Thesis Proposal
Towards a More Principled Compiler: Progressive Backend Compiler Optimization

David Ryan Koes

August 2006

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 © 2006 David Ryan Koes

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

Abstract

As we reach the limits of processor performance and architectural complexity increases, more principled approaches to compiler optimization are necessary to fully exploit the performance potential of modern architectures. Existing compiler optimizations are typically heuristic-driven and lack a detailed model of the target architecture. In this proposal I develop the beginnings of a framework for a principled backend optimizer.

Ideally, a principled compiler would consist of tightly integrated, locally optimal, optimization passes which explicitly and exactly model and optimize for the target architecture. Towards this end this proposal investigates two pivotal backend optimizations: register allocation and instruction selection. I propose to tightly integrate these optimizations in an expressive model which can be solved progressively, approaching optimality as more time is allowed for compilation.

I present an expressive model for register allocation based on multi-commodity network flow that explicitly captures the important components of register allocation such as spill code optimization, register preferences, coalescing, and rematerialization. I also describe a progressive solution technique for this model that utilizes the theory of Lagrangian relaxation and domain-specific heuristics to approach the optimal solution and provide optimality-bound guarantees on solutions. As future work, I discuss some improvements that can be made to this model and solution technique to improve their performance and usefulness, and I sketch how I believe this model and solution technique can be extended to incorporate instruction selection and present some preliminary results that indicate the benefit achievable from such an integration.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

As we reach the limits of processor performance and architectural complexity increases, more principled approaches to compiler optimization are necessary to fully exploit the performance potential of modern architectures. Existing compiler optimization frameworks are lacking in that

- many optimization passes use an extremely simplified model of the target architecture
- the various optimization passes are not tightly integrated, and
- not all optimization passes are internally optimal.

A more principled approach to compiler optimization must address all three of these points.

Many compiler optimization passes use a simplified model of the target architecture and, as a result, may actually produce less optimized code. As an example, partial redundancy elimination (PRE), which might be (falsely) considered a target-independent optimization, depends crucially on the register resources of the target architecture. This architectural dependency is typically simply modeled by a heuristic which crudely estimates the benefit of eliminating a redundant expression and the cost of introducing a long-lived temporary to hold the value of that expression. As a result, the application of PRE can sometimes reduce performance. For instance, the PRE pass of the GNU `gcc` version 3.4.4 compiler, which improves performance on some SPEC2000 benchmarks by as much as 4.3%, also decreases the performance of some benchmarks by as much as 2% (overall, it provides an average improvement). These slowdowns are caused by increased spilling within loops as a result of aggressive PRE. Even optimizations that are intrinsically linked to architectural features, such as register allocation, may use inappropriately simple architectural models. 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 would need to explicitly represent and optimize for these features.

The importance of leveraging architectural features can be seen even in today's compilers by comparing the performance of code compiled for a generic x86 processor and code compiled to run specifically on a Pentium 4. The performance of SPEC2000 benchmarks improves by as much as 33% using `gcc` 3.4.4 (although the overall average improvement is less than 1%) and as much as 100% using the Intel compiler `icc` version 9.0 (with an average improvement of 14%). The effect of simplistic architectural models and a lack of tight integration between compiler passes is further highlighted by studies in adaptive and iterative compilation [7, 46, 53, 70, 88, 103] that search for better orderings and combinations of existing compiler optimizations. These studies get between 5% and 20% average performance improvements on SPEC benchmarks, with some SPEC benchmarks increasing in performance by as much as 75%, and can get up to factors of four performance improvements on some numerical kernels.

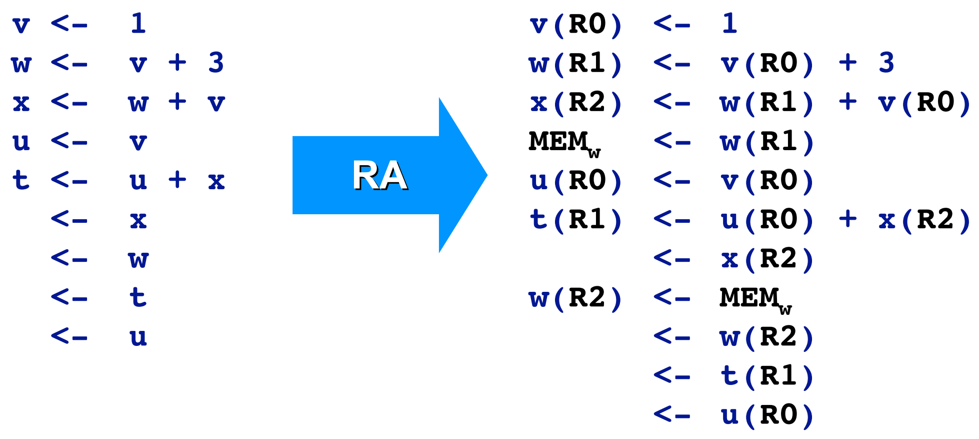
The general optimization problem, finding a correct instruction sequence that results in the shortest possible execution time, is clearly undecidable since such an optimizer could be used to solve the halting problem. However, if instead of seeking a result that is optimal in the most general case, we consider only the optimality of a specific optimization at performing its particular task, it potentially becomes tractable to design optimal algorithms. For example, dead-code elimination may not be able to remove all code that is not executed for all inputs of a program, but it can eliminate all code that is dead in a meets over all paths static analysis. In this sense dead-code elimination can be thought of as *internally optimal*; given a restricted, but reasonable, definition of the problem (remove all static dead code) it finds the optimal result. In contrast, some compiler optimizations, such as register allocation and instruction scheduling, are provably NP-hard for even simple representations of the problem. In these cases it is unlikely that internally optimal efficient algorithms exist. However, internal optimality can be approached and the trade off between compile time and optimality made explicit through the use of *progressive compilation*.

Progressive compilation bridges the gap between fast heuristics and slow optimal algorithms. A progressive optimization 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 of selecting an optimization level, a programmer *explicitly* trades compilation time for improved optimization.

The goal of the work in this proposal is to move towards a more principled compiler. Given their substantial influence on all other optimizations, I plan to investigate the crucial backend optimizations of register allocation and instruction selection. I propose to tightly integrate these optimizations in an expressive model which can be solved progressively. Successfully achieving this goal will result in compilers which can more capably exploit the performance potential of modern architectures.

1.2 Problem Description

Register allocation and instruction selection are essential passes of any compiler backend. Together they are responsible for finalizing a compiler's intermediate representation of code into machine executable assembly. As such, it is important to define what these passes entail and to characterize their difficulty.



```
v <-  1                  v(R0) <-  1
w <-  v + 3              w(R1) <-  v(R0) + 3
x <-  w + v              x(R2) <-  w(R1) + v(R0)
u <-  v                  MEM_w <-  w(R1)
t <-  u + x              u(R0) <-  v(R0)
  <-  x                  t(R1) <-  u(R0) + x(R2)
  <-  w                        <-  x(R2)
  <-  t                  w(R2) <-  MEM_w
  <-  u                        <-  w(R2)
                               <-  t(R1)
                               <-  u(R0)
```

Figure 1.1: 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.

1.2.1 Register Allocation

The *register allocation problem* is to find a desirable assignment of program variables to memory locations and hardware registers as illustrated by Figure 1.1. 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 with 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 unfortunately 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 only the available registers. That is, it is not necessary to *spill*, store to memory, a variable. It is this problem that Chaitin et. al. [29] 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 [16], which includes all programs written in Java and `goto`-free C [51, 101], the register sufficiency problem can be solved in linear time (but exponential in the constant number of registers) [15, 90] or constant factor approximation algorithms can be used [60, 101]. For programs that are in SSA form, the register sufficiency problem is also readily solved [24, 52], although converting out of SSA form remains difficult [92].

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 which stores and loads values to and from memory. Determining the minimum number of loads and stores needed is NP-hard even for the local case [38]. In some cases the register allocator may be able to avoid spilling by rematerializing a know 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]. 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. Solving the register sufficiency problem, even in the local case, in the presence of such constraints is NP-hard [107], although fixed parameter tractable in the number of registers [14]. In order to generate quality code, a register allocator must take register preferences into account.

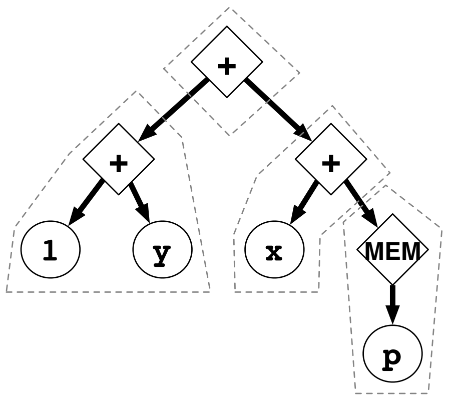
The register allocation problem is an NP-hard problem consisting of several important components. In order to generate quality code, a register allocator must not only perform register assignment, but also optimize spill code, perform coalescing and rematerialization, and take register preferences into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

1.2.2 Instruction Selection

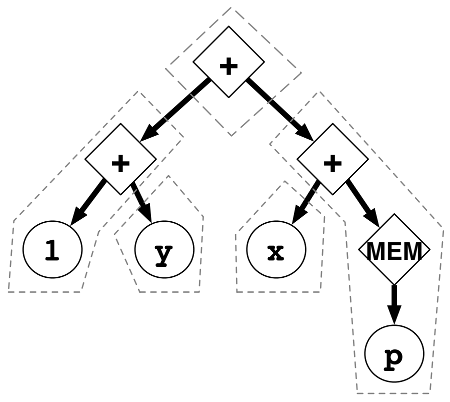
The *instruction selection problem* is to find an efficient conversion from the compiler's target-independent intermediate representation (IR) of a program to a target-specific assembly listing. An example of instruction selection, where a tree-based IR is converted to x86 assembly, is shown in Figure 1.2. 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.

In the most general case, 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 selection is usually defined as finding an optimal *tiling* of the intermediate code with predefined tiles of machine instructions. 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. The instruction selection problem is difficult even for basic blocks since straight-line code can be represented as a directed acyclic graph (DAG) [3] and optimal tiling of DAGs is known to be NP-complete even for simple machine models [25]. However, if the code is represented as a sequence of expression trees (i.e., there are no common sub-expressions explicit in the representation), then efficient optimal tiling algorithms exist [1].

Although optimal instruction selection algorithms exist for tree-based intermediate representations, the actual optimality of the result is limited by the accuracy of the costs associated with each tile. If instruction selection is done independently from register allocation, these tile costs are inherently inaccurate since spills, register preferences, and move coalescing may change the instructions corresponding to a tile. For example, in Figure 1.2, which instruction sequence is better is determined by the ability of the register allocator to coalesce certain variables and eliminate the cost of moves. An instruction selection algorithm that integrates with the register allocator would be able to assign more accurate costs to tiles, but would also inherit the NP-hardness of register allocation.



(a)



(b)

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

(c)

```
movl  x,t1
addl  t1,(p)
movl  y,t2
incl  t2
movl  t2,r
addl  r,t1
```

(d)

```
movl  (ecx),ebx
leal  (edx,ebx),edx
leal  1(eax),eax
leal  (edx,eax),eax
```

(e)

```
movl  edx,edx
addl  edx,(ecx)
movl  eax,eax
incl  eax
movl  eax,eax
addl  eax,edx
```

(f)

Figure 1.2: An example of instruction selection on a tree-based IR. Two possible tilings, (a) and (b), with their corresponding instruction sequences, (c) and (d), are shown. Although sequence (c) is shorter, it is possible that register allocation will be able to coalesce the move instructions in (d) resulting in an even shorter sequence as illustrated by (e) and (f).

Chapter 2

Related Work

Register allocation is a fundamental part of any compiler backend and has been extensively studied. The textbook [5, 8, 34, 83, 84] approach to register allocation is to represent the 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 such as the x86. Less limited methods of register allocation which use more 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.

Although instruction selection by itself has been extensively studied, the integration of instruction selection with register allocation remains an open problem. Through the use of register allocation aware instruction selection, the thesis will build on the existing body of knowledge to more tightly integrate these two key compiler passes.

2.1 Graph Coloring Register Allocation

A traditional graph coloring allocator constructs an interference graph which is then labeled with "colors" representing each of k available registers.

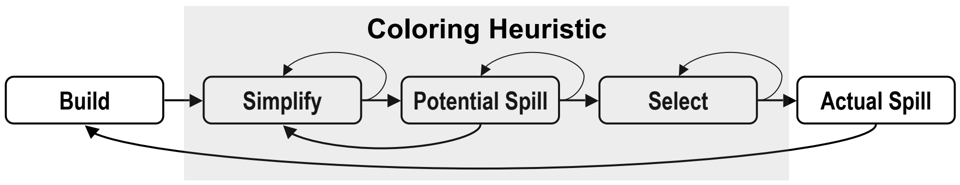


Figure 2.1: The flow of a traditional graph coloring algorithm.

2.1.1 Algorithm

The traditional optimistic graph coloring algorithm [20, 23, 28] consists of five main phases as shown in Figure 2.1:

Build An interference graph is constructed using the results of data flow analysis. A node in the graph represents a variable. An edge connects two nodes if the variables represented by the nodes interfere and cannot be allocated to the same register. 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. In the most commonly used heuristic [61] any node with degree less than k, where k is the number of available registers, is removed from the graph and placed on a stack. This is repeated until all nodes are removed, in which case we skip to the Select phase, or no further simplification is possible. More complicated heuristics [78, 104] can also be used to further simplify the graph.

Potential Spill If only nodes with degree greater than k are left, we mark a 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. We now pop the nodes off the stack. If the node was not marked as a potential spill node then there must be a color we can assign this node that does not conflict with any colors already assigned to this node's neighbors. If it 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 If any nodes are marked as actual spills, we generate spill code which loads and stores the variables represented by these nodes 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.

Note that 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 [34]. A "top-down" coloring uses high-level program information instead of interference graph structure to determine a priority coloring order [30, 31] 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.

2.1.2 Improvements

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

Web Building [28, 59] Instead of a node in the interference graph representing all the live ranges of a variable, a node can just represent 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. Each web can then be allocated to a different register, even though they represent the same variable.

Coalescing [23, 28, 47, 91] 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 [13] A heuristic is used when determining what node to mark in the Potential Spill stage. An ideal node to mark is one with a low spill cost (requiring only a small number of dynamic loads and stores to spill) but one whose absence will make the interference graph easier to color and therefore reduce the number of future potential spill nodes.

Improved Spilling [12, 23, 33] 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. Techniques that perform such live range splitting can be applied before or during register allocation [33, 71, 87].

2.1.3 Limitations

The graph coloring model of register allocation has several fundamental limitations. The basic graph coloring model is only effective at solving the register sufficiency problem. It must be extended in an *ad hoc* fashion in order to incorporate other components of the register allocation problem. The graph coloring model implicitly assumes a uniform register model and so must be further extended to target irregular architectures [21, 23, 58, 69, 100]. However, as we shall see, the graph coloring register allocation is not well suited for targeting irregular and constrained architectures.

Simply solving the register sufficiency problem is not enough to obtain quality code. As shown in Figure 2.2, 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 2.2 had to generate spill code, it is clearly important that the compiler explicitly optimize spill code. The importance of components besides register sufficiency and the shortcomings of the graph coloring model are further demonstrated in Figure 2.3, which shows the effect of replacing the heuristic coloring algorithm in a traditional graph coloring allocator with an optimal allocator as described in [65]. 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 (no explicit cost model is used), the results are mixed with the optimal-coloring based allocator performing more

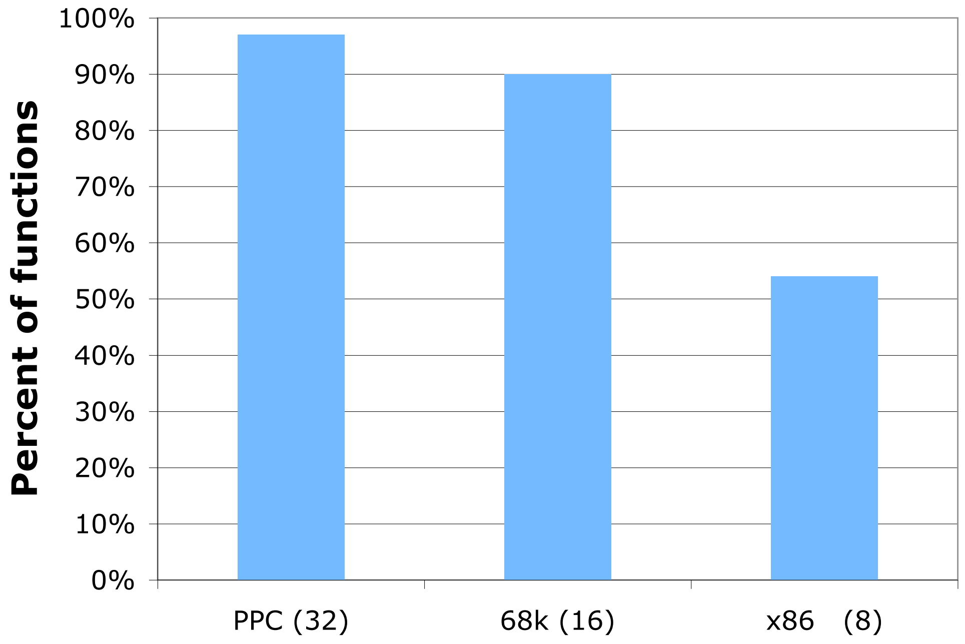


Figure 2.2: The percent of over 10,000 functions assembled from various benchmark suites for which no spilling is necessary. Three architectures with different numbers of registers are evaluated. In these cases a graph coloring based allocator successfully found an allocation without spilling. Note that 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 [65].

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

2.2 Alternative Allocators

Although graph-coloring based allocators are the textbook approach to register allocation, several other approaches have been studied and implemented in production compilers. Several allocators, including gcc, separate the register allocation problem into global allocation and local allocation, each of which is done separately, while other allocators attempt to exploit program structure when performing allocation. Linear scan techniques focus on improving the speed of the register allocator itself, usually in the context of a just-in-time compiler.

Although allocators which perform local and global register allocation separately may perform global allocation first [83], typically local allocation is performed first in order to take advantage of fast and effective local register allocation algorithms [38, 56, 77]. In probabilistic register allocation [95] and demand-driven allocation [96], the results of local allocation are used by the global allocator to determine which variables get registers. In the gcc (as of version 3.4.4

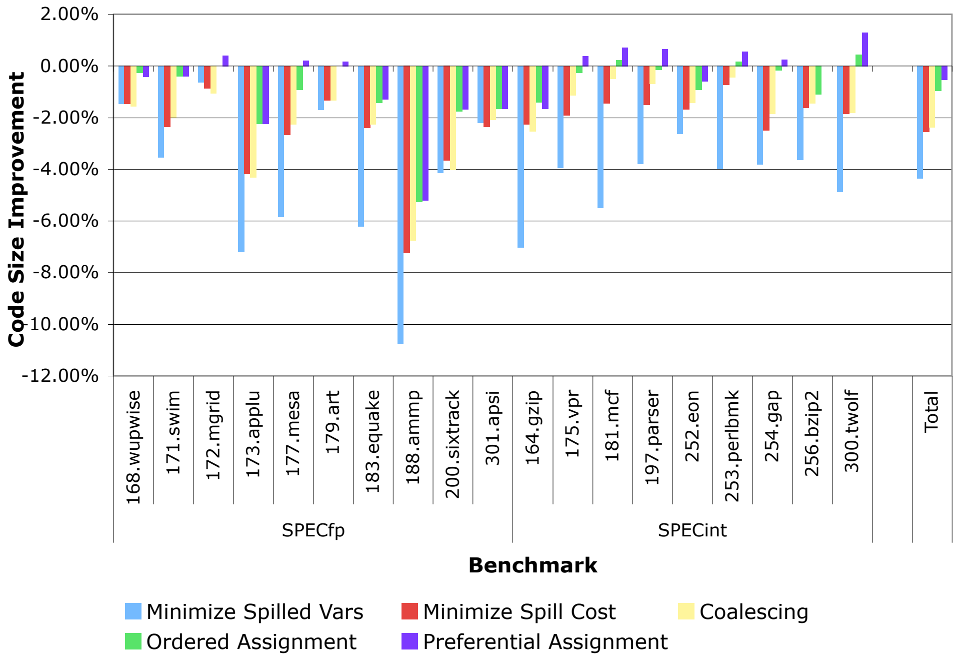


Figure 2.3: The effect of incorporating various components of register allocation into the coloring algorithm. The coloring heuristic of a traditional graph allocator is replaced with an optimal coloring algorithm. Results are shown for an algorithm that optimal minimizes the number of spilled variables, 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.

[43]) 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 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 which exploit program structure break the control flow graph into regions or tiles. In hierarchical register allocation [26, 32] 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 on a second top-down pass. A similar technique can also be used with regions derived from program dependence graphs [89]. 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 [79] 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.

Linear scan allocators find a register allocation in a single sweep of the program code. They are usually designed for just-in-time compilers and sacrifice code quality for compile-time speed. Each variable is represented by a single linear lifetime range and registers are assigned to lifetime ranges in a single quick pass [93]. This basic method can be extended to support holes in lifetime intervals, the splitting of intervals and other efficient optimizations [102, 106]. Although these improvements can result in significant benefits over the basic linear scan algorithm, linear scan allocators remain inferior to more traditional allocators in terms of code quality.

2.3 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 [56]. This algorithm has been extended to handle loops and irregular architectures [67] and multi-issue machines [82]. 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 register allocation can also be solved in polynomial space and exponential time using integer linear programming techniques [77].

The global register sufficiency problem has been solved optimally [15, 90] or approximately [101] by exploiting the bounded treewidth property of structured programs. The optimal solutions include 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 [44, 45, 50] and irregular [48, 68, 85, 86] 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 any solution (let alone the optimal solution) for most functions with more than 1000 instructions [45].

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) [55, 98]. 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.

2.4 Instruction Selection

Instruction selection, or code generation, converts the compiler's intermediate representation (IR) of code into a target-specific assembly listing. Instruction selection and register allocation are typically done as independent passes with instruction selection preceding register allocation. However, particularly with irregular architectures such as the x86, the interaction between these two passes can significantly impact code quality.

The textbook approach to instruction selection [5, 8, 34, 84] is to represent the problem as a tiling problem. 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. If the IR is a sequence of trees, this tiling can be done optimally using dynamic programming [1, 4, 99, 105], even for super-scalar machines [17]. Furthermore, code-generator generators based on this approach have been developed [4, 27, 37, 40, 41, 49, 94] which simplify the construction and maintenance of a compiler. If the IR consists of directed acyclic graphs (DAGs) then a simplified version of the problem can be solved within a constant approximation ratio using heuristics [2].

An alternative method for tiling instruction DAGs that is particularly relevant when targeting DSP [73] and SIMD [74] processors is to tile the IR as if it were a sequence of trees, but generate several possible tilings for each tree using partial tiles which my potentially be invalid. For example, two tilings, by themselves, may be invalid because they both contain half of a SIMD instruction. A pass after tiling attempts to reconcile the invalid tilings. For example, the two halves of the SIMD instruction would be combined to produce a valid tiling of the DAG. Unfortunately, the reconcile phase is itself NP-hard. When evaluated on small DSP kernels, this technique successfully increased the parallelism explicitly exposed to the processor, but, due to the use of an optimal integer-programming based reconcile phase, the compile-times did not scale beyond tree sizes of 40 operations.

Instruction selection on a DAG IR can also be represented as an instance of a binate covering problem [75, 76]. The binate covering problem is to find an assignment of boolean variables that satisfies a given set of logical clauses (consisting only of disjunctions of variables or their complement) that minimizes a given cost function. Optimal branch-and-bound based solvers can then be used to find solutions that are significantly better than optimal solutions that work on sequences of trees. (Approximately a 10% size improvement on selected basic blocks). Unfortunately, this technique does not scale well (several seconds are required to compile a single block with fewer than 100 instructions).

An alternative method of instruction selection, which is better suited for linear, as opposed to tree-like, IRs, is to incorporate instruction selection into peephole optimization [34, 36, 42, 63]. In peephole optimization [81], 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 it may be 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. Code-generator generators have also been developed using the peephole method of instruction selection [35, 39].

Several techniques that partially integrate instruction scheduling and register allocation have been developed. A tree-tiling based instruction selector can be extended to incorporate the notion of usable or efficient register classes into each tile [9, 105]. The AVIV retargetable code generator [54] performs instruction selection over a split-DAG, which additionally represents function unit resource constraints, and inserts spills (sub-optimally) during instruction selection if necessary to keep the number of live variables less than the number of registers. Similarly, instruction selection, combined with instruction duplication, has been used to reduce register pressure resulting in a better final register allocation [97]. Instruction selection and register assignment (no spilling) have been performed using an exhaustive search with memoization of the search space [62]. Due to the nature of instruction selection, if register allocation is performed, it is only a local register allocation. Otherwise, the instruction selector produces code that is hopefully easier to allocate and then the global register allocator runs independently.

Chapter 3

Completed Work

3.1 Expressive Model

In this section we describe a model of register allocation based on multi-commodity network flow. We first describe the general MCNF problem and show how to create an expressive model of register allocation for straight-line code using MCNF. We then extend the MCNF model to handle control flow. Finally, we discuss some limitations of the model. Overall, the our global MCNF model explicitly and exactly represents the pertinent components of the register allocation problem.

3.1.1 Multi-commodity Network Flow

The multi-commodity network flow (MCNF) problem is finding a minimum cost flow of commodities through a constrained network. The network is defined by nodes and edges where each edge has costs and capacities. Without loss of generality, we can also apply costs and capacities to nodes. The costs and capacities can be specific for each commodity, but edges also have *bundle constraints* which constrain the total capacity of the edge. For example, if an edge has a bundle constraint of 2 and commodities are restricted to a single unit of integer flow, at most two commodities can use that edge in any valid solution. Each commodity has a source and sink node such that the flow from the source must equal the flow into the sink. Although finding the minimum cost flow of a single commodity is readily solved in polynomial time, finding a solution to the MCNF problem where all flows are integer is NP-complete [6].

Formally, the MCNF problem is to minimize the costs of the flows through the network:

$$\min \sum_k c^k x^k$$

subject to the constraints:

$$\sum_k x_{ij}^k \leq u_{ij}$$

$$0 \leq x_{ij}^k \leq v_{ij}^k$$

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

where c^k is the cost vector containing the cost of each edge for commodity k, x^k is the flow vector for commodity k where x^k_{ij} is the flow of commodity k along edge (i, j), u_{ij} is the bundle constraint for edge (i, j), v^k_{ij} is an individual constraint on commodity k over edge (i, j), the matrix $\mathcal{N}$ represents the network topology, and the vector b^k contains the inflow and outflow constraints (source and sink information).

3.1.2 Local Register Allocation Model

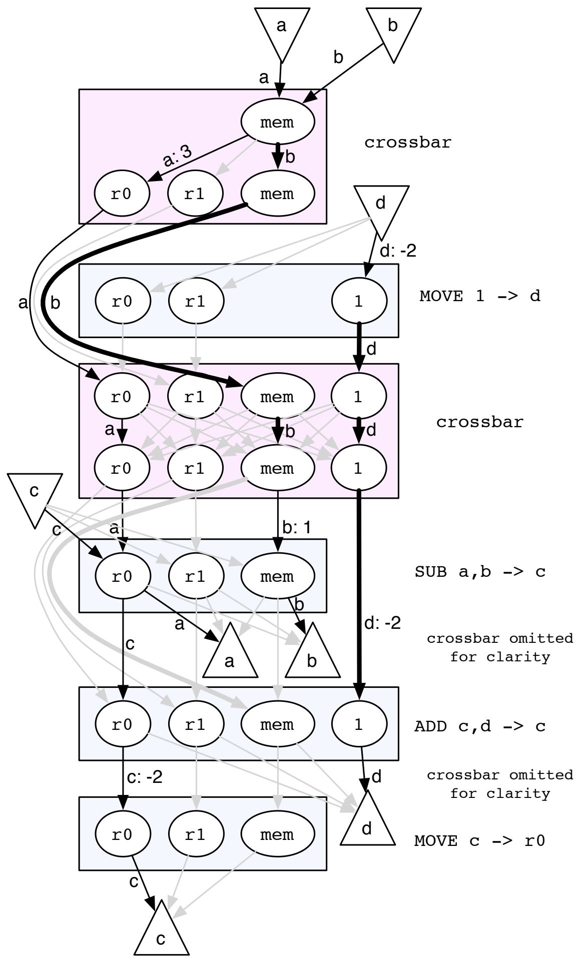
Multi-commodity network flow is a natural basis for an expressive model of the register allocation problem. A flow in our MCNF model corresponds to a detailed allocation of that variable. A simplified example of our MCNF model of register allocation is shown in Figure 3.1. Although simplified, this example demonstrates how our MCNF model explicitly represents spill costs, constant rematerialization, and instruction register usage constraints and preferences.

The commodities of the MCNF model correspond to variables. The design of the network and individual commodity constraints is dictated by how variables are used. The bundle constraints enforce the limited number of registers available and model instruction usage constraints. The edge costs are used to model both the cost of spilling and the costs of register preferences.

Each node in the network represents an allocation class: a register, constant class, or memory space where a variable's value may be stored. Although a register node represents exactly one register, constant and memory allocation classes do not typically correspond to a single constant or memory location. Instead they refer to a class of constants or memory locations that are all accessed similarly (e.g., constant integers versus symbolic constants).

Nodes are grouped into either instruction or crossbar groups. There is an instruction group for every instruction in the program and a crossbar group for every point between instructions. An instruction group represents a specific instruction in the program and contains a single node for each allocation class that may be used by the instruction. The source node of a variable connects to the network at the defining instruction and the sink node of a variable removes the variable from the network immediately after the last instruction to use the variable. The nodes in an instruction group constrain which allocation classes are legal for the variables used by that instruction. For example, if an instruction does not support memory operands, such as the load of the integer constant one in Figure 3.1, then no variables are allowed to flow through the memory allocation class node. Similarly, if only a single memory operand is allowed within an instruction, the bundle constraints of the instruction's memory edges are set to 1. This is illustrated in Figure 3.1 by the thin edges connecting to the memory node of the `SUB` instruction group. Variables used by an instruction must flow through the nodes of the corresponding instruction group. Variables not used by the instruction bypass the instruction into the next crossbar group. This behavior can been seen in the behavior of variables a and b in Figure 3.1. The flows of these variables bypass the first instruction but are forced to flow through the `SUB` instruction.

Crossbar groups are inserted between every instruction group and allow variables to change allocation classes. For example, the ability to store a variable to memory is represented by an edge within a crossbar group from a register node to a memory allocation class node. In Figure 3.1 the variable a, which is assumed to start as a parameter on the stack, flows from the memory node to `r0`, which corresponds to a load. The crossbar groups shown in Figure 3.1 are full crossbars which means that for some allocations the use of swap instructions, instead of a



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

Source code of example

```
MOVE 1 -> d
SUB a,b -> c
ADD c,d -> c
MOVE c -> r0
```

Assembly before register allocation

```
MOVE STACK(a) -> r0
SUB r0,STACK(b) -> r0
INC r0
```

Resulting register allocation

Figure 3.1: A simplified example of the multi-commodity network flow model of register allocation. Thin edges have a capacity of 1 (as only one variable can be allocated to a register and instructions only support a single memory operand). A thick edge indicates that the edge is uncapacitated. For clarity, edges not used by the displayed solution are in gray and much of the capacity and cost information is omitted. The commodity and cost along each edge used in the solution are shown if the cost is non-zero. In this example the cost of a load is 3, the cost of using a memory operand in the `SUB` instruction is 1, the benefit (negative cost) of allocating c to `r0` in the final `MOVE` instruction is 2 since the move can be deleted in this case. Similarly, allocating d to a constant when it is defined has a benefit of 2. If an operand of the `ADD` instruction is the constant one, then a benefit of 2 is accrued because the more efficient `INC` instruction can be used. The total cost of this solution is -2.

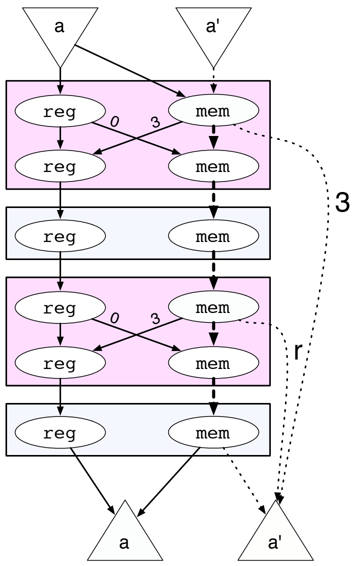
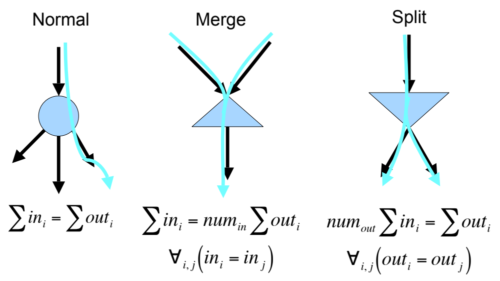


Figure 3.2: An example of anti-variables. The anti-variable of a, a', is restricted to the memory subnetwork (dashed edges). The edge r is redundant and need not be in the actual network. The cost of the second store can be paid by the first edge. If the r edge is left in the graph, it would have a cost of three, the cost of a store in this example. Multiple anti-variable eviction edges can also be used to model the case where stores have different costs depending on their placement in the instruction stream.

simple series of move instructions, might be necessary. If swap instructions are not available or are not efficient relative to simple moves, a more elaborate zig-zag crossbar structure can be used.

The cost of an operation, such as a load or move, can usually be represented by a cost on the edge that represents the 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. That is, values in memory are persistent, unlike those in registers which are assumed to be overwritten.

In order to model the persistence of data in memory, we introduce the notion of anti-variables which are used as shown in Figure 3.2. An anti-variable is restricted to the memory subnetwork and is constrained such that it cannot coexist with its corresponding variable along any memory edge. An anti-variable can either leave the memory sub-network when the variable itself exits the network or the cost of a store can be paid to leave the memory sub-network early. There is no cost associated with edges from registers to memory, but for these edges to be usable, the anti-variable must be evicted from memory. The cost of evicting the anti-variable is exactly the cost of a single store. In this way a variable may flow 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). An actual store is only generated for the first move to memory.



Normal
Merge
Split
$\sum in_i = \sum out_i$
$\sum in_i = num_{in} \sum out_i$
$\forall_{i,j}(in_i = in_j)$
$num_{out} \sum in_i = \sum out_i$
$\forall_{i,j}(out_i = out_j)$

Figure 3.3: The types of nodes in a global MCNF representation of register allocation. The merge/split nodes not only modify the traditional flow equations with a multiplier, but also require uniformity in the distribution of inputs/outputs.

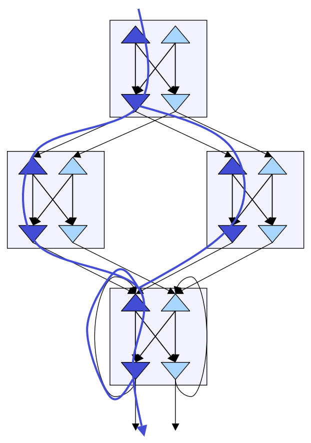


Figure 3.4: A GMCNF based representation of an register allocation with a sample allocation shown with the thicker line. Each block can be thought of as a crossbar where the cost of each edge is the shortest path between a given merge and split node.

3.1.3 Global Register Allocation Model

Although the described MCNF model is very expressive and able to explicitly model many important components of register allocation, it is unsuitable as a model of global register allocation since it does not model control flow. In order to represent the global register allocation problem, boundary constraints are added to link together the local allocation problems. These constraints are represented by *split* and *merge* nodes as shown in Figure 3.3.

Similar to normal nodes, split and merge nodes represent a specific allocation class. Merge nodes denote the entry to a basic block. A variable with a flow through a specific merge node is allocated to that allocation class at the entry of the relevant block. The merge property of the merge node, as enforced by the flow equations in Figure 3.3, requires that a variable be allocated to the same allocation class at the entry of a block as at the exit of all of the predecessors of the block. Similarly, a split node requires that an allocation of a variable at the exit of a block match the allocation at the entry to each of the successors to the block.

More formally, we add the following equality constraint for every commodity k and for every pair $(split, merge)$ of connected split and merge nodes to the definition of the MCNF problem given in Section 3.1.1:

$$x^k_{in,split} = x^k_{merge,out}$$

Note that split nodes are defined to have exactly one incoming edge and merge nodes to have exactly one outgoing edge. We refer to these constraints as the *boundary constraints*. These constraints replace the normal flow constraint between nodes for split and merge nodes.

A simplified example of a single allocation in the global MCNF model is shown in Figure 3.4. In this example, the full MCNF representation of each basic block is reduced to a simple crossbar. Unlike the local MCNF model, finding the optimal allocation for a single variable is not a simple shortest path computation. In fact, for general flow graphs the problem is NP-complete (by a reduction from graph coloring).

3.1.4 Limitations

Our global MCNF model can explicitly model instruction usage constraints and preferences, spill and copy insertion, and constant rematerialization. In addition, our model can model a limited amount of register-allocation driven instruction selection. For example, in Figure 3.1 the model explicitly encodes the fact that if an operand of the `ADD` instruction is the constant one, a more efficient `INC` instruction can be used. However, the model can not currently represent inter-variable register usage preferences or constraints. That is, the model can not 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 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 saving in code size cannot be exactly represented in our model because edge costs only apply to the flow of a single variable. If the instruction stream was modified so that a move from a to b were performed before the sign extension and the sign extension had b as its only operand, then the model would be capable of exactly representing the cost savings of allocating b to `eax` with the caveat of requiring a more constrained instruction stream as input.

Another example where inter-variable register usage preferences are useful is in the modeling of 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 currently is not capable of representing such a constraint.

Our model does not explicitly represent the benefits of move coalescing. Instead, 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. Inter-variable register usage preferences are necessary in order to exactly represent the move coalescing component of register allocation.

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 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. If desired, this limitation can be removed by using a technique similar to how anti-variables are used to model stores.

3.2 Progressive Solvers

In this section we present a progressive solver for the global MCNF problem. This solver quickly finds a solution using heuristic allocators and then uses iterative subgradient optimization to find the Lagrangian prices of the network. These prices are used in each iteration by heuristic allocators to find progressively better solutions. We first describe our use of the theory of Lagrangian relaxation and then describe and discuss two heuristic allocators (additional heuristic allocators are discussed in [64]).

3.2.1 Lagrangian Relaxation

Ideally, we would like to build a solution from simple shortest path computations. Each individual variable's shortest path would need to take into account not only the immediate costs for that variable, but also the marginal cost of that allocation with respect to all other variables. Lagrangian relaxation provides a formal way of computing these marginal costs.

Lagrangian relaxation is a general solution technique [6, 72] that removes one or more constraints from a problem and integrates them into the objective function using Lagrangian multipliers resulting in a more easily solved Lagrangian subproblem. In the case of multi-commodity network flow, the Lagrangian subproblem is to find a price vector w such that $L(w)$ is maximal, where $L(w)$ is defined:

$$L(w) = \min \sum_k c^k x^k + \sum_{(i,j)} w_{ij} \left(\sum_k x_{ij}^k - u_{ij} \right) \tag{3.1}$$

which can be rewritten as:

$$L(w) = \min \sum_{k} \sum_{(i,j)} \left(c_{ij}^{k} + w_{ij}\right) x_{ij}^{k} - \sum_{(i,j)} w_{ij} u_{ij} \tag{3.2}$$

subject to

$$x_{ij}^{k} \geq 0$$

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

$$\sum_{i} x_{i,split}^{k} = \sum_{j} x_{merge,j}^{k}$$

The bundle constraints have been integrated into the objective function. If an edge x_{ij} is over-allocated, then the term $\sum_k x_{ij}^k - u_{ij}$ will increase the value of the objective function, making it less likely that an over-allocated edge will exist in a solution that minimizes this objective function. The w_{ij} terms are the Lagrangian multipliers, called prices in the context of MCNF. The prices, w, are arguments to the subproblem and it is the flow vectors, x^k, that are the free variables in the minimization problem. The Lagrangian subproblem is still subject to the same network and individual flow constraints as in the MCNF problem. As can be seen in (3.2), the minimum solution to the Lagrangian subproblem decomposes into the minimum solutions of the individual single commodity problems.

Unfortunately, in our global MCNF model the individual single commodity problem remains NP-complete because of the boundary constraints. Fortunately, the boundary constraints can also be brought into the objective function using Lagrangian multipliers:

$$L(w) = \min \sum_{k} \sum_{(i,j)} \left(c_{ij}^{k} + w_{ij}\right) x_{ij}^{k} - \sum_{(i,j)} w_{ij} u_{ij} + \sum_{(split,merge)} w_{split,merge}^{k} \left(x_{merge,out}^{k} - x_{in,split}^{k}\right) \tag{3.3}$$

subject to

$$x_{ij}^{k} \geq 0$$

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

Since there are no normal flow constraints between split and merge nodes, the solution to (3.3) is simply a set of disconnected single commodity flow problems.

The function $L(w)$ has several useful properties [6]. Let $L^* = max_w L(w)$, then L^* provides a lower bound for the optimal solution value. Furthermore, a solution, x, to the relaxed subproblem which is feasible in the original MCNF problem is likely to be optimal. In fact, if the solution obeys the complementary slackness condition, it is provably optimal. The complementary slackness condition simply requires that any edge with a non-zero price be used to its full capacity in the solution.

We solve for L^* using an iterative subgradient optimization algorithm. At a step q in the algorithm, we start with a price vector, w^q, and solve $L(w^q)$ for x^k to get an optimal flow vector,

y^k, by performing a multiple shortest paths computation in each block. We then update w using the rules:

$$w_{ij}^{q+1} = \max\left(w_{ij}^q + \theta_q\left(\sum_k y_{ij}^k - u_{ij}\right), 0\right)$$

$$w_{split,merge}^{k\ \ q+1} = w_{split,merge}^{k\ \ q} + \theta_q\left(y_{merge,out}^k - y_{in,split}^k\right)$$

where θ_q is the current step size. This algorithm is guaranteed to converge if θ_q satisfies the conditions:

$$\lim_{q\to\infty} \theta_q = 0$$

$$\lim_{q\to\infty} \sum_{i=1}^{q} \theta_i = \infty$$

An example of a method for calculating a step size that satisfies these conditions is the ratio method, $\theta_q = 1/q$. More sophisticated techniques to calculate the step size and update the prices [10, 80] can also be used.

Although the iterative subgradient algorithm is guaranteed to converge, it is not guaranteed to do so in polynomial time. Furthermore, L^* does not directly lead to an optimal solution of the original, unrelaxed global MCNF problem. However, the Lagrangian prices can be used to effectively guide the allocation algorithms towards better solutions and to provide optimality guarantees.

3.2.2 Progressive Solver

We combine the Lagrangian relaxation technique with allocation heuristics to create a progressive solver. The solver first finds an initial solution in the unpriced network. Then, in each iteration of the iterative subgradient algorithm, the current set of prices are used to find another feasible solution. When finding solutions in the priced network, the allocation heuristics compute shortest paths using edge and boundary prices in addition to edge costs. Global information, such as the interference graph, is not used except to break ties between identically priced paths. Instead, the allocators rely exclusively 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 feasible solution to the global MCNF problem whose cost in the priced network is as close as possible to the cost of the unconstrained solution found during the update step of the subgradient algorithm. If the algorithm is successful and the found solution obeys the complementary slackness condition, then the solution is provably optimal. When selecting among similarly priced allocation decisions, we can increase the likelihood that the solution will satisfy the complementary slackness condition by favoring allocations with the lowest unpriced cost.

We present an iterative heuristic allocator, which attempts to find the best allocation on a variable-by-variable basis, a simultaneous heuristic allocator, which attempts to find the best allocation on a block-by-block basis, and a trace-based simultaneous allocator which extends the simultaneous allocator to work on traces of basic blocks.

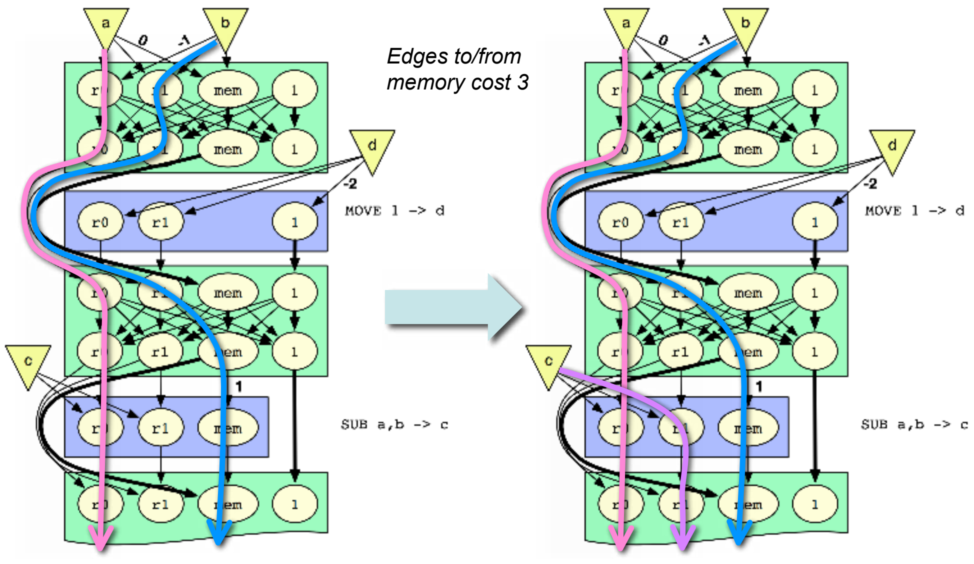


Figure 3.5: An example of a step in the iterative heuristic allocator. The variables a and b have already been allocated. The allocation for c is found through a shortest path computation. Note that the allocation of b avoided using the last available register in the SUB instruction group to avoid preventing the allocation of c.

Iterative Heuristic Allocator

The iterative heuristic allocator (Figure 3.5) allocates variables in some heuristically determined order. A variable is allocated by traversing the control flow graph in depth first order and computing the shortest path for the variable in the priced network of each block. Because the blocks are traversed in order, the split nodes at the exit of a processed block will fix the starting point for the shortest path in each successor block. Within each block we will always be able to find a feasible solution because the memory network is uncapacitated. We constrain our shortest-path algorithm to conservatively ignore paths that could potentially make the network infeasible for variables that still need to be allocated. For example, if an instruction requires a currently unallocated operand to be in a register and there is only one register left that is available for allocation, all other variables are required to be in memory at that point.

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 instructions, n, because each block is a topologically ordered directed acyclic graph. Therefore the worst case running time of the algorithm is $O(nv)$.

Simultaneous Heuristic Allocator

As an alternative to the iterative allocator, we describe a simultaneous allocator (Figure 3.6) which functions similarly to a second-chance binpacking allocator [102] but uses the global MCNF model to guide eviction decisions. The algorithm traverses the control flow graph in depth first order. For each block, it performs both a forwards and backwards shortest-path computation

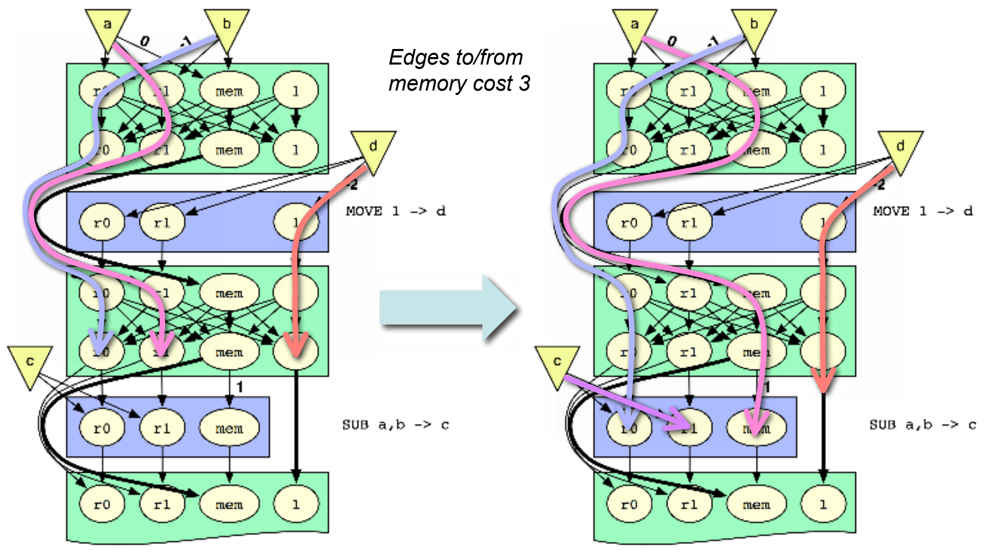


Figure 3.6: An example of a step in the simultaneous heuristic allocator. The variables a, b, and d have all been allocated up to the point before the `SUB` instruction. The definition of c requires that a variable be evicted from a register. Since it is cheapest to evict a, its allocation is changed to go through memory and c is allocated to `r1`.

for every variable. These paths take into account that the entry and exit allocations of a variable may have been fixed by an allocation of a previous block. Having performed this computation, the cost of the best allocation for a variable at a specific program point and allocation class in a block can be easily determined by simply summing the cost of the shortest paths to the corresponding node from the source and sink of the given variable.

After computing the shortest paths, the algorithm scans through the block, maintaining an allocation for every live variable. The allocations of live-in variables are fixed to their allocations at the exit of the already allocated predecessor blocks. At each level in the network, each variable's allocation is updated to follow the previously computed shortest path to the sink node of that variable (the common case is for a variable to remain in its current location). If two variables' allocations overlap, the conflict is resolved by evicting one of the variables to an alternative allocation.

When a variable is defined, the minimum cost allocation is computed using the shortest path information and a calculation of the cost of evicting any variable already allocated to a desired location. The cost of evicting a variable from its current location is computed by finding the shortest path in the network to a valid eviction edge (an edge from the previous allocation to a new allocation). In computing this shortest path we avoid already allocated nodes in the graph. That is, we do not recursively evict other variables in an attempt to improve the eviction cost. The shortest path is not necessarily a simple store immediately before the eviction location. For example, if the defining instruction of the variable being evicted supports a memory operand, it might be cheaper to define the variable into memory instead of defining it into a register and performing a more costly store later. When a variable is evicted to memory the cost of the corresponding anti-variable eviction is also computed and added to the total eviction cost. When

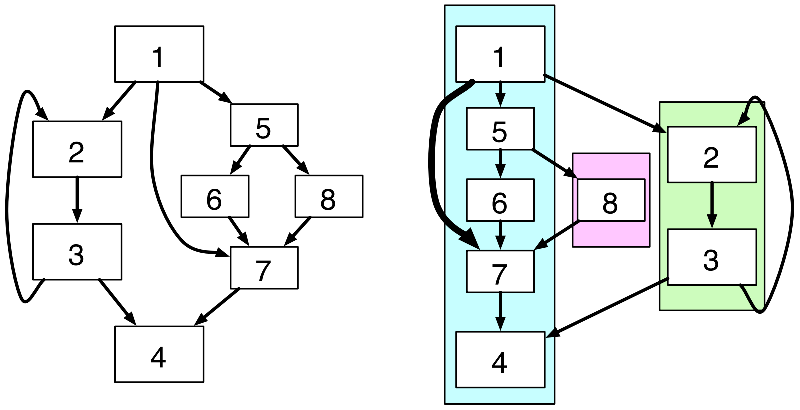


Figure 3.7: A example control flow graph (left) decomposed into traces (right). The bold edge is an example of control flow internal to a trace that complicates allocation.

choosing a variable to evict we break ties in the eviction cost by following the standard practice and choosing the variable whose next use is farthest away [11, 77].

Only intra-block evictions are allowed; the earliest a variable can be evicted is at the beginning of the current block. Because of this limitation, this allocator performs more poorly as the amount of control flow increases since poor early allocation decisions can not be undone later in the control flow graph.

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, although this results in an empirical performance improvement, the worst case asymptotic running time remains $O(nv)$.

Trace-Based Simultaneous Allocator

In an attempt to improve upon the simultaneous allocator we have developed a trace-based simultaneous allocator. Instead of processing each basic block independently, the trace-based allocator decomposes the control flow graph into linear traces of basic blocks, which may contain internal and external control flow, and allocates each trace similarly to how a single basic block is allocated by the simultaneous allocator. To construct our traces we simply find the longest possible traces using depth first search while ensuring that loop headers start a new trace (as in the example in Figure 3.7).

The presence of control flow within each trace creates some complications. When computing shortest paths care must be taken to take the correct edge spanning basic blocks within a trace (there may be holes in a trace where a variable is not live). 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 3.7 fixes the starting allocation of both blocks 5 and 7 and the exit allocation of block 6 in the same trace. Similarly, it may not be straightforward to evict a variable across block boundaries if doing so affects other blocks in the trace.

We consider two techniques for propagating boundary allocation decisions within a trace. The first, *easy-update*, does the minimal amount of recomputation necessary for correctness. Only blocks directly effected by the boundary allocation have their shortest path computations redone. For example, in Figure 3.7, after allocating block 1, only block 6 would have to be recomputed as its exit allocations have changed. Although these recomputations result in extra work compared to the original simultaneous allocator, they are necessary for the correct allocation of the trace. The second technique, *full-update*, recomputes the shortest paths for all unallocated blocks prior to and including the blocks effected by the boundary allocation. 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 3.7, if a variable were to spill to memory and then be loaded back into a register in block 5, it would likely be 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 6). With full-update the allocator would be aware of this cost since both blocks 5 and 6 would have been recomputed after the allocation of block 1.

3.2.3 Allocation Difficulties

There are several factors that prevent the allocation algorithms from finding the optimal solution given a priced network. Until the iterative subgradient method has fully converged, the prices in the network are only approximations. As a result, we may compute a shortest path for a variable that would not be a shortest path in a network with fully converged prices. The simultaneous allocators are less sensitive to this effect since they can undo bad allocation decisions. However, the values of the boundary prices are critical to the performance of the simultaneous allocators as allocation decisions get fixed at block or trace boundaries.

A potentially more significant impediment to finding an optimal solution is that the lower bound computed using Lagrangian relaxation converges to the value of the optimal solution of the global MCNF problem without integer constraints. If the difference between the value of the solution to the integer problem and the linear problem is nonzero, we will not be able to prove the optimality of a solution. Fortunately, it has been shown empirically that this difference is rarely nonzero [66].

Even given perfectly converged prices and an *a priori* knowledge that the integrality gap is not problematic, the allocation problem remains difficult. The allocators must choose among identically priced allocations, not all of which may be valid allocations in an optimal solution. Again, the simultaneous allocators are somewhat insulated from this difficulty since they can undo bad decisions within a block, but they still must rely upon the value of the boundary prices to avoid locally good, globally poor, allocation decisions.

The challenges faced by the allocators in converting a priced network into an optimal register allocation are not unexpected given the NP-completeness of the problem. However, as we shall see, as the iterative subgradient algorithm converges, the quality of solutions found by the allocation heuristics improve and the lower bound on the optimal solution value increases resulting in provably optimal or near-optimal solutions.

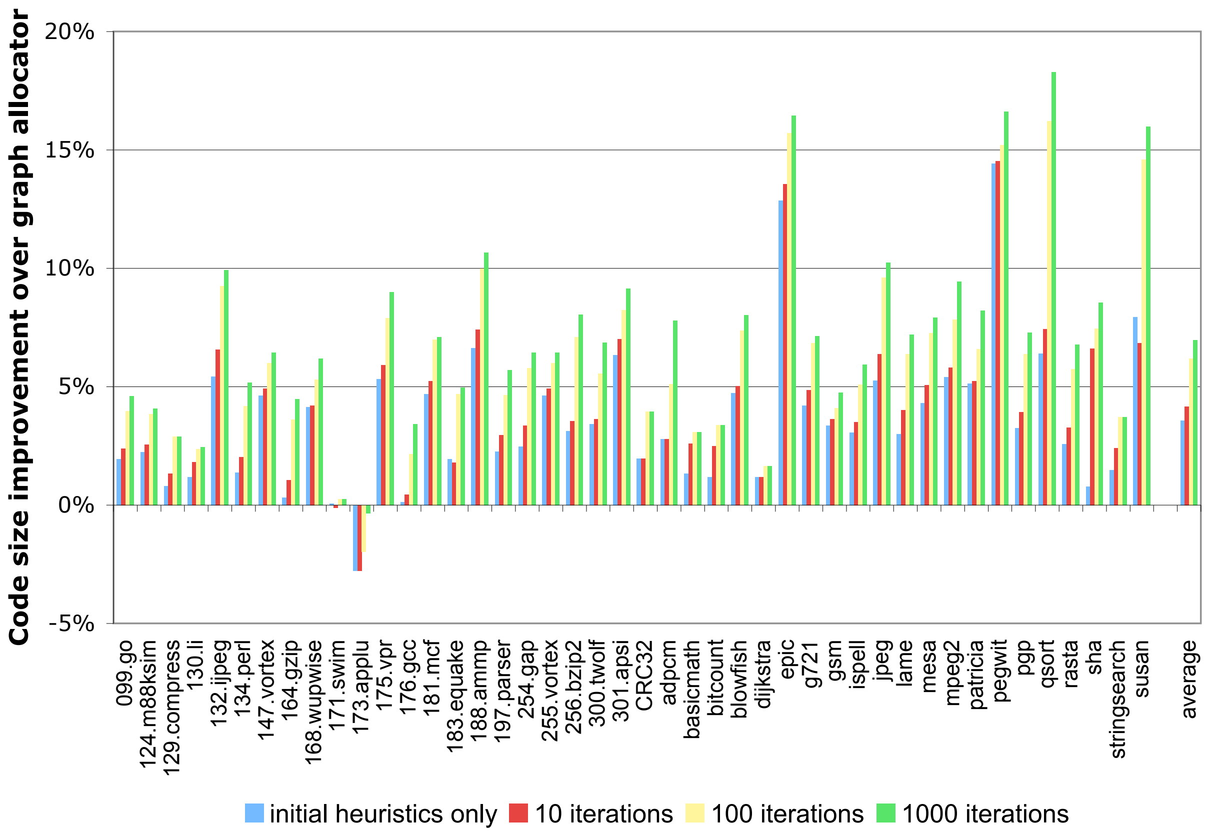


Figure 3.8: Code size improvement with our easy-update trace-based simultaneous allocator compared to a standard iterative graph coloring allocator. All benchmarks were compiled using the `-Os` optimization flag. Note the improvement over time with our allocator. The benchmark `qsort` had the largest improvement with a size improvement of 18.28% after 1000 iterations.

3.3 Results

3.3.1 Implementation

We have implemented our global MCNF allocation framework as a replacement for the register allocator in `gcc` 3.4.4 when targeting the Intel x86 architecture. Before allocation, we execute a preconditioning pass which aggressively coalesces moves and translates instructions that are not in an allocable form. For example, the compiler represents instructions as three operand instructions even though the architecture only supports two operand instructions. If all three operands are live out of the instruction, it is not possible to allocate these three variables to distinct registers and still generate an x86 two operand instruction. The preconditioning pass translates such instructions so that two of the three operands are the same variable.

We next build a global MCNF model for the procedure. In our model, crossbars are represented as zig-zags since `gcc` does not support the generation of the x86 swap instruction. We simplify the network by only permitting loads and stores of a variable to occur at block bound-

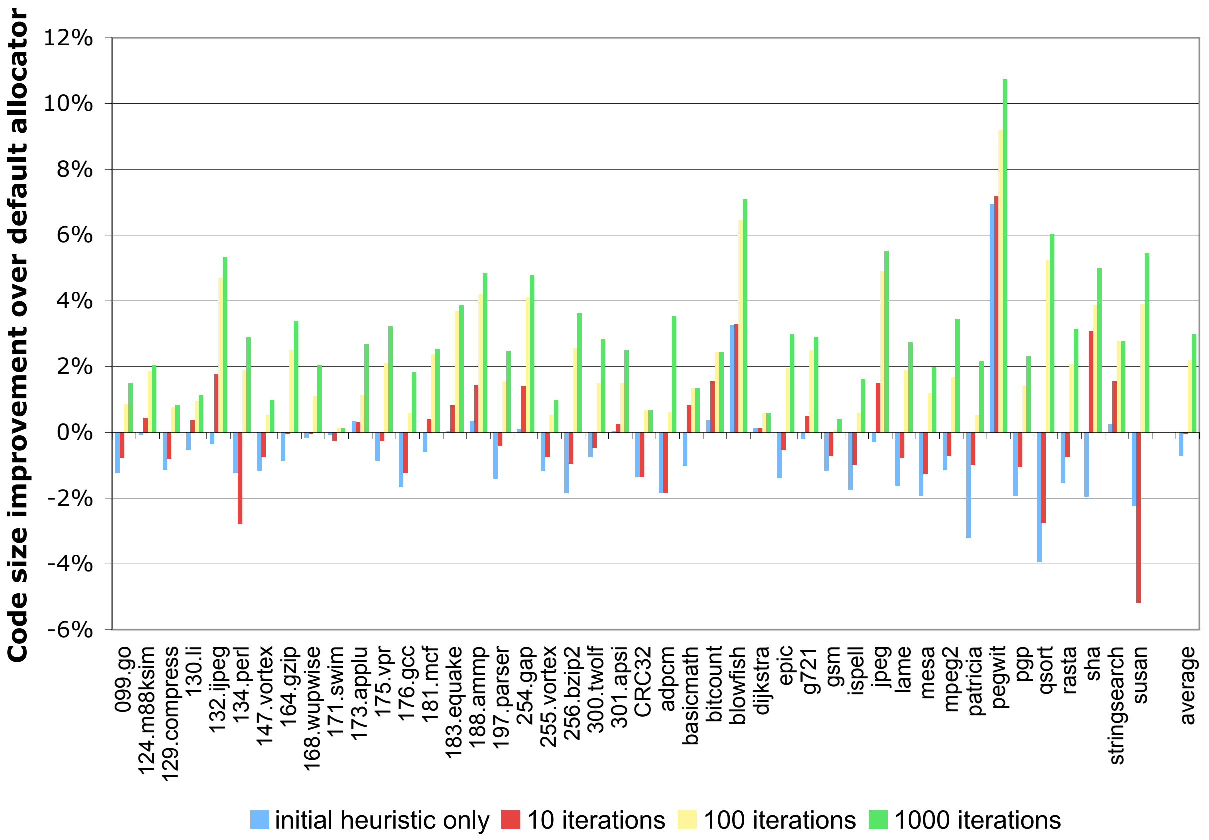


Figure 3.9: Code size improvement with our easy-update trace-based simultaneous allocator compared to the default allocator of `gcc`. All benchmarks were compiled using the `-Os` optimization flag. Our allocator outperforms the default allocator on all benchmarks after 100 iterations. The benchmark `pegwit` had the largest improvement with a size improvement of 10.78% after 1000 iterations.

aries and after a write to the variable (for a store) or before a read of the variable (for a load). This simplification does not change the value of the optimal solution.

We use code size as the cost metric in our model. This metric has the advantage that it can be perfectly evaluated at compile time and exactly represented by our model. We assume a uniform memory access cost model. Specifically, we assume that spilled variables will always fit in the 128 bytes below the current frame pointer unless this space is already fully reserved for stack allocated data (such as local arrays). As a result, for some large functions that spill more than 32 values the model is inaccurate. We only model constant rematerialization for integer constants. Although it is not required by the architecture, `gcc` requires 64-bit integer values to be allocated to consecutive registers. Since our model currently does not support such constraints, we ignore such values (resulting in all such variables being allocated to memory and fixed up by the reload pass).

We run both the iterative and trace-based simultaneous allocators on the initial unpriced network and then for each step of the iterative subgradient algorithm we apply only the trace-based simultaneous allocator to the priced network. In addition to being faster, the trace-based simulta-

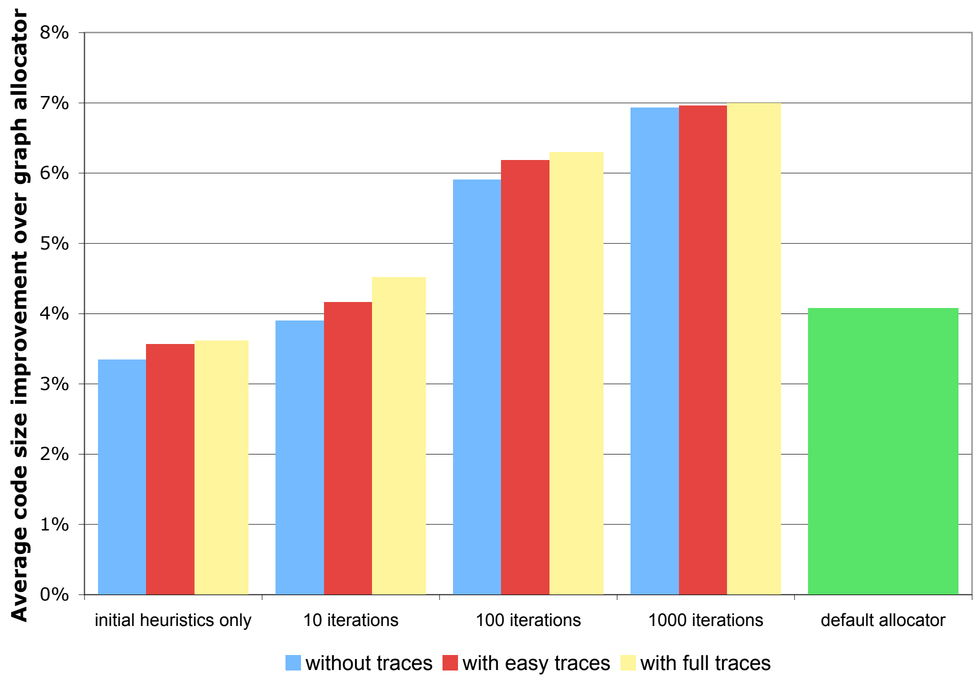


Figure 3.10: Average code size improvement over all the benchmarks relative to the graph allocator. The highly tuned default allocator outperforms both the graph allocator and our initial allocation, but, on average, when a trace-based allocator is used, our progressive allocator outperforms the default allocator after only 10 iterations.

neous allocator generally performs better than the iterative allocator once the Lagrangian prices start to converge. However, the iterative allocator does better on the unpriced graph because it allocates variables in order of decreasing priority. Unless otherwise specified, we use the easy-update technique for our trace-based allocator.

After running our solver, we insert the appropriate moves, stores, and loads and setup a mapping of variables to registers. The `gcc` reload pass is then run which applies the register map and modifies the instruction stream to contain only references to physical registers. This pass will also fix any illegal allocations that our allocator might make if the model of register preferences and usage constraints is not correct by generating additional fixup code (this is not common).

3.3.2 Code Quality

We evaluate our global MCNF model and progressive allocators on a large selection of benchmarks from the SPEC2000, SPEC95, MediaBench, and MiBench benchmark suits. Combined, these benchmarks contain more than 10,000 functions. We evaluate the quality of our solutions in terms of code size. Because our concern is with evaluating our model and our solver, all size results are taken immediately after the register allocation pass (including `gcc`'s reload pass) to avoid noise from downstream optimizations.

We compare our allocators to the standard iterative graph coloring allocator that can be enabled by passing `-fnew-ra` to `gcc`. This allocator implements standard graph coloring register allocation [23] with iterative coalescing [47] and interference region spilling [12]. The graph allocator generally does not perform as well as the highly-tuned default allocator. The default `gcc` register allocator divides the register allocation process into local and global passes. In the local pass, only variables that are used in a single basic block are allocated. After local allocation, the remaining variables are allocated using a single-pass graph coloring algorithm. Although the default allocator is algorithmically simple, it benefits from decades of development and improvement.

As shown in Figure 3.8 and Figure 3.10, our initial heuristic allocator, which runs both the iterative and trace-based simultaneous allocators and takes the best result, outperforms the graph allocator on all but one benchmark with an average improvement in code size of 3.57%. As expected, as more time is alloted for compilation, our easy-update trace-based simultaneous allocator does progressively better with average code size improvements of 4.17%, 6.18%, and 6.96% for ten, 100, and 1000 iterations respectively. As shown in Figure 3.9, our allocator does not initially do as well as the default allocator; at first we outperform the default allocator on only twelve benchmarks. However, we outperform or match even the default allocator on all 44 benchmark when we run our algorithm for 100 iterations. On average, we surpass the performance of the default allocator with only ten iterations. Figure 3.10 shows the advantage of using traces of blocks with our simultaneous allocator. Although the full-update trace-based allocator outperforms the easy-update allocator on a per iteration basis, as demonstrated in Figure 3.12, because of its worst-case quadratic complexity, it is not as beneficial on a per time unit basis.

3.3.3 Solver Performance

Progressiveness

The behavior of our progressive solver for a single function (`quicksort`) is shown in Figure 3.11. As expected, as the Lagrangian prices converge (resulting in a better lower bound), the quality of the allocation improves (the amount of register allocation induced overhead decreases). However, as the rate of convergence decreases, so does the progressive improvement of the best allocation.

Ideally, a progressive allocator would be competitive with both existing fast heuristic allocators and existing slow optimal allocators. In Figure 3.12 we compare several of our progressive allocators with `gcc`'s two allocators and an allocator that uses CPLEX version 10 [57] to solve our global MCNF problem. Our progressive allocators perform best in the absence of control flow; for the `squareEncrypt` function, which is a single basic block, our initial allocation is better than that of `gcc`'s heuristic allocators and we find a provably optimal allocation much more quickly than it takes CPLEX to solve the same problem. However, in control-flow intensive functions, such as `quicksort`, we aren't as competitive with `gcc`'s allocators initially, and, although we get close, we do not find an optimal solution.

Although our progressive allocator may not always be strictly better than heuristic allocators initially, or strictly better than optimal allocators eventually, our progressive allocator has the

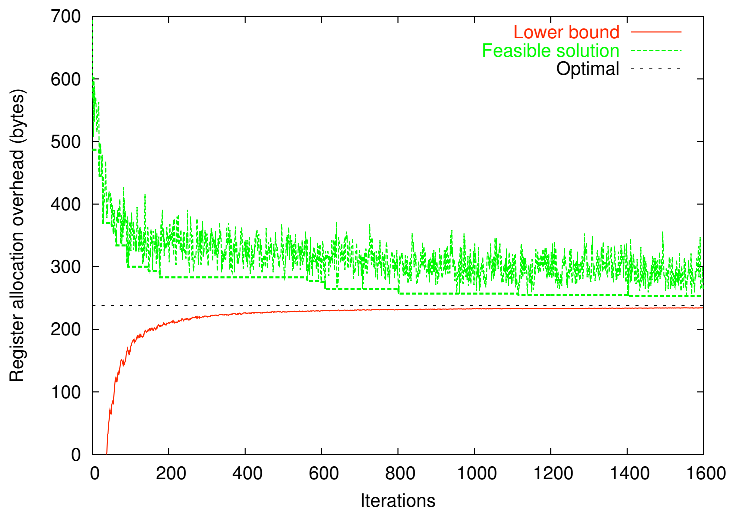
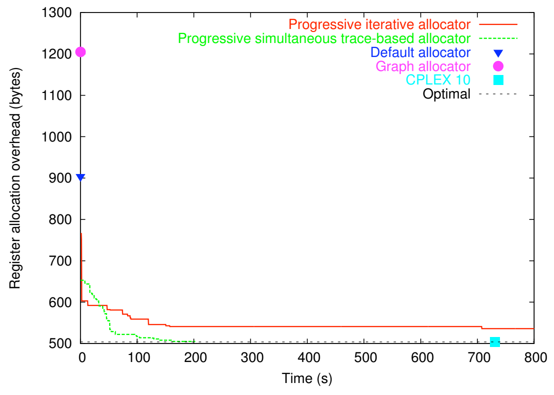


Figure 3.11: The behavior of our allocator on the quicksort function. Although the quality of the solution found by our allocator oscillates, as the lower bound computed using Lagrangian relaxation converges to the optimal value the value of the best solution progressively improves.



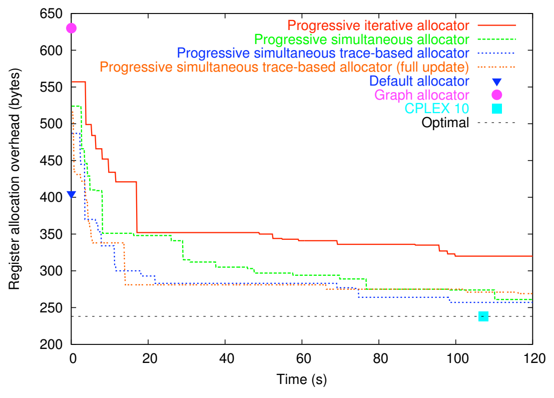


Figure 3.12: The behavior of our heuristic allocators as the Lagrangian prices converge executed on a 2.8Ghz Pentium 4 with 2GB of RAM. The `squareEncrypt` function from the pegwit benchmark consists of a single basic block and has 378 instructions, 150 variables, and an average register pressure of 4.99. The `quicksort` function is spread across 57 blocks, has 236 instructions, 58 variables, and an average register pressure of 3.14. Approximately a third of the final size of both functions is due to register allocation overhead. The iterative allocator performs better initially, but as the Lagrangian prices converge the simultaneous allocator performs better. In the case of the `squareEncrypt` function, which has no control flow, the simultaneous allocator find a better initial solution than both of `gcc`'s heuristic allocators and finds an optimal solution in about a quarter of the time it takes the CPLEX solver. The use of traces has no effect on the performance of the allocator in this case since there is only one basic block. None of our allocators succeeded in finding an optimal allocation for `quicksort` before CPLEX found the optimal solution at 107 seconds. The trace-based allocators clearly outperform both the block-based allocators, although as the prices converge the advantage of using traces decreases.

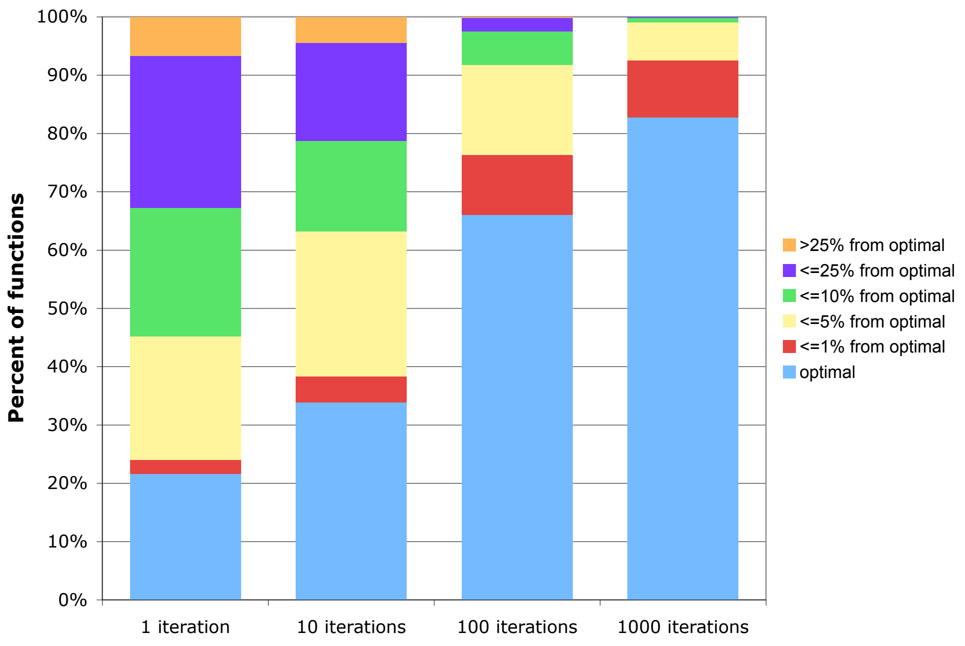


Figure 3.13: The proven optimality of solutions as the progressive allocator executes more iterations. The found solution is provably within 5% of optimal for 45.18%, 63.21%, 91.74%, and 99.06% of the functions after 1, 10, 100, and 1000 iterations respectively.

distinct advantage of successfully connecting these two extremes: decent solutions can be found quickly and optimality can be approached as more time is alloted for compilation.

Optimality

Ideally, a progressive solver is guaranteed to eventually find an optimal solution. Although our solver has no such guarantee, the Lagrangian relaxation technique lets us prove an upper bound on the optimality of the solution. As the iterative subgradient algorithm used to solve the Lagrangian relaxation converges, both a better lower bound on the optimal value of the problem is found and the quality of the solutions found by the Lagrangian-directed allocator improves. Consequently, as shown in Figure 3.13, as more iterations are executed, a larger percentage of compiled functions are proven optimal. After 1000 iterations, we have found a provably optimal register allocation for 82.74% of the functions and 99.06% of the functions have a solution that is provably within 5% of optimal.

Compile Time Performance

The worst case running time of $O(nv)$ of our heuristic solvers combined with the early developmental stage of our implementation leads us to expect that our allocator will not perform as well as existing allocators in terms of compilation time. Indeed, as shown in Figure 3.14, allocating with just one heuristic solver is almost ten times slower than the graph allocator, and a single iteration is clearly more expensive than an entire allocation in the graph allocator. These slowdowns

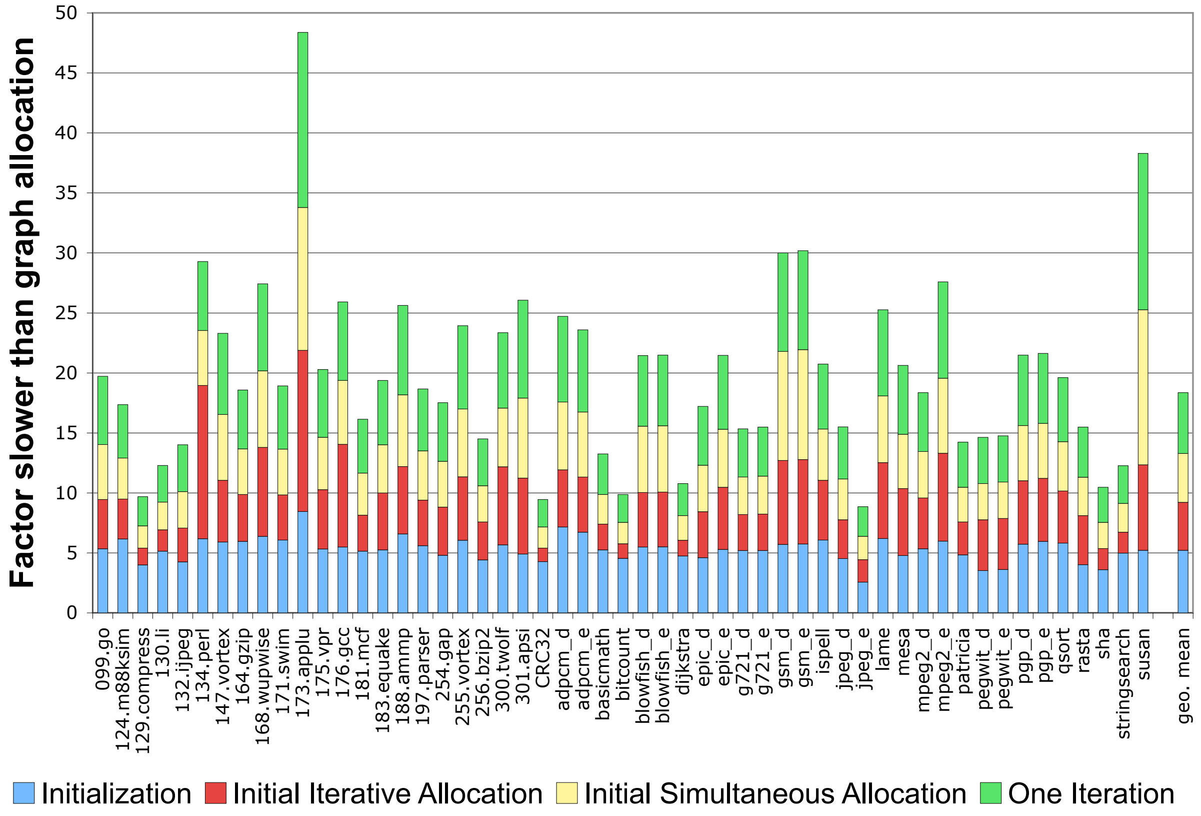


Figure 3.14: The slowdown of our register allocator relative to the graph allocator. On average, we take about 10 times longer than the graph allocator to find a solution.

are relative to the time spent by the graph allocator which accounts for between 10.5% and 46% of the total compile time (27.5% on average). The graph allocator is, on average, about four times slower than the default allocator. Although it is likely that these results will improve when we optimize our implementation, because our allocators solve a fundamentally harder problem than existing fast heuristics, it is unlikely that it is possible to be faster than existing allocators.

Chapter 4

Proposed Work

4.1 Model Improvements

Although the current global MCNF model successfully captures the pertinent features of register allocation, there remains room for improvement. The current model is not always 100% accurate and is needlessly expressive when modeling uninteresting program regions and representing uniformly accessed registers.

The existing model accurately models the costs of register allocation most of the time, as shown in Figure 4.1. We measure the accuracy of the model by comparing the predicted size of a function after ten iterations of our progressive algorithm to the actual size of the function immediately after register allocation. Approximately half of the compiled functions have their size exactly predicted and more than 70% of the functions have their size predicted to within 2% of the actual size.

I propose to improve the accuracy of the model. The biggest cause of under-prediction is the uniform memory cost model. Most of the severely under-predicted functions spill more variables than fit in the first 128 bytes of the frame resulting in incorrectly predicted costs in the model for memory operations. It is possible to exactly represent such behavior by using two memory allocation classes, one of which is capacitated, although it is not clear that the improved model accuracy is worth the increased model complexity. The biggest cause of the most severe over-predictions is `gcc`'s instruction-sizing function inaccurately reporting the size of certain floating point instructions prior to register allocation. This should be easily fixed by modifying the pre-register allocation code size analysis.

The current global MCNF model is uniformly expressive. I propose to modify the current model to be adaptively expressive; at each program point the model will be only as expressive as it needs to be. Reducing the size of the model in this way will result in better solver performance and memory efficiency. The most obvious cases where simplification is possible is in a local region of no register pressure as in Figure 4.2. In this case, it may be possible to summarize the entire region as a single meta-instruction group. This is straightforward to do in any region where the allocation on entry can be proven to be identical to the allocation on exit and where the constraints of the region can be represented by the normal instruction constraints. It may also be worthwhile to summarize more complex regions using more elaborate summary representations.

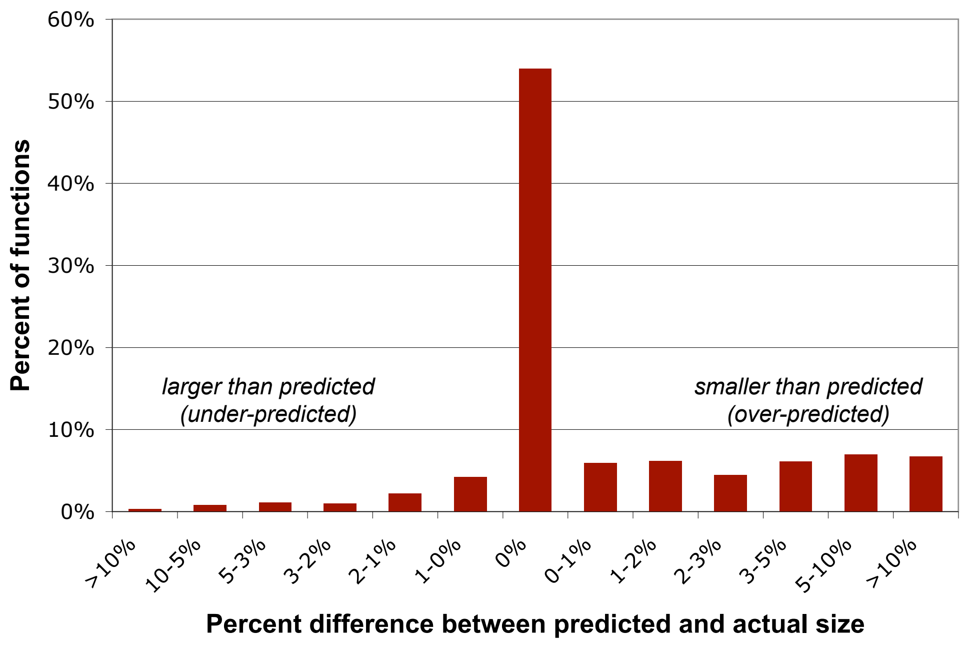


Figure 4.1: An evaluation of the accuracy of the global MCNF model that compares the predicted size after register allocation to the actual size.

I propose to implement a separate model simplification pass that runs after the creation of the full model. Although this approach is not as efficient as modifying the model creation routines to directly build a simplified model, it will make it easier to investigate different simplification techniques. In particular, it will be easier to compare safe simplification techniques, which reduce the expressiveness of the model only if doing so is guaranteed not to change the optimal solution, with aggressive simplification techniques, which may sacrifice some optimality for reductions in model complexity.

Further model simplification may be possible when targeting architectures with more uniform register files. The existing model implementation represents each register as its own allocation class. While this is useful and necessary for the x86 architecture, for RISC architectures such as the PowerPC, and even CISC architectures such as 68k/Coldfire (which has 8 uniformly accessed data registers and 8 uniformly accessed address registers), assigning each register its own allocation class increases the size of the model (especially in the crossbar groups where the number of edges is quadratic with the number of allocation classes). Unfortunately, this level of expressiveness is not completely unnecessary when performing global register allocation. As illustrated by Figure 4.3, in the presence of control flow the register assignment problem cannot be decoupled from the rest of register allocation.

I propose to investigate model simplification in the presence of uniformly accessed register sets. Such simplification will likely be necessary in order to efficiently support RISC-like architectures. I will investigate the need for such simplification in the presence of more general model simplifications as well as consider the impact of a less detailed model that does not directly solve the register assignment problem.

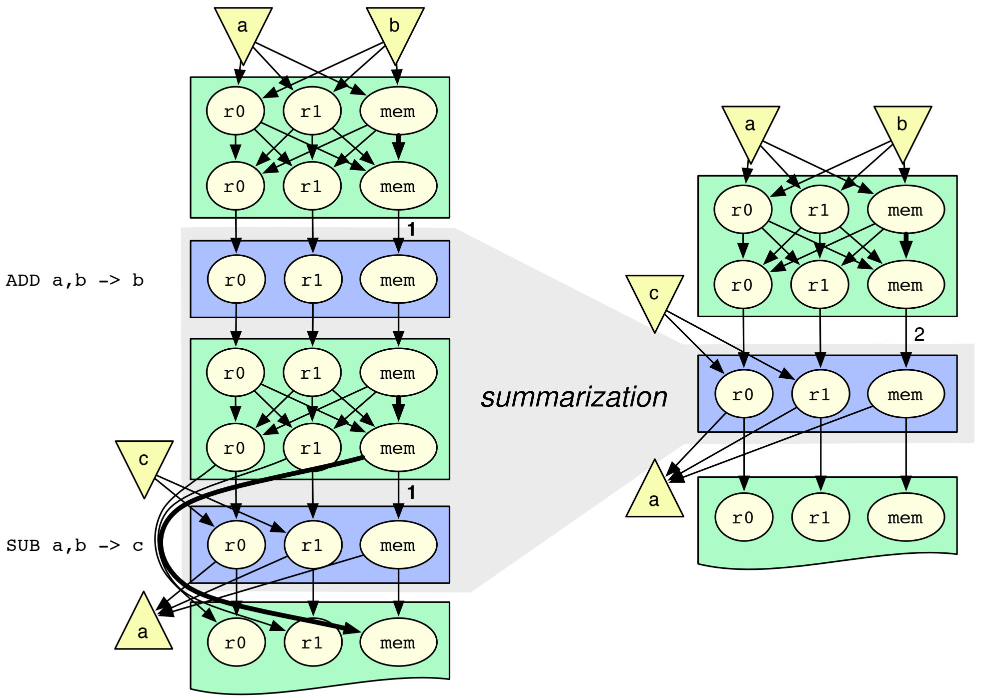


Figure 4.2: An example of model simplification. The `ADD` and `SUB` instructions and their intervening crossbar can be summarized as a single meta-instruction group without changing the value of the solution.

The purpose of the thesis is to investigate, develop, and evaluate new algorithms and approaches for performing register allocation and instruction selection. Code size is used as a metric of code quality since it can be precisely evaluated at compile time and, as a result, allows for noise-free evaluation of the register allocator and instruction selector. The techniques in this thesis should be equally valid for any code quality metric that can be evaluated at compile time. However, formulating other precise and accurate statically evaluatable code quality metrics, such as for speed and energy usage, is considered beyond the scope of this thesis. As a result, I propose to investigate only a straightforward speed metric where the cost of each operation is simply its predicted cycle time multiplied by its predicted execution frequency.

4.2 Solver Improvements

An ideal progressive register allocator would be able to quickly find an allocation that is as good as or better than allocations found by existing heuristic allocators and then progressively improve upon this solution until an optimal solution is found, ideally in less time than a standard optimization package. Although in terms of local register allocation the current allocator appears to meet or exceed this standard, it is unclear how closely this standard can be approached with global register allocation.

I propose to further investigate improvements to the current solver that either improve the initial solution or improve the quality of the solution as the prices converge. I do not propose to add a guarantee that the solver will eventually find the optimal solution since, unless P = NP,

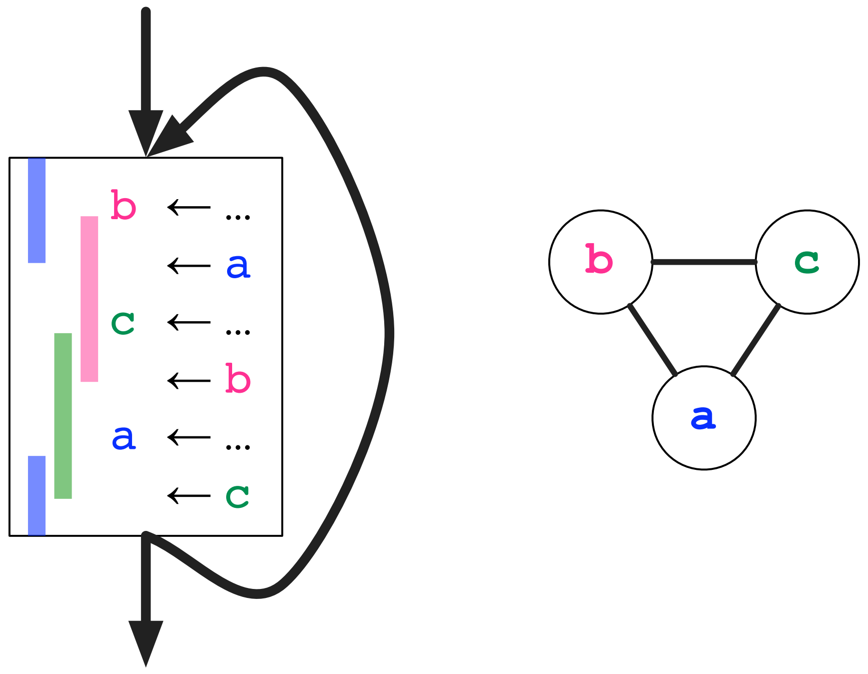


Figure 4.3: An example where register assignment cannot be decoupled from the rest of register allocation. In this example, if there are two uniformly accessed registers `r0` and `r1` making up some register class R, then while it is possible to allocate each variable to R such that at no point more than two variables are allocated to R (at most two live ranges overlap at any point), it is not possible to assign `r0` and `r1` to the three variables (the interference graph is a triangle).

such a guarantee would require the implementation of a branch and bound search which I view as an uninteresting extension of the work.

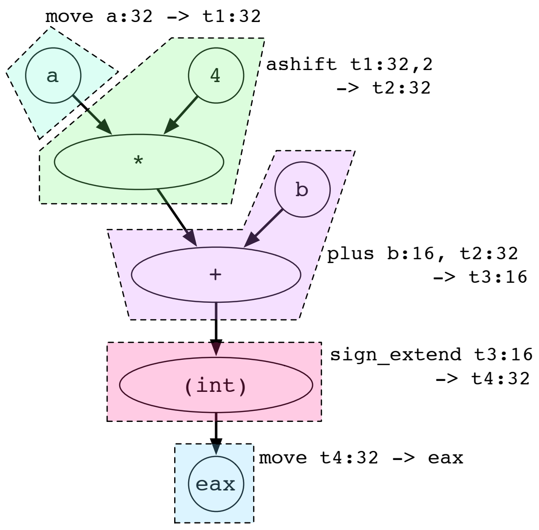
Although I do not intend to provide an optimality guarantee, it is likely that some theoretical bound on solution quality can be obtained. The current best approximation algorithm for local register allocation on a RISC-like machine is a 2x approximation [38]. I propose to investigate the approximation properties of my solution technique. At a minimum, I expect to be able to duplicate the existing 2x result within the context of the MCNF model of the problem.

4.3 Integrating Register Allocation and Instruction Selection

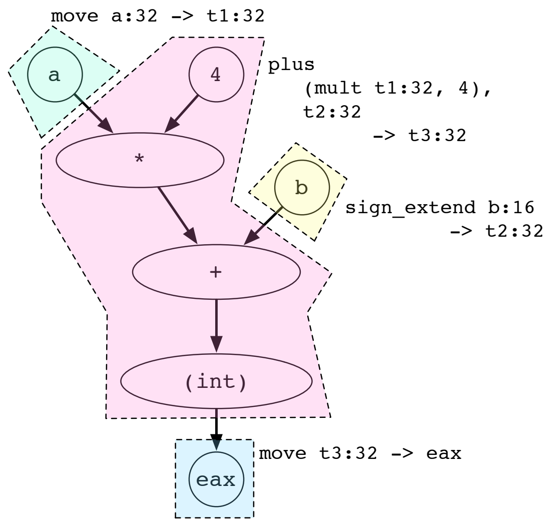
Instruction selection is an essential part of the compiler backend. Although optimal algorithms exist for tiling expression trees and heuristic algorithms exist for tiling expression DAGs, these algorithms rely on the accuracy of the assessments of the costs of each instruction pattern tile. These tile costs are inherently inaccurate since spills, register preferences, and move coalescing may change the instructions corresponding to a tile. For example, the shorter instruction sequence of Figure 4.4(e) may actually be worse than that of Figure 4.4(d) if the extra register pressure it introduces results in spill code. A principled backend optimizer requires a greater

```
int foo(int a, short b) { return a*4+b; }
```

(a)



(b)


move a:32 -> t1:32
plus (mult t1:32, 4), t2:32 -> t3:32
a
4
*
b
sign_extend b:16 -> t2:32
+
(int)
eax
move t3:32 -> eax

(c)

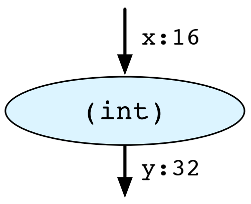
```
size
 4   movl  4(%esp), %eax
 3   sall  $2, %eax
 4   addl  8(%esp), %eax
 1   cwtl
 1   ret
```

(d)

```
size
 5   movswl  8(%esp),%edx
 4   movl    4(%esp), %eax
 3   leal    (%edx,%eax,4), %eax
 1   ret
```

(e)

Figure 4.4: Two possible instruction tilings for the code snippet shown in (a). Under ideal conditions (no register pressure), both tilings produce code of the same size, but (b) results in a longer instruction sequence, (d). Alternatively, (c) results in code, (e), which requires an additional register. In the presence of register pressure, it is not clear which tiling is better. The tiling (c) requires two registers, but (b) results in larger code if the `eax` register is not used preventing the generation of the efficient `cwtl` instruction.



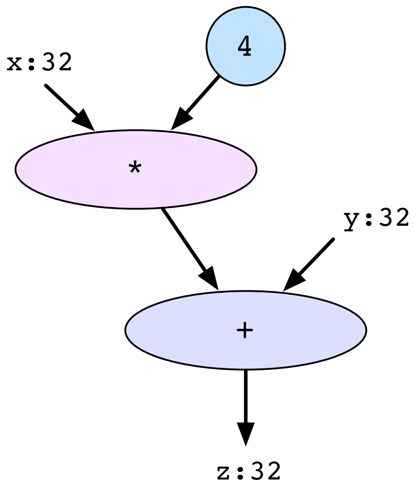
(a)

`sign_extend` $x \rightarrow y$

x\y	eax	edx	⋯	mem
eax	cwtl(1)	movsx(4)		
edx	movsx(4)	movsx(4)		
⋮				
mem	movsx(5)	movsx(5)		

(b)

Figure 4.5: A register allocation aware tile for sign extension. The tile (b), matches the IR operation (a). The cost of the tile (here determined by code size) depends on the eventual allocation of the input (x) and output (y) of the operation. In some cases multiple instructions might be necessary. For example, if both x and y are in memory, a store instruction has to be generated. However, the register allocator is assumed capable of generating this store, and so the tile need not represent this case.



(a)

`plus (mult` x`,4),` $y \rightarrow z$
z :eax

x\y	eax	edx	⋯	mem
eax		leal(3)		sall;addl(6)
edx	leal(3)			
⋮				
mem				

`plus (mult` x`,4),` $y \rightarrow z$
z :edx

x\y	eax	edx	⋯	mem
eax		leal(3)		
edx	leal(3)			sall;addl(6)
⋮				
mem				

⋮

(b)

Figure 4.6: A register allocation aware tile (a) for a more complicated expression tree (b). In this case it is impossible for x and y to be allocated to the same register since their live ranges overlap, but if y is in memory and x and z are allocated to the same register then y can be directly accessed with the `addl` instruction. Although this code sequence is no smaller than loading y into a register and using the `leal` instruction, it does require one less register. In RA^{2}ISE, the register allocator makes the final decision as to which sequence to generate.

degree of integration between the instruction selection phase and register allocation. Towards that end I propose *Register Allocation Aware Instruction SElection (RA^{2}ISE).*

In RA^{2}ISE, instruction selection is not finalized until the register allocation phase. Instead of tiling the expression trees with fixed cost (inaccurate) tiles, instruction selection uses variable-cost *register allocation aware tiles* (RAATs) whose final instruction sequence and cost depends upon the allocation of the tile's inputs and outputs. For example, the sign extension RAAT in Figure 4.5 explicitly encodes the benefit of both operands being in `eax` (a smaller instruction can be used) and the cost of the input operand being in memory (an additional byte is necessary to store the stack offset). Larger RAATs that potentially resolve to multiple instructions can also be used, as shown in Figure 4.6. Instruction selection is partially performed using RAATs. The information in these tiles is then incorporated into the expressive model used by the register allocator. The allocation found by the register allocator is then used to finalizes the instruction sequence.

In order to support the RA^{2}ISE framework, the current global MCNF model of register allocation needs to be extended to support inter-variable register usage preferences and constraints so that the register allocation aware instruction tiles can be exactly expressed in the model. Currently, the model can only exactly represent tiles if the cost of an allocation decision is determined solely by an individual variable's allocation. For example, in the tile 4.5(b) the cost of allocating x to memory can be represented but the cost of allocating x to `eax` can not be modeled exactly since this cost is determined by the allocation of y.

I propose to increase the expressiveness of the current global MCNF model to exactly express the costs and constraints represented by RAATs by adding side-constraints, which constrain the flows of variables with respect to each other. After adding support for side-constraints to the model, I will determine the necessary extensions to the existing solution algorithms to solve a model with side-constraints and evaluate their effectiveness. I will first use these side-constraints to exactly represent `gcc`'s existing instruction tiles and then implement a more general framework for specifying and generating RAATs.

I propose to investigate various algorithms for generating a tiling of RAATs. I will explore simple extensions of existing tiling algorithms that simply assign a fixed cost to each RAAT. For example, a RAAT might optimistically be assigned its minimum possible cost, its median cost, or a cost that is a heuristic function of the register pressure at that point. In addition, I will consider feedback directed algorithms where instruction selection and register allocation are performed multiple times, each time providing feedback for the next iteration.

4.3.1 Preliminary Results

In order to demonstrate the benefits of adding side constraints to our model we consider one straightforward application of side constraints: move coalescing. The goal of the NP-hard [18] move coalescing problem is to remove as many move instructions as possible by allocating the source and destination of the move instruction to the same register. In our current allocator we aggressively and greedily coalesce as many moves as possible prior to allocation. Our allocator is then capable of reinserting moves into the instruction stream if doing so aids allocation. However, as shown in Figure 4.7, it is not always possible to coalesce all available moves and the moves which our aggressive coalescing algorithm chooses may not be optimal.

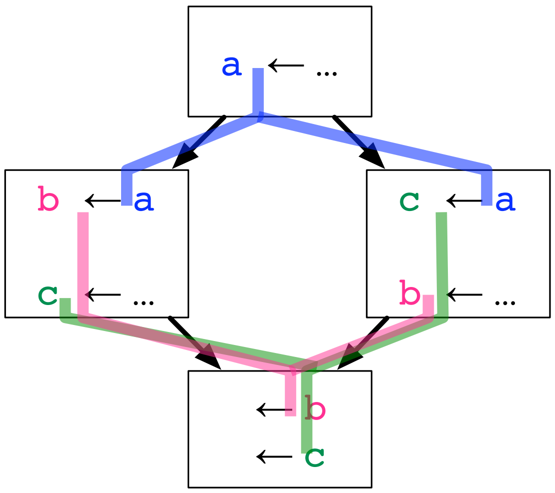


Figure 4.7: An example of a coalescing decision. Both the move from `a` to `b` and the move from `a` to `c` are candidates for coalescing, but since `b` and `c` conflict only one of the moves can be coalesced. Aggressive coalescing arbitrarily picks one of the moves, but the best move to coalesce depends on the register pressure properties within the various blocks.

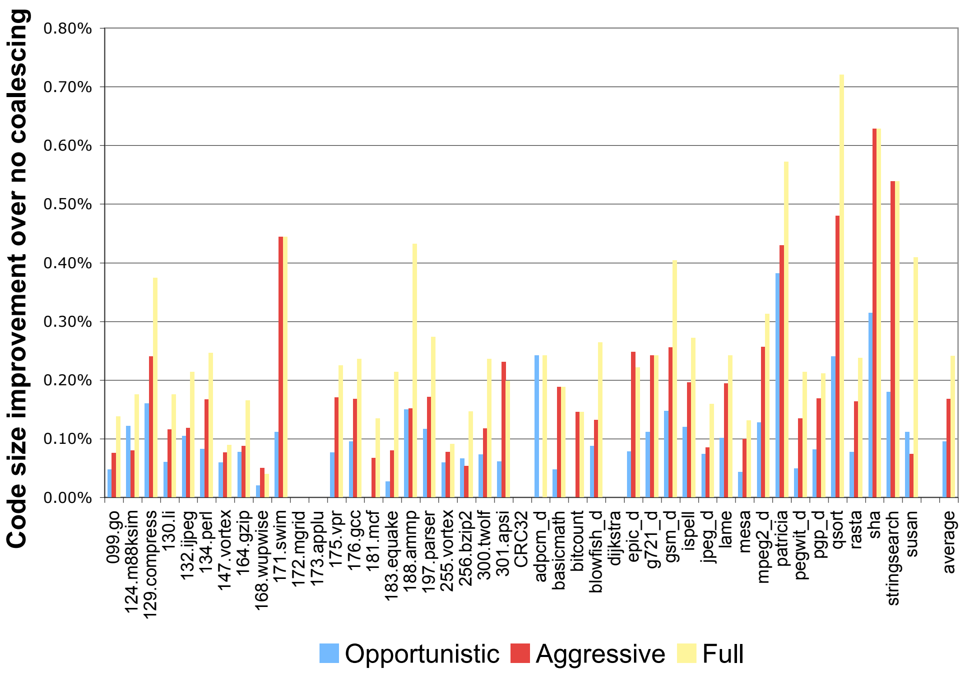


Figure 4.8: Size improvement using different coalescing methods relative to no coalescing. Results are computed using the CPLEX optimizer; results for functions where no solution could be found within 10 minutes are omitted.

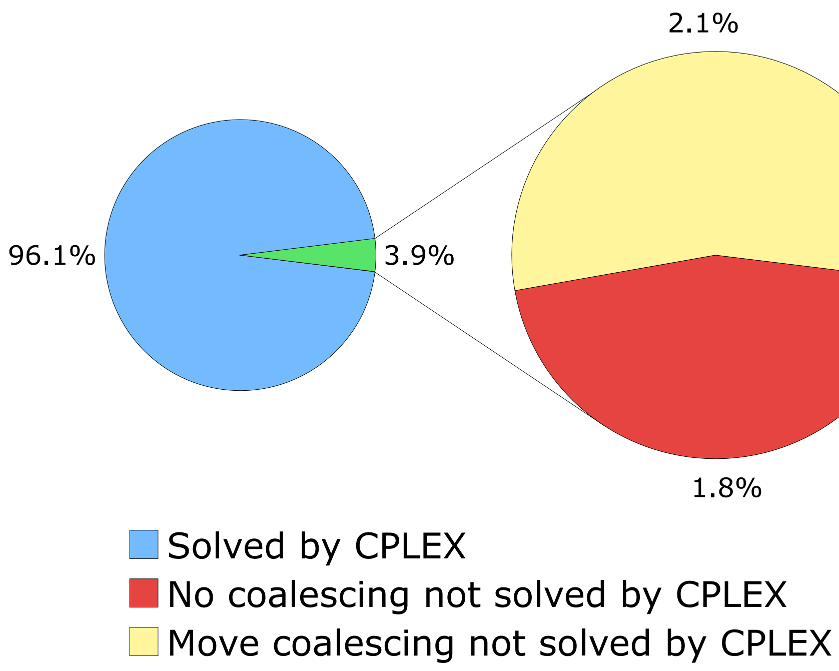


Figure 4.9: Percent of functions for which CPLEX could find an optimal solution within a 10 minute time limit. In some cases CPLEX found a solution for the no coalescing problem within the time limit but not for the full coalescing problem.

The move coalescing component of register allocation is easily added to our existing model through the use of side constraints. The allocation of the move source is simply required to be equal to the allocation of the move destination. Put another way, the RAAT for the move instruction would have a cost table with a diagonal of zeroes.

We have implemented move coalescing side constraints within the integer linear programming representation of our model. We evaluate four methods of coalescing:

No Coalescing Absolutely no coalescing is performed. Move instructions where both operands have the same allocation remain in the instruction stream.

Opportunistic Coalescing Coalescing is not performed until after the register allocation problem has been solved. The register allocation problem solved does not have a move coalescing component. Move instructions where both operands have the same allocation are removed from the instruction stream.

Aggressive Coalescing As many move instructions as possible are coalesced prior to solving the register allocation problem, which works on this modified instruction stream. The variables being coalesced become a single variable. This is the method currently used by our allocator.

Full Coalescing The move coalescing component of register allocation is incorporated into our model using side constraints. The current implementation has the side effect of forcing both variables of the move to be in a register. This is because we assume every variable has a unique location on the stack and memory to memory moves are not allowed by the x86 architecture. In contrast, the aggressive coalescing technique would allocate both variables to the same stack location allowing for the coalescing of a move when both operands are in memory. In this uncommon case it is possible for the aggressive coalescing technique to outperform our full coalescing technique.

We find the cost of the optimal register allocation using all four of these coalescing techniques by solving each problem using version 10 of the ILOG CPLEX optimizer [57]. The size improvement of each method relative to a baseline of no coalescing is shown in Figure 4.8. Relative to no coalescing, opportunistic coalescing, aggressive coalescing, and full coalescing improve code size on average by 0.10%, 0.17%, and 0.25% respectively. The addition of side constraints made some problems harder to solve. As shown in Figure 4.9, CPLEX could find an optimal solution to the problem without side-constraints but not with the side-constraints for 2.1% of the functions compiled. In some cases, such as the `quicksort` function, the addition of side-constraints resulted a two orders of magnitude slowdown in CPLEX solution time.

4.4 Evaluation

In order to determine the benefit of using an expressive model and progressive solver it is necessary to perform a comprehensive evaluation. I propose to evaluate the allocator for both code size and speed on a wide variety of benchmarks. In order to evaluate the value of progressive optimization in normal use, I will modify the compiler driver to accept a time limit for optimization (as opposed to the current iteration count limit). In addition, in order to demonstrate the generality and applicability of the allocator, I propose to evaluate it on multiple architectures: x86, which is a highly irregular architecture with only eight registers; 68k/ColdFire which has 16 registers divided equally between differently accessed address and data registers; and PowerPC which has 32 uniform registers.

Chapter 5

Contributions and Timeline

5.1 Expected Contributions

The expected contributions of the final thesis are:

- Register Allocation Aware Instruction Selection (RA^{2}ISE) which consists of
  - register allocation aware tiles (RAATs) which explicitly encode the effect of register allocation on an instruction sequence,
  - algorithms for performing instruction selection using RAATs,
  - an expressive model for global register allocation that operates on RAATs and explicitly represents important components of register allocation such as spill code insertion, register preferences, copy insertion, and constant rematerialization,
  - a progressive solver for this model that quickly finds solutions comparable to existing approaches and approaches optimality as more time is allowed for compilation,
  - algorithms for encorporating feedback from register allocation into instruction selection to more fully exploit the expressiveness of RAATs.
- A comprehensive evaluation of RA^{2}ISE
  - implemented in a production quality compiler (gcc),
  - targeting different architectures (x86, 68k/ColdFire, PowerPC), and
  - compiling for both the code size and code performance optimization metrics.

Overall, the thesis aims to improve the state of the art in backend compiler optimization by creating a new, principled, optimization framework that replaces the existing *ad hoc* heuristic approaches with expressive and explicit models coupled with progressive solution techniques.

5.2 Timeline

Fall 2006	• add simple speed metric option to model • begin model simplification work • improve model accuracy and solver performance
Winter 2006	• finish model simplification work • add side-constraints to model • implement existing `gcc` tiles as RAATs • improve model accuracy and solver performance
Spring 2007	• finish implementation of side-constraints and `gcc` RAATs • begin work on RA^{2}ISE infrastructure • create `gcc`-independent set of RAATs for x86 • improve model accuracy and solver performance
Summer 2007	• finish work on RA^{2}ISE • investigate and develop tiling algorithms • improve model accuracy and solver performance
Fall 2007	• add 68k/ColdFire and PowerPC targets • investigate uniform register set simplifications • improve model accuracy and solver performance
Winter 2007	• begin writing thesis • work on improving compile time performance
Spring 2008	• finish writing thesis

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. 1.2.2, 2.4

[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. 2.4

[3] Alfred Aho and Jeffrey Ullman. Optimization of stright line programs. *SIAM Journal on Computing*, 1(1):1–19, 1972. 1.2.2

[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. 2.4

[5] 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

[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. 3.1.1, 3.2.1, 3.2.1

[7] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-806-7. 1.1

[8] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1997. ISBN 0-521-58654-2. 2, 2.4

[9] Guido Araujo and Sharad Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 36–41, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-771-5. 2.4

[10] Barrie M. Baker and Janice Sheasby. Accelerating the convergence of subgradient optimisation. *European Journal of Operational Research*, 117(1):136–144, August 1999. 3.2.1

[11] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 3.2.2

[12] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 287–295, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-907-6. 2.1.2, 3.3.2

[13] 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 1989 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989. ISBN 0-89791-306-X. 2.1.2

[14] M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. i: Interval graphs. *Discrete Math.*, 100 (1-3):267–279, 1992. ISSN 0012-365X. 1.2.1

[15] 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. 1.2.1, 2.3

[16] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993. URL `citeseer.nj.nec.com/bodlaender93tourist.html`. 1.2.1

[17] 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. 2.4

[18] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. Technical Report RR2006-15, LIP, ENS-Lyon, France, April 2006. 1.2.1, 4.3.1

[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. 1.2.1

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

[21] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, 1992. ISSN 1057-4514. 2.1.3

[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. 2.1.2

[23] 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. 2.1.1, 2.1.2, 2.1.3, 3.3.2

[24] 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. 1.2.1

[25] John Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, 1976. ISSN 0004-5411. 1.2.2

[26] 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. 2.2

[27] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst.*, 2(2):173–190, 1980. ISSN 0164-0925. 2.4

[28] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982. ISBN 0-89791-074-5. 2.1.1, 2.1.2

[29] 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. 1.2.1

[30] 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. 2.1.1

[31] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 222–232, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. 2.1.1

[32] 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. 2.2

[33] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag. ISBN 3-540-64304-4. 2.1.2

[34] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004. 2, 2.1.1, 2.4

[35] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 111–116, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. 2.4

[36] 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. 2.4

[37] 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. 2.4

[38] 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. 1.2.1, 2.2, 4.2

[39] 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. 2.4

[40] 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. 2.4

[41] 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. 2.4

[42] 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. 2.4

[43] *GNU Compiler Collection (GCC) Internals*. Free Software Foundation, Boston, MA, 2006. URL `http://gcc.gnu.org/onlinedocs/gccint/`. 2.2

[44] 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. 2.3

[45] 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`. 2.3

[46] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002. 1.1

[47] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996. ISSN 0164-0925. 2.1.2, 3.3.2

[48] 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. 2.3

[49] 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. 2.4

[50] 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. 2.3

[51] 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. 1.2.1

[52] Sebastian Hack. Interference graphs of programs in ssa-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005. 1.2.1

[53] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. 1.1

[54] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 510–515, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-964-5. 2.4

[55] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003. 2.3

[56] 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. 2.2, 2.3

[57] ILOG. ILOG CPLEX. `http://www.ilog.com/products/cplex`. 3.3.3, 4.3.1

[58] Sven-Olof Nyström Johan Runeson. Retargetable graph-coloring register allocation for irregular architectures. *Lecture Notes in Computer Science*, 2826:240–254, October 2003. 2.1.3

[59] 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. 2.1.2

[60] 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. 1.2.1

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

[62] Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for clustered vliw architectures. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 102–111, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-527-0. 2.4

[63] Robert R. Kessler. Peep: an architectural description driven peephole optimizer. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 106–110, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. 2.4

[64] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, pages 269–280, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. 3.2

[65] 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`. 2.1.3, 2.2

[66] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 204–215, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. 3.2.3

[67] 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`. 2.3

[68] 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. 2.3

[69] 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. 2.1.3

[70] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-647-1. 1.1

[71] 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. 2.1.2

[72] 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. 3.2.1

[73] 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. 2.4

[74] 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. 2.4

[75] 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. 2.4

[76] 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. 2.4

[77] 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. 2.2, 2.3, 3.2.2

[78] 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. 2.1.1

[79] 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. 2.2

[80] 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. 3.2.1

[81] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965. ISSN 0001-0782. 2.4

[82] 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. 2.3

[83] C. Robert Morgan. *Building an Optimizing Compiler.* Butterworth, 1998. ISBN 1-55558179-X. 2, 2.2

[84] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4. 2, 2.4

[85] 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. 2.3

[86] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. *Trans. on Embedded Computing Sys.*, 3(1):163–181, 2004. ISSN 1539-9087. 2.3

[87] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-based global live-range splitting. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Pro-*

gramming language design and implementation, pages 216–227, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. 2.1.2

[88] Andy Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In *HPCN Europe*, pages 987–989, 1998. 1.1

[89] 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. 2.2

[90] 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. 1.2.1, 2.3

[91] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004. ISSN 0164-0925. 2.1.2

[92] 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. 1.2.1

[93] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. ISSN 0164-0925. 2.2

[94] Todd A. Proebsting. Burs automata generation. *ACM Trans. Program. Lang. Syst.*, 17(3):461–486, 1995. ISSN 0164-0925. 2.4

[95] 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. 2.2

[96] Todd A. Proebsting and Charles N. Fischer. Demand-driven register allocation. *ACM Trans. Program. Lang. Syst.*, 18(6):683–710, 1996. ISSN 0164-0925. 2.2

[97] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. Register-sensitive selection, duplication, and sequencing of instructions. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-410-X. 2.4

[98] 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. 2.3

[99] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970. ISSN 0004-5411. 2.4

[100] 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. 2.1.3

[101] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998. ISSN 0890-5401. 1.2.1, 2.3

[102] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 142–151, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-987-4. 2.2, 3.2.2

[103] Spyridon Triantafyllis, Manish Vachharajani, and David I. August. Compiler optimization-space exploration. *The Journal of Instruction-Level Parallelism*, 7:1–25, January 2005. URL `http://www.jilp.org/vol7`. 1.1

[104] 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. 2.1.1

[105] B. Wess. Automatic instruction code generation based on trellis diagrams. In *1992 IEEE International Symposium on Circuits and Systems*, volume 2, pages 10–13, May 1992. 2.4

[106] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-047-7. 2.2

[107] Thomas Zeitlhofer and Bernhard Wess. Optimum register assignment for heterogeneous register-set architectures. In *ISCAS '03. Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 3, pages III–252–III–244, May 2003. 1.2.1