. Every instruction is weighted by its execution frequency and all memory operations are further weighted by an additional multiplicative factor.

The optimality bound is a useful measure of the performance of our progressive allocator. As shown in Figure 6.23, when optimizing for code size and executing 1000 iterations of the progressive allocator, the majority of functions in our benchmark set have a provably optimal allocation and at least 95% are no more than 5% larger than the optimal solution. When optimizing for performance, Figure 6.24, the progressive allocator does not converge as quickly, but the

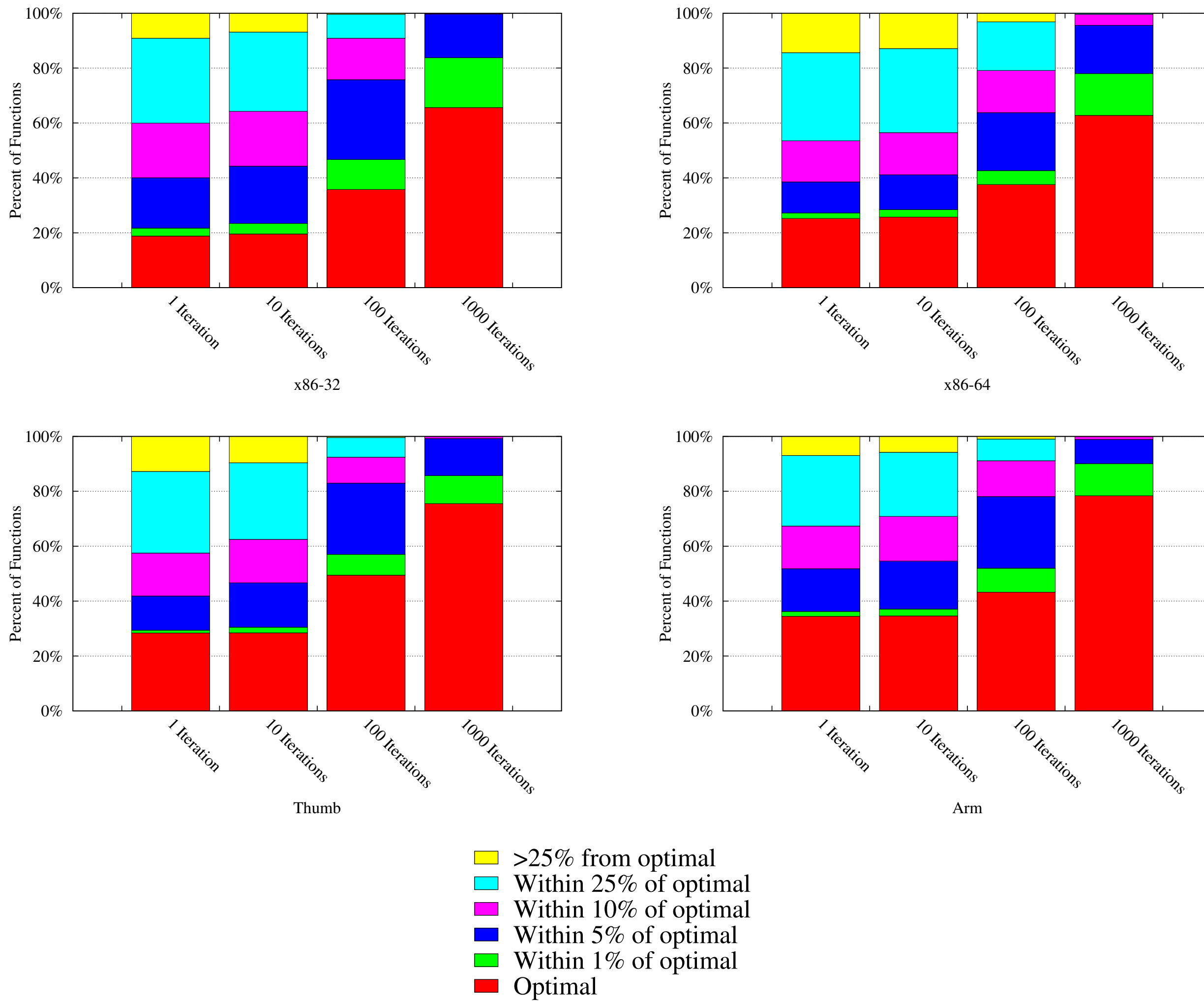


Figure 6.23: Optimality bounds of progressive allocator when optimizing for code size.

majority of functions are provably within 1% of optimal after 1000 iterations. These optimality measures demonstrate that our progressive allocator approaches the optimal solution as more time is allowed for compilation.

6.3.4 Compile Time

The time spent in each component of the progressive allocator relative to the default LLVM register allocator is shown in Figure 6.25. As with the heuristic allocators, we observe a substantial orders-of-magnitude slow down relative to the extended linear scan algorithm used within LLVM. Despite the orders of magnitude slowdown in register allocation, compiling with one

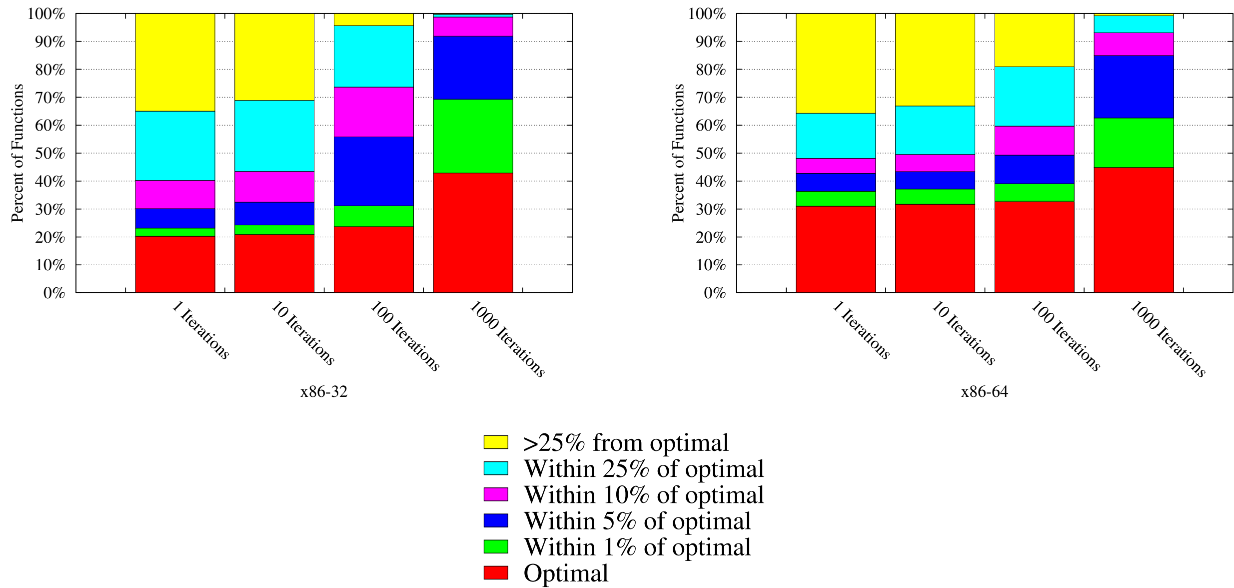


Figure 6.24: Optimality bounds of progressive allocator when optimizing for performance.

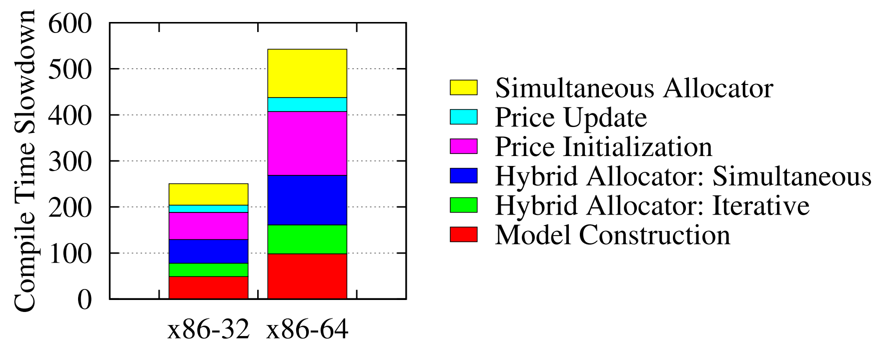


Figure 6.25: Break down of register allocation time slowdown of one iteration of the progressive allocator. The geometric mean across all benchmarks of the slowdown is shown. The total slowdown in compile time, not just register allocation, is 4x.

iteration of progressive allocation results in only a 4x slowdown in total compile time. The default LLVM allocator is designed for compile-time performance and is approximately 25 times faster than traditional graph coloring. Our progressive register allocation research framework is designed for flexibility, not for performance. Most importantly, our allocator is asymptotically more complex because of the high precision of our model. Values are computed and stored for every allocation class at every program point. Our detailed and expressive model is the key to our substantial improvements in code quality, but it also is the major factor in our two orders of magnitude slowdown relative to the LLVM allocator.

There are several optimizations that will improve the compile time of our progressive allocator. For example, we could replace our current graph data structure, which is implemented with structures and pointers, with a more efficient, cache-conscious data structure that is optimized for the global MCNF model. The shortest path algorithms, which contribute the most to the compile-time overhead of our allocator, are embarrassingly parallel. An efficient parallelization on modern many-core hardware would result in at least an order of magnitude improvement in compile-time. Alternatively, we can consider simplifications of the model that change the asymptotic behavior of our algorithms at the expense of models that do not map directly to an optimal register allocation.

6.4 Summary

In this chapter we have combined the heuristic allocators of the previous chapter, relaxation techniques, and subgradient optimization to create a progressive register allocator. Our progressive register allocator allows the programmer to explicitly manage the trade-off between compile-time and code quality. The initial allocation generated by our progressive allocator is competitive with the state-of-the-art LLVM extended linear scan allocator and, as more time is allowed for compilation, it generates allocations with substantially better code quality.

Chapter 7

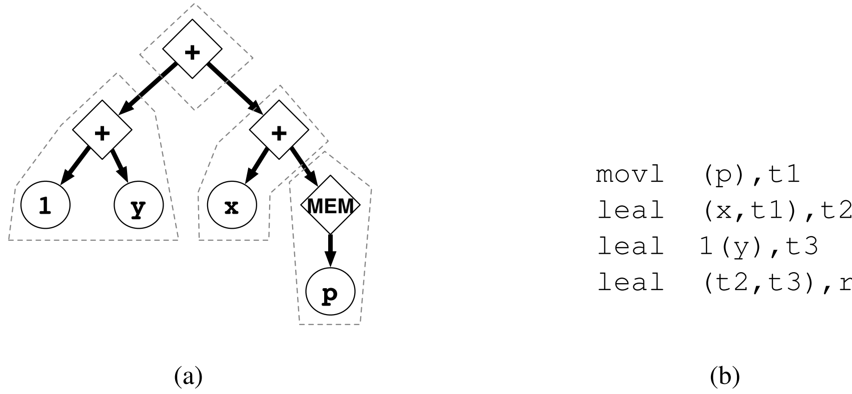
Near-Optimal Linear-Time Instruction Selection

In this chapter we consider the *instruction selection problem*. We model the problem using an expressive directed acyclic graph representation and present a novel algorithm for finding a near-optimal tiling in linear-time.

7.1 Problem Description and Hardness

The instruction selection problem is to find an efficient conversion from the compiler's target-independent intermediate representation (IR) of a program to a target-specific assembly listing. An example of instruction selection, where a tree-based IR is converted to x86 assembly, is shown in Figure 7.1. In this example, and in general, there are many possible correct instruction sequences. The difficulty of the instruction selection problem is finding the best sequence, where best may refer to code performance, code size, or some other statically determined metric.

Instruction selection is usually defined as finding an optimal tiling of the intermediate code with a predefined set of tiles. Each tile is a mapping from IR code to assembly code and has an associated cost. We consider the problem of tiling an expression DAG intermediate representation. Expression DAGs are more expressive than expression tree and linear intermediate representations since they explicitly model redundancies.




```
movl  (p),t1
leal  (x,t1),t2
leal  1(y),t3
leal  (t2,t3),r
```

(a) (b)

Figure 7.1: An example of instruction selection as a tiling problem. The shown tiling (a) corresponds to the assembly listing (b).

Given an expression DAG that represents the computation of a basic block and a set of architecture specific instruction tiles, we wish to find an optimal tiling of the DAG that corresponds to the minimum cost instruction sequence. The expression DAG consists of nodes representing operations (such as an add or load) and operands (such as a constant or memory location). We refer to a node with multiple parents as a *shared node*. The set of tiles consists of a collection of expression trees each with an assigned cost. Since we allow nodes within a tile to map to the same expression DAG node, essentially turning the expression tree tile into an expression DAG tile, there is no benefit in explicitly encoding tiles as expression DAGs. If a leaf of an expression tree tile is not an operand, it is assumed that the inputs for that leaf node will be available in a register.[1] Similarly, the output of the tree is assumed to be written to a register. A tile matches a node in the DAG if the root of the tile is the same kind of node as the DAG node and the subtrees of the tile recursively match the children of the DAG node. In order for a tiling to be legal the inputs of each tile must be available as the outputs of other tiles in the tiling. In order for a tiling to be complete all the root nodes of the DAG (those nodes with zero in-degree) must be matched to tiles. The optimal tiling is the legal and complete tiling where the sum of the costs of the tiles is minimized. More formally, we define an optimal instruction tiling as follows:

Definition Let K be a set of node kinds; $G = (V, E)$ be a directed acyclic graph where each node $v \in V$ has a kind $k(v) \in K$, a set of children $ch(v) \in 2^V$ such that $\forall_{c \in ch(v)} (v \rightarrow c) \in E$, and a unique ordering of its children nodes $o_v : ch(v) \rightarrow \{1, 2, ...|ch(v)|\}$; T be a set of tree

[1] These are unallocated temporaries, not actual hardware registers.

tiles $t_i = (V_i, E_i)$ where similarly every node $v_i \in V_i$ has a kind $k(v_i) \in K \bigcup \{\circ\}$ such that $k(v_i) = \circ$ implies $outdegree(v_i) = 0$ (nodes with kind $\circ$ denote the boundary of a tile and, instead of corresponding to an operation or operand, serve to link tiles together), children nodes $ch(v_i) \in 2^{V_i}$, and an ordering o_{v_i}; and $cost : T \rightarrow \mathbb{Z}^+$ be a cost function that assigns a cost to each tree tile.

We define the relation *matches* such that a node $v \in V$ *matches* tree t_i with root $r \in V_i$ iff $k(v) = k(r)$, $|ch(v)| = |ch(r)|$, and, for all $c \in ch(v)$ and $c_i \in ch(r)$, $o_v(c) = o_r(c_i)$ implies that either $k(c_i) = \circ$ or c *matches* the tree rooted at c_i. For a given matching of v and t_i and a tree tile node $v_i \in V_i$, we define $m_{v,t_i} : V_i \rightarrow V$ to return the node in V that matches with the subtree rooted at v_i. A mapping $f : V \rightarrow 2^T$ from each DAG node to a set of tree tiles is *legal* iff $\forall v \in V$:

$$t_i \in f(v) \implies v \; matches \; t_i$$

$$indegree(v) = 0 \implies |f(v)| > 0$$

$$\forall t_i \in f(v), \forall v_i \in t_i, k(v_i) = \circ \implies |f(m_{v,t_i}(v_i))| > 0$$

An *optimal instruction tiling* is a legal mapping f which minimizes

$$\sum_{v \in V} \sum_{t_i \in f(v)} cost(t_i)$$

In some versions of the instruction tiling problem the tiles are extended to include the notion of labels. Each tile includes a label for every input and output. Adjacent tiles must have identical labels. These labels represent the storage locations a tile writes or reads. For example, some tiles might write to memory or read from a specific register class. In this case, there is an additional constraint that a tile's inputs must not only match with other tiles' outputs, but the names of the respective input and output must also match. In practice, if instruction selection is performed independently of register allocation, the names of storage locations are irrelevant. Although previous proofs of the hardness of instruction selection have relied on complications such as storage location naming [108] or two-address instructions [2], we now show that even without these restrictions the problem remains NP-complete.

Theorem 7.1.1. *The optimal instruction tiling problem (is there an optimal instruction tiling of cost less than k_{const}?) is NP-complete.*

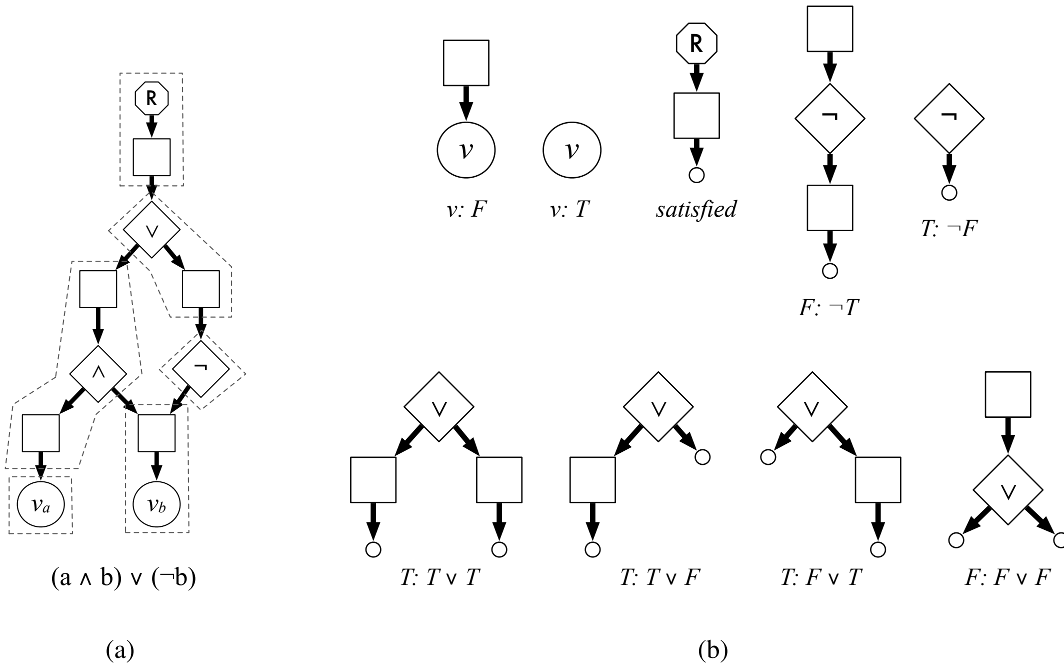


Figure 7.2: Expressing Boolean satisfiability as an instruction selection problem. (a) An example of an expression DAG that represents a Boolean expression. (b) The tiles used to cover such an expression DAG. Each tile has unit cost. The tiles representing $\wedge$ are omitted, but are similar to the $\vee$ tiles with the two middle tiles having an additional box node at the root.

Proof. Inspired by [108], we perform a reduction from Boolean satisfiability [50]. Given a Boolean expression consisting of variables $u \in U$ and Boolean connectives $\{\vee, \wedge, \neg\}$, we construct an instance of the optimal instruction tiling problem as follows:

Let the set of node kinds K be $\{\vee, \wedge, \neg, \Box, R, v\}$. We refer to nodes with kind $\Box$ as box nodes. For every variable $u \in U$, create two nodes u_1 and u_2 and a directed edge $(u_1 \rightarrow u_2)$ in G such that $k(u_1) = \Box$ and $k(u_2) = v$. Similarly, for every Boolean operator op create two nodes op_1 and op_2 and a directed edge $(op_1 \rightarrow op_2)$ such that $k(op_1) = \Box$ and $k(op_2)$ is the corresponding operation. Next, for every operation $a\ op\ b$ create edges $(op_2 \rightarrow a_1)$ and $(op_2 \rightarrow b_1)$ where $k(a_1) = k(b_1) = \Box$ (in the case of the unary $\neg$ operation a single edge is created). Note the ordering of child nodes is irrelevant since the Boolean operators are commutative. Finally, create a node r and edge $(r \rightarrow op)$ such that $k(r) = R$ and op is the root operation of the expression. An example of such a DAG is shown in Figure 7.2(a). Note that the only nodes with potentially more than one parent in this DAG are the box nodes that correspond to variables.

Now let the tree tile set T be as shown in Figure 7.2(b) where each tile contains a single non-box node and has unit cost. These tiles are designed so that it can be shown that a truth assignment of a Boolean expression corresponds directly with a legal tiling of a DAG constructed as described above. If a variable is true, then its corresponding node is covered with the tile $v : T$, otherwise it is covered with $v : F$. The rest of the tiling is built up in the natural way suggested from the tile names in Figure 7.2(b). This tiling is optimal since every leaf node of the DAG will have exactly one tile covering it (corresponding to the truth assignment of that variable) and, since the parents of leaf nodes are the only shared nodes in the DAG (they may have multiple parents), no other non-box node in the DAG can be covered by more than one tile in this tiling. Therefore, the cost of the tiling is equal to the number of non-box nodes and is optimal.

Given an optimal tiling of a DAG derived from a Boolean expression, if the cost of the tiling is equal to the number of non-box nodes, then we can easily construct a truth assignment that satisfies the expression by observing the tiles used to cover the leaves of the DAG. If the cost of the tiling is greater than the number of non-box nodes then the expression is not satisfiable. If it were, a cheaper tiling would have to exist..

We have shown the boolean satisfiability reduces to the optimal instruction tiling problem, and, therefore, the optimal instruction tiling problem is NP-complete. □

7.2 NOLTIS

The NP-completeness of the optimal instruction tiling problem necessitates that heuristics be used to perform instruction selection. The textbook approach is to first decompose the DAG into a forest of trees and then use an optimal tree tiling algorithm to tile each tree [5]. Every common subexpression in the DAG is therefore at the root of a tree in the forest. However, as we will show in Section 7.5, this approach is not as successful as algorithms which work directly on the DAG. For example, if all the tiles in Figure 7.3 were assigned a unit cost, the tree decomposition solution would be suboptimal.

In this section we present NOLTIS, a linear-time algorithm which obtains near-optimal tilings of expression DAGs. The algorithm applies tree tiling directly to the DAG without first perform-

```
 1: DAG : expression DAG
 2: bestChoiceForNode : Node → (Tile × int)
 3: fixedNodes : set of Node
 4: matchedTiles : set of Tile
 5: coveringTiles : Node → set of Tile

 6: procedure SELECT
 7:     fixedNodes ← {}
 8:     BOTTOMUPDP()                          ▷ initializes bestChoiceForNode
 9:     TOPDOWNSELECT()                       ▷ initializes coveringTiles
10:     IMPROVECSEDECISIONS()                 ▷ initializes fixedNodes
11:     BOTTOMUPDP()                          ▷ uses fixedNodes
12:     TOPDOWNSELECT()                       ▷ puts final tiling in matchedTiles

13: procedure BOTTOMUPDP
14:     for n ∈ reverseTopologicalSort(DAG) do
15:         bestChoiceForNode[n].cost ← ∞
16:         for t_n ∈ matchingTiles(n) do
17:             if ¬hasInteriorFixedNode(t_n, fixedNodes) then
18:                 val ← cost(t)+
                        Σ_{n′∈boundaryNodes(t_n)} bestChoiceForNode[n′].cost
19:                 if val < bestChoiceForNode[n].cost then
20:                     bestChoiceForNode[n].cost ← val
21:                     bestChoiceForNode[n].tile ← t_n

22: procedure TOPDOWNSELECT
23:     matchedTiles.clear()
24:     coveringTiles.clear()
25:     q.push(roots(DAG))
26:     while ¬q.empty() do
27:         n ← q.pop()
28:         bestTile ← bestChoiceForNode[n].tile
29:         matchedTiles.add(bestTile)
30:         for every node n_t covered by bestTile do
31:             coveringTiles[n_t].add(bestTile)
32:         for n′ ∈ boundaryNodes(bestTile) do
33:             q.push(n′)
```

Listing 7.1: *Dynamic programming instruction selection with modifications for near-optimal DAG selection*

```
1: procedure IMPROVECSEDECISIONS
2:   for n ∈ sharedNodes(DAG) do
3:     if coveringTiles[n].size() > 1 then                    ▷ has overlap
4:       overlapCost ← getOverlapCost(n, coveringTiles)
5:       cseCost ← bestChoiceForNode[n].cost
6:       for t_n ∈ coveringTiles[n] do
7:         cseCost ← cseCost + getTileCutCost(t_n, n)
8:       if cseCost < overlapCost then
9:         fixedNodes.add(n)
```

Listing 7.2: IMPROVECSEDECISIONS *Given a DAG matching that ignored the effect of shared nodes, decide if the solution would be improved by pulling shared nodes out into common subexpressions (eliminating tile overlap).*

```
1: function GETOVERLAPCOST(n)
2:   cost ← 0
3:   seen ← {}
4:   for t ∈ coveringTiles[n] do
5:     q.push(t)
6:     seen.add(t)
7:   while ¬q.empty() do
8:     t ← q.pop()
9:     cost ← cost + cost(tile)
10:    for n′ ∈ boundaryNodes(t) do
11:      if n′ is reachable from n then
12:        t′ ← bestChoiceForNode[t′].tile
13:        if coveringTiles[n′].size() = 1 then
14:          cost ← cost + bestChoiceForNode[n′].cost
15:        else if t′ ∉ seen then
16:          seen.add(t′)
17:          q.push(t′)
18:  return cost
```

Listing 7.3: GETOVERLAPCOST *Given a shared node n with overlapping tiles, compute the cost of the tree of tiles rooted at the tiles overlapping n without double counting areas where the tile trees do not overlap.*

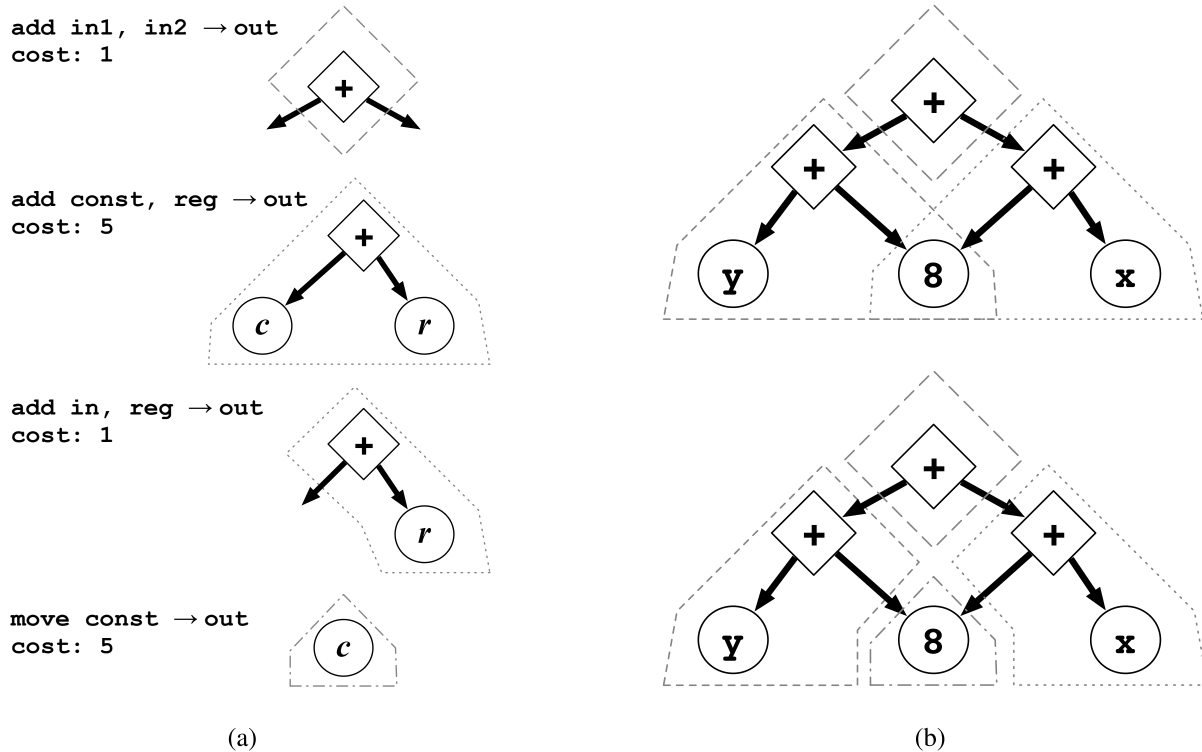


Figure 7.3: An example of instruction selection on a DAG. (a) The tile set used (commutative tiles are omitted). (b) Two possible tilings. In a simple cost model where every tile has a unit cost the top tiling would be optimal, but with the cost model shown the lower tiling is optimal.

ing tree decomposition, uses this tiling to decide which parts of the DAG can be productively decomposed into trees, and then retiles the partially decomposed DAG.

First we apply dynamic programming on the DAG ignoring the presence of shared nodes using the procedure BOTTOMUPDP of Listing 7.1. Conceptually, we are tiling the tree that would be formed if every shared node (and its descendants) had been duplicated to convert the DAG into a potentially exponentially larger tree. However, the algorithm remains linear since each node is visited only once. Once dynamic programming has labeled each node with the best tile for that node, a top down pass, TOPDOWNSELECT from Listing 7.1, creates a tiling of the DAG. The existence of shared nodes may result in a tiling where nodes are covered by multiple tiles (i.e., the tiles overlap). However, since no node will be at the root of two tiles (this would imply that the exact same value would be computed twice), the number of tiles in a tiling is proportional to the number of nodes. Consequently, the top down pass, which traverses tiles, has linear time complexity.

```
1:  function GETTILECUTCOST(t,n)
2:      bestCost ← ∞
3:      r ← root(tile)
4:      for t′ ∈ matchingTiles(r) do
5:          if n ∈ boundaryNodes(t′) then
6:              cost ← cost(t′)
7:              for n′ ∈ boundaryNodes(t′) ∧ n′ ≠ n do
8:                  cost ← cost + bestChoiceForNode[n′].cost
9:              if cost < bestCost then
10:                 bestCost ← cost
11:     for n′ ∈ boundaryNodes(t) do                ▷ Subtract edge costs of original tile
12:         if path r ⇝ n′ ∈ t does not contain n then
13:             bestCost ← bestCost
                                −bestChoiceForNode[n′].cost
14:     return bestCost
```

Listing 7.4: GETTILECUTCOST *Given a tile t and node n, determine the cost of cutting t at node n so that the root of t remains the same but n becomes an boundary node.*

The tiling found by the first tiling phase ignores the impact of shared nodes in the DAG and therefore may have excessive amounts of overlap. In the next step of the algorithm, we identify shared nodes where removing overlap locally improves the overall tiling. These nodes are added to the *fixedNodes* set. We then perform another tiling pass. In this pass, tiles are prohibited from spanning nodes in the *fixedNodes* set; these nodes must be matched to the root of a tile.

The procedure IMPROVECSEDECISIONS (Listing 7.2) is used to determine if a shared node should be fixed. For each shared node n with overlap we compute the cost of the overlap at n using the GETOVERLAPCOST function in Listing 7.3. This function computes the cost of the local area of overlap at n. Note that, in the rare case that the area of overlap covers another shared node, it is possible that IMPROVECSEDECISIONS will have super-linear time complexity; however, this can be addressed through the use of memoization, a detail that is not shown in the pseudocode.

The next step is to compute the cost that would be incurred if the tiles covering n were cut so that only a single tile, rooted at n, covered n. A tile is cut at an edge by decomposing it into smaller tiles such that the cut point is no longer internal to a tile. In this case, we seek to find the minimum cost collection of such smaller tiles that cover the same nodes as the original overlapping tile and have a tile rooted at n. The cost of the tile tree rooted at n can be determined from

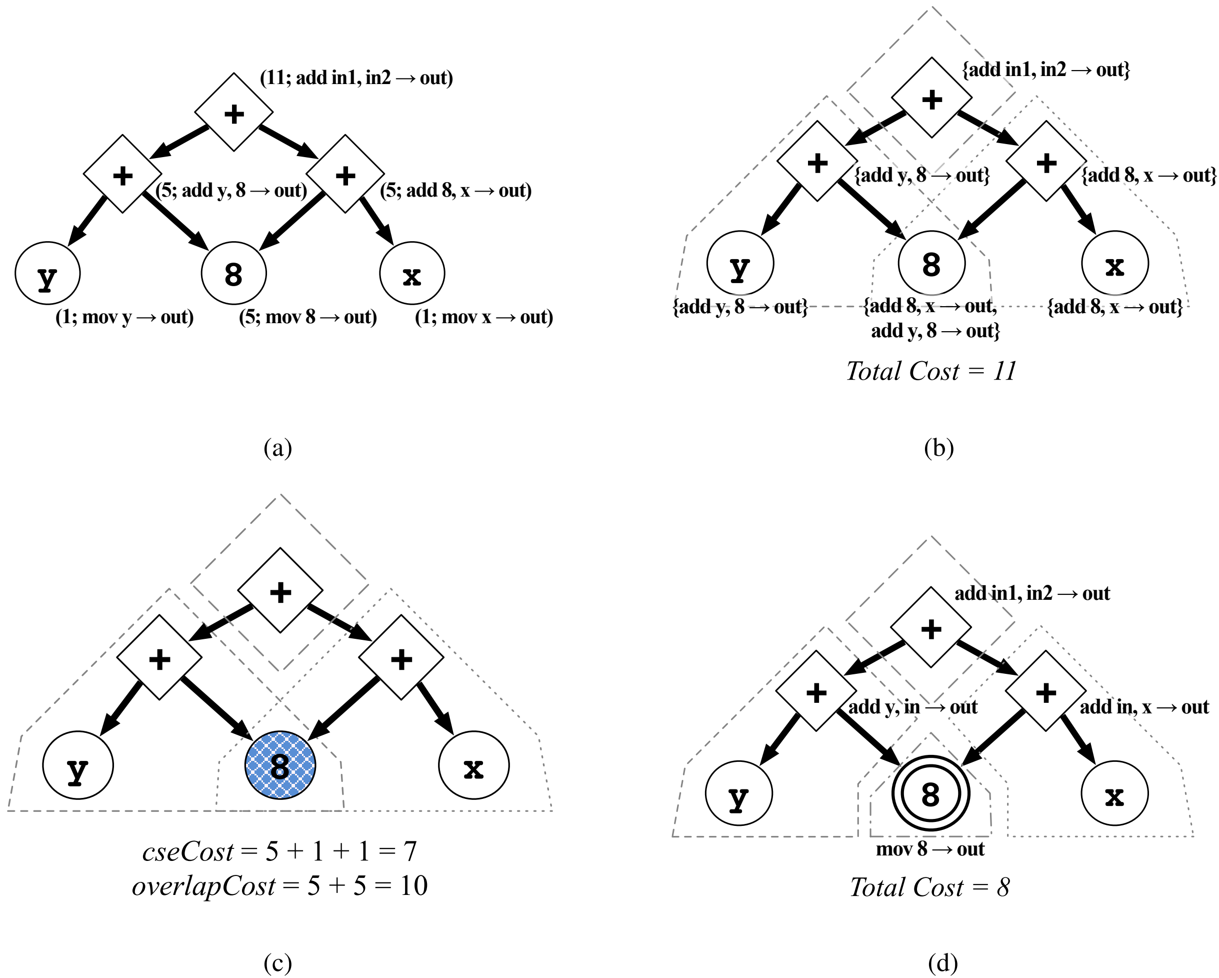


Figure 7.4: The application of the NOLTIS algorithm to the example from Figure 7.3. (a) BOTTOMPUPDP *computes the dynamic programming solution for the DAG, initializing* $bestChoiceForNode$*. (b)* TOPDOWNSELECT *determines a tiling from the dynamic programming solution, initializing* $coveringTiles$*. (c)* IMPROVECSEDECISIONS *evaluates the shared node and determines it should be fixed. (d) The dynamic programming solution is then recomputed with the requirement that the fixed node not be overlapped and the optimal solution is found.*

the results of dynamic programming. To this cost we add the costs of cutting the tiles currently covering n, which are computed using the function GETTILECUTCOST shown in Listing 7.4. In determining the cost of cutting a tile t with root r at node n, we consider every tile which also matches at r and has n as a boundary node. We then compute the cost difference between using this tile to match r and using t. We choose the minimum cost difference as the cost of cutting the tile. If the cost of the current overlapping tiling is more than the cost of removing the overlap and cutting the tiles, then we have found a local transformation that improves the existing tiling.

Instead of immediately applying this transformation, we choose to fix node n, disabling overlap when we compute a new tiling. This results in a potentially better solution as the new tiling need not be limited to tiles rooted at r. Figure 7.4 shows the execution of the NOLTIS algorithm on the example from Figure 7.3.

The NOLTIS algorithm is not optimal as it depends upon several assumptions that do not necessarily hold. We assume that it is always possible to cut tiles at a shared node without affecting the tileability of the DAG. We assume that the best place to cut tiles to eliminate overlap is at a shared node. We assume the decision to fix a shared node can be made independently of other shared nodes. When deciding to fix a shared node we assume we can represent the impact of fixing the node by examining simple tile cuts. Despite these assumptions, in practice the NOLTIS algorithm achieves near-optimal results.

7.3 0-1 Programming Solution

In order to establish the near-optimality of our algorithm, we formulate the instruction tiling problem as a 0-1 integer program and find solutions using a commercial solver. The formulation of the problem is straightforward. For every node i and tile j we have binary variable $M_{i,j}$ which is one if tile j matches node i (the root of tile j is at node i) in the tiling, zero otherwise. Let $cost_j$ be the cost of tile j, $roots$ be the root nodes of the DAG, and $boundaryNodes(i, j)$ be the nodes at the boundary of tile j when rooted at node i, then the optimal instruction tiling problem is:

$$\min \sum_{i,j} cost_j M_{i,j}$$

subject to

$$\forall_{i \in roots} \sum_{j} M_{i,j} >= 1 \tag{7.1}$$

$$\forall_{i,j} \forall_{i' \in boundaryNodes(i,j)} M_{i,j} - \sum_{j'} M_{i',j'} \leq 0 \tag{7.2}$$

where (7.1) requires that the root nodes of the DAG be matched to tiles and (7.2) requires that if a tile matches a node, then all of the inputs to that tile must be matched to tiles.

7.4 Implementation

We have implemented our algorithm in the LLVM 2.1 [87] compiler infrastructure targeting the Intel x86 32-bit architecture. The default LLVM instruction selector constructs an expression DAG of target independent nodes and then performs maximal munch [8]. Tiles are selected from the top-down. The largest tile (the tile that covers the most nodes) is greedily selected. Tile costs are only used to break ties. We have modified the LLVM algorithm to use code size when breaking ties.

In addition to the default LLVM algorithm and the NOLTIS algorithm, we have implemented three other algorithms that we believe to be representative of existing practice:

cse-all The expression DAG is completely decomposed into trees and dynamic programming is performed on each tree. That is, every shared node is fixed. This is the conventional method for applying tree tiling to a DAG [5].

cse-leaves The expression DAG is partially decomposed into trees and dynamic programming is performed. If the subgraph rooted at a shared node can be fully covered by a single tile, the shared node remain unaltered, otherwise shared nodes become roots of trees to be tiled. That is, shared nodes are fixed unless they represent an expression that can be implemented by a single tile whose only inputs are leaf expressions such as constants or variables.

cse-none The expression DAG is not decomposed into trees and dynamic programming is performed. That is, no shared nodes are fixed (this is equivalent to the solution found before the IMPROVECSEDECISIONS procedure is executed in the NOLTIS algorithm).

All algorithms use the same tile set. The cost of each tile is the size in bytes of the corresponding x86-32 instruction(s). We do not allow tiles to overlap memory operations (i.e., a load or store node in the expression DAG will only be executed once). Similarly, as an implementation detail,[2] overlap of function call addresses is not allowed. Valueless token edges enforce ordering dependencies in the expression DAG. Despite the two-address nature of the x86 architecture, all tiles represent three-address instructions. A pass after instruction selection converts the code to

[2] LLVM's support for different relocation models requires that function call addresses be part of the call instruction.

two-address form. A scheduling pass, which converts the code from DAG form into an assembly listing, attempts to minimize the register pressure of the schedule using Sethi-Ullman numbering [118].

7.5 Results

We evaluate the various instruction selection algorithms by compiling the C and C++ benchmarks of the SPEC CPU2006 [123], MediaBench [79], MiBench [57], and VersaBench [113] benchmark suites and observing both the immediate, post-selection cost of the tiling and the final code size of the benchmark. We evaluate the optimality of the NOLTIS algorithm, demonstrate its superiority compared to existing heuristics, investigate its impact on the code size of fully compiled code, and describe its compile-time behavior.

7.5.1 Optimality

In order to determine an optimal solution for an expression DAG, we create a 0-1 integer programming problem as described in Section 7.3 and then solve it using ILOG CPLEX 10.0 [64]. We evaluated all the basic blocks of the SPEC CPU2006 benchmarks, resulting in nearly a half million tiling problems. We utilized a cluster of Pentium 4 machines ranging in speed from 2.8Ghz to 3.0Ghz to solve the problems. CPLEX was able to find a provably optimal solution within a 15 minute time limit for 99.8% of the tiling problems. Of the problems with provably optimal solutions, the NOLTIS algorithm successfully found the optimal solution 99.7% of the time. Furthermore, suboptimal solutions were typically very close to optimal (only a few bytes larger). Of the 0.2% of problems where CPLEX did not find a provably optimal solution, the NOLTIS algorithm found a solution as good as, and in some cases better than, the best solution found by CPLEX 75% of the time implying our algorithm is effective even for very difficult tiling problems.

The overall improvement obtained by using the best CPLEX solution versus using the NOLTIS algorithm was a negligible 0.05%. We feel these results clearly demonstrate that the NOLTIS algorithm is, in fact, near-optimal.

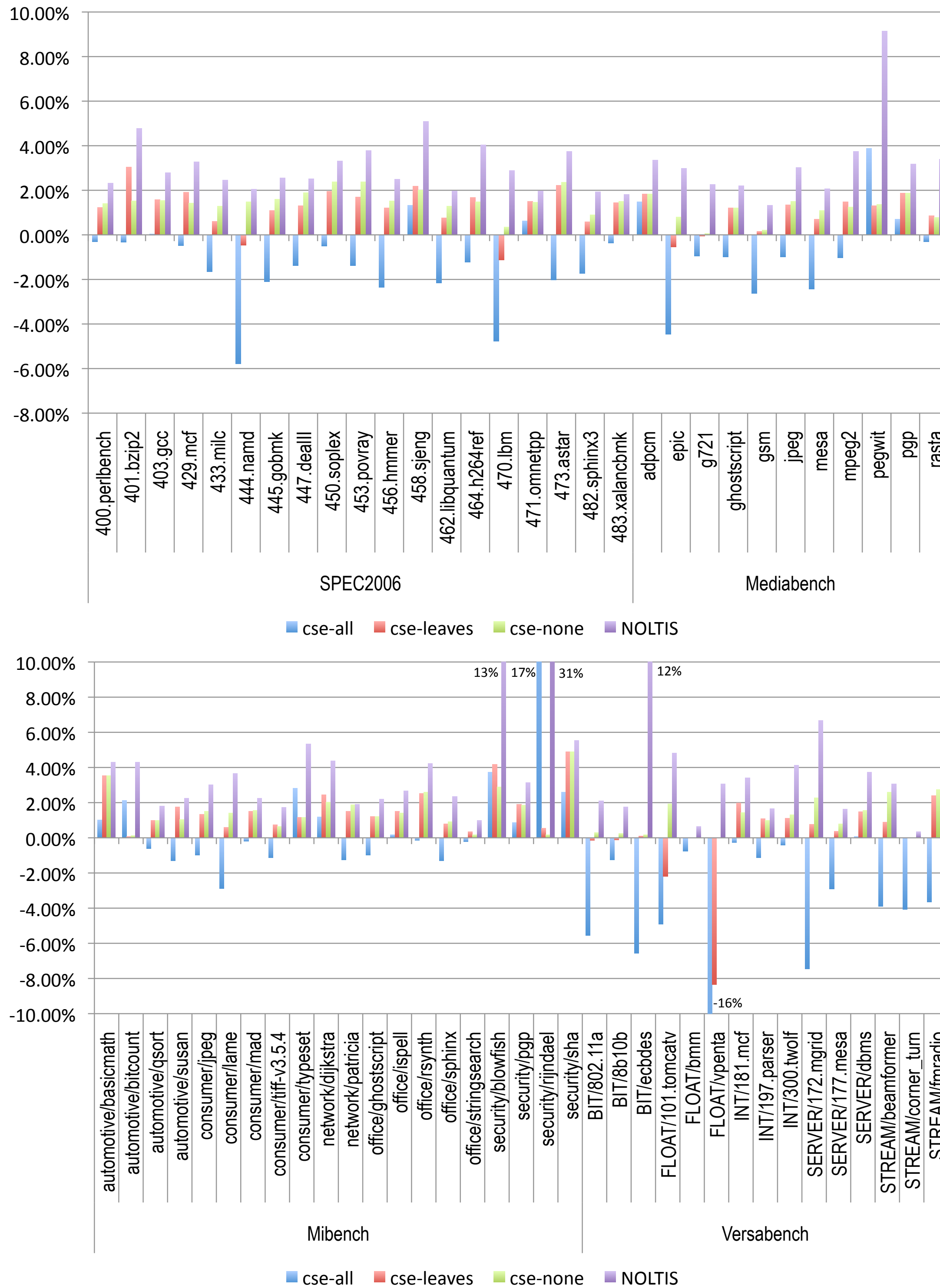


Figure 7.5: Improvement in tiling cost relative to the default maximal munch algorithm for individual benchmarks. In all cases the NOLTIS algorithm yields the greatest improvement. Very small benchmarks can exhibit large relative changes in code size as a result of a small absolute difference. The geometric means across all benchmarks for the cse-all*,* cse-leaves*,* cse-none*, and NOLTIS algorithms are -1.04% , 1.05%, 1.39%, and, 3.89% respectively.*

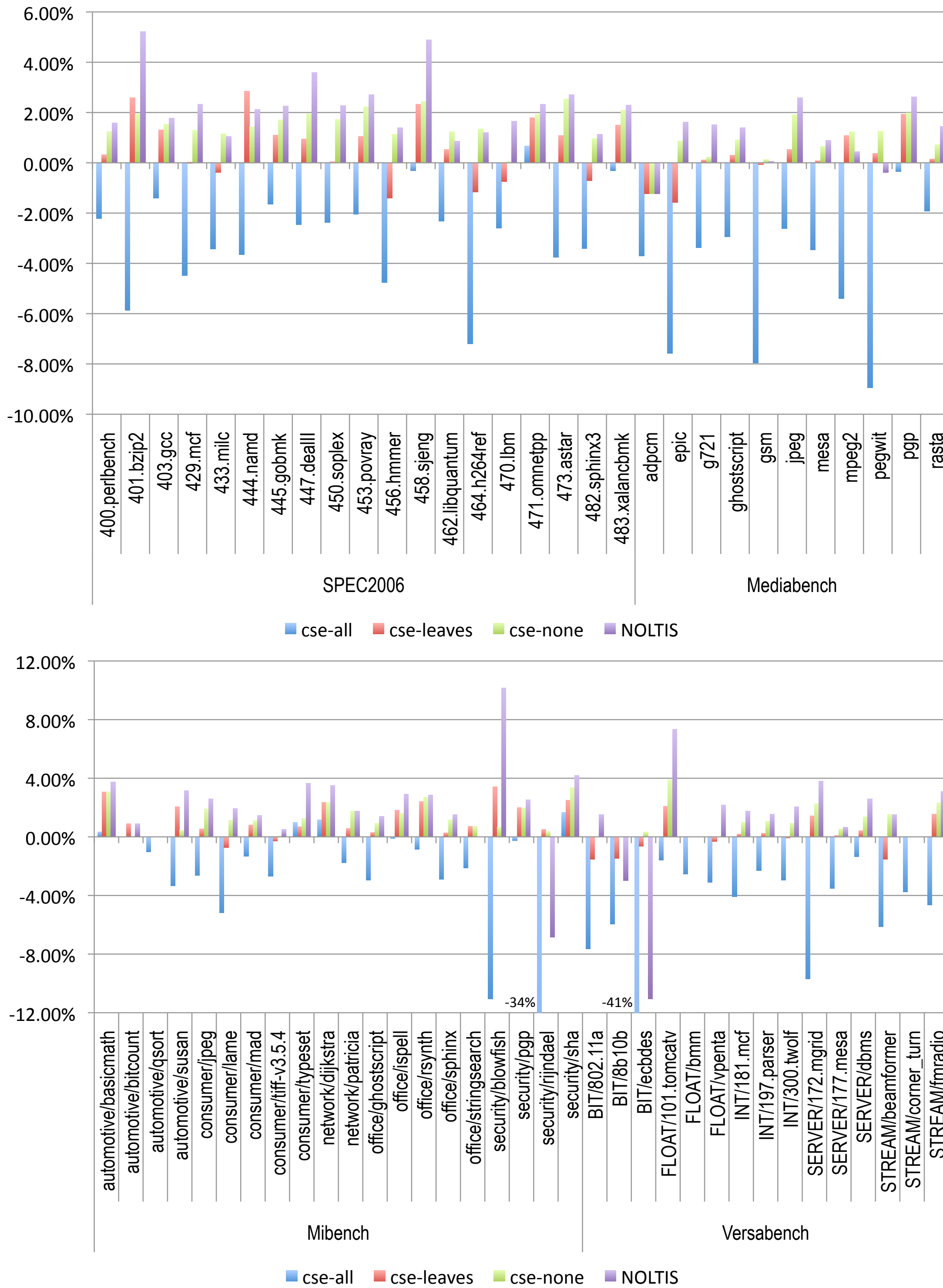


Figure 7.6: Final code size improvement relative to the default maximal munch algorithm for individual benchmarks. In 54 of the 65 benchmarks the NOLTIS algorithm has the greatest improvement. Very small benchmarks can exhibit large relative changes in code size as a result of a small absolute differences. The geometric means across all benchmarks for the cse-all, cse-leaves, cse-none, and NOLTIS algorithms are -3.98%, 0.61%, 1.24%, and 1.74% respectively.

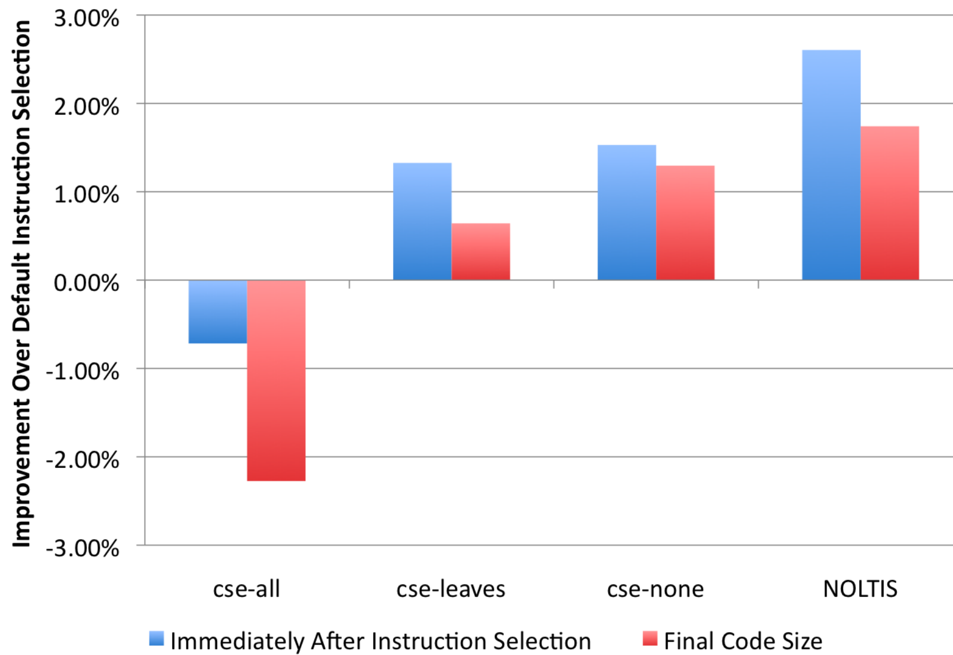


Figure 7.7: The cumulative total improvement relative to maximal munch for the entire collection of benchmarks. Both the improvement in tiling cost and final code size improvement are shown. The near-optimality of the NOLTIS algorithm results in excellent tiling solutions. However, interactions with successive passes of the compiler, such as register allocation, reduce the advantage of using a near-optimal tiling.

7.5.2 Comparison of Algorithms

In addition to being near-optimal, the NOLTIS algorithm provides significantly better solutions to the tiling problem than conventional heuristics. Detailed results for individual benchmarks are shown in Figure 7.5 and overall improvements are shown in Figure 7.7. The *cse-all* algorithm, despite finding an optimal tiling for each tree in the full tree decomposition of the DAG, performs poorly relative to all other algorithms suggesting that DAG tiling algorithms are necessary for maximum code quality. Both the *cse-leaves* and *cse-none* algorithms benefit from using dynamic programming and outperform the greedy algorithm, although neither algorithm is clearly superior to the other. The NOLTIS algorithm, as expected, significantly outperforms the other algorithms and has the greatest improvement in every benchmark with a cumulative overall improvement of 2.6% and an average improvement of 3.89%.

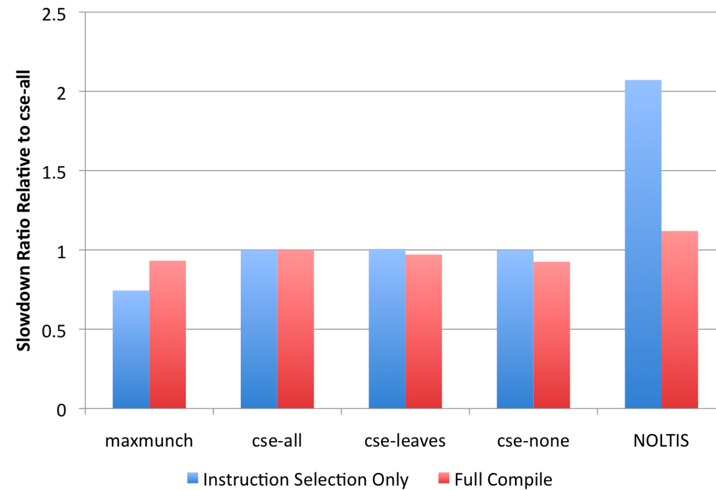


Figure 7.8: The average slowdown of each instruction selection algorithm relative to cse-all.

7.5.3 Impact on Code Size

Instruction tiling is only one component of code generation. The two-address conversion pass, scheduling pass, and register allocation pass all further impact the final quality of the compiled code. Final code size results for the individual benchmarks are shown in Figure 7.6 and overall improvements are shown in Figure 7.7. The cumulative overall code size improvement across all benchmarks for the NOLTIS algorithm is 1.741% and the average improvement is 1.736%.

Although the average code size improvements exhibited by the NOLTIS algorithm may seem marginal, even such seemingly small code size reductions may be significant when targeting highly constrained embedded architectures. Furthermore, it is important to note that these results are relative to an algorithm that has already been adapted to work directly on expression DAGs. Compared to the classical textbook approach of tree decomposition (the *cse-all* algorithm), the NOLTIS algorithm exhibits an overall cumulative code size improvement of 3.9%.

7.5.4 Compile Time Performance

As shown in Figure 7.8, the two pass nature of the NOLTIS algorithm means that its running time is slightly more than twice that of the other dynamic programming based algorithms. Despite this doubling in instruction selection time, the full compilation time only slows down by 12%

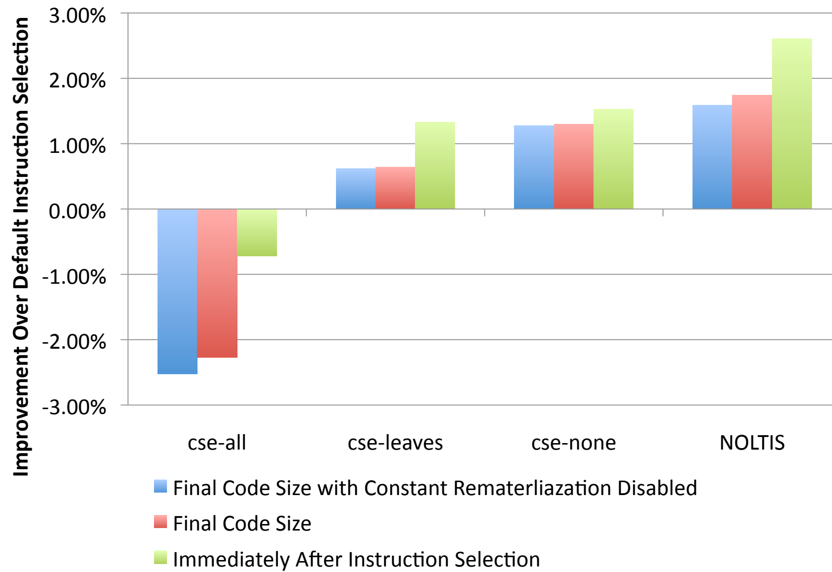


Figure 7.9: The influence of constant rematerialization on final code size. Constant rematerialization during register allocation can eliminate register pressure caused by decomposing a constant expression into its own tree by regenerating a constant directly within an instruction. Both the cse-all *and NOLTIS algorithms noticeably benefit from constant rematerialization, however there remains a significant gap between the final code size improvement and the tiling cost improvement immediately after instruction selection.*

relative to the single-pass *cse-all* algorithm. The single-pass dynamic programming algorithms are approximately 30% slower than the greedy algorithm since they must perform a tile selection operation at every node of the expression DAG. The greedy algorithm can ignore any nodes which are completely covered by the greedily selected tile.

7.6 Limitations and Future Work

Instruction selection algorithms have been used successfully to solve the technology mapping problem in the automated circuit design domain. It remains an open question whether the NOLTIS algorithm can be successfully adapted to this domain where multiple optimization goals (area, delay, routing resources) must be simultaneously addressed.

Although the NOLTIS algorithm is linear in the size of the program, its running time is largely determined by how efficiently the matching of a single node to a set of tiles can be performed. The algorithm, as we have presented it, uses a simple, but inefficient, matching

algorithm. More efficient algorithms, such as tree parsing, exist [4, 46, 104, 110] and should be used in a production implementation. Additionally, the second pass of dynamic programming could be made more efficient by intelligently recomputing only portions of the DAG.

The classical representation of instruction selection as a tiling problem relies on instructions being represented by tree tiles. In some cases, such as with single instruction multiple data (SIMD) instructions and instructions with side-effects, an instruction cannot be represented as a tree of data dependences. Additional, non-tiling, techniques are required to handle such instructions.

7.7 Interaction with Register Allocation

Despite demonstrating near-optimal results, the NOLTIS algorithm does not always result in the smallest code size. The mixed nature of the final code size results appears to be mostly caused by the interaction with the register allocator, in particular the number of loads and stores the allocator inserts. Decomposing the graph into trees results in the creation of temporaries with multiple uses. These potentially long-lived temporaries result in more register pressure and more values must be spilled to memory. Hence the *cse-all* algorithm performs particularly poorly.

In some cases the register allocator can intelligently reverse the bad effects of a tree decomposition. For example, if the decomposed tree is a constant expression, constant rematerialization can decrease register pressure by regenerating the constant expression instead of storing it in a temporary. The effect of constant rematerialization on the final code size is shown in Figure 7.9. As expected, both the *cse-all* and NOLTIS algorithms, which perform tree decomposition, demonstrate some improvement from constant rematerialization. In contrast, the *cse-none* and *cse-leaves* algorithms are largely unaffected. Constant rematerialization provides a potential model of improving the interaction between instruction selection and register allocation. The instruction selector optimistically chooses a selection that potentially has a negative impact on register pressure, and then the register allocator is capable of intelligently modifying the instruction selection within the full context of register allocation.

Allowing unlimited overlap can also have a negative effect on register allocation as the inputs of overlapping tiles are also potentially long lived temporaries. Another factor influencing register allocation is the number of tiles. If more, smaller, tiles are used, there are correspondingly more temporaries to allocate. It is likely that architectures with complex instruction sets but plentiful (e.g., more than eight) registers would see more benefit from the NOLTIS algorithm.

7.8 Summary

In this chapter we have described NOLTIS, an easy to implement, fast, and effective algorithm for finding an instruction tiling of an expression DAG. We have shown empirically that the NOLTIS algorithm achieves near-optimal results and significantly outperforms existing tiling heuristics.

Chapter 8

Conclusion

We have presented a principled approach for understanding, evaluating, and solving backend optimization problems. In our principled approach we

- develop a comprehensive and *expressive model* of the backend optimization problem, and
- design model solution techniques that *achieve or approach the optimal solution*.

The foundation of our approach is the development of an expressive model that fully captures the complexities of the problem. This is in contrast to conventional approaches where the focus is on developing simplified, easy to solve, abstractions, such as graph coloring or tree tiling. The conventional approach inevitably leads to greater complexity as compiler developers are forced to develop *ad hoc* heuristic extensions in the pursuit of improved code quality.

In our approach, we develop a model that is as complex as it needs to be to fully represent the problem. If an optimal solution to the model cannot be computed efficiently, we do not reduce the complexity of the model, but instead develop solution techniques that are capable of approaching an optimal solution. Given sufficiently advanced solution techniques, there is no need for the compiler developer to develop *ad hoc* extensions. Code quality improvements come from refining the model to more accurately reflect the target architecture and code quality metric.

We successfully apply our principled approach to the critical backend compiler optimizations of register allocation and instruction selection. Our progressive register allocator utilizes a novel network flow representation of the register allocation problem that is expressive enough to fully model all the pertinent features of the problem. Our progressive solution technique, based on La-

grangian relaxation and subgradient optimization, approaches the optimal solution as more time is allowed for compilation. It also provides feedback about the quality of the allocation in the form of an optimality bound. Compared to the state-of-the-art extended linear scan allocator of LLVM when targeting the highly constrained x86-32 architecture, our progressive allocator initially improves code size by 1.6% on average and then, as more time is allowed for compilation, improves code size by more than 5% on average with some benchmarks improving by more than 7%. When optimizing for performance, our allocator can reduce the number of memory operations executed by as much as 20% on average and for several benchmarks results in performance improvements in excess of 15%.

We use the expressive, but computationally complex, directed acyclic graph representation of the instruction selection problem as the foundation of our near-optimal linear-time instruction selection algorithm (NOLTIS). The NOLTIS algorithm nearly always finds the optimal instruction selection and achieves an average code size improvement of 1.7% relative to the LLVM instruction selection algorithm and 3.9% relative to the textbook approach to instruction selection.

Our principled approach could also be applied to other aspects of compiler optimization. For example, existing expressive models of instruction scheduling [129] would benefit from progressive solution techniques. We also believe that our expressive model of register allocation would be a valuable starting point for developing a principled approach for balancing resource usage and parallelism when compiling for GPUs [114].

The substantial improvement in code quality achieved with our approach validates our thesis: *our principled approach results in better code quality and a better compiler.*

Bibliography

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23 (3):488–501, 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321958.321970. 1.1.2, 2.2

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321992.322001. 2.2, 7.1

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN 0-201-10088-6. 2.2

[4] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, 1989. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/69558.75700. 2.2, 7.6

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools (2nd Edition).* Addison-Wesley, 2006. ISBN 978-0321486813. 2.1, 2.2, 7.2, 7.4

[6] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., 1993. ISBN 0-13-617549-X. 3.1, 5.1.1, 6.1.1, 6.1.1, 6.1.2, 6.2, 6.2.4

[7] Anonymous ARM Compiler Engineer. Personal Communication, 2005. 4.2.1

[8] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1997. ISBN 0-521-58654-2. 2.1, 2.2, 7.4

[9] Barrie M. Baker and Janice Sheasby. Accelerating the convergence of subgradient optimisation. *European Journal of Operational Research*, 117(1):136–144, August 1999. 6.2.3

[10] Thomas Ball and James R. Larus. Branch prediction for free. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 300–313, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: http://doi.acm.org/10.1145/155090.155119. 3.5

[11] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/

258915.258941. 2.1.1

[12] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263. ACM Press, 1989. ISBN 0-89791-306-X. doi: http://doi.acm.org/10.1145/73141.74841. 2.1.1

[13] M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. i: Interval graphs. *Discrete Math.*, 100(1-3):267–279, 1992. ISSN 0012-365X. doi: http://dx.doi.org/10.1016/0012-365X(92)90646-W. 1.1.1

[14] Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 574–583. Society for Industrial and Applied Mathematics, 1998. ISBN 0-89871-410-9. 1.1.1, 2.1.5

[15] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993. URL `citeseer.nj.nec.com/bodlaender93tourist.html`. 1.1.1

[16] Pradip Bose. Optimal code generation for expressions on super scalar machines. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 372–379, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-4743-4. 2.2

[17] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École Normale Supérieure de Lyon, France, December 2008. 2.1.2

[18] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. Technical Report RR2006-15, LIP, ENS-Lyon, France, April 2006. 1.1.1

[19] Florent Bouchez, Alain Darte, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove. In *Workshop on Duplicating, Deconstructing, and Debunking*, 2006. 1.1.1, 2.1.2

[20] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *Acm sigplan/sigbed conference on languages, compilers, and tools for embedded systems (lctes'07)*, San Diego, USA, jun 2007. URL `http://doi.acm.org/10.1145/1273444.1254782`. 1.1.1

[21] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992. 2.1.1

[22] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 311–321, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-475-9. doi: http://doi.acm.org/10.1145/143095.143143. 2.1.1

[23] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, 1992. ISSN 1057-4514. doi: http://doi.acm.org/10.1145/130616.130617. 2.1.1

[24] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/177492.177575. 2.1.1

[25] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005. 1.1.1, 2.1.2

[26] John Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3): 502–510, 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321958.321971. 2.2

[27] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. *SIGPLAN Not.*, 30(6):79–92, 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/223428.207118. 3.5

[28] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 192–203. ACM Press, 1991. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113462. 2.1.4

[29] P.M. Camerini, L. Fratta, and F. Maffioli. On improving relaxation methods by modified gradient techniques. *Mathematical Programming Study*, 3:26–34, 1975. 6.2.3

[30] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst.*, 2(2):173–190, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357094.357097. 2.2

[31] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–101. ACM Press, 1982. ISBN 0-89791-074-5. 2.1.1

[32] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6 (1):47–57, 1981. ISSN 0096-0551. 1.1.1

[33] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/88616.88621. 2.1.1

[34] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 222–232, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. doi: http://doi.acm.org/10.1145/502874.502896. 2.1.1

[35] Keith Cooper, Anshuman Dasgupta, and Jason Eckhardt. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, October 2005. 2.1.4

[36] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag. ISBN 3-540-64304-4. 2.1.1

[37] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004. 2.1, 2.1.1, 2.2

[38] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320. 2.1.2

[39] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, 1984. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1780.1783. 2.2

[40] H. Emmelmann, F.-W. Schröer, and L. Landwehr. Beg: a generation for efficient back ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 227–237, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-306-X. doi: http://doi.acm.org/10.1145/73141.74838. 2.2

[41] M. Anton Ertl. Optimal code selection in dags. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-095-3. doi: http://doi.acm.org/10.1145/292540.292562. 2.2

[42] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9. 1.1.1, 2.1.4

[43] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 79–84, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.53998. 2.2

[44] Christopher W. Fraser and Alan L. Wendt. Integrating code generation and optimization. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 242–248, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-197-0. doi: http://doi.acm.org/10.1145/12276.13335. 2.2

[45] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992. ISSN 1057-4514. doi: http://doi.acm.org/10.1145/151640.151642. 2.2

[46] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/131080.131089. 2.2, 7.6

[47] *GNU Compiler Collection (GCC) Internals*. Free Software Foundation, Boston, MA, 2006. URL `http://gcc.gnu.org/onlinedocs/gccint/`. 2.1.4

[48] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256. IEEE Computer Society Press, 2002. ISBN 0-7695-1859-1. 2.1.5

[49] Changqing Fu, Kent Wilken, and David Goodwin. A faster optimal register allocator. *The Journal of Instruction-Level Parallelism*, 7:1–31, January 2005. URL http://www.jilp.org/vol7. 2.1.5

[50] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447. 7.1

[51] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/229542.229546. 2.1.1

[52] Lal George and Matthias Blume. Taming the ixp network processor. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-662-5. doi: http://doi.acm.org/10.1145/781131.781135. 2.1.5

[53] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254, New York, NY, USA, 1978. ACM Press. doi: http://doi.acm.org/10.1145/512760.512785. 2.2

[54] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience*, 26(8):929–965, 1996. 2.1.5

[55] Daniel Grund and Sebastian Hack. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction*, Lecture Notes In Computer Science. Springer, March 2007. URL http://rw4.cs.uni-sb.de/~grund/papers/cc07-opt_coalescing.pdf. Braga, Portugal. 2.1.2

[56] Jens Gustedt, Ole A. Mæhle, and Jan Arne Telle. The treewidth of java programs. In *ALENEX '02: Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*, pages 86–97. Springer-Verlag, 2002. ISBN 3-540-43977-3. 1.1.1, 3.7

[57] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001. 3.5, 7.5

[58] Sebastian Hack. Interference graphs of programs in ssa-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005. 1.1.1

[59] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006. URL http://dx.doi.org/10.1016/j.ipl.2006.01.008. 1.1.1, 2.1.2

[60] Sebastian Hack and Gerhard Goos. Register Coalescing by Graph Recoloring. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, New York, NY, USA, 2008. ACM. doi:

10.1145/1375581.1375610. 2.1.2

[61] Michael Held, Philip Wolfe, and Harlan P. Crowder. Validation of subgradient optimization. *Mathematical Programming*, 6:62–88, 1974. 6.2

[62] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003. 2.1.5

[63] Wei-Chung Hsu, Charles N. Fisher, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng.*, 15(10):1252–1260, 1989. ISSN 0098-5589. 2.1.4, 2.1.5

[64] ILOG. ILOG CPLEX. `http://www.ilog.com/products/cplex`. 6.1.1, 6.1.1, 7.5.1

[65] Intel. Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/products/processor/manuals/`, 2009. 4.3.1, 4.3.2

[66] Sven-Olof Nyström Johan Runeson. Retargetable graph-coloring register allocation for irregular architectures. *Lecture Notes in Computer Science*, 2826:240–254, October 2003. 2.1.1

[67] Mark S. Johnson and Terrence C. Miller. Effectiveness of a machine-level, global optimizer. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 99–108, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-197-0. doi: http://doi.acm.org/10.1145/12276.13321. 2.1.1

[68] Sampath Kannan and Todd Proebsting. Register allocation in structured programs. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–368. Society for Industrial and Applied Mathematics, 1995. ISBN 0-89871-349-8. 1.1.1

[69] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193–200, September 1879. 2.1.1

[70] Christoph Kessler and Andrzej Bednarski. A dynamic programming approach to optimal integrated code generation. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 165–174, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-425-8. doi: http://doi.acm.org/10.1145/384197.384219. 2.2

[71] Robert R. Kessler. Peep: an architectural description driven peephole optimizer. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 106–110, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. doi: http://doi.acm.org/10.1145/502874.502884. 2.2

[72] David Koes and Seth Copen Goldstein. An analysis of graph coloring register allocation. Technical Report CMU-CS-06-111, Carnegie Mellon University, March 2006. URL `http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-111.html`. 2.6, 2.1.6

[73] David Koes and Seth Copen Goldstein. Register allocation deconstructed. In *SCOPES '09: Proceedings of the 12th international workshop on Software & compilers for embed-*

ded systems, 2009. 2.7, 3.17, 3.19

[74] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Transactions on Design Automation of Electronic Systems.*, 1(2):251–279, 1996. URL `citeseer.nj.nec.com/kolson96optimal.html`. 2.1.5

[75] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307. IEEE Computer Society Press, 1998. ISBN 1-58113-016-3. 2.1.5

[76] Akira Koseki, Hideaki Komatsu, and Toshio Nakatani. Preference-directed graph coloring. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 33–44, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512535. 2.1.1

[77] Rick Kufrin. Perfsuite: an accessible, open source performance analysis tool for linux. In *Proceedings of the 6th international conference on Linux Clusters*, April 2005. 4.2.2

[78] Steven M. Kurlander and Charles N. Fischer. Zero-cost range splitting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 257–265, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178420. 2.1.1

[79] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997. URL `http://www.icsl.ucla.edu/~billms/Publications/mediabench.ps`. 3.5, 6.1.1, 7.5

[80] Claude Lemaréchal. *Computational Combinatorial Optimization: Optimal or Provably Near-Optimal Solutions*, volume 2241 of *Lecture Notes in Computer Science*, chapter Lagrangian Relaxation, pages 112–156. Springer-Verlag Heidelberg, 2001. 6.1.2

[81] Rainer Leupers. Code generation for embedded processors. In *Proceedings of the 13th international symposium on System synthesis*, pages 173–178. ACM Press, 2000. ISBN 1080-1082. doi: http://doi.acm.org/10.1145/501790.501827. 2.2

[82] Rainer Leupers. Code selection for media processors with simd instructions. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 4–8, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-244-1. doi: http://doi.acm.org/10.1145/343647.343679. 2.2

[83] Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):794–814, 2000. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/362652.362661. 2.2

[84] S. Liao, K. Keutzer, S. Tjiang, and S. Devadas. A new viewpoint on code generation for directed acyclic graphs. *ACM Trans. Des. Autom. Electron. Syst.*, 3(1):51–75, 1998. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/270580.270583. 2.2

[85] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binate covering for code size optimization. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 393–399, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7213-7. 2.2

[86] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *CC'99: 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*. Springer, March 1999. ISBN 3-540-65717-7. 2.1.4, 2.1.5

[87] LLVM. The LLVM compiler infrastructure. `http://llvm.org`. 2.1.3, 4, 5.1.2, 7.4

[88] Guei-Yuan Lueh and Thomas Gross. Call-cost directed register allocation. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 296–307. ACM Press, 1997. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258942. 2.1.1

[89] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst.*, 22(3):431–470, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/353926.353929. 2.1.4

[90] Ananth R. Madabushi. Lagrangian relaxation / dual approaches for solving large-scale linear programming problems. Master's thesis, Virginia Polytechnic Institute and State University, February 1997. 6.2.2, 6.2.3

[91] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/364995.365000. 2.2

[92] Waleed M. Meleis and Edward S. Davidson. Optimal local register allocation for a multiple-issue machine. In *Proceedings of the 8th international conference on Supercomputing*, pages 107–116. ACM Press, 1994. ISBN 0-89791-665-4. doi: http://doi.acm.org/10.1145/181181.181318. 2.1.5

[93] C. Robert Morgan. *Building an Optimizing Compiler.* Butterworth, 1998. ISBN 1-55558179-X. 2.1, 2.1.4

[94] T. S. Motkin and I.J. Schoenberg. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6:393–404, 1954. 6.2.2

[95] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4. 2.1, 2.2

[96] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 120–129. ACM Press, 2002. ISBN 1-58113-527-0. doi: http://doi.acm.org/10.1145/513829.513851. 2.1.5

[97] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. *Trans. on Embedded Computing Sys.*, 3(1):163–181, 2004. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/972627.972635. 2.1.5, 2.2

[98] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-based global live-range splitting. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN confer-*

ence on Programming language design and implementation, pages 216–227, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134007. 2.1.1

[99] George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, NH, USA, 1999. ISBN 0-471-35943-2. 6.2.2

[100] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 266–277, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178427. 2.1.4

[101] Mizuhito Ogawa, Zhenjiang Hu, and Isao Sasano. Iterative-free program analysis. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 111–123. ACM Press, 2003. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944716. 1.1.1, 2.1.5

[102] Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of the thirteenth Australasian symposium on Theory of Computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. ISBN 1-920-68246-5. 1.1.1, 2.1.2

[103] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1011508.1011512. 2.1.1

[104] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application burs theory. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73586. 2.2, 7.6

[105] Fernando Quintao Pereira and Jens Palsberg. Register allocation after classical ssa elimination is np-complete. In *Proceedings of FOSSACS'06, Foundations of Software Science and Computation Structures*. Springer-Verlag (LNCS), March 2006. 1.1.1, 2.1.2

[106] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/330249.330250. 2.1.3

[107] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-43108-5. 6.2.3

[108] Todd Proebsting. Least-cost instruction selection in dags is NP-complete. `http://research.microsoft.com/~toddpro/papers/proof.htm`. 2.2, 7.1, 7.1

[109] Todd A. Proebsting. Simple and efficient burs table generation. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 331–340, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-475-9. doi: http://doi.acm.org/10.1145/143095.143145. 2.2

[110] Todd A. Proebsting. Burs automata generation. *ACM Trans. Program. Lang. Syst.*, 17(3): 461–486, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/203095.203098. 2.2, 7.6

[111] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 300–310, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-475-9. doi: http://doi.acm.org/10.1145/143095.143142. 2.1.4

[112] Todd A. Proebsting and Charles N. Fischer. Demand-driven register allocation. *ACM Trans. Program. Lang. Syst.*, 18(6):683–710, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/236114.236117. 2.1.4

[113] Rodric M. Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, June 2004. 7.5

[114] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345220. 8

[115] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *Proceedings of the 2007 International Conference on Compiler Construction*, March 2007. 2.1.3, 2.1.3, 4, 5.5

[116] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 139–148. ACM Press, 2002. ISBN 1-58113-527-0. doi: http://doi.acm.org/10.1145/513829.513854. 2.1.5

[117] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201737191. 4.3.3, 4.3.4

[118] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321607.321620. 2.2, 7.4

[119] Hanif D. Sherali and Gyunghyun Choi. Recovery of primal solutions when using subgradient optimization methods to solve lagrangian duals of linear programs. *Operations Research Letters*, 19:105–113, September 1996. 6.2.3

[120] Hanif .D. Sherali and Osman Ulular. A primal-dual conjugate algorithm for specially structured linear and convex programming problems. *Applied Mathematics and Optimization*, 20:193–221, 1989. 6.2.3, 6.2.3

[121] N. Z. Shor, Krzysztof C. Kiwiel, and Andrzej Ruszcayǹski. *Minimization methods for non-differentiable functions*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-12763-1. 6.2

[122] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/996893.996875. 2.1.1

[123] SPEC. SPEC CPU2006 benchmark suite. `http://www.spec.org`, 2006. 3.5, 4.1, 7.5

[124] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1997.2697. 1.1.1, 2.1.5, 3.7

[125] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277714. 2.1.3, 2.1.3

[126] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Department of operations and research and financial engineering, Princeton university, 2001. URL `www.princeton.edu/~rvdb/LPbook/onlinebook.pdf`. 6.1.1, 6.1.1

[127] Steven R. Vegdahl. Using node merging to enhance graph coloring. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 150–154, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-094-5. doi: http://doi.acm.org/10.1145/301618.301657. 2.1.1

[128] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1064998. 2.1.3, 2.1.3

[129] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–55, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: http://dx.doi.org/10.1109/MICRO.2007.10. 8

[130] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. doi: http://doi.acm.org/10.1145/192724.192725. 3.5

[131] T. Zeitlhofer and B. Wess. A comparison of graph coloring heuristics for register allocation based on coalescing in interval graphs. *Proceedings of the 2004 International Symposium on Circuits and Systems*, 4:IV–529–32 Vol.4, May 2004. 2.1.2

[132] Thomas Zeitlhofer and Bernhard Wess. Optimum register assignment for heterogeneous register-set architectures. In *Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 3, pages III–252–III–244, May 2003. 1.1.1, 2.1.2