RICE UNIVERSITY

Register Allocation via Graph Coloring

by

Preston Briggs

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Keith D. Cooper, Associate Professor, Chair
Computer Science

Ken Kennedy, Noah Harding Professor
Computer Science

Linda Torczon, Senior Research Associate
Computer Science

John Bennett, Assistant Professor
Electrical and Computer Engineering

Robert Michael Lewis, Research Associate
Mathematical Sciences

Houston, Texas

April, 1992

Register Allocation via Graph Coloring

Preston Briggs

Abstract

Chaitin and his colleagues at IBM in Yorktown Heights built the first global register allocator based on graph coloring. This thesis describes a series of improvements and extensions to the Yorktown allocator. There are four primary results:

Optimistic coloring Chaitin's coloring heuristic pessimistically assumes any node of high degree will not be colored and must therefore be spilled. By optimistically assuming that nodes of high degree will receive colors, I often achieve lower spill costs and faster code; my results are never worse.

Coloring pairs The pessimism of Chaitin's coloring heuristic is emphasized when trying to color register pairs. My heuristic handles pairs as a natural consequence of its optimism.

Rematerialization Chaitin *et al.* introduced the idea of rematerialization to avoid the expense of spilling and reloading certain simple values. By propagating rematerialization information around the SSA graph using a simple variation of Wegman and Zadeck's constant propagation techniques, I discover and isolate a larger class of such simple values.

Live range splitting Chow and Hennessy's technique, priority-based coloring, includes a form of live range splitting. By aggressively splitting live ranges at selected points before coloring, I am able to incorporate live range splitting into the framework of Chaitin's allocator.

Additionally, I report the results of experimental studies measuring the effectiveness of each of my improvements. I also report the results of an experiment suggesting that priority-based coloring requires $O(n^2)$ time and that the Yorktown allocator requires only $O(n \log n)$ time.

Finally, I include a chapter describing many implementation details and including further measurements designed to provide an accurate intuition about the time and space requirements of coloring allocators.

Acknowledgments

I wish to thank my committee: Keith Cooper, John Bennett, Ken Kennedy, Michael Lewis, and Linda Torczon. It is exceptionally difficult to acknowledge the help of my principal advisor, Keith Cooper, without simultaneously acknowledging the help of Linda Torczon. Their advice, support, and friendship helped make my graduate school career the most productive and enjoyable period of my life.[1] Ken Kennedy helped directly, with advice and encouragement, and indirectly, with his role in the establishment and direction of the Department of Computer Science at Rice. John Bennett and Michael Lewis served as outside members of my committee; I thank them for their interest and their criticism.

Naturally, I often talked to other members of the faculty and I wish to thank Bob Hood, John-Mellor Crummey, and Scott Warren for their help and advice. I also wish to thank my fellow students, especially Alan Carle, Steve Carr, Ben Chase, Mary Hall, Kathryn McKinley, and Chau-Wen Tseng for interesting conversation of all kinds. Vernon Lee gets special recognition for his courageous role as my "user community." Furthermore, I thank all those who worked on the $\mathbb{R}^n$ programming environment; without the foundation provided by their efforts, none of my experiments would have been possible. I also thank the systems support group, especially Vicky Dean, for handling all my problems over the years. Finally, I thank Ivy Jorgensen for her careful proofreading and for all her help with the required administrative details.

There are many people outside Rice, in both academia and industry, who have taken time to read and comment on my work. I'd like to thank David Callahan, Greg Chaitin, Fred Chow, Marty Hopkins, Brian Koblenz, and Chuck Lins for their help and encouragement. I also thank Randy Scarborough for his attention over the course of many years; his questions and comments prompted much of the work in this thesis.

[1] Therefore, the use of "we" throughout the text should be recognized as a further acknowledgement of their contribution.

Unlike many of the graduate students at Rice, I grew up in Houston and have a number of friends in the community. I would like to thank Paul and Kathryn Baffes, Jerry and Theresa Fuqua, Dave and Renee Odom, and Jamie, Donna, and Catherine Rickenbacker. Their friendship has eased my graduate years.

Financially, I have been supported from several sources. Support for my first year was provided by a Presidential Fellowship, as part of a program initiated by Norman Hackerman during his service as President of Rice University. Thereafter, I was supported as part of projects funded by IBM and DARPA.

Finally, I gratefully acknowledge my family's love and encouragement. Surely I owe them more than I can say.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Compilers and Optimization

Classically, an optimizing compiler is divided into three stages:

The *front-end* translates the source language into an intermediate form. This translation may be accomplished in one or more passes over the code, depending on the structure of the source language. Compile-time error checking is usually performed at this stage. Ideally, the front-end is language dependent and machine independent.

The *optimizer* consists of several passes, each performing specific transformations on the intermediate form. While the transformations are intended to improve performance of the final code, there is no question of achieving any sort of real optimality. We are interested in global optimizations; that is, optimizations that use information gathered from an entire routine to guide transformations. Common global optimizations include strength reduction, loop-invariant code motion, and common subexpression elimination [4]. The optimizer is intended to be both language and machine independent.

The *back-end* translates the intermediate form into a machine-specific form, usually object code. This translation, also called *code generation,* may require several passes, including instruction selection, instruction scheduling, and register allocation. The back-end is largely language independent and machine dependent.

This division provides a useful separation of concerns, simplifying the development and maintenance of each stage. Additionally, there is the possibility of reusing each stage in several compilers. For example, a completely machine-independent front-end for FORTRAN might be used in compilers for many different machines.

Of course, this is an idealized view. In practice, each stage tends to exhibit both language and machine dependencies. These dependencies inhibit reuse and maintenance and are therefore the target of compiler designers. Typically, we see such reuse only when compiling closely related languages (*e.g.*, FORTRAN and C) for closely related machines (*e.g.*, the common 32-bit RISC processors).

1.2 Optimization and Register Allocation

A register is one of a small number of high-speed memory locations in a computer's CPU. Registers differ from ordinary memory locations in several respects.

- The register set is small; a register may be directly addressed with a few bits. Memory can be quite large; a memory location is usually specified indirectly, using an "addressing mode" that includes one or more register references.
- Registers are fast; typically, two registers can be read and a third written – all in a single cycle. Memory is slower; a single access can require several cycles.

The limited size and high speed of the register set makes it one of the critical resources in most computer architectures. A register allocator, typically one phase of the back-end, controls utilization of the register set by a compiled program.

Registers, and therefore register allocators, must serve many purposes. In the simplest case, operands for primitive machine instructions must appear in registers. Intermediate results, arising during the evaluation of complex expressions, are held in registers by even naive compilers. More sophisticated compilers attempt to place frequently used variables in registers to avoid repeated fetches and stores. For an optimizing compiler, registers are the ideal place to hold values for reuse after common subexpression elimination or loop-invariant code motion. It is in connection with optimization that register allocation becomes crucially important.

During the development of the first FORTRAN compiler, John Backus suggested that the optimization of subscript expressions should be considered separately from the question of allocating index registers [7]. This idea has since been extended beyond the problems of optimizing subscript expressions; our approach to the design of optimizing compilers says:

> *During optimization, assume an infinite set of registers; treat register allocation as a separate problem.*

This important, perhaps essential, separation of concerns enables optimization to proceed in a relatively simple fashion, optimistically avoiding difficult choices caused by limited resources. This point of view was promoted by John Cocke and led to the development of the influential PL.8 compiler and 801 computers [50, 6].

When there are enough registers, this separation of concerns looks like a good idea. When there are not enough registers, the assumption underlying the separation of concerns breaks down and we see cases where optimization causes degradation due to lack of registers.

The task of register allocation may be attacked at one of several levels:

- Register allocation may be performed over *expressions*. This technique is a form of instruction scheduling, with the goal of reducing register requirements. Work by Aho, Johnson, Sethi, and Ullman considers how to minimize register requirements by careful ordering of expression evaluation [60, 2].
- More aggressive allocators can manage registers over a complete *basic block*. Work by Freiburghouse suggests one practical approach [37]. Further work by Aho, Johnson, and Ullman proves the difficulty of generating optimal code in the presence of common subexpressions [3].
- *Global* allocators work over an entire routine. Chaitin's allocator operates at this level. Other examples include work by Chow and Hennessy and work by Callahan and Koblenz [26, 16].
- *Interprocedural* register allocation works over a collection of routines, usually an entire program. Examples include work by Wall and work by Santhanam and Odnert [63, 58].

We believe that global register allocation is required to support global optimization.

1.3 Register Allocation and Graph Coloring

Unfortunately, good register allocation is difficult. Idiosyncratic machine details complicate even the simplest allocators. Robust allocators must also deal gracefully with complex programs and inadequate numbers of registers. Furthermore, attempts to achieve optimal solutions for any of these problems invariably lead to combinatorial explosion.

Graph coloring offers a simplifying abstraction. By building and coloring an *interference graph* representing the constraints essential to register allocation, we are able to handle many apparently disparate details in a unified fashion. Nodes in the interference graph represent live ranges (*e.g.*, variables and temporaries) and edges represent interferences between live ranges. Roughly, if two live ranges are both *live* at some point in the routine, they are said to *interfere* and cannot occupy the same register.[1] If the nodes in the graph can be colored in k or fewer colors, where any pair of nodes connected by an edge receive different colors and k is the number of registers available on the machine, then the coloring corresponds to an allocation. If

[1] Several definitions of *live* and *interfere* are possible. See Section 2.2.2 for more discussion.

a k-coloring cannot be discovered, then the code must be modified and a new coloring attempted. A global register allocator based on this approach was developed by Greg Chaitin and his colleagues at IBM [20].

1.3.1 Minimizing Register Usage

Informally, the goal of register allocation is to minimize the number of loads and stores that must be executed. Reducing the register allocation problem to the graph coloring problem subtly changes the goal; instead of minimizing memory traffic, the "reduced" goal is minimizing register usage. In other words, by shifting our attention to graph coloring, we are attacking a nearby problem. Note though, that this new goal is well suited to the style of optimization advocated by Backus and Cocke (*i.e.*, optimistically assuming an infinite register set during optimization). Many transformations are justified only if there is a register available to hold a temporary value. Proceeding optimistically, the optimizer will always assume such transformations are profitable. If the register allocator is able to map all of the registers used by the optimizer onto the finite set of machine registers, then all of the optimizer's assumptions will be correct. Therefore, the goal of minimal register usage is desirable, as is a large register set – each support the optimizer.

1.3.2 Minimizing Spill Code

Even with optimal coloring and a large register set, it will sometimes be necessary to *spill* certain values to memory. Several difficult problems arise. Overall, we wish to minimize the dynamic cost of inserted spill instructions (loads and stores). We must somehow choose live ranges to spill that are cheap to spill and that relax pressure in the graph, allowing coloring to progress. Furthermore, we must choose where to place spill instructions. These problems are all complex and highly interrelated; nevertheless, good solutions are required.

1.4 Overview

This thesis presents a series of extensions to Chaitin's work on register allocation via graph coloring. Chapter 2 presents an introduction to Chaitin's work and a brief history of the field. The main results of the thesis are contained in Chapters 3 though 6; they are briefly introduced in the sections below. Chapter 7 describes the framework used to compare allocation techniques and summarizes the results of a series of experiments testing the efficacy of our improvements. Additionally, we briefly compare the Yorktown allocator with priority-based coloring, a competitive approach developed by Chow and Hennessy [26]. Chapter 8 describes many of the important details required for efficient implementation of a graph coloring allocator. Additionally, a variety of measurements are included to help provide intuition about the expected costs, in time and space, of a coloring allocator.

1.4.1 Improved Coloring and Spilling

Optimal graph coloring is unlikely to be practical. The problem of determining the minimal number of colors needed to color an arbitrary graph is NP-complete [48]. Furthermore, the problem of finding a k-coloring for some fixed $k \geq 3$ is also NP-complete [39]. Finally, Garey and Johnson have shown that unless P = NP, no polynomial-time heuristic can guarantee using less than twice the minimal number of colors [38].

Due in part to these pessimistic results, there has been little study in the area of *optimal* coloring for global register allocation. Instead, researchers have concentrated on finding efficient *heuristic* approaches [18, 46, 9]. Useful heuristics extend naturally to help solve the spill problem; that is, they provide guidance when live ranges must be spilled.

In Chapter 3, we present a refinement to Chaitin's coloring heuristic. Our heuristic may be considered *optimistic* in contrast to Chaitin's *pessimistic* approach. The optimistic heuristic spills a subset of the live ranges spilled by the pessimistic heuristic.

1.4.2 Coloring Pairs

On many important architectures, a pair of single-precision floating-point registers may be treated as a double-precision register. Additionally, some machines provide instructions that load and store pairs and quadruples in a single instruction. Unfortunately, there is no adequate way to take advantage of these features using Chaitin's allocator.

Chapter 4 discusses the problem in some detail. We show why Chaitin's coloring heuristic overspills in the presence of register pairs and why our optimistic coloring heuristic avoids overspilling.

1.4.3 Rematerialization

Many important details must be handled correctly for best results from a global allocator. Stated more strongly, they must be handled correctly to achieve simply acceptable results. One example, mentioned briefly by Chaitin *et al.*, is the idea of *rematerialization*. This is a technique required for acceptably clean spilling of live ranges defined by constants and other simple expressions.

In Chapter 5, we introduce an extension to Chaitin's allocator allowing precise spilling and rematerialization of a wider class of live ranges.

1.4.4 Live Range Splitting

Fabri and Chow independently observed that splitting a single live range into several pieces and considering the new, smaller live ranges separately can produce an interference graph that colors with fewer colors [34, 25]. Chow and Hennessy used this idea, called *live range splitting*, as the basis for a new allocator that avoided spilling when splitting was possible.

Live range splitting has several merits. If an entire live range is spilled, as in Chaitin's work, its value will reside in a register only for short periods around each definition and use. Splitting allows the value to stay in a register over longer intervals – often an entire block or over several blocks.

Unfortunately, live range splitting is difficult. There are two fundamental problems: picking live ranges to split and picking places to split them. While optimal solution of either of these problems is certainly NP-hard, Chapter 6 extends the ideas introduced in Chapter 5 to attack both of these problems.

Chapter 2

Background

The first implementation of a graph coloring register allocator was described by Chaitin *et al.* [20]. This chapter explains their allocator in some detail. The first section introduces the general concept of register allocation via graph coloring. The second concentrates on the Yorktown allocator, including explanations of the individual phases. The last section gives a brief history of the area.

2.1 Register Allocation via Graph Coloring

We assume that the allocator works on low-level intermediate code, similar to assembly. The code has been shaped by an optimizer, addressing modes have been determined, and an execution order has been fixed. Of course, these assumptions ignore the possibility of cooperation between allocation and other parts of the compiler; see Chapter 9 for a discussion of these opportunities. For simplicity when discussing the generation of spill code, we assume a load-store architecture; however, provisions can be made for more complex target architectures (see Chapter 8).

Before allocation, the intermediate code can reference an unlimited number of registers. We refer to this unrestricted set of "pre-allocation" registers as *virtual registers*. The goal of allocation is to rewrite the intermediate code so that it uses only the registers available on the target machine – the *machine registers*. Note that both virtual registers and machine registers serve simply as names, much like variables in C and FORTRAN. In a manner common to other portions of the compiler, we care little about names *per se*; instead, we care about the named objects.

In the case of register allocation, we are concerned with *values* and *live ranges*. A value corresponds to a single definition. A live range is composed of one or more values, connected by common uses. On input to the allocator, all the values comprising a single live range will be named by the same virtual register. Furthermore, a single virtual register may also name several other live ranges. Similarly, any machine register will usually name several live ranges after allocation.

To model register allocation as a graph coloring problem, the compiler first constructs an interference graph G. The nodes in G represent live ranges and the edges represent *interferences*. Thus, there is an edge in G from node i to node j if and only if live range l_i *interferes* with live range l_j; that is, they are simultaneously live at some point and cannot occupy the same register. The live ranges that interfere with a live range l_i are called *neighbors* of l_i in the graph; the number of neighbors in the graph is the *degree* of l_i – denoted l_i°.

To find an allocation from G, the compiler looks for a k-coloring of G – an assignment of colors to the nodes of G such that neighboring nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring into a feasible register assignment. Because finding a k-coloring of an arbitrary graph is NP-complete, the compiler uses a heuristic technique to search for a coloring; it is not guaranteed to find a k-coloring for all k-colorable graphs.

Of course, some routines are sufficiently complex that no k-coloring is possible, even with an exhaustive coloring algorithm; their interference graphs are simply not k-colorable. If a k-coloring cannot be found, some live ranges are *spilled*; that is, kept in memory rather than registers.

2.2 The Yorktown Allocator

The first implementation of a global register allocator based on graph coloring was done by Chaitin and his colleagues as part of the PL.8 compiler at IBM Research in Yorktown Heights [6]. Further work by Chaitin yielded an improved coloring heuristic that attacked the problems of coloring and spilling in an integrated fashion [18].

This thesis builds directly upon the work of Chaitin and his colleagues; therefore, it is important to establish a clear understanding of (our interpretation of) their work. Figure 2.1 illustrates the overall flow of the Yorktown allocator.

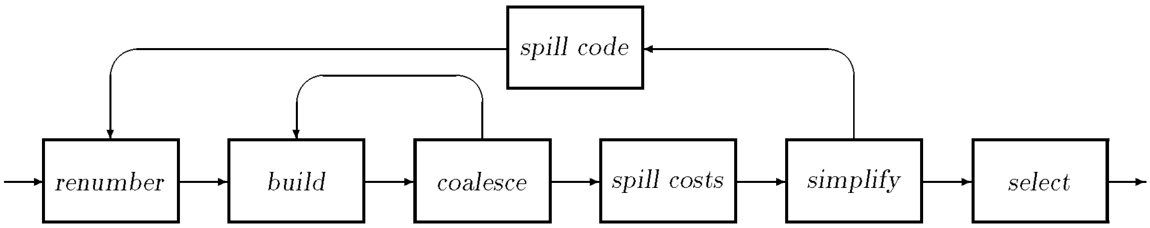


Figure 2.1 The Yorktown Allocator

Renumber This phase finds all the live ranges in a routine and numbers them uniquely. In the papers on the PL.8 compiler, this type of analysis is referred to as getting "the right number of names."

Build The next step is to construct the interference graph G. For efficiency, G is simultaneously represented in two forms: a triangular bit matrix and a set of adjacency vectors.

Coalesce The allocator removes unneeded copies, eliminating the copy instruction itself and combining the source and target live ranges. A copy may be removed if the source and target live ranges do not interfere. We denote the coalesce of l_i and l_j as l_{ij}.

Since the removal of a copy instruction can change the interference graph, we repeat *build* and *coalesce* until no more copies can be removed. However, when the allocator combines l_i and l_j, it can quickly construct a conservative approximation to the set of interferences for l_{ij}. The conservative update of G lets the allocator perform many combining steps before rebuilding the graph; in practice, we make a complete pass over the code before rebuilding.

Spill Costs In preparation for coloring, a spill cost estimate is computed for every live range l. The spill cost for l is an estimate of the cost of load and store instructions that would be required to spill l. The cost of each instruction is weighted by 10^d where d is the instruction's loop nesting depth, giving a simple approximation of the actual impact at run-time.

Simplify This phase, together with *select,* cooperates to color the interference graph. *Simplify* repeatedly examines the nodes in G, removing all nodes with degree $< k$. As each node is removed, its edges are also removed (decrementing the degree of its neighbors) and it is pushed on a stack s.

If we reach a point where every node remaining in G has degree $\geq k$, a node is chosen for spilling. Rather than spilling its corresponding live range immediately (requiring updates of the code and recomputation of the interference graph), it is simply removed from G and marked for spilling.

Eventually, G will be empty. If any nodes have been marked for spilling, they are spilled in *spill code* and the entire allocation process is repeated. Otherwise, no spill code is required and s is passed on to *select*.

Select Colors are chosen for nodes in the order determined by *simplify*. In turn, each node is popped from s, reinserted in G, and given a color distinct from its neighbors. The success of *simplify* ensures that a color will be found for each node as it is inserted.

Spill Code In a single pass over the routine, spill code is inserted for each spilled live range. Since we are assuming a load-store architecture, spilling requires (approximately) a load instruction before each reference to a spilled live range and a store after each definition of a spilled live range. Refinements to this simple policy are introduced below.

Note that values are spilled to locations in the current stack frame. There are several reasons for this policy. First, many values have no natural location in memory; *e.g.,* compiler-generated temporaries. Furthermore, by spilling to the stack, we are able to handle recursive and reentrant routines. Finally, locations in the stack frame can typically be accessed quickly.

The following sections give further detail about the various phases of allocation.

2.2.1 Discovering Live Ranges

In a given routine, a variable i may be used many times, for many different tasks. Similarly, a routine expressed in intermediate form after optimization may use the same virtual register for several purposes. However, there is no need for the allocator to assign each disjoint use of some virtual register to the same machine register. In fact, such behavior is undesirable since it constrains the possible colorings.

Each disjoint use of a virtual register is a unique *live range* and it is the live ranges in a routine that are colored by the allocator. Therefore, the first task of the allocator is to discover the live ranges in a routine. This procedure is called *getting the right number of names* by Chaitin and *web analysis* by Johnson and Miller [46]. In our implementation, each live range is given a unique index and the intermediate code is rewritten in terms of live range indices instead of the original virtual register numbers – hence the term *renumber*.

Conceptually, live ranges are discovered by finding connected groups of *def-use chains*. A single def-use chain connects the definition of a virtual register to all of its uses. When several def-use chains share a single use (in other words, when several definitions *reach* a single use), we say they are connected by the use. Of course, all the chains originating at a given definition are considered connected.

Consider the example shown in Figure 2.2. The upper half is an abstract control-flow graph with a few low-level statements representing code before renumbering. The lower half illustrates the same code, but rewritten to illustrate the effect of renumbering. Notice that four different live ranges have been discovered, all originally represented by $r50$. The simple cases ($r0$ and $r2$) are restricted to a single basic block.

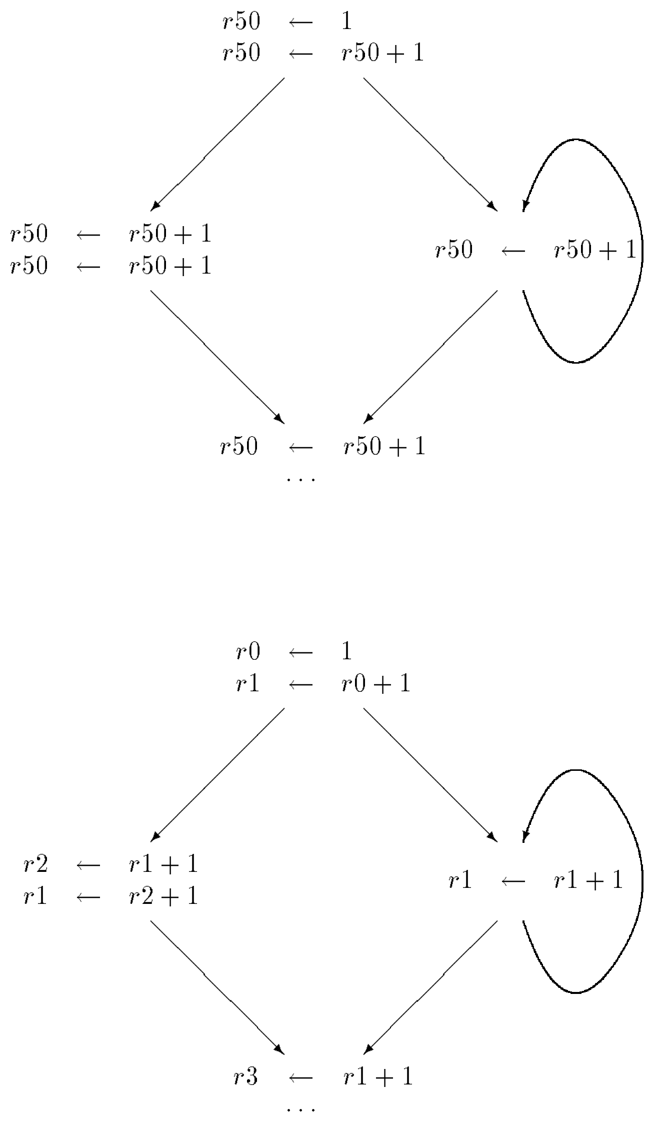


Figure 2.2 Renumbering

The live range represented by $r1$ is more complex – three definitions in different basic blocks are connected by uses in three other basic blocks. It is precisely this sort of code that makes global allocators more powerful than allocators that are restricted to expressions or basic blocks. The efficient implementation of *renumber* is discussed in Section 8.4.

2.2.2 Interference

The concept of *interference* is an important key to understanding graph coloring allocators. Intuitively, if the allocation of two live ranges to the same register changes the meaning of the program, they interfere. Chaitin *et al.* give a precise set of conditions for interference, noting that two live ranges interfere if there exists some point in the procedure and a possible execution of the procedure such that:

1. both live ranges have been defined,
2. both live ranges will be used, and
3. the live ranges have different values.

Each of these conditions is generally undecidable, as is their intersection. Chaitin's approach is to approximate interference by noting which live ranges are both *live* and *available* (in the data-flow sense) at each assignment.

We say that a live range l for a variable v is live at some statement s if there exists a path from s to some use of v and there is no assignment to v on the path. Similarly, l is available at s if there is a path from a definition of v leading to s. Note that availability and liveness correspond to conditions 1 and 2 above; they are conservative approximations of the exact but undecidable conditions required for interference.[2]

By handling copy instructions specially, Chaitin is also able to achieve a conservative approximation of condition 3. Since the source and destination live ranges will certainly have the same value at a copy, they need not interfere. In fact, for there to be any possibility of coalescing, they must not interfere. Of course, if they interfere for other reasons (perhaps one is incremented in a loop), then an interference will be added at another point in the code and coalescing will be correctly inhibited.

[2]Condition 2 specifies that a value *will* be used; liveness says that it *may* be used. It is the absolute guarantee of "will be used" that makes condition 2 undecidable in the general case. Similar arguments hold for conditions 1 and 3.

An alternative approach is to add interferences at each block by making all live ranges that are both live and available at the end of each basic block interfere with each other. However, this approach is less precise than Chaitin's idea of adding interferences at each assignment due to Chaitin's careful handling of copy instructions. Furthermore, the block-level approach can require much more time, since it can require adding $O(n^2)$ interferences at each block versus $O(n)$ at each assignment. The exact tradeoff is difficult to determine, since it depends on the number of assignments and the average number of live ranges (n) alive across each point in the routine.

2.2.3 The Interference Graph

One of the central data structures in the Yorktown allocator is the interference graph. Viewed as an abstract data type, the interference graph must provide five operations:

new(n) Return a graph with n nodes, but no edges.

add(g, x, y) Return a graph including g with an edge between the nodes x and y.

interfere(g, x, y) Return *true* if there exists an edge between the nodes x and y in the graph g.

degree(g, x) Return the degree of the node x in the graph g.

neighbors(g, x, f) Apply the function f to each neighbor of node x in the graph g.

In practice, the interference graph is implemented using two representations: a triangular bit matrix and a set of adjacency vectors. The bit matrix supports constant-time implementations of *add* and *interfere* while the adjacency vectors support the efficient implementation of *neighbors*. While initialization of the bit matrix requires $O(n^2)$ time, the constant is small in practice (see Sections 7.2.2 and 8.5).

Note that the dual representation is important for efficiency. Without the bit matrix, the speeds of *interfere* and *add* degrade sharply. Alternatively, without the adjacency vectors, the cost of visiting all the neighbors of a node increases, raising the cost of coloring from $O(n + e)$ to $O(n^2)$. While e is theoretically bounded by n^2, in practice, $e \ll n^2$.

An alternative implementation, based on a hash table of interfering pairs, offers the same asymptotic efficiencies and avoids the $O(n^2)$ space requirements of the bit matrix. In practice, the time considerations favor the bit-matrix representation. Space considerations also favor the bit-matrix representation for small graphs; for large graphs, the hash-table representation may become desirable.

Early versions of the Yorktown allocator constructed the interference graph in a single pass, storing adjacencies in a linked list of short vectors. Later versions adopted a two-pass approach, storing adjacencies in a continuous vector.[3]

1. Initially, the bit matrix is allocated and cleared. A pass is made over the code, filling in the bit matrix and accumulating each node's degree.
2. After the degree of every node is known, adjacency vectors are allocated and the bit matrix is reinitialized. In a second pass over the code, interferences are recorded in the bit matrix and the adjacency vectors.

After each pass of *coalesce*, the graph must be reconstructed. In practice, only step 2 must be repeated, since the degree of each node can be incrementally maintained while coalescing.

In our implementation, each pass runs backward over each basic block in the control-flow graph, incrementally maintaining a set s of all live ranges that are currently live and available. At each definition, edges are added between the defined value and all members of s. See Section 8.5 for more details on the efficient construction of the interference graph.

After completing the *build-coalesce* loop, the memory required for the bit matrix may be deallocated. The adjacency vectors are required for further use during coloring, both by *simplify* and *select*.

2.2.4 Coalescing

After building the interference graph, we are in a position to perform *coalescing* (also called subsumption and copy propagation). The code is traversed in any convenient order. At each copy instruction, we check to see if the source and target live ranges interfere; if not, they may be coalesced and the copy instruction may be deleted. To coalesce two live ranges l_x and l_y forming a third l_{xy}, we simply replace every mention of either live range with a reference to the result; that is, we replace every mention of l_x and l_y with l_{xy}.

To perform coalescing efficiently, we establish equivalence classes for each live range. As live ranges are coalesced, their equivalence classes are unioned, using a fast disjoint-set union algorithm [1, pages 129–139].

[3] The reasoning was that the vectors offered quicker traversal and better locality.

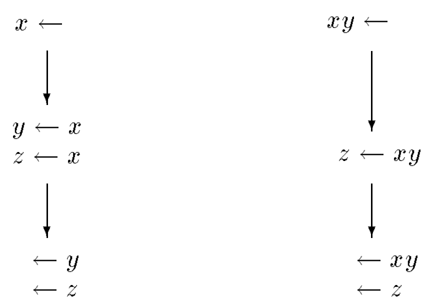


Figure 2.3 Effects of Coalescing

When two live ranges l_x and l_y are coalesced, we must update the interference graph so that l_{xy} interferes with the neighbors of l_x and with the neighbors of l_y. While we are able to perform this update accurately, we cannot accurately reflect updates due to copy instructions being removed. Recall that interferences are added at each assignment. When we remove a copy instruction (which is an assignment), some interferences may also be removed. Figure 2.3 illustrates such a case.

In the left column, y and z interfere since y is live across the definition of z. While x is live across the definition of y, we are certain that there is no interference since it is clear that x and y have the same value. The right column shows the result of coalescing x and y. We have rewritten the code in terms of xy and remove the now-useless copy. When the interference graph is updated, xy will interfere with z since the result of a coalesce is made to interfere with all the neighbors of the coalesced ranges. However, it is clear that xy and z do not interfere, so the interference graph is imprecise. Therefore, we must rebuild the interference graph from scratch to ensure accuracy.

In practice, *coalesce* makes a complete pass through the intermediate code, coalescing wherever possible and updating the interference graph in the conservative fashion described above. If any copies are removed, the interference graph is rebuilt and more coalescing attempted. This cycle repeats until no more copies can be removed. While the *build-coalesce* cycle is bounded by the number of copies in the code, convergence is usually quite rapid – typically two or three iterations suffice. See Section 8.6 for measurements on real code.

Uses of Coalescing

Much of the power and generality of the Yorktown allocator is due to the wide applicability of coalescing. Uses suggested by Chaitin and our own experience include:

- Removing copies introduced during optimization allows use of simpler forms of some optimizations. For example, coalescing is extremely useful in cleaning up the copies resulting from the removal of ϕ-nodes after using SSA-form [29].
- Coalescing can be used to achieve *targeting,* which attempts to compute arguments in the correct register for passing to a called procedure. In a called procedure, coalescing enables easy handling of incoming arguments passed in registers.
- Similarly, coalescing enables easy handling of the operands and results of machine instructions with special register requirements; *e.g.*, a multiply instruction that requires its operands to be in a particular pair of registers.
- Coalescing enables natural handling of common 2-address instructions; *e.g.*, instructions of the form $r_x \leftarrow r_x + r_y$ where the destination is constrained to match the first operand.

2.2.5 Spilling

The roughest possible version of *spill* would spill a live range l by inserting a store after every definition of l and a load before every use of l. Chaitin *et al.* give two important refinements to this coarse approach.

First, they note that certain live ranges are easy to recompute; for example, live ranges defined by constants. These live ranges should not be stored and reloaded; instead, they should be recomputed before each use. Of course, it is trivial to "recompute" a constant, but the technique also applies to certain expressions involving the frame pointer and the constant pool.

Second, it is not necessary to spill around *every* mention of a live range. Chaitin describes several situations that should be handled using local analysis.

- If two uses of a spilled live range are close together, it is unnecessary to reload for the second use; simply use the same register for both uses.
- If a use follows closely behind the definition of a spilled live range, there is no need to reload before the use.
- Similarly, if all uses of a live range are close to the definition, the live range should not be spilled.

In Chaitin's work, two references are considered "close" if no live range goes dead between them. Alternatively, if the last use of any interfering live range occurs between two references, those references are considered distant. See Section 8.7 for details and the accurate computation of spill costs.

2.2.6 Coloring

The core of the Yorktown allocator is the coloring algorithm. Since the problem of finding a k-coloring for an arbitrary graph is NP-complete, we rely on non-optimal heuristic techniques.[4] When reading modern descriptions of graph coloring heuristics, it is easy to forget the difficulty of devising good heuristic approaches to difficult problems. Schwartz presents two algorithms (one attributed to Cocke, the other to Ershov) illustrating some of the early attempts [59, pages 327–329].

The heuristic employed in the Yorktown allocator was devised by Chaitin [19]. It requires $O(n+e)$ time, where n is the number of live ranges to be colored and e is the number of edges in the interference graph. Chaitin also shows how his heuristic can be used to accomplish both coloring and spilling in an integrated fashion [18]. In our framework, Chaitin's coloring heuristic is distributed between *simplify* and *select*.

Why does it work? *Simplify* repeatedly removes nodes from the graph and pushes them on a stack. In *select*, the nodes are popped from the stack and added back to the graph. A node l_i is only moved from the graph to the stack if $l_i^\circ < k$. Therefore, when *select* moves l_i from the stack back to the graph, l_i will still have less than k neighbors. Clearly there will be a color available for l_i in that graph.

Simplify only removes a node when it can prove that the node will be assigned a color in the current graph. As each live range is removed, the degrees of its neighbors are lowered. This, in turn, may prove that they can be assigned colors. In *select*, the nodes are assigned colors in reverse order of removal. Thus, each node is colored in a graph where it is trivially colorable – *simplify* ordered the stack so that this must be true. In one sense, the ordering forces the allocator to color the most constrained nodes first – l_i gets colored before l_j precisely because *simplify* proved that l_j was colorable independent of the specific color chosen for l_i.

As an example, consider finding a three-coloring for the simple graph shown in Figure 2.4. The left column (working from the top down) illustrates one possible sequence of simplifications. In the initial graph, $l_1^\circ < 3$, so we are assured that a

[4]Chaitin *et al.* give a proof that *any* graph can be generated by some routine.

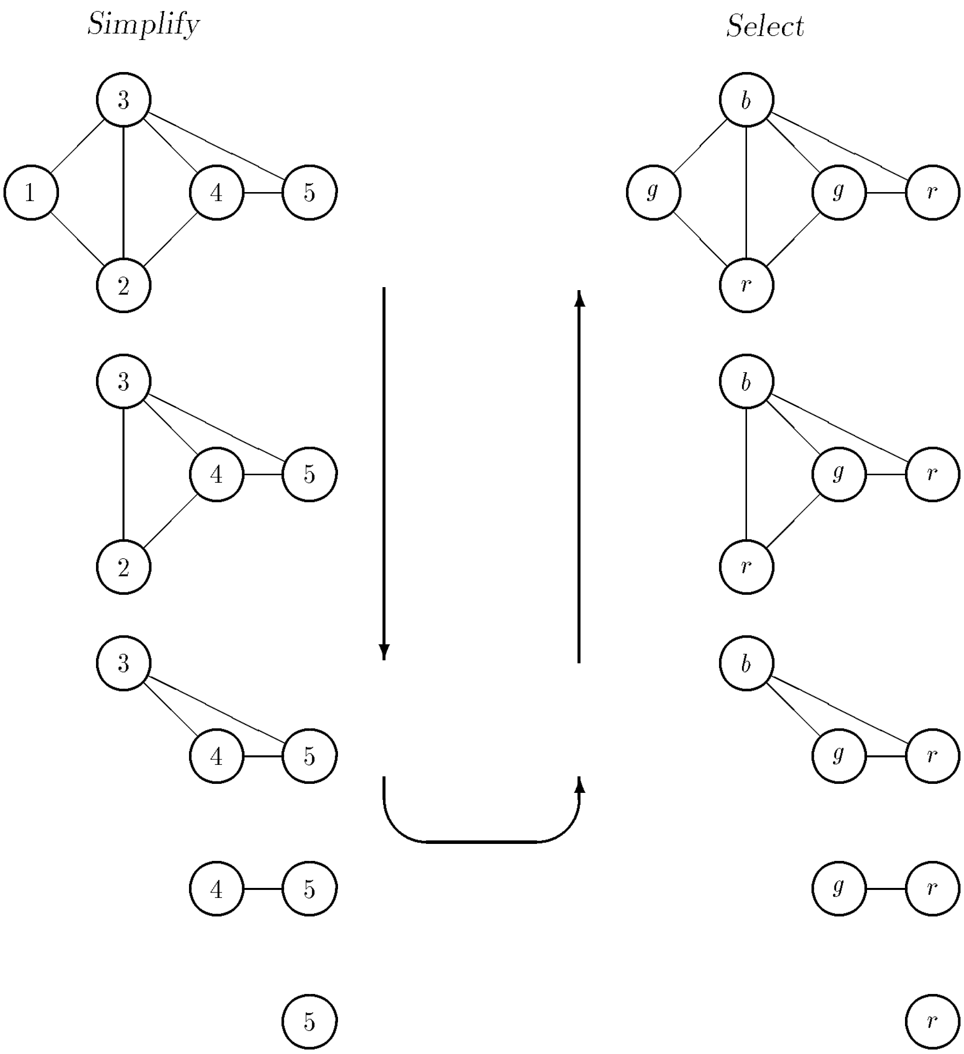


Figure 2.4 Coloring a Simple Graph

color will be available during *select*. When l_1 is removed, the degrees of l_2 and l_3 are lowered. Since l_2° is now < 3, it is removed in turn. The process is repeated until the graph is completely empty. No spilling is required in the case since there are nodes with degree < 3 available at every step. The right column (working back up) shows how *select* reconstructs the graph, coloring each node as it is added to the graph.

If *simplify* encounters a graph containing only nodes of degree $\geq k$, then a node is chosen for spilling (see next section). The spill node is removed from the graph and marked for spilling. One alternative at this point is to immediately insert spill code for the spill node, rebuild the interference graph, and attempt a new coloring. This approach is precise but expensive since some routines may require spilling many live ranges. Chaitin uses a less precise approach, continuing simplification after choosing a spill node, potentially marking many nodes for spilling in a single pass.

Choosing Spill Nodes

The metric for picking spill nodes is important. Chaitin suggests choosing the node with the smallest ratio of spill cost divided by current degree.

$$m_n = \frac{cost_n}{degree_n} \tag{2.1}$$

Note that $degree_n$ is the *current* degree of the node n; that is, the degree of n in the partial graph remaining after removing all nodes of low degree. This metric reflects a desire to minimize total spill costs coupled with a desire to simplify the graph by lowering the degree of many nodes (the neighbors of node n).

Later work by Bernstein *et al.* at Haifa explores other spill choice metrics [9]. They present three alternative metrics:

$$m_n = \frac{cost_n}{degree_n^2} \tag{2.2}$$

$$m_n = \frac{cost_n}{degree_n \mathrm{area}_n} \tag{2.3}$$

$$m_n = \frac{cost_n}{degree_n^2 \mathrm{area}_n} \tag{2.4}$$

In equations 2.3 and 2.4, $area_n$ represents an attempt to quantify the impact n has on live ranges throughout the routine.

$$area_n = \sum_{\substack{i \,\in\, instructions \\ n \text{ is alive at } i}} 5^{depth_i} width_i \tag{2.5}$$

Here $depth_i$ is the number of loops containing the instruction i and $width_i$ is the number of live ranges live across the instruction i.

The experiments of Bernstein *et al.* suggest that no single spill metric completely dominates the others. Therefore, they propose using a "best of 3" technique. They repeat *simplify* three times, each time with a different spill metric, and choose the version giving lowest total spill cost. The reason behind the practical success of the "best of 3" heuristic is perhaps subtle. Choosing the best node to spill is NP-complete; therefore, we expect counter-examples for *any* spill metric. By using a combination of three, we gain some measure of protection from the worst-case examples. To an extent, we view the "best of 3" heuristic as a filter that helps to smooth some of the *NP-noise* in our results.[5]

2.3 History

The idea of solving allocation problems by abstracting to graph coloring has a surprisingly long history. Ershov notes that early interest in graph theory among programmers (at least in the Soviet Union) was due to the reduction of storage packing problems to the graph coloring problem [33, page 174]. Historically, we see two partially overlapping threads: work in memory allocation and work in register allocation.

2.3.1 Memory Allocation

By memory allocation, we mean the problem of laying out storage for variables (scalars and arrays) in main memory so that they require minimal space. On early machines, this was an important concern, given their small memories.

Apparently, the earliest work on memory allocation and graph coloring was published by Lavrov in 1961 [52]. The work is difficult to understand, hampered somewhat by translation, but more significantly by the lack of common vocabulary (*e.g.*, there are no instructions, basic blocks, live ranges, or control-flow graphs – instead we see operators, routes, carriers, data paths, and areas of effect). Nevertheless, it is clear that the definition of an *incompatibility graph* is key to his approach.[6]

Inspired by Lavrov, Ershov also explored the correspondence between memory allocation and graph coloring [30]. A coloring-based memory allocator was described

[5]The term *NP-noise* was coined by Linda Torczon to describe the annoyingly large variations in spill costs that occur with even the smallest adjustments to the coloring algorithm.

[6]Those interested in reading Lavrov should certainly consult Ershov as a guide [33, pages 170–173].

as part of the Alpha compiler [31, 32]. It is also interesting to read the *Editor's Note* appended to Ershov's *JACM* paper.

> *The attainment of "global memory economy" by means of the "inconsistency matrix" is a novel scheme for minimizing the number of storage cells assigned to variables. An incidence matrix (inconsistency matrix) is constructed which shows which variables may not occupy the same cell. This permits extreme compression of the storage area for variables.*
>
> Ascher Opler

An extensive description of Ershov's work on memory allocation and graph coloring is included in his book [33].

Extensions to Ershov's work in this area were reported by Fabri [34, 35]. However, this general approach to conserving memory seems to be of less interest recently. There are perhaps several reasons:

- relatively large, cheap main memories now available,
- increased reliance on stack-based allocation, with its naturally conservative approach (however, see [56]), and
- almost exclusive use of separate compilation, making the whole-program analysis required for memory allocation seem painfully awkward.

Nevertheless, the increasing importance of memory locality together with approaches to convenient whole-program analysis [27] may lead to a renewed interest in the problems of packing main memory.

It is important to note that the work of Lavrov, Ershov, and Fabri attacked the problem of packing arrays in memory. This is *not* a trivial extension of the scalar packing problem nor is it naturally expressed as graph coloring. On the other hand, register allocation is a much closer fit to coloring.

2.3.2 Register Allocation

The idea of managing global register allocation via graph coloring is apparently due to Cocke [49, 19]. We find some limited discussion of graph coloring in the early 1970's; however, it seems to concentrate more on the search for useful coloring heuristics than on the problems arising in register allocation [59]. Chaitin points out that early work was fatally hampered by the relatively small memories available at the time [19].

The first complete global register allocator based on graph coloring was built by Chaitin and his colleagues at IBM [20]. Chaitin later described an improved coloring heuristic that handled the problems of coloring and spilling in a natural and coordinated fashion [18].

There has been a fair amount of work building on the foundation provided by Chaitin. We have reported an improvement to Chaitin's coloring heuristic [12]. Other improvements were introduced by Bernstein *et al.* [9]. Extensions to enable allocation of register pairs have been discovered by Nickerson and as part of our own work [57, 13]. Recently, we have described a technique for improving the accuracy of spill code estimation and placement [14].

An alternative form of global register allocation via graph coloring is described by Chow and Hennessy [22, 25, 26]. Their work, while based on coloring, differs in many respects from the work of Chaitin and his successors. They introduce the concept of *live range splitting* as an alternative to the spilling techniques originally used by Chaitin *et al.* The idea of live range splitting was independently discovered by Fabri in connection with her work on memory allocation [34]. Extensions and improvements have been reported by Larus and Hilfinger, Chow, and by Gupta, Soffa, and Steele [51, 23, 41].

A recent paper by Callahan and Koblenz describes a hierarchical approach to global register allocation via coloring [16]. Their approach decomposes the control-flow graph into a tree of *tiles,* colors each tile individually, and merges the results in two passes over the tree. It represents an attempt to gain the precision of Chaitin's approach to allocation together with a structured approach to live range splitting.

In this thesis, we have avoided extensive comparisons with the work of Callahan and Koblenz. This is not because we are unaware of their work or because we do not appreciate its value; rather it is because their work was done largely in parallel with ours – we have little perspective. As the community gains experience with their work and ours, we expect to be better able to understand how they compare.

Chapter 3

Improved Coloring and Spilling

At the heart of a graph coloring allocator is the algorithm used for coloring. Since the problem of finding a k-coloring is NP-complete, the coloring algorithm must employ a heuristic. One of the many strengths of the Yorktown allocator is Chaitin's coloring heuristic, which attacks the problems of coloring and spill choice in an integrated fashion. However, since it is a *heuristic* approach to an NP-complete problem, we are not surprised to find examples where its performance is not optimal.

In the next section, we present two examples where Chaitin's coloring heuristic is not optimal. The examples inspired a variation to Chaitin's heuristic, reported in Section 3.2, which offers significant improvements. The final two sections describe two further variations enabled by our new heuristic.

3.1 Problems

As a part of the $\mathbb{R}^n$ programming environment, we built an optimizing FORTRAN compiler [27, 17]. When the project was begun (*many* years ago), Chaitin's approach was new and elegant, so we decided to use it in our new compiler. While discussing details of the allocator, Ken Kennedy constructed a small example showing how Chaitin's heuristic could be forced to spill when a k-coloring was actually possible.

Later, when our allocator was working, we discovered a second interesting example – this time resulting from real code. In this case, spill code would always be required; however, Chaitin's heuristic obviously forced more spills than necessary.[7]

The two examples, one small and one large, demonstrate a single weakness in Chaitin's heuristic. The next two subsections present and explain both examples; Section 3.2 shows how to overcome the problem.

[7]The fact that the extra spills were obvious made it possible to detect that there even was a problem. Usually the sheer bulk of assembly code makes it difficult to detect such mistakes. After all, the code *is correct*; it simply runs a little slower than it might.

3.1.1 The Smallest Example

Suppose we want to find a 2-coloring for the graph shown in Figure 3.1. Clearly one exists; for example, x and y could be colored *red* and w and z could be colored *green*. If we apply Chaitin's heuristic though, *simplify* is immediately forced to spill – there are no nodes with degree < 2. If we assume for the example that all spill costs are equal, then some arbitrary node can be chosen for spilling; for example x. After x is removed from the graph and marked for spilling, the remaining nodes are removed by *simplify*.

This example illustrates that Chaitin's heuristic does not always find a k-coloring, even when one exists. Of course, we are not surprised, since the problem is NP-complete. The small size of the example *is* perhaps surprising. Of course, we might wonder how often such examples arise in real code, given the relatively large register sets typically available on modern processors.

3.1.2 A Large Example

In the process of isolating a bug elsewhere in the compiler, we carefully examined the code generated for a routine named SVD from the software distributed with Forsythe, Malcolm, and Moler's book on numerical methods [36]. The routine implements the singular value decomposition of Golub and Reinsch. It has 214 non-comment lines of code and 37 DO-loops organized into five different loop nests. The first loop nest is a simple array copy, shown at the top of Figure 3.2.

Close examination of the code generated for SVD revealed that the allocator was spilling a large number of short live ranges in deference to the larger, longer live

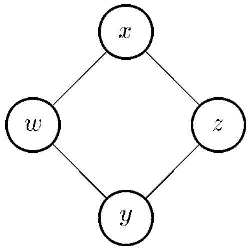


Figure 3.1 A Simple Graph Requiring Two Colors

```
subroutine SVD(M, N, ...)
do I = 1, N
  do J = 1, M
    A(I, J) = B(I, J)
  enddo
enddo
do
  many deeply-nested loops
enddo
do
  ...
enddo
do
  ...
enddo
do
  ...
enddo
```

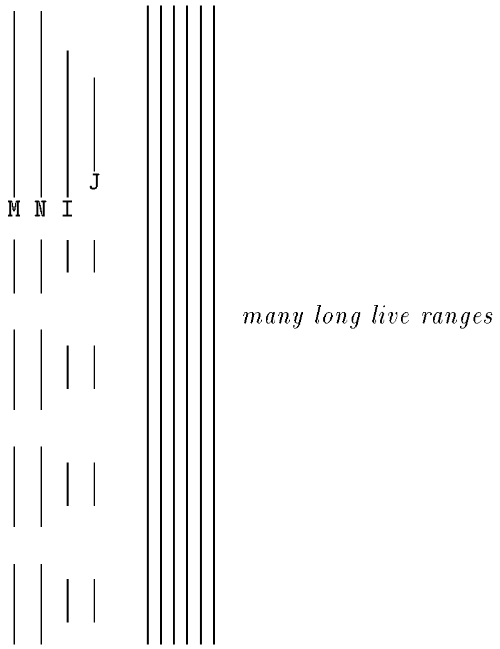


Figure 3.2 The Structure of SVD

ranges. The loop indices and limits of the array-copy loop were spilled despite the fact that there were several unused registers at that point in the code. After some study, we were able to understand why the register allocator over-spilled so badly and what situations provoked this behavior.

After optimization, about a dozen long live ranges (parameters specifying arrays and their sizes) extend from the initialization portion, through the array copy, and into the large loop nests. During coloring, these live ranges restrict the graph so much that some registers must be spilled. The estimated spill costs for `I`, `J`, `M`, and `N` (the indices and limits on the array-copy loops) are smaller than those for the longer live ranges – quite properly, since `I`, `J`, `M`, and `N` are only used over a small range that is less deeply nested than the rest of the routine. Unfortunately, spilling `I`, `J`, `M`, and `N` does not lower register pressure in the large loop nests and more live ranges must be spilled. Eventually, most of the longer live ranges have been spilled and coloring proceeds. The final result: the code has almost no register utilization during the array copy.

3.2 An Improvement

The two examples highlight different problems:

1. The allocator fails to find a two-coloring for the simple diamond graph. By inspection, we can see that the graph is two-colorable. The problem here is fundamental: the allocator uses too weak an approximation to decide whether or not x will get a color.

   In looking for a k-coloring, the allocator approximates "x gets a color" by "$x^\circ < k$." This is a sufficient condition for x to get a color but by no means a necessary condition. For example, x may have k neighbors, but two of those neighbors may get assigned the same color. This is precisely what happens in the diamond graph.

2. In SVD, the allocator must spill some live ranges. The heuristic for picking a spill candidate selects the small live ranges used in shallow loop nests because they *are* less expensive to spill. Unfortunately, spilling them is not productive – it does not alleviate register pressure in the major loop nests.

   When the spill decisions are made, the allocator cannot recognize that the spills do not help.[8] Similarly, the allocator has no way to retract the decisions. Thus, these live ranges get spilled and stay spilled.

While discussing the simple diamond graph, Kennedy pointed out that the coloring heuristic proposed by Matula and Beck will find a two-coloring of the diamond graph [54]. Their algorithm differs only slightly from Chaitin's approach. To simplify the graph, they repeatedly remove the node of smallest current degree, versus Chaitin's approach of removing *any* node n where $n^\circ < k$. After all nodes have been removed, they select colors in the reverse of the order of deletion, in the same fashion as Chaitin.

Applied to the diamond graph, this heuristic generates a two-coloring. Chaitin's heuristic fails because it pessimistically assumes that all the neighbors of a node will get different colors. Using Matula and Beck's heuristic, we have the opportunity to discover when some of the neighbors of a node n receive the same color, leaving a spare color for n itself.

Unfortunately, this scheme simply finds *a* coloring; there is no notion of finding a k-coloring for some k, and therefore no mechanism for producing spill code. For a register allocator, this is a serious problem. Many procedures require spill code – their interference graphs are simply not k-colorable.

[8] Rather, it cannot without expensive lookahead.

Thus, what is needed is an algorithm that combines Matula and Beck's stronger coloring heuristic with Chaitin's mechanism for cost-guided spill selection. To achieve this effect, we made two modifications to Chaitin's original algorithm:

1. *Simplify* removes nodes with degree $< k$ in an arbitrary order. Whenever it discovers that all remaining nodes have degree $\geq k$, it chooses a spill *candidate*. That node is removed from the graph; but instead of marking it for spilling, *simplify* optimistically pushes it on the stack, hoping a color will be available in spite of its high degree. Thus, nodes are removed in the same order as Chaitin's heuristic, but spill candidates are included on the stack for coloring.

2. *Select* may discover that it has no color available for some node. In that case, it leaves the node uncolored and continues with the next node. Note that any uncolored node would also have been spilled by Chaitin's allocator.

   If all nodes receive colors, the allocation has succeeded. If any nodes are uncolored, the allocator inserts spill code for the corresponding live ranges, rebuilds the interference graph, and tries again.

The resulting allocator is shown in Figure 3.3. Spill decisions are now made by *select* rather than *simplify*. The rest of the allocator is unchanged from Chaitin's scheme. In this form, the allocator can address both of our example problems.

Deferring the spill decision has two powerful consequences. First, it eliminates some non-productive spills. In Chaitin's scheme, spill decisions are made during *simplify*, before any nodes are assigned colors. When it selects a node to spill, the corresponding live range *is* spilled. In our scheme, these nodes are placed on the stack as spill candidates. Only when *select* discovers that no color is available is the live range actually spilled. This mechanism, in effect, allows the allocator to reconsider spill decisions.

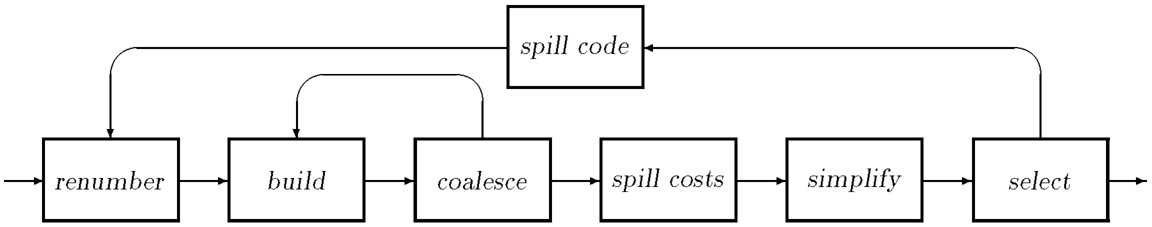


Figure 3.3 The Optimistic Allocator

Second, late spilling capitalizes on specific details of the color assignment to provide a stronger coloring heuristic. In selecting a color for node x, it examines the colors of all x's current neighbors. This provides a direct measure of "does x get a color?" rather than estimating the answer with "is $x^\circ < k$?" If two or more of x's neighbors receive the same color, then x may receive a color even though $x^\circ \geq k$.[9] On the diamond graph, this effect allows the allocator to generate a two-coloring.

Recall SVD. The live ranges for `I`, `J`, `M`, and `N` are early spill candidates because their spill costs are small. However, spilling them doesn't alleviate register pressure inside the major loop nests. Thus, the allocator must spill some of the large live ranges; this happens after the small live ranges have been selected as spill candidates and placed on the stack. By the time the small live ranges come off the stack in *select*, some of these large live ranges have been spilled. The allocator can easily determine that colors are available for these small live ranges in the early array-copy loops; it simply looks at the colors used by their neighbors.

Optimistic coloring is a simple improvement to Chaitin's pessimistic scheme. Assume that we have two allocators, one optimistic and one pessimistic, and that both use the same spill choice metric – for example, Chaitin's metric of $\frac{cost}{degree}$. The optimistic allocator has a stronger coloring heuristic, in the following sense: it will color any graph that the pessimistic allocator does and it will color some graphs that the pessimistic allocator will not. When spilling is necessary, both allocators will spill the same set of live ranges, except when optimistic coloring helps. In those cases, our allocator will spill a proper subset of the live ranges spilled by Chaitin's allocator.

Note that the comparisons between the optimistic heuristic and Chaitin's heuristic are predicated on both versions of *simplify* removing the same nodes in a given situation. This won't necessarily happen, but the assumption is necessary for comparison. For our experimental comparisons (see Chapter 7), we have been careful to implement both versions so they remove nodes in the same order. [10]

[9] Early versions of priority-based coloring considered only degree when assigning colors, despite having a single-pass algorithm where the actual colorings are available [22, 25]. Later descriptions correct this mistake [26].

[10] At least, the order will be identical on the first trip through the *build-color-spill* loop. Later iterations will present different graphs for coloring.

Results

Optimistic coloring helps generate better allocations. In a few cases, this eliminates all spilling; the diamond graph is one such example. In many cases, the total spill cost for the procedure is reduced.

In Section 7.1, we report results of a study comparing our optimistic allocator with our implementation of the Yorktown allocator. In a test suite of 70 FORTRAN routines, we observed improvements in 27 cases and a single loss (an extra load and store were required). Improvements ranged from tiny to quite large, sometimes reducing spill costs by over 40%.

The single loss was disappointing, since we have claimed that the optimistic coloring heuristic can never spill more than Chaitin's heuristic. However, we must recall the structure of the allocator. After each attempt to color, spill code is inserted and the entire *built-coalesce-color* process is repeated. The optimistic coloring heuristic will perform as well as Chaitin's heuristic on any graph; but after spilling, the two allocators will be facing different problems.

At least one independent confirmation of our results exists. Addition of the optimistic coloring heuristic to the back-end of the IBM XL compiler family for the RS/6000 machines resulted in a decrease of about 20% in estimated spill costs over the SPEC benchmark suite [44].

The optimistic heuristic is superior theoretically since it can never spill more than Chaitin's heuristic on a given graph. It is no more complex asymptotically than Chaitin's heuristic. Furthermore, it is no more difficult to implement than Chaitin's heuristic. Finally, our experimental results show that the improvement is significant on a large number of routines.

3.3 Limited Backtracking

Once we have the optimistic coloring heuristic, another refinement is possible. Recall *select*. Given a stack of nodes created by *simplify*, each node n is removed from the stack and added to the graph. After adding n to the graph, *select* first examines n's neighbors, noting which colors have been used, then chooses a different color for n. If no color remains from the k-palette, then n is left uncolored and will be spilled.

As an alternative to simply leaving n uncolored, we can sometimes attempt a limited form of backtracking, re-coloring a neighbor of n and thus freeing a color for

n. We note that such backtracking must be carefully constrained to avoid the chance of combinatorial explosion.

By limiting backtracking to a single level, we can maintain our linear time-bound for coloring. While examining n's neighbors, we can accumulate the number of uses of each color (rather than simply noting when each color is used) and note which neighbor uses each color. If no colors remain for n, we check for colors that have only been used by a single neighbor. If m is the only neighbor of n using a color c, the we try re-coloring m. If successful, we can then give n the color c.

Another possibility is trying to re-color several neighbors that all use the same color c. This seems to have less potential. For instance, it would be annoying to successfully re-color three neighbors only to have the fourth block the use of c.

Results

The advantages of limited backtracking are its low cost, easy implementation, and the fact that it rarely loses.[11] On the other hand, the results have been disappointing; limited backtracking almost never pays off.

In Section 7.1, we compare an allocator with limited backtracking to the optimistic allocator. In (only) three routines out of seventy, limited backtracking offers a slight improvement. In the best case, there is a 1.2% reduction in spill cost.

Why so little improvement? When a node is selected as a spill candidate, it is selected from a graph where every node in the graph has at least k neighbors (otherwise, *simplify* would have continued removing them before being forced to choose a spill candidate). Therefore, when *select* is unable to color a node n (where n is always a spill candidate), n has at least k neighbors and those neighbors have at least k neighbors. Therefore, any neighbor of n is relatively constrained and there is only a small chance that limited backtracking will be able to free a color.

3.4 Alternative Spill-Choice Metrics

In Section 2.2.6, we introduced the "best of 3" heuristic originally suggested by Bernstein *et al.* [9]. In essence, they run *simplify* three times, each time using a different spill choice metric, and choose the ordering that gives the lowest spill cost.

---

[11] We have never seen it lose, though such situations are conceivable. Saving an expensive spill now versus possibly saving less-expensive spills in the future is usually a safe bet.

This idea extends naturally to our optimistic coloring heuristic. We simply run the combination of *simplify* and *select* three times, being careful with our accounting, and choose the best result.

The three specific metrics used by Bernstein *et al.* (Equations 2.2 through 2.4) are not important; we can invent many similar metrics, perhaps using more extensive combinations to give a "best of 10" approach. Of course, there is a tradeoff of increased compile-time against the diminishing returns offered by such an attack.

An attractive idea is to experiment with different spill cost metrics, attempting to take advantage of the optimistic coloring heuristic. Rather than trying to remove nodes with high degree (hoping to greatly simplify the graph), we can try removing nodes of low degree. The hope is that a node of low degree (but still greater than k) will be more likely to color since only a few of its neighbors need to overlap (or spill) to create space in the k-palette. There seem to be several possibilities:

- Search for a node n such that $n^\circ < k + t$, where t is some small constant. If many such nodes are found, choose the one with lowest spill cost. If no such nodes are found, use one of the traditional spill metrics.
- Search for a node n that minimizes the product $cost_n \times degree_n$.
- Search for a node that minimizes

$$\frac{cost_n \, degree_n}{area_n}$$

We have performed a few limited experiments with these ideas; however, the results have been unimpressive. Why? Each time *simplify* must choose a spill candidate, any one of the spill metrics might indicate the best choice. For an entire sequence of spill choices (that is, for an entire simplification), it is unlikely that any one spill metric would be best for every choice. Each run of *simplify* in a "best of 3" or "best of 10" sequence is therefore a compromise – a series of acceptable choices that work out reasonably well together.

Additionally, there is a problem of diminishing returns. By adding a "best of 2" choice, we expect some amount of improvement. With "best of 3", we expect a further (though smaller) improvement. As we continue, the rate of improvement will continue to slow. Furthermore, the improvements will appear on fewer and fewer routines. Nevertheless, there seems to be some possibility of improvement in this area for those patient enough to explore it thoroughly.

3.5 Summary

The primary contribution of Chaitin's second allocation paper was a coloring heuristic able to make spill decisions based upon the structure of the interference graph. In this chapter, we have presented an improvement to Chaitin's heuristic. The improved heuristic is able to color more graphs with no spilling and able to color many other graphs with less spilling. The key difference is that the new heuristic optimistically attempts to color, even when faced with apparently complex graphs. We also presented two additional improvements made possible by the same insight that motivates our optimistic coloring heuristic.

The optimistic heuristic is valuable. We have reported experimental results showing that the optimistic heuristic pays off in many real routines and can reduce spill costs by up to 40%. In contrast, limited backtracking and the alternative spill-choice metrics were less valuable.

Chapter 4

Coloring Register Pairs

Register pairs are a fundamental part of many machine architectures. For example, many machines use pairs of single-precision registers to hold double-precision values. The two most common constraints imposed on register pairs are requiring that the registers be adjacent (named with consecutive integers) and that an adjacent pair be aligned (typically requiring the first register to have an even number).

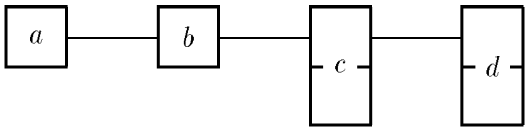
Previous descriptions of this problem in the literature have been rather limited. In 1986, Hopkins described a method to handle the register pair constraints that arise in the shift instructions on the ROMP microprocessor – the engine in RT/PC workstations [43]. In 1990, Nickerson published a method for allocating structures into an aggregate set of adjacent registers. He observed that an allocator based on our 1989 paper (see Chapter 3) produced good allocations under his scheme [57, 12].

This chapter describes work performed independently of Nickerson and completed at approximately the same time. We consider various ways to represent register pairs in the interference graph. We show why Chaitin's coloring heuristic over-estimates demand for registers and how the heuristic introduced in Chapter 3 naturally avoids this problem. Finally, we provide a simple rationale for deciding how many edges are required to correctly represent an interference between a single register and an aggregate set of registers.

4.1 Why Are Pairs Hard to Color?

In Chapter 3, we showed examples where Chaitin's coloring heuristic over-spilled. When the program includes operations requiring pairs of adjacent registers, this over-spilling is exaggerated. The reason is simple – introducing pairs of registers requires modifying the interference graph or changing the way the allocator interprets it. In Chaitin's scheme, such modification distorts the allocation – the allocator consistently overestimates register demand. This causes it to spill values in many cases where registers are available to hold them.

To help in the discussion, we use the following simple example to illustrate our points. Imagine four live ranges, a, b, c, and d; where a and b are single-precision values and c and d are double-precision values. Assume that b interferes with a and c and that c also interferes with d. The interference graph for this example looks like:



We have drawn the nodes for c and d larger than those for a and b as a reminder that they require pairs; this reflects a fundamental fact that the allocator must handle. Such a graph can be viewed as a *weighted* graph; each node has an integer weight associated with it. In our problem, the weights correspond to the number of colors (or registers) needed for each node.

From a graph coloring perspective, a weighted graph is fundamentally harder to color than an unweighted graph. For example, consider the interference graphs that result from straight-line code – they are *interval* graphs. A minimal coloring for an unweighted interval graph can be found in linear time [40, page 14]. In contrast, the problem of finding a minimal coloring for a weighted interval graph (which is identical to the *shipbuilding problem*) has been shown to be NP-complete even when the weights are constrained to be either 1 or 2 [40, page 204]. This is exactly the situation arising when allocating register pairs. In any case, a global register allocator must be prepared to handle procedures having more complex control flow.[12]

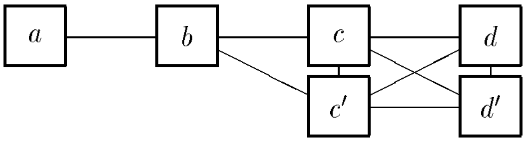
Fabri explored variations of this problem in the context of packing arrays in memory [34, 35]. While the work is interesting, it is not directly applicable to our problem. For example, her problem has no analog for the problems of spill choice and spill placement. Thus, for the purposes of register allocation, we have continued to work with allocators styled after Chaitin's work.

4.1.1 Unconstrained Pairs

Initially, consider the simplest case. Assume that the target machine places no adjacency or alignment restrictions on pairs. In this case, the graph shown previously

[12]Recall that Chaitin *et al.* showed that any arbitrary interference graph can be generated by some procedure.

overconstrains the coloring; instead, we can simply handle the two halves of each register pair separately. This yields the following graph:

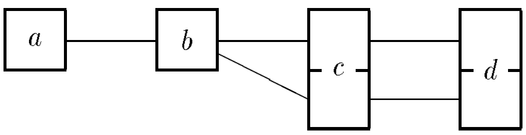


This simple graph suffices because the machine places no restrictions on a register pair beyond the obvious requirement that the two halves of the pair occupy different registers (the edges from c to c' and from d to d' embody this constraint). Because the simple constraints can be encoded directly into the interference graph without any additional interpretation, this graph can be colored directly with Chaitin's algorithm. It may over-spill, but the over-spilling will be limited to the kind found in programs containing no register pairs. Chaitin's method always finds a four-coloring for the example graph.

4.1.2 Adjacent Pairs

Extensions to handle adjacent register pairs correctly are more difficult. For the moment, assume that the target machine requires that pairs be allocated to aligned adjacent registers. The problem arises during color selection; the allocator must coordinate the colors for two nodes that appear unrelated. If it assigns a color to one and cannot assign an adjacent color to the other node, it must either reconsider the colors that it has already assigned or report complete failure. Neither of these is a good alternative. For *select* to reconsider colors requires backtracking, which can require exponential time. Reporting failure seems unhelpful; it provides no clear direction for recovery.

Changing our representation for pairs appears to be the best alternative. We should consider treating the pairs as indivisible units and assigning the pair two colors in *select*. This gives us the following graph (more accurately *multigraph*):



It resembles our original graph from Section 4.1, with additional edges to represent necessary interferences. The simplicity of this representation is appealing. Intuitively, the extra edge between b and c reflects the additional constraint placed on b. Similarly, the extra edge between c and d balances their extra width.

Multigraph Representation

So, why have we moved to a multigraph representation, besides the intuitive appeal of the pictures? For stronger justification, we must consider the role edges play in the coloring process. First, edges represent interferences – they are critical to the correctness of the resulting allocation. Second, they trigger spilling in *simplify*.

Recall that *simplify* examines the graph and repeatedly removes nodes with fewer than k neighbors, where k is the number of available colors. A node having fewer than k neighbors always receives a color independent of context. If, during the simplification process, a node always has k or more neighbors, *simplify* marks it for spilling. The number of neighbors is the node's *degree*. Thus, for the allocator to work correctly, a node's degree should accurately reflect its colorability. For register pairs, we must add enough edges to ensure proper behavior. Too few edges lead to a situation where *simplify* fails to reserve enough registers; too many edges leads to excessive spilling.

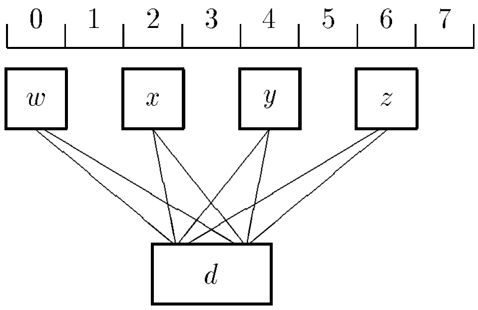
The graph shown above correctly models the colorability of each of node. Any interference that involves a value stored in a pair of registers adds two edges to the graph. Thus, the interference between b and c creates a pair of edges, as does the interference between c and d.

This rule makes sense. On a four-register machine, two single registers that interfere with a register pair raise the pair's degree to four. Placed correctly, the singles could block allocation of registers to the pair. Similarly, three register pairs that all mutually interfere create a situation where all three have a degree of four. This reflects the fact that three register pairs cannot fit into four registers.

Sometimes it is convenient to introduce an interference between a single register and one half of a pair. Often, one half of a register pair may be used as the source or destination of an operation. For example, the real half of a complex pair might be copied to another register. In this case, the target of the copy should interfere only with the imaginary half of the complex pair. An interference between the target register and the real half of the pair would prevent *coalesce* from combining them and eliminating the copy.

A Problem

Unfortunately, Chaitin's coloring heuristic performs poorly on graphs of this form. The graph is more constrained because of the pairs of edges between pairs and their neighbors; this triggers earlier application of the spill mechanism. To elucidate this problem requires a somewhat more complex example. Consider a machine with eight single-precision floating-point registers that requires double-precision values to be allocated to adjacent pairs of registers. If we have a double-precision value, four single-precision values are sufficient to force a spill with Chaitin's allocator. The following picture shows why.



Assume that w, x, y, and z each have at least six other interferences. Faced with this situation, Chaitin's algorithm invokes the spill metric to choose a value for spilling. It selects the node that minimizes $\frac{cost}{degree}$ and spills it. Of course, many of the possible assignments would leave space for all five values; for example, placing w, x, y, and z in the first four registers leaves the last four to hold d. Unfortunately, the early decision on spilling prevents the allocator from finding such an allocation.

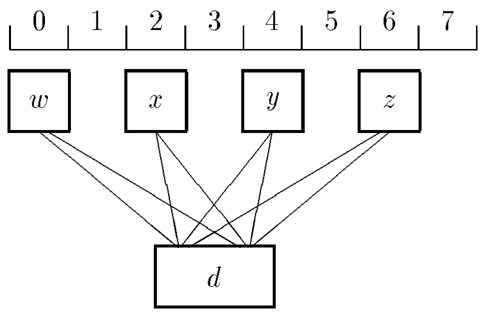
Why does this happen? There exist colorings of w, x, y, and z, like the one suggested above, that preclude d's allocation. The "extra" edges in the interference graph, the second edge from d to each of the other nodes, account for this possibility. In Chaitin's scheme, *simplify* constructs an order in which *select* is guaranteed to succeed. Such an order does not exist for the example, so *simplify* spills one of the values. Thus, in any region where there is strong competition for registers and a mixture of single registers and register pairs, the allocator will consistently overestimate the demand for registers and spill values for which registers are available.

4.2 The Optimistic Coloring Heuristic

Originally, we thought that this problem was endemic to all coloring allocators. Fortunately, Randy Scarborough asked us a question that caused us to consider the problem of adjacent pairs in the context of our optimistic coloring heuristic.

The optimistic allocator behaves differently than the pessimistic allocator with respect to spilling adjacent pairs. Because it defers spill decisions into *select*, it only spills a node when it discovers that it cannot color the node. With adjacent pairs, the behavior is the same; it only spills a pair when it discovers that no adjacent pair is available.

This change eliminates the over-spilling that arises with Chaitin's heuristic. To see this more clearly, reconsider the graph that caused problems in Section 4.1.2.

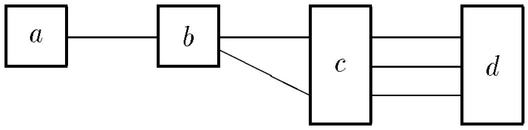


The pessimistic allocator would spill because the single-precision values w, x, y, and z might be assigned to registers in a way that precludes successful coloring of d. The optimistic allocator would simply push a node, say d, on the stack, because the actual coloring of w, x, y, and z may leave a pair available for d. This can happen in many ways; for example, w and x might be assigned the same color, y might be spilled, or y and z might be assigned to consecutive registers. In fact, the *only* way that d could be blocked is by an even spacing of the sort suggested by the figure.

The optimistic allocator often succeeds on graphs where the pessimistic allocator fails. *Simplify* determines an order for assigning colors; it treats single nodes and pairs identically. The difference between them is encoded in the number of edges. *Select* makes the actual spill decisions; it spills a node only after discovering that it cannot find the needed color(s).

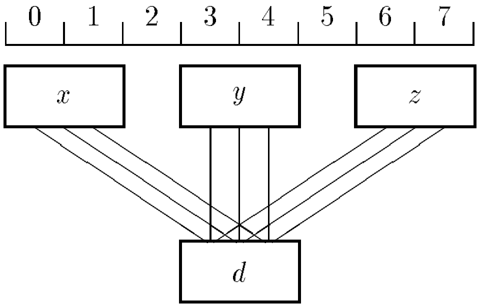
4.3 Unaligned Pairs

A few architectures allow the use of unaligned adjacent pairs of registers. The interference graph required for this situation is slightly more complex than for the aligned case. For our continuing example, the following interference graph captures all of the needed properties:

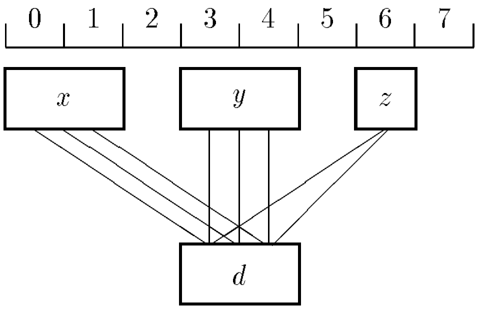


The extra edge (between c and d) is required to correctly trigger the selection of a spill candidate in *simplify*. The next two graphs help show why this third edge is necessary between unaligned pairs.

Note that three pairs (x, y, and z) can be colored so that there is no adjacent pair of colors for d. The fact that d has nine edges will trigger the spill heuristic in *simplify*, causing it to select a spill candidate. Of course, the candidate will not necessarily be spilled – this is the key difference between the optimistic and pessimistic approaches.



Even two pairs and a single may be placed so that d cannot be colored, as shown below. Again, d's eight edges are sufficient to warn that d may have to be spilled during *select*.



Finally, we note that requiring four edges between each interfering pair would be too conservative. This would suggest that only two pairs (say x and y) would suffice to preclude coloring d.

These examples show that the extra flexibility offered by eliminating alignment restrictions complicates the allocation process by enlarging the graph. Furthermore, because of the additional constraints that it adds, it can actually lead to worse allocations.

4.4 Using Pairs for Memory Access

In some cases, it is desirable to use adjacent registers for loads and stores. For example, complex numbers are often represented in storage as a pair of adjacent single-precision floating-point numbers. On some machines, it is advantageous to load these values into an adjacent register pair using a double-precision floating-point load. Unfortunately, tying all subsequent uses of the component parts of the complex number to the adjacent registers restricts the allocator's freedom. Our compiler handles this issue by carefully shaping the code before the allocator sees it. It generates a double-precision load into an adjacent pair of virtual registers, and then immediately copies the component values into single registers.[13] This allows the allocator to keep the values in adjacent registers at points where they are loaded and stored, while offering it the chance to keep them in non-adjacent registers during the rest of their lifetimes. It decides between these choices during coalescing, based on the structure of the interference graph. The allocator is the proper place to make this decision – it relies on information that cannot be made available earlier in compilation.

These ideas may become more important in the future. Architects can make more memory bandwidth accessible through the use of wider load and store instructions. For example, on Intel's i860 XP, the quad-word, floating-point load `fld.q` loads four registers at a time, allowing a program to move twice as much data as the double-word version `fld.d` and four times as much as the single-word version `fld.l` [45]. Naturally, any appreciable use of this feature ties down a large number of registers and allocating them carefully becomes important.

[13] These extra copies are only inserted when loading and storing complex numbers. Ordinary double-precision values are managed with no extra copies, since extra copies would provide no additional flexibility.

Previous work by Nickerson focuses on allocating aggregate data structures to adjacent sets of registers [57]. Nickerson assumes that the data items *must* remain adjacent throughout their lifetimes and that the components of the aggregate can have different lifetimes. In our work, we handle cases where the components of an aggregate have different lifetimes by enforcing adjacency only at those *instructions* where it is required. Pairs are used at instructions that require them; when a component is used for some different lifetime, we separate out that non-adjacent use. This can be accomplished by copying the component into another register; this lets *coalesce*, *simplify*, and *select* determine whether or not to preserve adjacency. Note that this is only important for values having components with *different* lifetimes.

4.5 Summary

This chapter examines the problem of dealing with register pairs in a graph coloring register allocator. To work with register pairs, the allocator needs a way to represent them in the interference graph; this entails either changing the interference graph or its interpretation. We have shown how to represent different sets of constraints: unconstrained pairs, adjacent pairs, and unaligned adjacent pairs. The key issue is determining the number of edges to add between an aggregate node and each of its neighbors. Our scheme extends in a straightforward way to larger aggregate register groupings.

Unfortunately, when presented with a multigraph, Chaitin's allocator consistently over-estimates the demand for registers. This results in allocations that underuse the register set, spilling values even when registers are available to hold them. An optimistic allocator avoids such over-spilling. This allows the compiler-designer to use a simple representation for adjacent register pairs without provoking underuse of the register set. The previous chapter has shown that the optimistic coloring heuristic produces better allocations than Chaitin's pessimistic heuristic; this chapter shows that optimism also improves the treatment of register pairs.

Chapter 5

Rematerialization

This chapter examines a specific problem that arises in global register allocation – *rematerialization*. When a value must be spilled, the allocator should recognize those cases when it is cheaper to recompute the value than to store and retrieve it from memory. While our discussion is set in the context of the Yorktown allocator, the same questions arise in all global allocators.

The next section introduces the problem and suggests why it is important. We give a high-level view of our approach in Section 5.2 and describe the necessary low-level modifications to the allocator in Section 5.3. Results are discussed in Section 5.4.

5.1 Introduction

Chaitin *et al.* discuss several ideas for improving the quality of spill code [20]. They point out that certain values can be recomputed in a single instruction and that the required operands will always be available for the recomputation. They call these exceptional values *never-killed* and note that such values should be recalculated instead of being spilled and reloaded. They further note that an uncoalesced copy of a never-killed value can be eliminated by recomputing it directly into the desired register. Together, these techniques are called *rematerialization*. Many opportunities for rematerialization arise in practice, including:

- immediate loads of integer constants and floating-point constants,
- computing a constant offset from the frame pointer or the static data pointer,
- loads from a constant location in the stack frame or the static data area, and
- loading non-local frame pointers from a *display* [4, Section 7.4].

The values must be cheaply computable from operands that are available throughout the procedure.

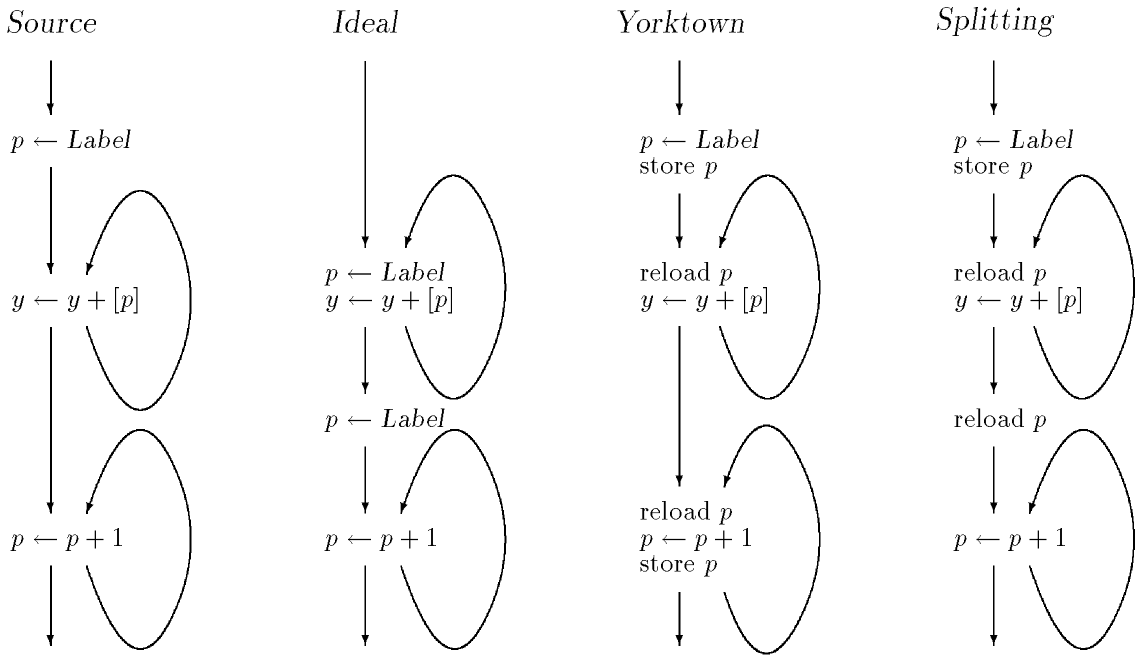


Figure 5.1 Rematerialization versus Spilling

Consider the code fragments shown in Figure 5.1.[14] Examining the *Source* column, we note that p is constant in the upper loop, but varying in the lower loop. The register allocator should take advantage of this situation.

Imagine that high demand for registers in the upper loop forces p to be spilled; the *Ideal* column shows the desired result. In the upper loop, p is loaded just before it is needed (using some sort of "load-immediate" instruction). For the lower loop, p is loaded just before the loop, again using a load-immediate.

The third column illustrates the code that would be produced by the Yorktown allocator. The entire live range of p has been spilled to memory, with loads inserted before the uses and stores inserted after the definitions.

The final column shows code we would expect from a "splitting" allocator [26, 51, 41, 16]; the actual code might be worse.[15] Unfortunately, examples of this sort are not discussed in the literature on splitting allocators and it is unclear how best to extend these techniques to achieve the *Ideal* solution.

[14]The notation $[p]$ means "the contents of the memory location addressed by p."

[15]In fact, our work on rematerialization was motivated by problems observed during our experiments with splitting.

5.2 Rematerialization

It is important to understand the distinction between *live ranges* and *values.* A live range may comprise several values connected by common uses. In the *Source* column of Figure 5.1, p denotes a single live range composed from three values: the address *Label,* the result of the expression $p+1$, and (more subtly) the merge of those two values at the head of the second loop.

The Yorktown allocator correctly handles rematerialization when spilling live ranges with a single value, but cannot handle more complex cases; *e.g.*, the variable p in Figure 5.1. Our task is to extend the Yorktown allocator to take advantage of rematerialization opportunities for complex, multi-valued live ranges. Our approach is to tag each value with enough information to allow the allocator to handle it correctly. To achieve this, we

1. split each live range into its component values,
2. propagate rematerialization tags to each value, and
3. form new live ranges from connected values having identical tags.

This approach allows correct handling of rematerialization, but introduces the new problem of minimizing unnecessary splits. The following sections describe how to find values, how to propagate tags, how to split the live ranges, and how to remove unproductive splits.

5.2.1 Discovering Values

To find values, we construct the procedure's *static single assignment* (SSA) graph, a representation that transforms the code so that each use of a value references exactly one definition [29]. To achieve this goal, the construction technique inserts special definitions called ϕ-nodes at those points where control-flow paths join and different values merge. We actually use the *pruned* SSA, with dead ϕ-nodes (ϕ-nodes with no uses) eliminated [21].

A natural way to view the SSA graph for a procedure is as a collection of values, each composed of a single definition and one or more uses. Each value's definition is either a single instruction or a ϕ-node that merges two or more values. By examining the defining instruction for each value, we can recognize never-killed values and propagate this information throughout the SSA graph.

5.2.2 Propagating Rematerialization Information

To propagate tags, we use an analog of Wegman and Zadeck's *sparse simple constant* algorithm [64].[16] We modify their lattice slightly to represent the necessary rematerialization information. The new lattice elements may have one of three types:

- $\top$ *Top* means that no information is known. A value defined by a copy instruction or a ϕ-node has an initial tag of $\top$.
- *inst* If a value is defined by an appropriate instruction (*never-killed*), it should be rematerialized. The value's tag is simply a pointer to the instruction.
- $\perp$ *Bottom* means that the value cannot be rematerialized. Any value defined by an "inappropriate" instruction is immediately tagged with $\perp$.

Additionally, their *meet* operation $\sqcap$ is modified in an analogous fashion. The new definition is:

$$\begin{array}{cccccl} \text{any} & \sqcap & \top & = & \text{any} & \\ \text{any} & \sqcap & \perp & = & \perp & \\ inst_i & \sqcap & inst_j & = & inst_i & \text{if } inst_i = inst_j \\ inst_i & \sqcap & inst_j & = & \perp & \text{if } inst_i \neq inst_j \end{array}$$

Note that $inst_i = inst_j$ compares the instructions on an operand-by-operand basis. Since our instructions have at most 2 operands, this modification does not affect the asymptotic complexity of propagation.

During propagation, each value will be tagged with an *inst* or $\perp$. Values defined by a copy instruction will have their tags *lowered* to *inst* or $\perp$, depending on the value that flows into the copy. Values defined by ϕ-nodes will be lowered to *inst* if and only if all the values flowing into the node have identical *inst* tags; otherwise, they are lowered to $\perp$.

This process tags each value is the SSA graph with either an instruction or $\perp$. If a value's tag is $\perp$, spilling that value requires a normal, heavyweight spill. If, however, its tag is an instruction, it can be rematerialized by inserting the instruction specified by the tag. The tags are used in two phases of the allocator: *spill costs* uses the tags to compute more accurate spill costs and *spill code* uses the tags to emit the desired code.

---

[16] The more powerful *sparse conditional constant* algorithm is unnecessary; by this point in the compilation, all constant conditionals have been folded.

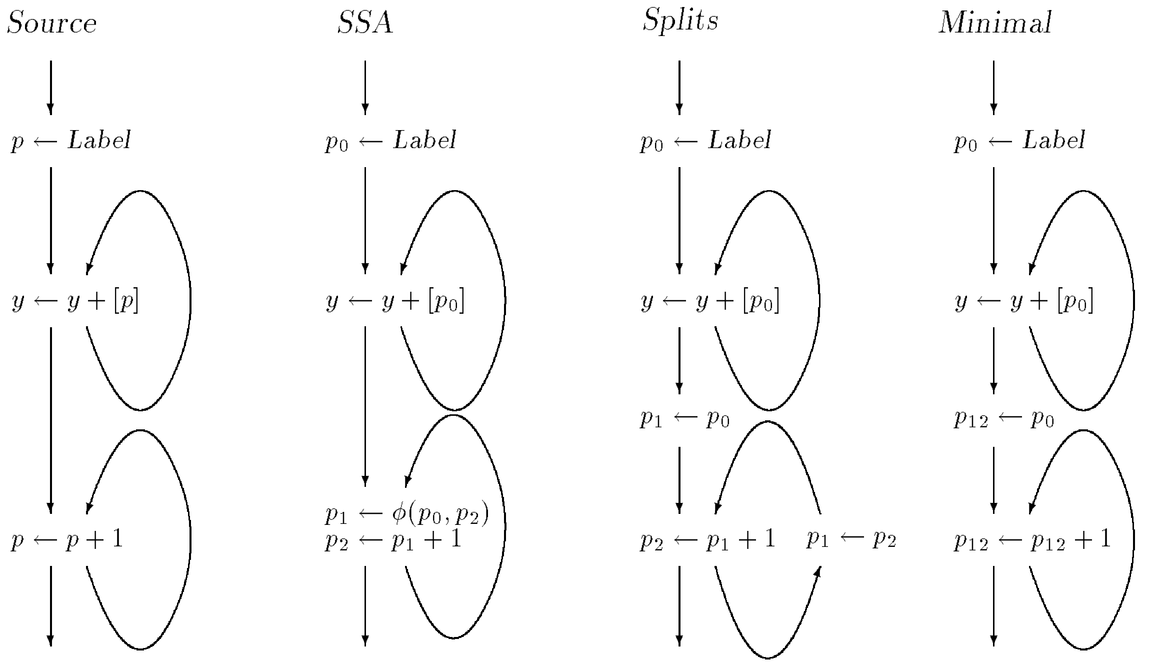


Figure 5.2 Introducing Splits

5.2.3 Inserting Splits

After propagation, the ϕ-nodes must be removed and values renamed to recreate an executable program. Consider the example in Figure 5.2. The *Source* column simply repeats the example introduced in Figure 5.1. The *SSA* column shows the effect of inserting a ϕ-node for p and renaming the different values comprising p's live range. The *Splits* column illustrates the copies necessary to distinguish the different values without ϕ-nodes. The final column (*Minimal*) shows the single copy required to isolate the never-killed value p_0 from the other values comprising p. We avoid the extra copy by noting that p_1 and p_2 have identical tags after propagation (both are $\perp$) and may be treated together as a single live range p_{12}. Similarly, two connected values with the same *inst* tag would be combined into a single live range.

For the purposes of rematerialization, the copies are placed perfectly – the never-killed value has been isolated and no further copies have been introduced. The algorithm for removing ϕ-nodes and inserting copies is described in Section 5.3.1. In Chapter 6, we discuss the possibility of including *all* the copies suggested in the *Splits* column.

5.2.4 Removing Unproductive Splits

Our approach inserts the minimal number of copies required to isolate the never-killed values. Nevertheless, coloring can make some of these copies superfluous. Recall the *Minimal* column in Figure 5.2. If neither p_0 nor p_{12} are spilled and they both receive the same color, the copy connecting them is unnecessary. Because it has a real run-time cost, the copy should be eliminated whenever possible. Of course, *coalesce* would remove *all* of the copies, losing the desired separation between values with different tags. So, we use a pair of limited coalescing mechanisms to remove unproductive copies:

Conservative coalescing is a straightforward modification of the standard *coalesce* phase. Conceptually, we add a single constraint to *coalesce* – only combine two live ranges if the resulting single live range will not be spilled.

Biased coloring increases the likelihood that live ranges connected by a copy get assigned to the same register. Conceptually, *select* tries to assign the same color to two live ranges connected by a copy instruction.

Taken together, these two mechanisms remove most of the unproductive copies.

5.3 Implementation

The Yorktown allocator can be extended naturally to accommodate our approach. The high-level structure depicted in Figure 3.3 is unchanged, but a number of low-level modifications are required. The next sections discuss the enhancements required in *renumber*, *coalesce*, and *select*.

5.3.1 Renumber

Chaitin's version of *renumber* (termed "getting the right number of names") was based on def-use chaining. Long before our interest in rematerialization, we adopted an implementation strategy for *renumber* based on the pruned SSA graph. The old implementation has four conceptual steps:

1. Determine liveness at each basic block using a sparse data-flow evaluation graph [21].

2. Insert ϕ-nodes based on dominance frontiers. Avoid inserting dead ϕ-nodes.

3. Renumber the operands in every instruction to refer to values instead of the original virtual registers. At the same time, accumulate availability information for each block. The intersection of *live* and *avail* is needed at each block to allow construction of a precise interference graph.

4. Form live ranges by unioning together all the values reaching each ϕ-node using a fast disjoint-set union. The disjoint-set structure is maintained while building the interference graph and coalescing (where coalesces are further union operations).

In our implementation, steps 3 and 4 are performed during a single walk over the dominator tree. Using these techniques, *renumber* completely avoids the use of *bit-vectored* data-flow analysis. Despite the apparent complexity of the algorithms involved, it is fast in practice and requires only a modest amount of space (see Section 8.4 for more details on the implementation of *renumber* as well as measurements and discussion of required compile time and space).

Because *renumber* already uses the SSA graph, only modest changes are required to support rematerialization. The modified *renumber* has six steps:

1. Determine liveness at each basic block using a sparse data-flow evaluation graph.

2. Insert ϕ-nodes based on dominance frontiers, still avoiding insertion of dead ϕ-nodes.

3. Renumber the operands in every instruction to refer to values. At the same time, initialize the rematerialization tags for all values.

4. Propagate rematerialization tags using the sparse simple constant algorithm as modified in Section 5.2.2.

5. Examine each copy instruction. If the source and destination values have identical *inst* tags, we can union them and remove the copy.

6. Examine the operands of each ϕ-node. If an operand value has the same tag as result value, union the values; otherwise, insert a *split* (a distinguished copy instruction) connecting the values in the corresponding predecessor block.[17]

Steps 5 and 6 are performed in a single walk over the dominator tree.

---

[17]During the initial construction of the control-flow graph, we insert extra basic blocks to ensure a unique predecessor wherever splits may be required.

5.3.2 Conservative Coalescing

To prevent coalescing from removing the splits that have been carefully introduced in *renumber*, we must limit its power. Specifically, it should never coalesce a split instruction if the live range that results may be spilled. In normal coalescing, two live ranges l_i and l_j are combined if l_j is defined by a copy from l_i and they do not otherwise interfere. In conservative coalescing, we add an additional constraint: combine two live ranges connected by a split if and only if l_{ij} has $< k$ neighbors of "significant degree," where significant degree means a degree $\geq k$.

To understand why this restriction is safe (indeed, conservative), recall Chaitin's coloring heuristic. Before any spilling, nodes of degree $< k$ are removed from the graph. When a node is removed, the degrees of its neighbors are reduced, perhaps allowing them to be removed. This process repeats until the graph is empty or all remaining nodes have degree $\geq k$. Therefore, for a node to be spilled, it must have at least k neighbors with degree $\geq k$ in the initial graph.

In practice, we perform two rounds of coalescing. Initially, all possible copies are coalesced (but not split instructions). The graph is rebuilt and coalescing is repeated until no more copies can be removed. Then, we begin conservatively coalescing split instructions. Again, we repeatedly build the interference graph and attempt further conservative coalescing until no more splits can be removed.

In theory, we should not intermix conservative coalescing with unrestricted coalescing, since the result of an unrestricted coalesce may be spilled. For example, l_i and l_j might be conservatively coalesced, only to have a later coalesce of l_{ij} with l_k provoke the spilling of l_{ijk} (since the significant degree of l_{ijk} may be quite high). In practice, this may not prove to be a problem, permitting a slight simplification of the entire process.

Conservative coalescing directly improves the allocation. Each coalesce removes an instruction from the resulting code – a split instruction that was introduced by the allocator. In regions where there is little competition for registers (a region of low register pressure), conservative coalescing undoes all splitting. It cannot, however, undo all of the non-productive splits by itself.

5.3.3 Biased Coloring

The second mechanism for removing useless splits involves changing the order in which colors are considered for assignment. Before coloring, the allocator finds *partners* – values connected by splits. When *select* assigns a color to l_i, it first tries colors already assigned to one of l_i's partners. With a careful implementation, this is no more expensive than picking the first available color; it really amounts to biasing the spectrum of colors by previous assignments to l_i's partners.

The biasing mechanism can combine live ranges that conservative coalescing cannot. For example, l_i might have $2k$ neighbors of significant degree; but these neighbors might not interfere with each other and thus might all be colored identically. Conservative coalescing cannot combine l_i with any of its partners; the resulting live range would have too many neighbors of significant degree. Biasing may be able to combine l_i and its partners because it is applied after the allocator has shown that both live ranges will receive colors. At that late point in allocation, combining them is a matter of choosing the right colors. By virtue of its late application, the biasing mechanism uses a detailed level of knowledge about the problem that is not available any earlier in the process – for example, when coalescing is performed.

Limited Lookahead and Backtracking

Of course, biased coloring will not always succeed in assigning adjacent partners to the same register. The vagaries of the coloring process ensure that cases will arise when l_i and l_j will be adjacent partners and be assigned to different registers. To help cope with these cases, we can add a final improvement to *select*.

When selecting a color for l_i, the allocator can try to select a color that is still available for each of its adjacent uncolored partners. This increases the likelihood that biased coloring will succeed. We call this technique *limited lookahead.*

Similarly, when selecting a color for l_i, the allocator may discover that none of the available colors matches its already colored adjacent partners. In this case, the allocator can try to change the colors assigned to those partners. We call this technique *limited backtracking.*

Notice that this form of backtracking differs from the style suggested in Section 3.3. In that case, we were attempting to avoid spills; in this case, we are attempting to remove splits. Unfortunately, neither form of backtracking cooperates well with biased coloring. Having carefully selected a color for a node, perhaps matching a partner's color, it would be disappointing if some sort of backtracking disturbed our artfully arranged coloring. While it is possible to account for the direct effects of recoloring, the extended case involves exponential exploration of the graph. Therefore, in the presence of biased coloring, we attempt no backtracking.

In practice, we try each heuristic in succession. First, we try to find a color matching a colored partner. If that fails, we try limited lookahead, seeking to avoid colors that uncolored partners cannot use. If all else fails, we take any free color.

5.4 Results

Our recognition and exploration of this problem was prompted by poor spilling observed in our experimental splitting allocator (see Chapter 6). However, Randy Scarborough pointed out that our approach was really orthogonal to the splitting question. Therefore, it seems natural to compare our new approach to the simpler scheme used in the Yorktown allocator and our optimistic allocator.

In Section 7.1, we present a comparison of the optimistic allocator with the enhanced allocator described here. In our test suite of 70 routines, we observed improvements in 28 cases and two cases of degradation. One loss was small (2 loads, 2 stores, and an extra copy); the other was somewhat larger. Improvements ranged from tiny (after all, some routines may offer no opportunities for rematerialization) to reasonably large (typically reducing spill costs by 10 to 20%). It is possible to see a pattern of trading loads for load-immediates: we often see a fairly large reduction in load instructions offset by an increased number of load-immediate instructions. Since loads are usually more expensive, we win in the balance.

Typically, the number of stores and copies is also reduced. The reduction in copy instructions suggests that our various heuristics for removing unhelpful splits are "good enough."

5.5 Summary

The primary contribution of this chapter is a natural extension of Chaitin's ideas on rematerialization. We show how to handle complex live ranges that may be completely or partially rematerialized. We describe a technique for tagging the component values of a live range with correct rematerialization information. We introduce heuristics, conservative coalescing and biased coloring, that are required for good results. Finally, we report experimental results showing the effectiveness of our extensions.

Our work extends the work described by Chaitin *et al.* and recalls an approach suggested by Cytron and Ferrante [28].

Chaitin et al. introduce the term *rematerialization* and discuss the problem briefly. Because their allocator cannot split live ranges, they handle only the simple case where all definitions contributing to a live range are identical. Our work is a direct extension and is able to handle each component of a complete live range separately and correctly.

Cytron and Ferrante suggest splitting based on (the equivalent of) the SSA. Their goal is minimal coloring in polynomial-time – achieved at the cost of introducing extra copies. There is no direct discussion of rematerialization; indeed, they do not consider the possibility of spilling. In contrast, we are concerned primarily with quality of spill-code. Nevertheless, their work might be considered a direct ancestor of our approach.

It is also interesting to compare our approach to other published alternatives; for example, the splitting allocator of Chow and Hennessy and the hierarchical coloring allocator of Callahan and Koblenz [26, 16]. The published work does not indicate how they handle rematerialization. It is possible that they make no special provisions, trusting their splitting algorithm to do an adequate job.[18]

Some colleagues have suggested the possibility of more extensive rematerialization, perhaps recomputing entire expressions to avoid excess spilling. The difficulty is avoiding the introduction of additional register pressure in the attempt to save a spill (which was due to excess pressure in the first place).

[18]Inspired by a draft of our paper [14], Brian Koblenz has added rematerialization to their allocator.

Chapter 6

Aggressive Live Range Splitting

Consider the example shown in Figure 6.1. The left side sketches a pair of loops, each updating a variable. If we assume that each loop has only one register available for either x or y, then the right column illustrates the ideal allocation. The Yorktown allocator can never produce this ideal allocation; a value is either held in a register for its entire lifetime or it is spilled for its entire lifetime, with appropriate loads and stores inserted *immediately* before and after each use and definition. Since neither x nor y can be held in a register across both loops, the Yorktown allocator will spill both variables and the resulting code will require many more loads and stores than the ideal allocation.

This chapter explores ways of extending the Yorktown allocator to handle problems similar to those illustrated in Figure 6.1. We propose an aggressive approach to splitting and consider alternative implementations.

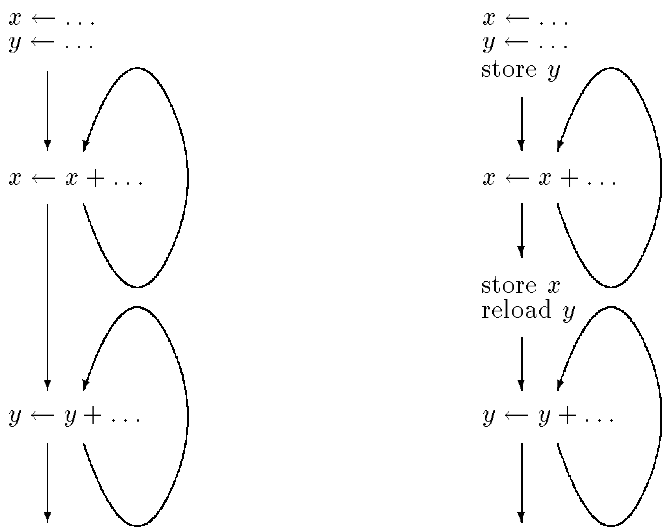


Figure 6.1 Splitting

6.1 Live Range Splitting

As an alternative to the "spill everywhere" approach used in the Yorktown allocator, Chow and Hennessy describe a scheme called *live range splitting* [26]. They observe that breaking a live range into several pieces and considering the pieces separately can produce an interference graph that colors with less spilling.[19] Thus, when their allocator cannot assign a color to some live range l_i, it splits l_i into smaller live ranges, one for each basic block in which l_i appears. These new, smaller live ranges become independent candidates for coloring; eventually, they will be colored or spilled.

To decrease the amount of fragmentation introduced by splitting, Chow and Hennessy also included a method for combining some of these small live ranges. After splitting a live range, their allocator examines the resulting set of smaller live ranges. If it finds two adjacent live ranges that would have $degree < k$ when combined, they are pasted together.

Live range splitting has several merits. The splitting process often creates live ranges of lower degree and the limitation on combining keeps degrees low. If an entire live range is spilled, as in Chaitin's work, its value will reside in a register only for trivial periods around each definition or use. Splitting allows the live range to stay in a register over longer intervals – often an entire block or, if combinations are possible, over several blocks. With luck, the new live range can be large enough to extend over all of an important construct, like an inner loop.

6.1.1 Theoretical Difficulties

Two theoretically hard problems arise in splitting: choosing the right live ranges to split and the right places to split them. Chow and Hennessy use simple heuristics to attack both problems.

- They choose live ranges to split based upon failure of their coloring heuristic. Unfortunately, there is no assurance that this scheme will select the best live ranges to split. For example, in the code from Figure 6.1, their technique will fail to produce the ideal allocation. One of x and y will be colored successfully and the other will be split. However, the ideal allocation requires that *both* live ranges be split. Their allocator never backtracks to consider splitting a successfully colored live range.

[19] Fabri used this same observation to improve the coloring in her work on storage optimization [34].

- Splits are recombined based solely on degree; this can lead to unfortunate locations for split points. For example, Chow might build a live range that extends into a loop, but does not encompass the whole loop, thus requiring a split on the loop's back edge.

Note that split points tend to become spill points; therefore, the correct placement of split points is crucial.

6.1.2 Practical Difficulties

There is a third problem we must consider – efficiency. Chow and Hennessy are able to perform live range splitting relatively efficiently; but to do so, they must sacrifice many of the desirable features of the Yorktown allocator (see Section 7.2). Retaining the precision and algorithmic efficiency of the Yorktown allocator is a challenge. The key problem seems to be the difficulty and expense of maintaining the interference graph as live ranges are split.

6.2 Aggressive Live Range Splitting

Our approach to all these problems is to aggressively split live ranges *before* attempting to color. This idea, combined with our earlier ideas for undoing excess splits (recall Sections 5.3.2 and 5.3.3), seems to offer a useful tack. The following sections provide more detail on splitting and the complications it introduces for the rest of the allocator.

6.2.1 Splitting

In our search for a splitting technique that produced good results with a reasonable running time, we were forced to reconsider the fundamental basis of the coloring approach to register allocation. The key insight is that *the interference graph captures none of the structure of the control-flow graph.* In reducing the allocation problem to a coloring problem, the compiler loses almost all information about the topography of the code. There is no representation for locality. Estimates of execution frequency get factored into estimated spill costs, but because the information is computed over the whole procedure, it gives equal weight to both near and distant references. Thus, a live range that is heavily used in some critical inner loop may get spilled in deference to a value that is live across the loop and used in one or more distant but deeply nested loops.

In an effort to recapture geographic locality, we advocate:

1. finding those points in the code where we would like to spill, if spilling is actually required, and
2. splitting *every* live range at those points.

Thus, we avoid the difficult problem of picking the optimal live ranges to split by splitting all the live ranges that cross a split point. We choose split points based on the structure of the control-flow graph. Of course, such a general statement admits many specific interpretations. Possibilities include:

- splitting at every basic block,[20]
- splitting around high-level control structures, such as loops and if-then-else statements, or
- as suggested in Chapter 5, splitting based on the SSA-graph.

In Section 6.4, we consider several alternatives in detail.

6.2.2 Spilling

Our approach to splitting has some subtle consequences. In the original interference graph, all of the live ranges are independent. After splitting, some of the live ranges are related – they are partners. Recognition and proper handling of partners is critical if the allocator is to produce high-quality spill code. For example, each set of partners should spill to the same location.

Consider the example in Figure 6.2. The single live range in (a) is split in (b) by the introduction of a copy. The resulting live ranges, l_j and l_k, are partners. If l_j is spilled, we should get (c). Alternatively, (d) illustrates the result of spilling l_k. Note that each partner spills to the same location. Finally, (e) shows the result of spilling both partners.

Now consider the costs for the sequence from (b) through (c) to (e). Moving from (b) to (c) costs one store and one load, but saves one copy. The transition from (c) to (e) saves one load at the split point and costs one load at the use point. No new instructions are required; instead, the load is effectively moved. Therefore, the cost

[20]This can be even more effective if we artificially limit the size of basic blocks to some relatively small size [51].

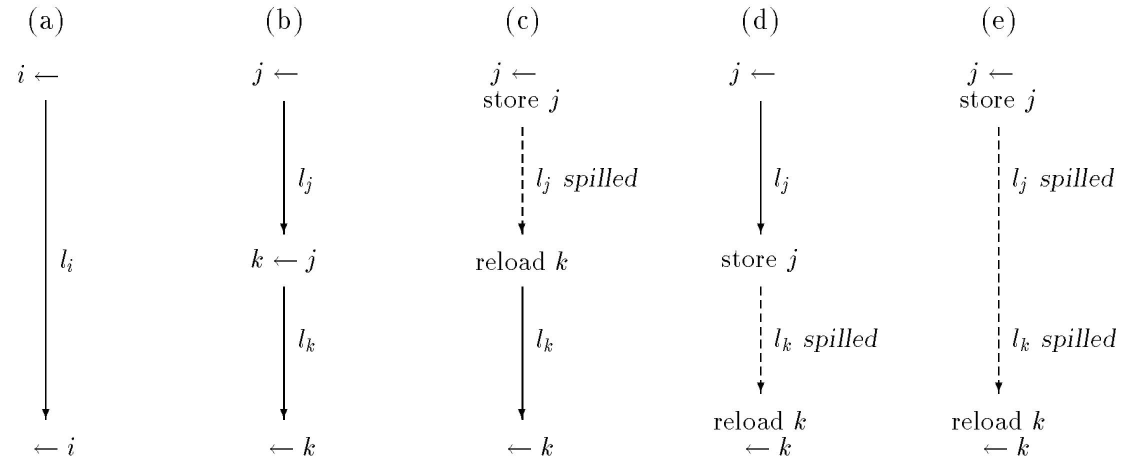


Figure 6.2 Spilling Partners

of spilling l_k at (c) is determined by the relative loop nesting depth of the split point and the use. If the split point is nested more deeply than the use, it will be *profitable* to spill l_k.

We account for these situations while computing spill costs and inserting spill code. Additionally, we update spill costs incrementally during *simplify*. In terms of the example in Figure 6.2, if l_j cannot be colored and must be spilled, the cost of spilling its partner is immediately adjusted, increasing the probability of spilling l_k. Furthermore, if any live range has a negative spill cost, it will be spilled immediately and its partners' costs updated appropriately.

The incremental adjustment to spill costs during *simplify* is really just a simple heuristic that has proven effective in practice. Since a node chosen as a spill candidate may not actually be spilled (indeed, we hope it is not), the adjustments are in some sense premature. We have also experimented with adjusting spill costs during *select*, when we actually mark nodes for spilling. However, the results were nearly always disappointing.

The effect of this careful handling of partners is important. Aggressive splitting can divide long live ranges into long chains of partners. If one partner is spilled, it tends to drag its immediate partners along. Conversely, when a partner is kept in a register, it tends to hold its immediate partners in registers. Together with register pressure from competing live ranges, this works to force spill points out of loop nests.

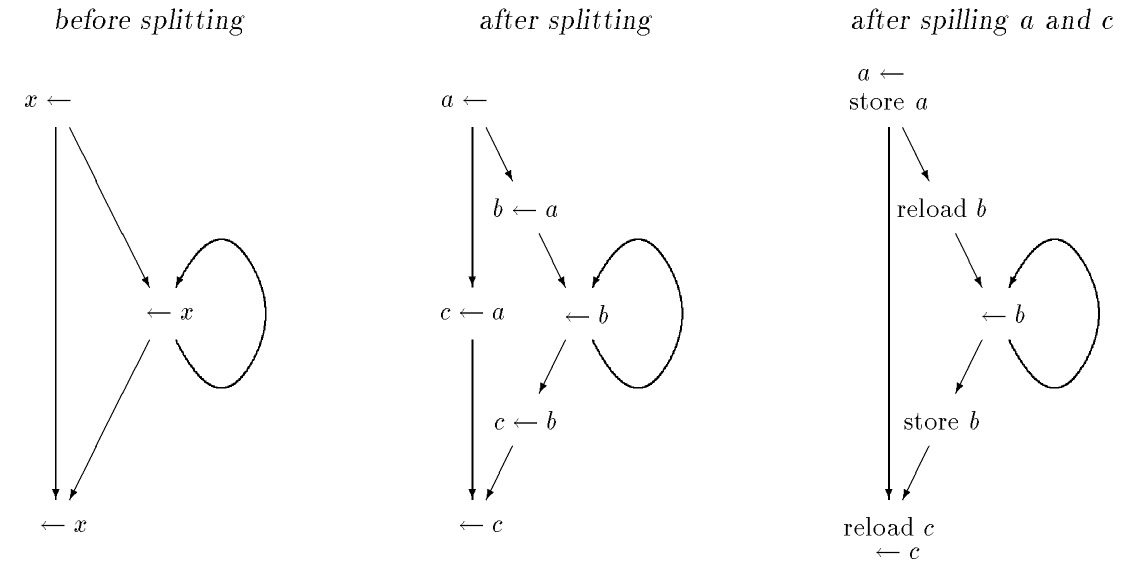


Figure 6.3 Splitting and Spilling

6.2.3 Cleanup

Consider the example in Figure 6.3. The left illustrates the code for a small DO-loop, where all details except for references to x have been omitted. The center illustrates the effect of splitting x into three ranges labeled a, b, and c. The right illustrates the effect of spilling a and c outside the loop. Note that the splits (copy instructions) are converted into loads and stores, just outside the loop as desired. Unfortunately, the store of b at the loop exit is unnecessary.

While the specific case shown in Figure 6.3 is easy to recognize; the general problem is global in nature. Consider the example sketched in Figure 6.4. On the left side, a single live range has been split into four components separated by copy instructions. On the right side, the fourth component l_z has been spilled. The question arises: How do we handle the copy $z \leftarrow y$? One possibility is to convert it to a store instruction; certainly the spill location in memory must have the correct value when z is finally loaded. However, suppose l_w has also been spilled. In this case, the value in memory would already be initialized and the copy instruction can simply be deleted. The difficulty is that the correct handling of $z \leftarrow y$ doesn't depend on l_y, l_z, or even their immediate partners.

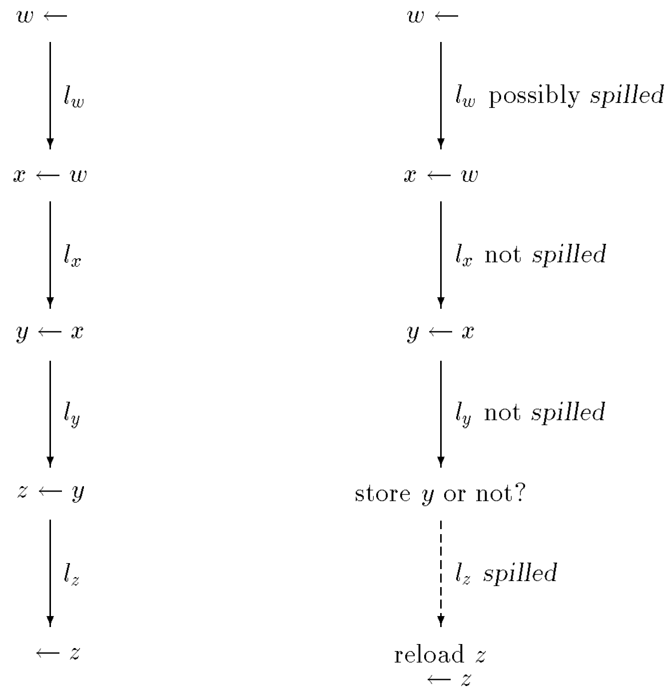


Figure 6.4 Globally Unnecessary Stores

For best results, we would like to see such facts reflected immediately in the spill costs for each component, similar to the heuristic used to maintain spill costs for immediate partners. However, it seems difficult to accomplish this updating on the fly. Therefore, we simply insert redundant stores when they may be necessary, accepting the imprecision.

However, we are able to detect and eliminate redundant stores in a separate pass by solving a global data-flow problem for each set of partners (recall that all the partners split from a single live range share the same spill location). Redundant stores are discovered and removed immediately after spill code has been inserted, in a separate phase called *cleanup*. *Partially* redundant stores are still a problem; see Figure 6.6 for an example.

Note that this problem is apparently shared by other allocators that attempt splitting [26, 16]. Actually, the other allocators seem to accept the extra stores, with no attempt at later cleanup.[21] It seems to be a difficulty inherent in splitting.

[21]In conversation, David Callahan mentioned that they were aware of the problem and were considering solutions, though he knew of nothing better than our batch approach.

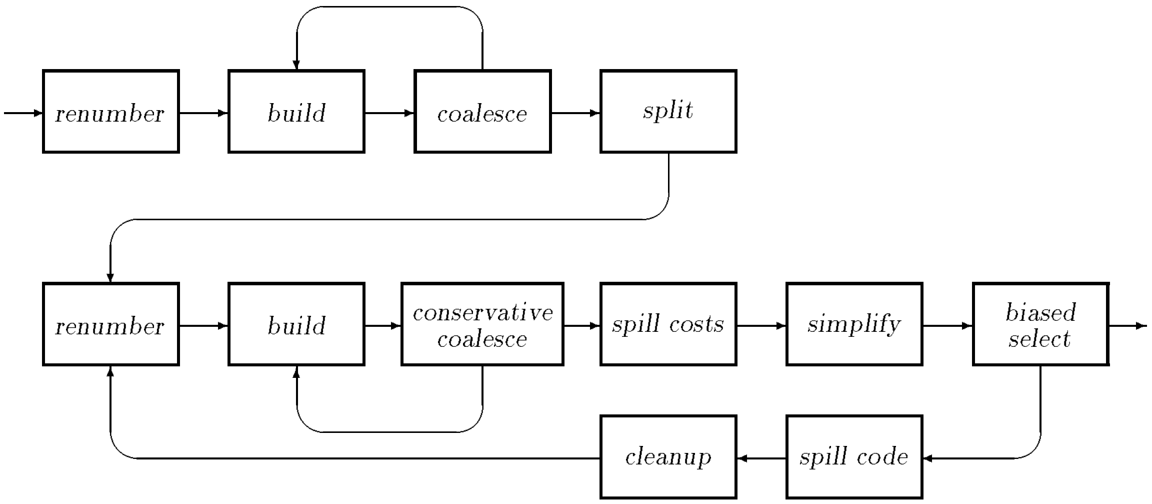


Figure 6.5 The Splitting Allocator

6.3 Implementation

Integrating these ideas into our allocator was a major task. Figure 6.5 shows a high-level view of the resulting allocator. Several points have changed from the optimistic allocator depicted in Figure 3.3. While there are many components, they may be partitioned into three major phases:

before splitting The portion is lifted directly from our earlier implementations of the Yorktown allocator. The intent here is to use the unrestricted *coalesce* to remove any extraneous copies from the code before introducing new splits. Thus, before the splitter is ever invoked, the allocator will reduce the number of live ranges to some canonical set. In the simplified code, any remaining copy instructions are meaningful.

splitting This is a generic splitting phase. For our experiments, we can employ any one of several possible splitting heuristics. Each splitter does some combination of data-flow analysis and control-flow analysis and introduces splits (distinguished copies) to guide the remainder of the allocation.

after splitting This portion of the allocator is nearly identical to the optimistic allocator shown in Figure 3.3. The major differences are the use of *conservative coalesce* and *biased select* (already introduced to support rematerialization) and global *cleanup* after spilling.

The final phase of the allocator must carefully maintain the distinction between live ranges (discovered by the first phase) and partners (determined by splitting). Furthermore, the allocator must maintain the relation between partners and their original live ranges, since all partners split from a single live range should spill to the same location. This relation is also required by *cleanup*.

6.4 Splitting

We have considered many possible approaches to splitting. The two next sections describe several heuristic approaches that appeared attractive. Section 6.4.3 describes some important details common to all our implementations.

6.4.1 Loop-Based Splitting

Our exploration of live range splitting was motivated by the intuition that we often want to split live ranges around loops.

Splitting Around All Loops

Our initial approach was simple and aggressive:

1. Find all loops in the code using Tarjan's algorithm for testing reducibility [62]. For our experimental purposes, this approach is adequate; production implementations will require extensions to handle irreducible control-flow graphs.

2. Edges leading in and out of loops are marked as split points. We insert empty basic blocks at each split point (we refine this notion in Section 6.4.3).

3. Split *all* the live ranges that cross each split point by inserting copies in the new basic block.

Our early implementations were exploratory; many of the heuristics we now employ were discovered in the course of our experiments. For instance, our work on rematerialization was prompted by examples observed in practice – for example, in handling the many COMMON blocks in the SPEC program `doduc`.

Despite some encouraging results, our overall impression is that splitting around all loops gives unstable results; that is, it performs well in some cases and poorly in other cases. Even more disappointing were the compile-time costs. In some cases, space and time requirements were increased by a factor of ten. While some increase in compile-time or space might have been acceptable if we could count on better results; more expensive allocations *and* inferior results were not attractive.

We have tried other, more conservative, possibilities. For example, we can split around outermost loops only, approximately splitting the routine into manageable pieces. Alternatively, we can split only around innermost loops, attempting to isolate computationally intensive areas in the code. However, all these approaches share the common weakness of ignoring all information about the location of uses and definitions.

Splitting Based on Inactivity

An attractive alternative is to split only *inactive* live ranges around loops. The intuition here is that some live ranges extend across a loop without being mentioned (used or defined) in the body of the loop. It seems clear that they should be spilled first if there is excess pressure in the loop. Therefore, we can do a simple scan of the code in each loop, accumulating the set of live ranges mentioned in the loop. Given this set, we can split any unmentioned live range at the entrance and exit of the loop.

This simple approach extends naturally to loop nests. For each loop, we accumulate the same information. Then, working from the outermost loop inward, we split unmentioned live ranges around a loop – but only if the live range was not split around an enclosing loop. Figure 6.6 illustrates the desired effect on a nest of two DO-loops. In this case, x is referenced in the innermost loop; therefore, it is not split at all. There is a use of y in the outermost loop, but it is unreferenced in the innermost loop; therefore, y is split on the edges leading in and out of the innermost loop. There are no mentions of z in either loop; therefore, z is split around the outermost loop (but not the innermost).

We implemented a version of our allocator that split unmentioned live ranges around loops. During limited tests, the results were almost uniformly disappointing. Reconsidering Figure 6.6 with a more sceptical eye, we can see possible reasons for the poor results:

Splitting z There was little benefit gained by splitting z. If the unsplit z is spilled, the results will be nearly identical to the result of spilling z'. Of course, if the outer loop is never executed, the split is preferred; but these cases are perhaps uncommon in practice (especially with FORTRAN routines).

Splitting y If y' is spilled, the results will actually be worse than simply spilling the unsplit y. The two split points will be converted into a store and a load inside

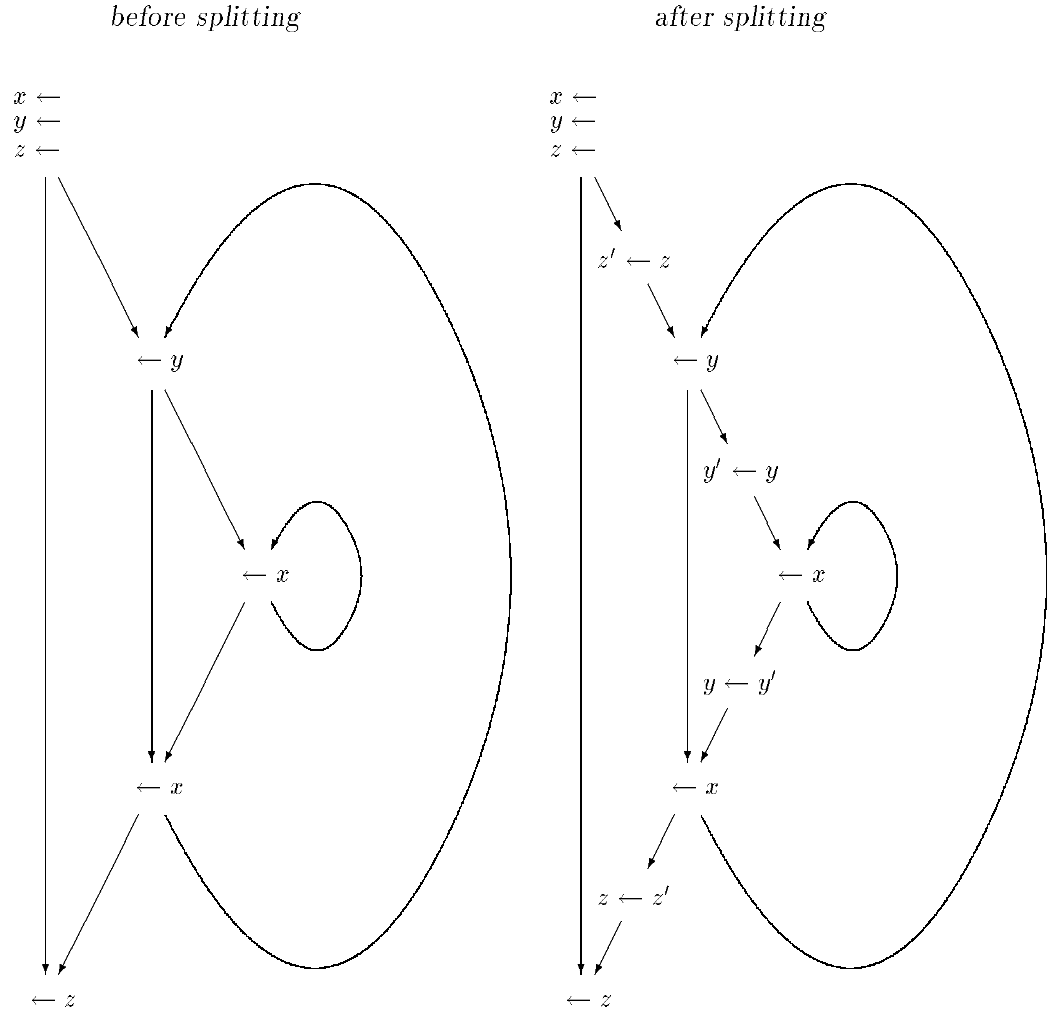


Figure 6.6 Splitting Unmentioned Live Ranges

the outer loop, whereas spilling the unsplit y would only require a load in the loop. This case is annoying since y is not modified inside the loop; all but the first store will be useless. Note that *cleanup* is unhelpful in this case, since the store is required on the initial iteration. Handling this case optimally is difficult – it requires cloning the entire loop nest.[22] Again, if the inner loop is never executed, splitting y is helpful.

Of course, one loop nest is not an entire routine. It is certainly easy to construct cases where splitting y or z as shown in Figure 6.6 can be profitable. These points are simply mentioned to help illustrate the difficulties of proper splitting.[23]

6.4.2 Splitting Based on Dominance

An alternative is to split live ranges based on the location of ϕ-nodes in the pruned SSA graph. This idea was suggested by several people in discussions about splitting (including Jeanne Ferrante and Mike Lake). Furthermore, it is a natural extension to our work with rematerialization. Exploration of this alternative leads to several approaches based on the fundamental idea of *dominance*.

Dominance and Dominance Frontiers

Cytron *et al.* give a fast method for building the SSA-graph based on the idea of *dominance frontiers* [29]. Dominance frontiers, as the name suggests, are based on the idea of *dominance*.

In a flow graph (a directed graph with a designated node *start*), a node x dominates a node y if all paths from *start* to y include x. Additionally, if $x \neq y$, then x *strictly dominates* y. In the control-flow graph for a large routine, a given basic block often dominates many other blocks; for example, the header of a loop will dominate all members of the loop.

The dominance frontier of a node x (DF_x) is the set of nodes y such that x dominates a predecessor of y but does not strictly dominate y. Notice that the last clause allows x to be a member of its own dominance frontier. Intuitively, the dominance frontier of x does not include nodes dominated by x; rather, it includes the nodes *just outside* the dominion of x.

[22] This discussion suggests the possibility of splitting only undefined live ranges around loops; that is, live ranges that are not defined inside the loop. This is a conservative approach that avoids some of the problems of cleaning up extra loads. We have not yet explored this avenue.

[23] Certainly these difficulties were not obvious to us when we began work on this problem.

To insert ϕ-nodes for a variable v, we find basic blocks containing definitions of v. For each such basic block b, we insert a ϕ-node for v in each block of DF_b. Since we are building a *pruned* SSA-graph, we actually insert a ϕ-node in a block d only if v is live on entrance to d. Of course, a ϕ-node represents a new definition of v; therefore, further ϕ-nodes are inserted in DF_d. Cytron *et al.* give an efficient worklist algorithm for placing ϕ-nodes in this *iterated dominance frontier*.

Splitting at Dominance Frontiers

Recall the discussion in Section 5.2.3. In that case, we were attempting to isolate *never-killed* values by inserting a minimum number of splits. We inserted splits on edges leading into a ϕ-node if the incoming value had a different tag than the value defined by the ϕ-node. In the present case, we simply insert splits on *all* edges leading to a ϕ-node. The effect is to isolate all the values in each live range, allowing the allocator to handle each value individually.

This approach is attractive for several reasons. Of course, we already have all the required machinery as a result of our work on rematerialization. Furthermore, we avoid the awkwardness of finding loops (for loop-based splitting) in irreducible control-flow graphs. Finally, this approach is intuitively appealing because it depends on the structure of the control-flow graph and the location of definitions in the routine. It seems to achieve much of the effect of loop-based splitting with much less actual splitting. Additionally, we can hope to obtain some benefit in other regions of the control-flow graph; for example, around IF and CASE constructs.

We were able to build a splitting allocator that split live ranges based on the pruned SSA-graph. In Section 7.1, we compare the SSA-based allocator with the rematerializing allocator discussed in Chapter 5. From a total of 70 routines, we found a difference in 35 cases, with 21 improvements and 14 degradations. These numbers are pessimistic in that the improvements appear larger in magnitude than the degradations. On the other hand, the relatively large losses on `twldrv` and `tomcatv` are disappointing. We also note that the number of copy instructions almost always increases when using SSA-based splitting. This suggests that our heuristics for removing excess copies, while adequate for rematerialization, are less satisfactory in this case.

Splitting at Reverse Dominance Frontiers

Splitting at ϕ-nodes (or dominance frontiers) is appealing since it accounts for both control structure and the location of definitions; however, there is no allowance for the location of uses. Recall the example loop nest shown on the left side of Figure 6.6. If we attempt to split based on the location of ϕ-nodes, we get no splits at all. Remember that ϕ-nodes are inserted based on the location of definitions. In this example, the definitions of x, y, and z are all located in the first block, a block that dominates every other block in the graph. Again, this is only a small example, but we can imagine realistic cases. For instance, this same situation arises when parameters are defined in the entrance to a subroutine and remain constant throughout the routine. Since the entrance block of a routine certainly dominates the entire routine, none of the constant parameters will be split. This seems unhelpful – an effective splitting allocator should have some provision for splitting these live ranges.

These considerations suggest additional splitting based on the location of uses and *reverse* dominance frontiers. The reverse dominance frontier for a node is defined to be the dominance frontier, but computed on the reverse control-flow graph; that is, the control-flow graph with all edges reversed. The algorithm for placing reverse ϕ-nodes would be similarly analogous to the algorithm invented by Cytron *et al.* for placing ϕ-nodes. To achieve the desired effect (splitting at forward and reverse dominance frontiers), we maintain two worklists simultaneously, with each ϕ-node placement contributing to both worklists.

Figure 6.7 shows two rather simple examples that help illustrate the effect of splitting at both forward and reverse dominance frontiers. In the upper example, we show a loop containing a use (and no definitions) of a live range x that is live across the loop. The lower example illustrates the effect if x is dead at the end of the loop (any further uses would be masked by the second definition of x). Note that we avoid splits when the result would be unused. In both cases we have completely separated the uses and definitions; if desired, it would be possible to allocate a, b, and c to different registers.

The intuition behind splitting at dominance frontiers is not to achieve complete separation of all uses and definitions (indeed, it does not); instead, it relates to the traditional use of dominators in optimization. Consider the lower half of Figure 6.7. If a is spilled, the load of b occurs at the split point, at a point leading *inevitably* to a use of b.

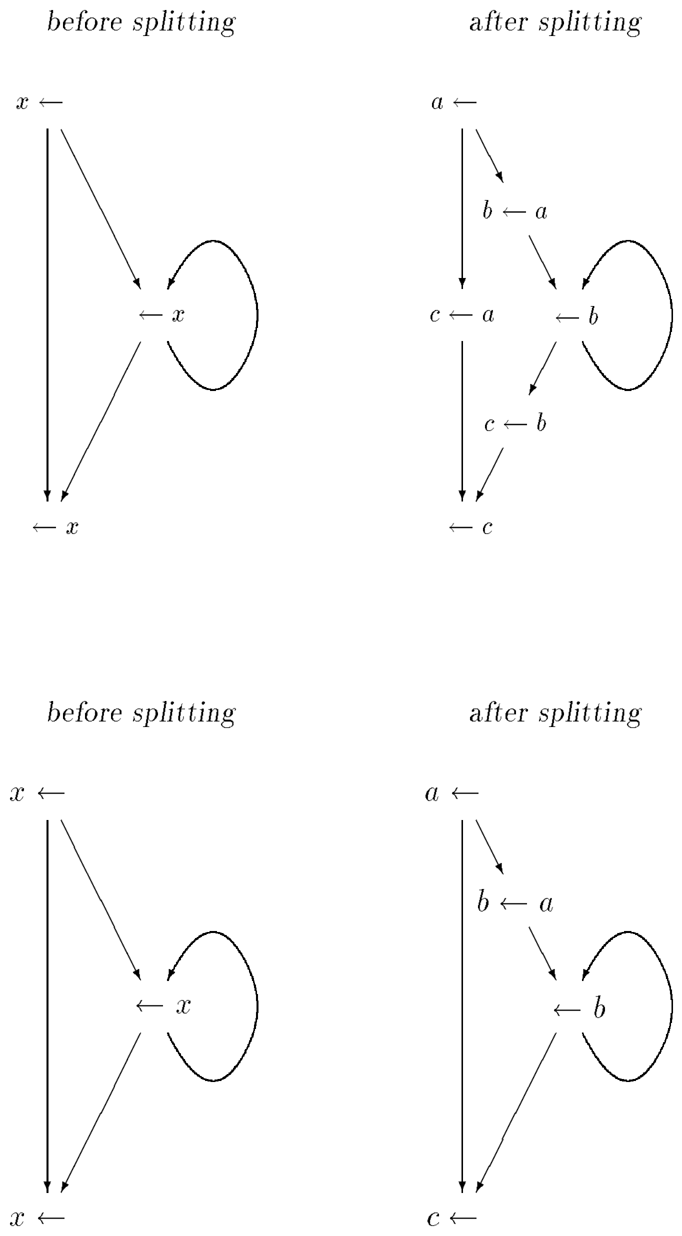


Figure 6.7 Splitting at Dominance Frontiers

We built another version of our splitting allocator that split live ranges at both forward and reverse dominance frontiers. Section 7.1 contains a comparison with our rematerializing allocator. The results are mediocre. Out of 70 routines, there were differences in 42 cases: 13 improvements and 29 degradations. Furthermore, many of the degradations were quite large. On the other hand, there was a 22% reduction in spill cost for `tomcatv`; in raw cycles, this improvement probably dominates all other effects in the entire thesis. Once again, we see significant losses due to excess copies.

6.4.3 Mechanics

There are many mechanical details that must be handled correctly while splitting to achieve best results. They are largely independent of any specific splitting heuristic.

Inserting New Basic Blocks

Conceptually, we would split live ranges on edges in the control-flow graph. In practice, splits may require actual instructions: copies or perhaps loads and stores. To accommodate the split instructions, we must create basic blocks along some of the edges in the control-flow graph. While it may be possible to create the new blocks when and where desired, we simply create them when the control-flow graph is constructed. After allocation, empty blocks can be quickly deleted.[24]

Recall that splits are always inserted before a join or after a fork. If the edge leading into a join has a source with no other successor, then we simply insert the split at the end of the source block. Similarly, if the edge leading from a fork has a destination with a single predecessor, than we insert the split at the beginning of the successor block. Therefore, new blocks are only required on edges whose source has more than one successor *and* whose destination has more than one predecessor.

It may be difficult (in a practical sense) to split some edges. For example, in a FORTRAN routine, it seems difficult to split edges leading from an ASSIGNed GOTO to a join point. Fortunately, the use of ASSIGN is apparently rare. FORTRAN's computed GOTO statement, Pascal's CASE statement, and C's `switch` statement are all manageable, given an adequate intermediate representation.

---

[24]Deletion of empty blocks is required anyway, since *coalesce* is sometimes able to delete every instruction in a basic block.

Splits to Avoid

It is clear that we ought to avoid splitting a live range immediately after it is defined or immediately before it is used. In fact, if we recall the details of spill code generation, it becomes clear that we ought not split a live range if there have been no deaths between its definition and the end of the block. Similarly, we avoid splitting a live range if there are no deaths between the beginning of the block and its use. Finally, we must be careful not to split a live range at both the beginning and end of a block if there are no deaths within the block.

Ordering Splits

Many split instructions may be inserted in a single block. Before spilling, the ordering of individual splits is unimportant; however, after some live ranges have been spilled, the order of resulting loads, stores, and the remaining splits is significant – stores should be scheduled first, then splits (copies), and finally loads. The insight here is that a store is the end of a live range and a load is the beginning of a live range – by placing loads after stores, we minimize interferences.

Unfortunately, we discovered this idea too late. None of our experimental implementations take advantage of the insight; instead, splits are inserted in some arbitrary order and the order remains unchanged while spilling.

6.5 Summary

The approach to spilling used in the Yorktown allocator (and in the variations discussed in earlier chapters) is quite coarse. It is easy to give examples where some form of live range splitting is desirable: Figure 6.1 illustrates a typical artificial example; a more realistic example is provided by SVD (Figure 3.2). In this chapter, we have described an approach that allows finer control over spilling within the general context of the Yorktown allocator.

The fundamental problem with the Yorktown allocator is that the reduction of the register allocation problem to graph coloring throws away a large amount of information; for example, the structure of the control-flow graph. Of course, an advantage of the reduction to coloring *is* the distillation of large amounts of programmatic detail to the essential concept of interference. When there are adequate registers, the loss of structure information is unimportant. When there are insufficient registers, the information about control structure could have been used to help minimize spilling.

Our general approach is to split live ranges into finer components before attempting to color. Thus, instead of being forced to spill an entire live range, the allocator can simply spill the troublesome components individually.

The bulk of our work has been exploring the consequences of our aggressive approach to splitting. We were able to identify a number of important details that must be handled correctly for best results. These have never been published before and it seems likely that other splitting allocators may be improved by correctly handling these cases.

Our approach is sensitive to the heuristic employed for splitting. In Section 6.4, we considered several alternative heuristics. Despite the intuitions and rationalizations supporting each heuristic, none were completely satisfactory. While there were some notable successes on individual routines, each splitting heuristic seemed too unstable for production use. Furthermore, our heuristics for removing excess splits seemed inadequate. Of course, our implementations also suffer from uncontrolled placement of splits, as discussed in the previous section.

Future work will certainly begin with new implementations, taking advantage of our new ideas for split placement. Additionally, there are many unexplored heuristics for splitting; given the difficulty in removing excess copies, we plan to explore splitting heuristics that perform less unnecessary splitting. In our implementation, splitting is only performed once; therefore, we can afford to spend a fair amount of time deciding which live ranges to split and where to split them. The sensitivity of our allocator to the splitting heuristic (as shown by the widely varying results) emphasizes the need for accurate splitting.

There are other approaches to live range splitting. Chow and Hennessy are able to accomplish splitting by sacrificing much of the precision and efficiency offered by Chaitin's approach (see Section 7.2). Callahan and Koblenz also describe an approach based on a hierarchical decomposition of the control-flow graph [16]. However, our experience suggests that other approaches to splitting may not be achieving the high-quality code they desire. While studying the (many) problems discovered during the course of our experiments, we have often referred to published descriptions of other splitting allocators – neither the problems nor their solutions are discussed.[25]

[25]The problems solved by *rematerialization* and *cleanup* are two good examples.

One possibility is that the problems have not been noticed, perhaps due to lack of comparison with other allocators. In our work, we have been able to compare against a high-quality allocator; therefore, cases of poor performance are immediately exposed. Without reference to the standard set by the Yorktown allocator and our improvements, we would be unable to evaluate the performance of our approach.

Chapter 7

Measurements and Comparisons

There are many qualities to consider when comparing register allocators – implementation difficulty, compile-time and space requirements, and allocation quality. Since our work largely concentrates on improving the Yorktown allocator, we are interested in comparing allocation quality; the implementation difficulty and compile-time and space requirements are relatively constant across our variations. The next section presents a series of experiments used to compare the Yorktown allocator and several of the improvements discussed in the previous chapters.

A second well-known approach to global register allocation, priority-based coloring, was introduced by Chow and Hennessy [26]. The final section compares the Yorktown allocator with priority-based coloring and presents an experiment used to study their compile-time behavior.

7.1 Measuring Allocation Quality

When building a register allocator for an optimizing compiler, we are primarily concerned with the speed of the generated object code. A high quality allocation will result in faster code than a low quality allocation. Usually, slower code will be the result of extra instructions; *e.g.*, load and store instructions required for spill code.[26] In this section, we describe a series of experiments designed to compare the allocation quality of various versions of our allocator. We perform comparisons by measuring the number of instructions executed by routines compiled using each allocator. The following sections present a description of the experiments followed by a summary and discussion of the results. The complete results are given in Appendix A.

[26] On pipelined machines, register allocation may sometimes have an adverse impact on instruction scheduling; however, these considerations are beyond the scope of this work.

7.1.1 Methodology

To support our research, we have written an optimizing compiler for FORTRAN. The compiler is part of the $\mathbb{R}^n$ programming environment and includes support for interprocedural analysis and a variety of traditional optimizations [27]. We currently generate code for the IBM RT/PC and have experimental code generators for the Sparc, i860 and RS/6000. To experiment with register allocation, we have written a series of allocators that are independent of any particular architecture.

The optimizer is organized as a collection of independent programs, each accomplishing a distinct transformation. During optimization, a routine is represented in a low-level intermediate form, ILOC. Each pass of the optimizer reads in a routine via `stdin`, performs the necessary analysis and transformations, and writes the result to `stdout`. Since each pass consumes and produces ILOC, they may be organized in any order, with specific passes added, repeated, or omitted as desired.

After optimization, which includes passes to accomplish the effects of instruction selection, we run register allocation. The register allocator is also organized as an independent pass, consuming and producing ILOC, but it is always run after the optimization passes and may not be omitted or repeated.

ILOC is a low-level intermediate language designed to allow extensive optimization. It resembles the assembly language for a simple RISC machine, with the addition of hooks for interprocedural information and certain higher-level operations representing FORTRAN intrinsics. During optimization, the high-level operations may be expanded to a sequence of lower-level operations or subroutine calls if required. The design of ILOC and the entire optimizer was heavily influenced by the PL.8 compiler [6].

Each register allocator is built around the same basic framework. An ILOC routine that assumes an infinite register set is rewritten in terms of the target machine's register set, with spill code added as necessary. The stack frame size may be adjusted to accommodate spilled values and copy instructions may be added and deleted. The target register set is specified in a small table and may be varied (within limits) to allow convenient experimentation with a wide variety of register sets.

After register allocation, each ILOC routine is translated into a complete C routine. Each C routine is compiled and the resulting object files are linked into a complete program. There are several advantages to this approach:

- By inserting appropriate instrumentation during the translation to C, we are able to collect accurate, *dynamic* measurements.
- Compilation to C allows us to test a single routine (perhaps from a library) in the context of a complete program running with real data.
- We are able to perform our tests in a machine-independent fashion, potentially using a variety of register sets.

Simply timing actual machine code is inherently machine-dependent and tends to bury the important issues (loads and stores) under the sometimes massive costs of incidental floating-point computation.

The idea was originally suggested to us by Hans Boehm as a way to avoid having to retarget the compiler for every new architecture we encountered (this approach was used by researchers at Xerox PARC to aid in porting Cedar [5]). We were further influenced by Mills *et al.*, who describe a *compiled* instruction set simulator [55].

During the translation into C, we are able to add instrumentation to accumulate the dynamic occurrences of any class of ILOC instruction. For the purposes of comparing register allocators, we are concerned with the number of loads, stores, copies, load-immediates, and add-immediates.[27] At the entrance of an instrumented routine, five integer variables are initialized. At each instruction, the appropriate counter is incremented. When the routine exits, the values of all five counters, along with the name of the routine, are printed to `stderr`.

Figure 7.1 shows a small sample of ILOC code and the corresponding C translation. Usually there is a one-to-one mapping between the ILOC statements and the C translations, though some additional C is required for the function header and declarations of the "register" variables; *e.g.*, `r14` and `f15`. Also note the simple instrumentation appearing immediately after several of the statements. The number of load-immediates is accumulated in the variable `i`, the number of copies in `c`, the number of loads in `l`, the number of add-immediates in `a`, and the number of stores in `s` (not shown in the example). Of course, this code is simple, but the majority of ILOC is no more complex. However, some care is required in handling procedure call and return.

[27]The numbers of load-immediates and add-immediates are affected by rematerialization.

```
LLE3:  nop                                LLA4:
LLA4:  ldi   r14   8                      LLE3:  r14 = (int) (8); i++;
       add   r9    r15   r11                     r9 = r15 + r11;
       mvf   f15   f0                            f15 = f0; c++;
       bc    L0023                               goto L0023;

L0023: lddrr f14   r14   r9               L0023: f14 = *((double *) (r14 + r9)); l++;
       dabs  f14   f14                           f14 = fabs(f14);
       dadd  f15   f15   f14                     f15 = f15 + f14;
       addi  r14   r14   8                       r14 = r14 + (8); a++;
       sub   r7    r10   r14                     r7 = r10 - r14;
       br    ge    r7    N6    N7                if (r7 >= 0) goto N6; else goto N7;
```

Figure 7.1 ILOC and C

The Target Machine

For the tests reported here, our target machine is defined to have sixteen integer registers and sixteen floating-point registers. Each floating-point register is capable of holding a double-precision value, so no distinction is made between single-precision and double-precision values once they are held in registers. Integer register 0 and floating-point register 0 are both defined to be 0. Integer register 1 is reserved as the frame pointer. Up to four integer registers may be used to pass arguments (recall that arguments are passed by reference in FORTRAN; therefore, the argument registers hold pointers to the actual values); any remaining arguments are passed in the stack frame. Function values are returned in an integer or floating-point register, as appropriate. Ten of each register class are designated as callee-saves; the remaining six (including the argument registers) are not preserved by the callee.

When reporting costs, we assume that each load and store requires two cycles; all other instructions are assumed to require one cycle. Of course, these are simply convenient assumptions that reflect no actual machine architecture. The raw data is given in Appendix A to allow the interested reader to recalculate results for different relative instruction weightings.

Spill Costs

Since our instrumentation reports dynamic counts of *all* loads, stores, *etc.*, we need a mechanism for isolating the instructions that arise due to register allocation decisions – after all, a typical routine will perform some loads and stores even with an infinite register set. A difficulty is that some spills are profitable. In other cases, the allocator removes instructions; *e.g.*, copy instructions. Therefore, we tested each routine on a hypothetical "huge" machine with 128 registers, assuming this would give a nearly perfect allocation. The difference between the "huge" results and the results for one of the allocators targeted to our "standard" machine should therefore equal the number of cycles added by the allocator to cope with insufficient registers.

In reality (or when dealing with NP-complete problems), problems arise. A few routines from our test suite require more instructions with the "huge" machine than the "standard" machine. In these cases (notably `yeh`, from the program `doduc`), we simply present raw results, with no attempt to determine spill contributions.

The Test Suite

Our test suite is a collection of seventy routines contained in eleven programs. Eleven routines are from Forsythe, Malcolm, and Moler's book on numerical methods [36]. They are grouped into seven programs with simple drivers. The remaining fifty-nine routines are from the SPEC benchmark suite [61]. Four programs were used: `doduc` (41 routines), `fpppp` (12 routines), `matrix300` (5 routines), and `tomcatv` (1 routine). The two other FORTRAN programs in the suite (`spice` and `nasa7`) require language extensions not yet supported by our front-end.

7.1.2 Results

In earlier chapters, we have described several improvements to the basic Yorktown allocator, giving a sequence of allocators. Similarly, we have organized our results as a sequence of comparisons: each variation is compared with the previous approach. We present five tables comparing six allocators:

- Table 7.1 shows a comparison of test results for the Yorktown allocator (*Chaitin*) and the optimistic allocator described in Chapter 3 (*Optimistic*).
- Table 7.2 shows the results of adding *limited backtracking* (as described in Section 3.3) to the optimistic allocator. In this case, the improvements were small, so the percentages are reported with a extra digit of precision.
- Table 7.3 shows the results of our improved approach to rematerialization (see Chapter 5). In this case, we compare the optimistic allocator using the Yorktown allocator's limited rematerialization to a version of the optimistic allocator using our enhanced approach to rematerialization.

- Experimental results for our splitting allocator (see Chapter 6) are shown in Table 7.4 and Table 7.5. In each case, we compare against our best previous allocator (the optimistic allocator with our approach to rematerialization). Table 7.4 shows the effect of splitting based on the SSA-form. Table 7.5 shows the results of splitting at both forward and reverse dominance frontiers.

In each case, the tables show only routines where a difference was found (some routines were so short as to require no spill code). Furthermore, we have omitted routines when the differences comprised less than one percent of the spill code (approximately ten cases total, over the five tables).

Each table is organized identically. The first two columns give the program and subroutine name. The third and fourth column give the observed spill costs for the two allocators being compared. These costs are calculated from dynamic counts of instructions as described earlier. The last column (*total*) gives the percent improvement in spill costs from the old allocator to the new one, where

$$\%\ improvement = \frac{old - new}{old} \times 100$$

Therefore, large positive numbers indicate significant improvements. Middle columns show the contribution of each instruction type to the total.

All percentages have been rounded to the nearest integer. Insignificant results are reported as 0 (or, for an insignificant but negative result, -0). In cases where the result *is* zero, we simply show a blank. Since results are rounded, the *total* entry may not equal the sum of the individual instruction entries.

For an example, consider the first row in Table 7.1. This row presents results for the routine `fmin` from the program `fmin`. The Yorktown allocator generated an allocation requiring 551 cycles of spill code; the optimistic allocator required only 370 cycles. 16% of the savings came from having to execute fewer loads and 16% arose from fewer stores. A further insignificant fraction from fewer load-immediates. The total improvement was 32%.

A few rows are distinguished by a $\star$ in the *total* column. In these cases, we were unable to determine the spill contribution of the total cost since the allocation for the "huge" machine required more cycles than the allocations for the "standard" machine. In these few cases, we simply show the total costs (all loads, stores, *etc.*) and report the actual cycles saved for each instruction type.

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Chaitin*	*Optimistic*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
fmin	fmin	551	370	16	16		0		32
seval	spline	125	117	5	2				6
solve	decomp	362	305	10	6	0			16
svd	svd	2,509	1,977	16	7	−2			21
doduc	colbur	25	19	8	8		8		24
	dcoera	29	15	21	21		7		48
	ddeflu	443	335	13	12	−0			24
	debflu	1,939	1,131	20	20				42
	debico	463	459	0	0				1
	deseco	5,500	4,957	4	2	−1	3	0	10
	drepvi	252	218	7	6				14
	ihbtr	452	400	6	6				12
	inithx	714	579	11	4		4		19
	integr	526	502				5		5
	lectur	257	221	11	2		2		14
	paroi	1,780	1,433	9	6			5	20
	prophy	1,954	1,531	13	9			0	21
	repvid	651	599	4	4				8
	supp	146	149	−1	−1		0		−2
	yeh	353	297	32	24				⋆
fpppp	d2esp	51	35	16	16	−2	2		31
	efill	173	94	21	23		2		46
	fpppp	1,472	1,444	1	1				2
	twldrv	13,731,802	11,311,624	12	7	0			18
matrix300	lbmk14	136	132	1	1				3
	sgemm	12,321	9,905	10	10				20
	sgemv	3,027	1,808	40	0	−0			40
tomcatv	tomcat	394,397,732	367,995,733	3	3	−0			7

Table 7.1 Effects of Optimistic Coloring

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Optimistic*	*Backtrack*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
doduc	ddeflu	335	327	1.2	1.2				2.4
	drepvi	218	214	0.9	0.9				1.8
matrix300	sgemv	1,808	1,804	0.1	0.1				0.2

Table 7.2 Effects of Limited Backtracking

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Optimistic*	*Rematerial*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
rkf45	fehl	68	50	26		7	−7		27
seval	spline	117	102	10	2	2	−1		13
solve	decomp	305	286	4	3		−1		6
svd	svd	1,977	1,966	1		0	−0		1
zeroin	zeroin	236	234	2		−1			1
doduc	bilan	1,046	966	5	3				8
	bilsla	16	15				6		6
	colbur	19	24	−11	−11	−5			−26
	ddeflu	335	375	−5	−7	1	1		−12
	debico	459	418	6	0	1	2		9
	deseco	4,957	4,636	7	2		−2	0	7
	drepvi	218	175	4	14	0	2		20
	drigl	32	31				3		3
	heat	34	31	6		1			9
	ihbtr	400	395	1	0		−0		1
	inideb	50	48				4		4
	inisla	31	28		6		3		10
	inithx	579	437	17	10		−2		25
	integr	502	372	18	12		−3		26
	lectur	221	166			2	23		25
	orgpar	39	35		5	−3	8		10
	paroi	1,433	1,383	8	0	−1	−4		4
	pastem	289	220	20	10	13	−19		24
	repvid	599	404	9	13	11			33
	yeh	297	290	4	4	2	−3		⋆
fpppp	d2esp	35	34	6			−3		3
	main	210	199			0	5		5
	twldrv	11,311,624	11,198,058	2	0		−1		1
matrix300	sgemm	9,905	8,398	12	6		−3		15
tomcatv	tomcat	367,995,733	355,039,258	4	0	−0			4

Table 7.3 Effects of Rematerialization

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Rematerial*	*SSA*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
fmin	fmin	370	359	10		−7			3
seval	spline	102	101	1		−0			1
solve	decomp	286	278	3	3	−3	−1		3
svd	svd	1,966	1,883	−0	4	1	−0		4
zeroin	zeroin	234	143	39	12	−12			39
doduc	cardeb	83	85			−2			−2
	colbur	24	25			−4			−4
	ddeflu	375	257	12	17	−2	4		32
	debflu	1,131	1,290	−3	−5	−7		1	−14
	debico	418	431				−3		−3
	deseco	4,636	4,363	2	3	0	0	−0	6
	drepvi	175	160	6	5	−2			9
	drigl	31	32				−3		−3
	heat	31	26	6	13	−3			16
	ihbtr	395	319	21	2	−3	−0		19
	inideb	48	25				48		48
	inithx	437	435		1	−0			1
	integr	372	383		1	−4			−3
	lectur	166	145			−2	14		13
	paroi	1,383	1,366	2	2	−0		−4	1
	pastem	220	218	12	1	−12			1
	prophy	1,525	1,678	−7	−3	−1	2		−10
	repvid	404	429			−6			−6
	saturr	106	114	−4			−4		−8
	sigma	12	13			−8			−8
	sortie	20	24	−10	−10				−20
	supp	149	137	4	4				8
	yeh	290	283	2	4	1			⋆
fpppp	d2esp	34	38	−6	−6				−12
	efill	94	70	2	26	−2			26
	main	199	198			1			1
	twldrv	11,198,058	13,165,137	−11	−1	−5			−18
matrix300	sgemm	8,398	6,308	21	7	−3			25
	saxpy	46	38	9	9				17
tomcatv	tomcat	355,039,258	406,856,259	−7	−7	−0			−15

Table 7.4 Effects of SSA-Based Splitting

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Rematerial*	*Dominance*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
fmin	fmin	370	427	31	−21	−40	14		−15
rkf45	fehl	50	66	−32	4	−4			−32
seval	spline	102	53	53	6	−10	−2	1	48
solve	solve	85	90	−2	−2	−1			−6
	decomp	286	308	9	−3	−13	−1		−8
svd	svd	1,966	2,991	−15	−19	−17	−1		−52
zeroin	zeroin	234	128	36	8	−43	45		45
doduc	bilan	966	1,038	−5	−2		0		−8
	cardeb	83	155			−2	−84		−87
	coeray	133	141			−6	−2		$\star$
	colbur	24	29	−8		−4	−8		−21
	dcoera	15	11	53	13	−33	7		27
	ddeflu	375	298	13	15	−11	4		21
	debflu	1,131	1,273	0	0	−13	−0	0	13
	debico	418	606	6			−51		−45
	deseco	4,636	5,079	4	2	−3	−11	−2	−10
	drepvi	175	194	−1	−2	−4	−3		−11
	drigl	131	134			−2			−2
	dyeh	106	91	6	6	3			$\star$
	ihbtr	395	528	−1	−5	−32	4		−34
	inideb	48	117				−144		−144
	inithx	437	558		1	−5	−23		−28
	integr	372	447	20	8	−14	−34		−20
	inter	22	12	27	18				45
	lectur	166	171	17	2	−8	−21	7	−3
	paroi	1,383	1,738	−6	−4	−9	−2	−4	−26
	pastem	220	451	12	1	−57	−61		−105
	prophy	1,525	2,662	−9	−9	−43	−14	−0	−75
	repvid	404	537	13	7	−25	−28		−33
	saturr	106	141	−13	−5	−7			−33
	sortie	20	26			−10	−20		−30
	supp	149	110	17	17		−9		26
	yeh	290	269	18	2	−6			$\star$
fpppp	d2esp	34	28	12	6	−9	9		18
	efill	94	133	−9	23	−56			−41
	fmtgen	143	251	−24	−22	−47	−5		$\star$
	gamgen	2,083	4,478		0			−115	−115
	twldrv	11,198,058	25,826,498	−33	−37	−45	−15	0	−130
matrix300	sgemm	8,398	6,015	21	7	−0			28
	sgemv	1,808	2,109	−0		−17			−17
	saxpy	46	42	9	9	−9			9
tomcatv	tomcat	355,039,258	276,432,455	8	15	−0	−0		22

Table 7.5 Effects of Splitting at Dominance Frontiers

7.2 Priority-Based Coloring

An important alternative approach to global register allocation via graph coloring was developed by Chow and Hennessy [25, 26]. Given the existence of competitive techniques, it is naturally interesting to compare them. Unfortunately, it is difficult to perform useful comparisons of global register allocators. Ideally, each would run on the same machine, target the same machine, use the same input language produced by the same optimizer, and be implemented and tuned with equal care. While we have performed a large number of comparisons under exactly these strictures (see Section 7.1), our comparisons are all between relatively similar allocators; adding a significantly different allocator (*e.g.*, priority-based coloring) to the mix would require a large amount of time. In addition, it is difficult to achieve the best possible performance without extensive testing and tuning.

Nevertheless, some comparisons are possible. In the next two sections, we present a brief discussion comparing the priority-based coloring with the Yorktown allocator in terms of allocation quality and the results of an experiment comparing the two allocators in terms of speed.

7.2.1 Allocation Quality

While both allocators are based on graph coloring, they differ in most respects. Chow and Hennessy have published another comparison of the Yorktown allocator and priority-based coloring; the interested reader should consult their paper for another viewpoint [26, pages 513–517].

Intermediate Representation The Yorktown allocator works on code that has already been massaged into its final form; all optimization, address mode selection, and instruction scheduling has been completed (though instruction scheduling will be repeated after allocation) [6]. Chow's implementation of priority-based coloring runs much earlier in the compilation process, on a relatively high-level intermediate language [22]. This seems like a mere detail of implementation rather than a requirement of either allocation scheme. Certainly Larus and Hilfinger demonstrate that it is possible to do priority-based coloring on low-level code [51]. The advantage to using the low-level form is greater accuracy in allocation; the advantage to using the high-level form is allocation speed (typically many less live ranges will have to be handled) and greater machine-independence.

The Yorktown allocator accepts code referencing an unlimited number of *virtual* registers. The virtual registers are used to hold temporaries generated by the optimizer and variables allocated by the front-end. Note that the front-end must load all values into registers before they may be referenced (recall our assumption of a load-store architecture). This exposes the load instructions to optimization; for example, common subexpression elimination and loop-invariant code motion. The result is that even a global variable may appear temporarily (perhaps across a loop) in a virtual register and therefore be subject to allocation. As a result of allocation, some of the virtual registers may be spilled to memory. Priority-based coloring (as described by Chow and Hennessy) takes a different approach. Before allocation, their intermediate code has all variables and temporaries in memory. As a result of allocation, some of these values are promoted to registers. The effect of these different viewpoints on allocation quality is unclear. From a practical standpoint, Chow and Hennessy note that their scheme produces working code without ever running the register allocator.[28] Again, this is an implementation issue; Larus and Hilfinger's version of priority-based coloring is much closer to Chaitin's work in this respect.

Spilling and Splitting When the Yorktown allocator must spill, it spills an entire live range at once (modulo the local spilling considerations described in Section 2.2.5). There is no possibility of splitting a live range into two or more pieces, some spilled and others kept in registers. The great power of priority-based coloring is exactly this ability to perform live range splitting.

To support live range splitting, several tradeoffs seem to be required. The interference graph constructed by the Yorktown allocator is precise whereas the interference graph used during priority-based coloring is relatively coarse. In priority-based coloring, two live ranges appear to interfere if they share a common basic block. This is conservative, since one might be live only at the beginning and the other live only at the end, with no actual overlap at the instruction level. As a result, the graphs built by the Yorktown allocator will tend to have a smaller coloring, leading to less spill code. On the other hand, the coarse graphs constructed by priority-based coloring are relatively easy to update during live range splitting.

[28] On a load-store architecture, operands must appear in registers. In Chow's compiler, unallocated variables are loaded into temporary registers by the code generator whenever they are required as operands.

Coalescing Given such a coarse interference graph, it seems impossible to incorporate *coalescing* into priority-based coloring. Recall that coalescing examines pairs of live ranges connected by a copy. If the live ranges do not interfere with each other, they can be coalesced and the copy removed. However, in priority-based coloring, two live ranges connected by a copy *always* interfere (since they share a copy instruction, they must share a basic block and therefore interfere in the coarse graph).

We believe that coalescing is important. It is a powerful mechanism providing natural solutions to several real problems arising during code generation.

- Parameters passed in registers are naturally managed by coalescing. In priority-based coloring, these parameters are handled by a special-purpose mechanism called *precoloring* [26, Section 7.1].

- Coalescing removes unnecessary copies; Chow relies on a separate optimization called *copy propagation* [22, pages 39–41]. We note that coalescing *is* copy propagation. Furthermore, coalescing is a particularly effective form of copy propagation because it is performed at a very late stage of compilation. Finally, we note that Chow's approach to global copy propagation uses iterative data-flow analysis to attack a problem that is inherently not *rapid* [47]. In this case, the problem is formulated so that a conservative solution is found quickly. A more precise solution could require longer analysis times.

- Chaitin *et al.* describe how to use coalescing to handle 2-address instructions. Similar extensions can be used to manage idiosyncratic instructions that require their operands in particular registers. Priority-based coloring seems less adaptable to these situations.

Coloring Since the problem of finding a k-coloring for a graph is NP-complete, both allocators employ an efficient heuristic to determine an approximate solution. The Yorktown coloring heuristic is guided entirely by the structure of the graph; but, spills are determined by a combination of graph structure and estimated spill cost. Priority-based coloring is guided by a combination of estimated profit and live range size. An attempt is made to color high-priority live ranges first, where priority is determined by the apparent profit of allocating a live range to a register, with preference given to shorter live ranges. This approach seems weaker because it ignores the structure of the graph. As a result, we would expect the Yorktown coloring heuristic to find better colorings with less spilling. Of course, this tendency is mitigated (or perhaps overwhelmed) by the benefits of splitting instead of spilling.

Coverage Finally, we note that the Yorktown allocator controls the assignment of the entire register set. With priority-based coloring, the machine's register set is divided into three subsets:

locals Live ranges that include only one basic block are allocated, before coloring, into the set of local registers.

globals These are reserved for use by the coloring algorithm for live ranges that initially include more than one basic block.

temporaries A final group of registers is reserved for use by the code generator, since a few temporaries may be required when expanding the high-level intermediate language statements into their low-level assembly language components.

The size of each subset is fixed, potentially forcing further inefficiencies in register usage.

7.2.2 Allocation Time

While it is certainly possible to determine the asymptotic complexity of many of the sub-components of each allocator, it is difficult to draw useful conclusions about the relative speeds of each allocator in practice. In an effort to address this question, we have conducted an experiment attempting to determine the expected behavior of each allocator.

Methodology

The overall plan is to measure the time required to perform register allocation on a wide variety of routines, using both the Yorktown allocator and an allocator based on priority-based coloring. While we have written our own implementation of the Yorktown allocator, we rely upon Chow's own implementation of priority-based coloring that is embedded in the MIPS `f77` compiler.

This situation is not completely ideal. While we can give each compiler the same FORTRAN source, the code seen by the allocators is surely different in many respects. Perhaps more importantly, Chow's implementation allocates registers on fairly high-level intermediate code, whereas our allocator operates on code at the level of assembly language. The result is that our allocator must consider *many* more live ranges for a given routine. Note that this difference is a matter of implementation choice rather than an essential characteristic of the techniques.

Attempting to minimize unnecessary differences, we ported our code to a MIPS R3000, where all timings were performed. We configured our allocator to allocate registers for the MIPS register model (32 integer registers and 16 double-precision floating-point registers). Because our optimizer performs no loop unrolling, we suppressed loop-unrolling in the MIPS optimizer to avoid a gross mismatch in the amount of work presented to each allocator. The command used for the MIPS compiler was

`f77 -c -O2 -Wo,-loopunroll,0 -Wo,-l,`*log-file source-file*

where `-Wo,-loopunroll` controls loop-unrolling and `-Wo,-l` enables logging of various optimizer statistics. We used the `-Olimit` flag when necessary to force optimization of exceptionally large routines. All tests used version 2.20 of the MIPS compiler.

For test data, we collected 181 FORTRAN routines. The routines were drawn from several sources, including the SPEC benchmark suite, the LAPACK library, and the RiCEPS benchmark suite. The routines ranged in size from 48 bytes of object code to more than 41,000 bytes of object code.

Results

Figure 7.2 is a scatter plot summarizing the results of the experiment. Note immediately that both axes are log-scales. The vertical axis represents time required for allocation; the horizontal axis reflects the object code size (the size of the `.text` segment, as reported by the `size` command). The curves are created from the points of the scatter plot, smoothed by the `lowess` function provided as part of the S data analysis package [8, pages 497–498].

Each test was repeated ten times on an unloaded machine and the resulting times were averaged; each point in the scatter plot represents the average allocation time for a single routine. The timer granularity was 10 milliseconds. Reported times include cache miss times, but exclude paging [24]. The times shown for the Yorktown allocator include all control-flow and data-flow analysis, construction of the interference graph, coalescing, coloring, and spilling. The times shown for priority-based coloring are the sum of two times reported by the MIPS compiler:

`reg alloc preparation` This is the time spent in data-flow analysis and building all the data structures for register allocation, including the interference graph.

`global coloring` This is the time required for coloring and splitting.

Note that we have *not* included the cost of copy propagation when reporting times for priority-based coloring, though the Yorktown allocator achieves much of the same effect via coalescing. There are two reasons. First, it was impossible to separate the cost of local copy propagation from other local optimizations. Second, Chow's copy propagator actually propagates expressions, achieving a larger effect than that achieved by simply coalescing [22, pages 39–41].

Discussion

While there are a variety of points to be made about the data shown in Figure 7.2, the speed comparisons are the prime concern. We see that priority-based coloring is far faster on the smallest routines, sometimes as much as a factor of eight (remember we are using a logarithmic scale). At the other end of the scale, the Yorktown allocator is about six times faster than priority-based coloring on the largest routines. Of course, they meet in the middle, requiring similar time for routines with five to ten thousand bytes of object code (perhaps 300 lines of FORTRAN, depending on details of comments, declarations, and so forth).

Caveats This data should be interpreted carefully, with due consideration for the following points.

At the large end of the size scale, there are unfortunately few samples, primarily due to the difficulty of finding large routines. Furthermore, some of the largest routines (`fpppp` and `twldrv`) are somewhat artificial, with the bulk of their code contained in a few huge basic blocks. Nevertheless, we have specifically included them because of their notoriety among compiler writers.

We must also remember that the two allocators implement different algorithms. Priority-based coloring performs live range splitting, which we expect to allow more precise control of spilling; *i.e.,* better code quality. So, while priority-based coloring requires more time for large routines, the results may justify the effort.[29]

Complexity On a log-log graph, a straight line is defined by the equation $y = kx^c$ where c and k are constants defining the slope and position of the line, respectively.

[29]Note that the superiority of priority-based coloring is not assured; recall the discussion in Section 7.2.1.

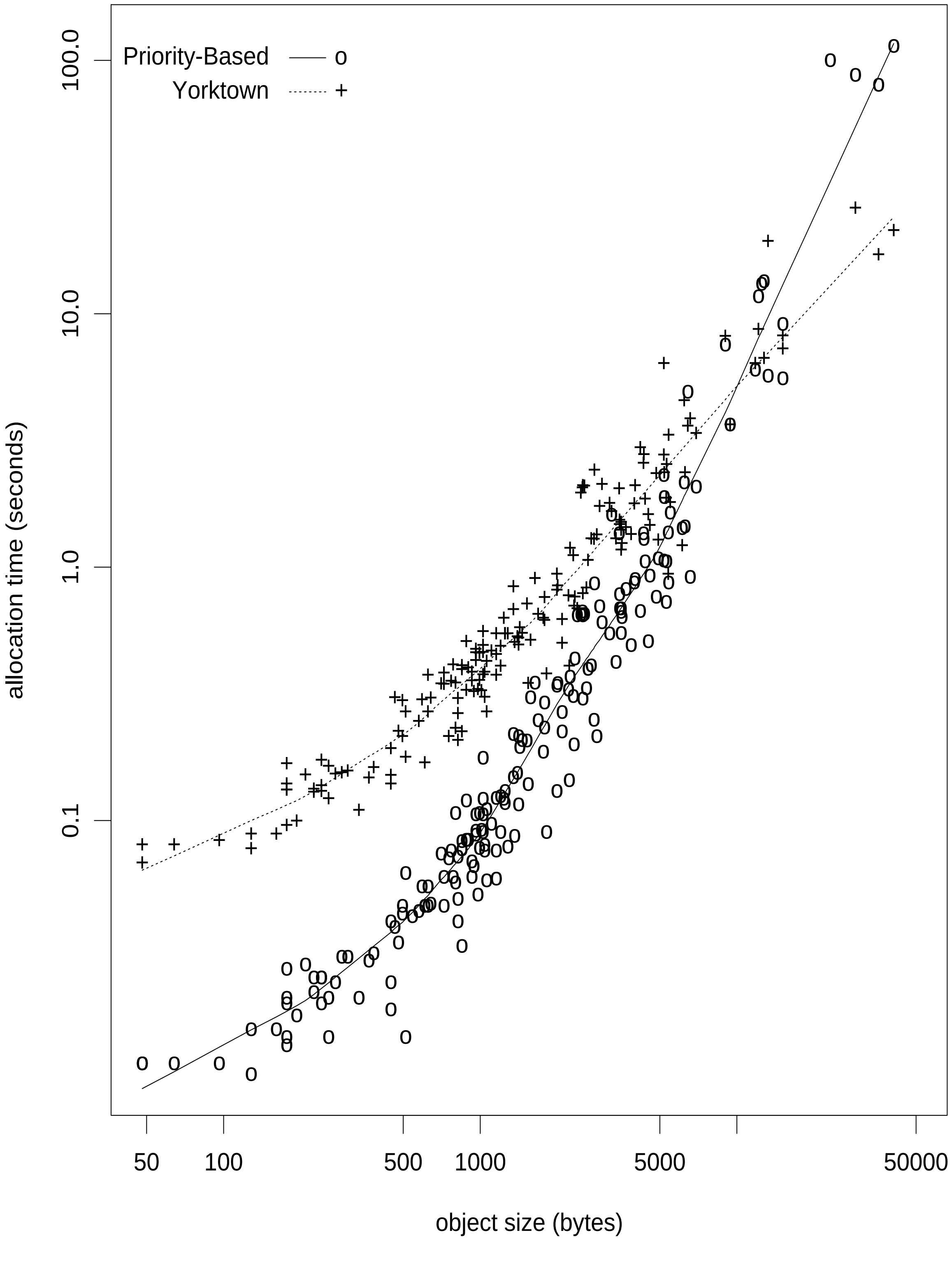


Figure 7.2 Allocation Times

The slope of the line is completely determined by the value of c. Abstracting to "big-O" notation, we obtain the equation $y = \mathrm{O}(x^c)$ where the constant c determines the slope. Therefore, by observing the slopes of lines representing the time required for allocation versus the size of the routine, we can determine each allocator's *observed time complexity*.

Priority-Based Coloring Figure 7.3 repeats the graph shown earlier, but includes only the data for priority-based coloring. The dotted line illustrates the slope of a line parallel to $y = x^2$. The distribution of measurements suggests that the time required for priority-based coloring is, in practice, $\mathrm{O}(n^2)$, where n is the size of the routine, with a constant-time overhead dominating the smallest cases.

The Yorktown Allocator Figure 7.4 shows only the data for our implementation of the Yorktown allocator. The dotted line illustrates the slope of a line parallel $y = x$ and the dashed line illustrates the slope of a line parallel to $y = x \log x$. The plots suggest that, in practice, the time required by the Yorktown allocator is $\mathrm{O}(n \log n)$, where n is the size of the routine. The flattening of the curve for the smallest routines is again explained by a constant-time overhead (a much greater overhead than required by priority-based coloring).

The wide variation in times for a given size routine may be in part explained by recalling that the number of iterations of the two major loops in the allocator (the *build-coalesce* loop and the *build-color-spill* loop) will vary independently of the routine size. Chapter 8 explores these questions and additional performance details in greater depth.

Conclusions Both techniques are fast, especially for small and medium-sized routines. Given the overall costs of compilation, there seems to be little excuse for not using one of the global allocation techniques in an optimizing compiler. Of course, as machines get faster, the time required for allocation will continue to shrink (consider the results in Chapter 8). Note also that we have considered only FORTRAN routines; presumably other languages with better facilities for expressing modularity will tend to have smaller routines, leading to even smaller allocation times in practice.

For small routines, Chow's implementation of priority-based coloring shows that it can be very fast indeed. Unfortunately, our implementation of the Yorktown allo-

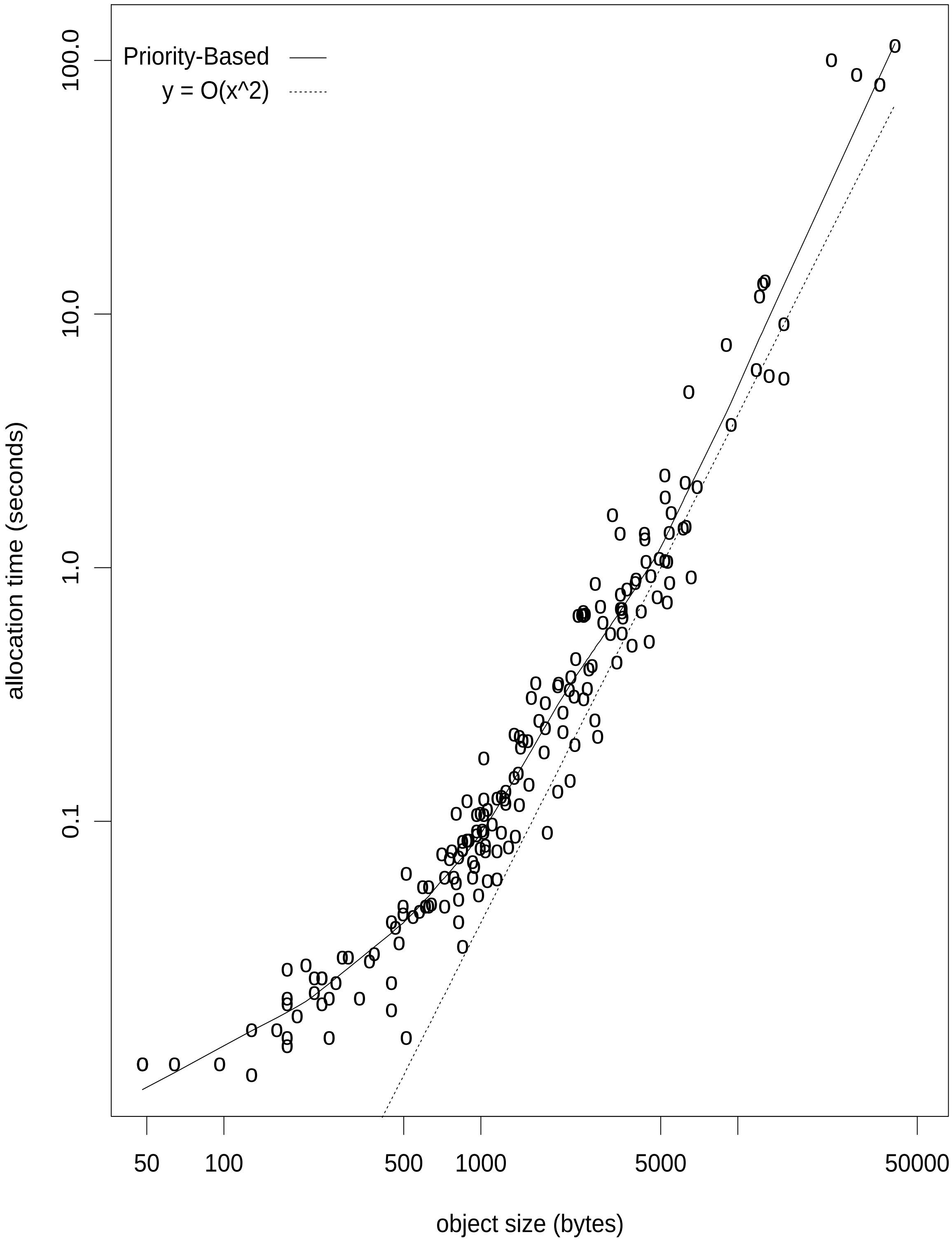


Figure 7.3 Observed Complexity of Priority-Based Coloring

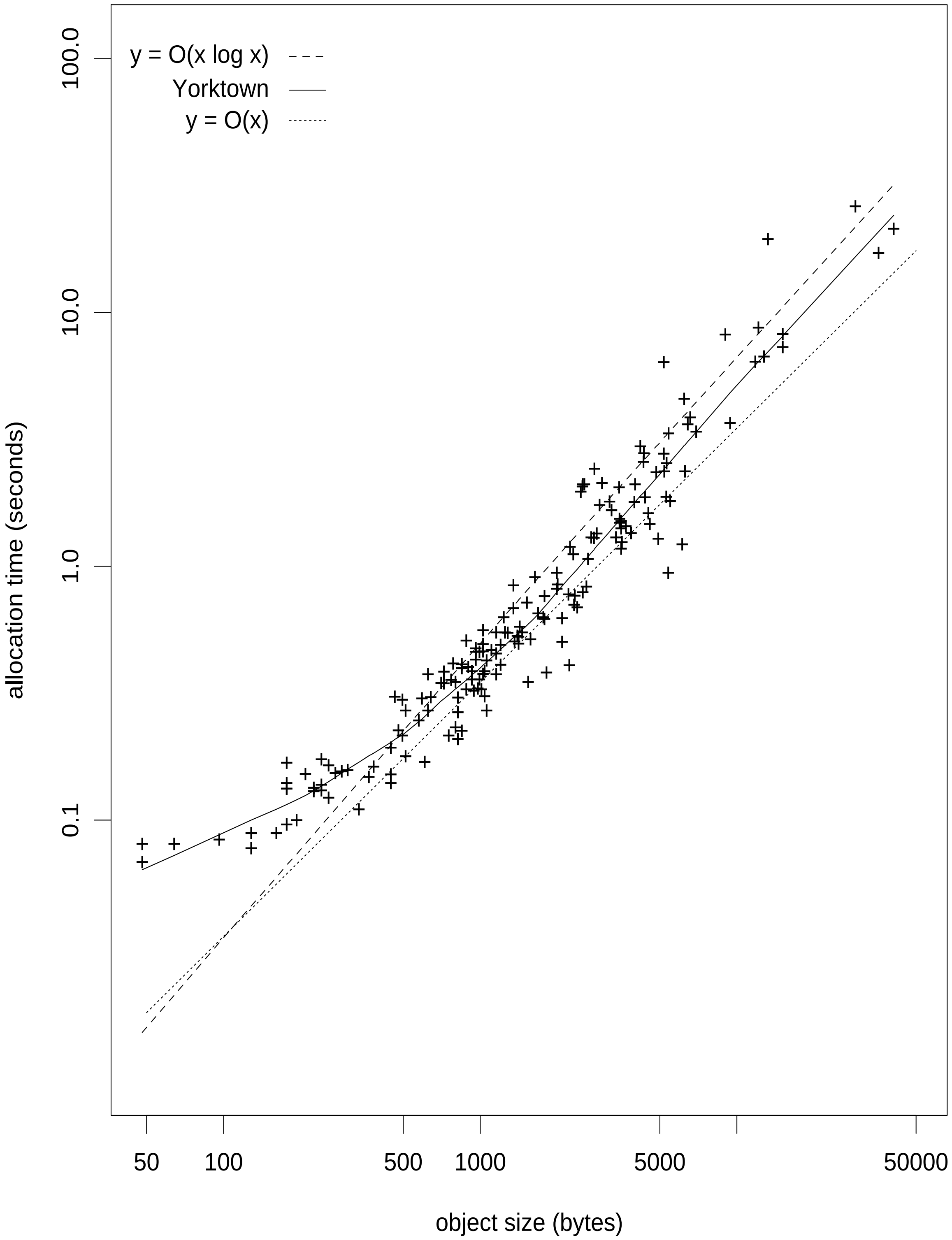


Figure 7.4 Observed Complexity of the Yorktown Allocator

cator seems slower. However, it is likely that this problem is susceptible to tuning.[30] For large routines, the $O(n^2)$ behavior of priority-based coloring makes its use problematic for production compilers. While tuning alone cannot improve its algorithmic complexity, it is possible that some form of "damage control" could be used to avoid explosive allocation times, trading away some allocation quality for improved allocation time.

[30]To be fair, it should be noted that we worked hard to make our implementation fast on large routines. On the other hand, relatively little attention was paid to handling small cases.

Chapter 8

Engineering

To support our experiments, we have written an optimizing compiler for FORTRAN. The optimizer is organized as a collection of independent programs, each accomplishing a distinct transformation. Each routine is represented in a low-level intermediate form, ILOC. Each pass reads in a routine via `stdin`, performs the necessary analysis and transformations, and writes the result to `stdout`. Since each pass consumes and produces ILOC, they may be organized in any order, with specific passes added, repeated, or omitted as desired.

After optimization, which includes passes to accomplish the effects of instruction selection, we run register allocation. The register allocator is also organized as an independent pass, consuming and producing ILOC, but it is always run after the optimization passes and may not be omitted or repeated.

ILOC is a low-level intermediate language designed to allow extensive optimization. It resembles the assembly language for a simple RISC machine, with the addition of hooks for interprocedural information and certain higher-level operations representing FORTRAN intrinsics. The design of ILOC and the entire optimizer was heavily influenced by the PL.8 compiler [6].

A multi-pass design, with many simple passes reading and writing ILOC, is not efficient in terms of compile time; however, it is ideal for our work. We are able to experiment with high-level issues involving the ordering of optimizations and with asymptotically efficient implementations of the individual optimizations.

A naive implementation of a coloring register allocator is not difficult to build; an efficient implementation is more challenging. Over the course of several implementation efforts, we have discovered efficient algorithms and data structures for many parts of the Yorktown allocator. While some of the ideas are surely known to other implementors, they do not appear in the literature. In the following sections, we describe many of the details of our implementation.

8.1 The Internal Representation

The ILOC for a routine is represented internally in a fairly straightforward fashion. As the routine is read in, a control-flow graph is constructed. The control-flow graph is a directed graph with a set of nodes, a set of edges, and a distinguished node *start*. Every node in the graph is reachable from *start*. Each node in the graph represents a basic block. Each block has:

- a list of labels,
- a set of incoming edges,
- a set of outgoing edges,
- a sequence of instructions, and
- slots for holding the results of various forms of analysis.

The start block actually contains no instructions; instead, it is artificially inserted as a predecessor to all entry points in the routine.[31] Each edge in the graph is represented explicitly. An edge contains:

- a source block,
- a destination block, and
- slots for holding the results of various forms of analysis.

Figure 8.1 shows the C declarations for blocks and edges. Notice that the predecessors of a block are organized as a linked-list. Note also that each edge is simultaneously the predecessor of a block and the successor of a block; therefore, each edge is part of two lists.

The instructions in a basic block are organized in a doubly-linked list to allow constant-time insertion and deletion. The list is also circularly-linked to allow easy traversal in either direction. By convention, each block has at least one instruction: an unconditional jump, a conditional branch, or a return. Therefore, the header of the circularly-linked list can always be the terminating instruction.

Individual instructions also have information about the kind of instruction and the exact registers and constants used in the instruction. Figure 8.2 illustrates how

[31] Recall that we are dealing with FORTRAN routines that may have more than one entry point to a given subroutine.

```
typedef
  struct {
    Labels labels;
    Edges preds;
    Edges succs;
    Blocks next;
    Insts inst;
    ...
  } Block, *Blocks;
```

```
typedef
  struct {
    Blocks pred;
    Blocks succ;
    Edges nextPred;
    Edges nextSucc;
    ...
  } Edge, *Edges;
```

Figure 8.1 Blocks and Edges

```
typedef
  struct {
    Insts prevInst;
    Insts nextInst;
    Kinds kind;
    short parm[4];
    ...
  } Inst, *Insts;
```

Figure 8.2 Instructions

instructions are declared. Parameters are stored in a short vector at the end of the structure. If more than four are required, we simply allocate extra space and address beyond the end of the array. Parameters are either register numbers or short indices into a table of constants.

Each instruction `kind` is simply a pointer into a constant table of information about each possible ILOC instruction. The declaration of `Kind` is shown in Figure 8.3, along with an excerpt of the constant instruction table. Each kind of instruction has a string representation and defines zero or more registers, uses zero or more registers, and employs zero or more constants. We use an incremental style (due to Chaitin [18]) to specify the number of each parameter. For example, the `add` instruction defines one register and uses two registers – the defined register is found in `parm[0]` and the used registers are found in `parm[1]` and `parm[2]`. This style of specification avoids the need for many additions when examining instructions. The `details` field is simply used as a small bit vector for compactly representing a variety of useful facts.

While the control-flow graph is relatively stable, many instructions are created and deleted during the course of allocation. In addition, many analysis results must be recorded temporarily – usually in small structures hung off relevant blocks or edges. Storage management is therefore an important concern. We have tried several schemes. Initially, we used a variety of *ad hoc* approaches, handling each problem as it appeared. When this proved difficult to maintain, we took advantage of a conservative garbage collector, suitable for use with arbitrary C programs [10]. In our most recent implementations, we have experimented with a fast, phase-oriented approach to memory management promoted by Hanson [42].

```
typedef
  struct {
    char *str;
    char defs;
    char uses;
    char parms;
    char details;
  } Kind, *Kinds;
```

```
Kind table[] = {
  ...
  { "add", 1, 3, 3, 0 },
  { "mv",  1, 2, 2, COPY },
  { "ld",  1, 2, 2, MEM },
  { "ldi", 1, 1, 2, 0 },
  { "st",  0, 2, 2, MEM },
  { "bc",  0, 0, 1, LAST|COND },
  ...
}
```

Figure 8.3 Instruction Kinds

8.2 Set Representations

Sets are a fundamental abstraction widely used in programming. Many implementations are possible, each offering different advantages. The choice of implementation for a given set depends on the exact operations required and their relative frequency.

When building the interference graph, we require fast versions of *clear-set, add-member,* and *delete-member,* as well as a quick way to enumerate the members of a set. Bit vectors are inefficient in this application since they require $O(u)$ time to clear the set and to enumerate the members. Here u is the size of the universe. This requirement is distressing when the average number of members in a set is small, but the universe is relatively large. We have developed an alternative implementation that supports these operations efficiently.

Using the matrix initialization idea suggested in a homework problem by Aho, Hopcroft, and Ullman [1, problem 2.12], we are able to achieve constant-time operations for *clear-set, add-member, delete-member, member?, choose-one,* and *cardinality*. Furthermore, the following operations may be performed in $O(n)$ time, where n is the number of members: *set-union, set-intersection, set-difference, set-copy, set-equality,* and enumeration of the members. *Set-complement* requires $O(u)$ time, but is rarely necessary given the existence of an efficient *set-difference*. Of course, the tradeoff is that the new representation requires much more space than a bit vector – in our case 32 times more space. Nevertheless, for many applications, this tradeoff is justified. In our implementation, we use up to four of these sets simultaneously. Together, they typically comprise about 1% of the total working set during allocation. Furthermore, it is easy and efficient to convert between our set representation and a packed vector form (a vector of shorts containing only the members of the set).

In our representation, a set has three components: two vectors, each u elements long, and an additional scalar that records the number of members in the set. Figure 8.4 illustrates an example set with a single member 3. The scalar `members` delimits the initialized portion of the `stack`. Initialized elements in `stack` point to members in `array`, which point back into the stack. To test for membership of some number in the set, we can use the following routine:

```
int member(Set s, int n) {
   int a = s->array[n];
   return 0 <= a && a < s->members && s->stack[a] == n;
}
```

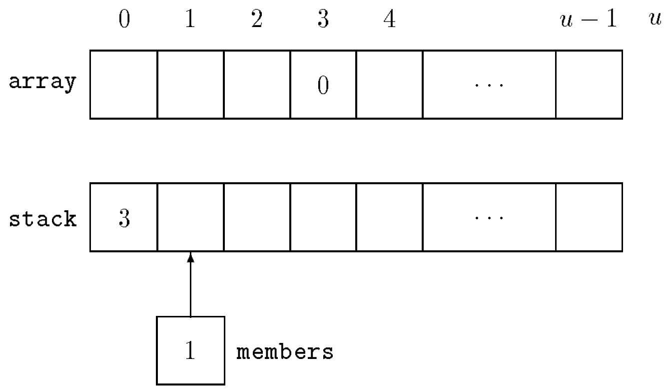


Figure 8.4 Set Representation

Since all members of the set appear between 0 and `members` in `stack`, clearing the set requires simply setting `members = 0`. Enumeration of all the members is accomplished by iterating over the elements of `stack`. Adding a member involves first checking for membership and then extending the stack to point at the new member. Similarly, the appropriate entry in `array` is made to point at the new stack entry. For a final example, consider the code for deleting a member:

```
void deleteMember(Set s, int n) {
   int a = s->array[n];
   int m = s->members;
   if (0 <= a && a < m && s->stack[a] == n) {
      int e = s->stack[--m];
      s->members = m;
      s->stack[a] = e;
      s->array[e] = a;
   }
}
```

In this case, we first check for membership, then pop one element `e` from the stack. The popped element replaces the deleted member in `stack[a]` and the link from `array[e]` to `stack[a]` is updated.

8.3 Liveness

To support *renumber*, *build*, *spill costs*, and *spill code*, we must know which values are live at each point in the routine. More precisely, for *renumber*, we must know which virtual registers are live at each point; for *build, spill costs,* and *spill code,* we need to know which live ranges are live at each point. In our implementation, we are able to translate from virtual register numbers to live range indices. Thus, we calculate liveness information in terms of virtual registers before each invocation of *renumber*; all other phases are supported by the results of this analysis. Note however, that the computation of live virtual registers must be repeated after each invocation of *spill code,* since spilling alters the results.

Our early implementations employed iterative data-flow analysis to solve the liveness problem [4, Section 10.6]. We used a traditional bit-vector representation for sets of live registers, where the liveness of each virtual register was represented by a bit. This approach requires two bit vectors per basic block, with $\lceil n/32 \rceil$ words per bit vector, where n is the number of virtual registers. We also face the problem of traversing the bit vectors when building the interference graph. While we can use the set representation discussed in Section 8.2 for handling basic blocks efficiently, we must somehow convert between representations.

Choi, Cytron, and Ferrante describe a method for performing data-flow analysis using *sparse data-flow graphs* [21]. Their approach is closely related to the work on SSA, but allows solution of forward and backward data-flow problems. Our positive experiences with SSA and our frustration with the bit-vector approach prompted consideration of their sparse approach. While a description of their approach is outside the scope of this thesis, there are a few points that deserve mention:

- We make an initial pass over the code to discover which virtual registers are live at the beginning of at least one block; other registers are uninteresting globally. This same information is used to speed construction of the pruned SSA graph.
- We build all the sparse graphs at once, using a single walk of the dominator tree (the procedure *Search* [21, Figure 3]). This speeds construction, but requires more space.[32]

In limited tests, the sparse approach seems to be slightly faster than the bit-vector implementation. The exact tradeoff depends on the machine (the bit-vector approach benefits from long cache lines) and the code being analyzed. The sparse approach seems require about half the memory of the bit-vector approach, depending on the sparsity of the problem. Finally, the sparse approach enables efficient initialization of the live sets when building the interference graph.

[32]Ron Cytron pointed out this possibility and clarified several details in the paper.

8.4 Live Ranges

In our implementation, the handling of virtual registers, values, and live ranges is complex and requires explanation. In all cases, the operands of each ILOC instruction are small integers; but the meaning of the integers changes from phase to phase. Recall the flow graph shown in Figure 3.3. On input to *renumber,* the routine is always expressed in terms of virtual registers, where a given register number may appear in many places and have many distinct uses. During the *build-coalesce* loop, the integers represent values. Each definition in the routine introduces a distinct value; each value is represented by a distinct integer. The values serve as indices into a disjoint-set structure, where each set represents a different live range. The meaning of the routine may not be recovered from the values alone; the disjoint-set tree is required. Before *spill costs,* the routine is rewritten in terms of live ranges and the disjoint-set tree is discarded. A single live range is a connected web of definitions and uses. Each live range is numbered uniquely and the definitions and uses refer directly to the live range. Of course, spilling a live range can cause it to become disconnected; therefore, we say that the routine, after *spill code,* is again expressed in terms of virtual registers. Finally, after successfully coloring the interference graph, the routine is rewritten in terms of machine registers.

Values and live ranges are determined by *renumber.* Our implementation is based upon construction and manipulation of the pruned SSA graph [21]. The SSA form is constructed as described by Cytron *et al.,* except that no dead ϕ-nodes are inserted [29]. Furthermore, when renaming the virtual registers, we simply use distinct integers for each definition [29, Figure 12]. Thus, each integer corresponds to a single definition (perhaps a ϕ-node) and a single value.

Live ranges are composed of values connected by common uses. In the pruned SSA graph, values are connected only by ϕ-nodes. To determine live ranges, we union the operands of each ϕ-node together with the value defined by the node, using a fast disjoint-set union [1, Section 7.4]. Once all unions have been performed, the ϕ-nodes can be discarded. To determine the live range for a given value, we would perform a *find* operation. However, we do not immediately rewrite the routine in terms of live ranges; instead, we maintain the disjoint-set tree for use during *coalesce.*

In Chapters 5 and 6, we used SSA to guide the placement of split points. In those cases, we were primarily taking advantage of the fact that ϕ-nodes are placed at dominance frontiers, though the propagation of rematerialization tags uses the single-birthpoint property of SSA to achieve asymptotic efficiency. To compute live ranges, we are again taking advantage of the single-birthpoint property to achieve efficiency. Without the ϕ-nodes, we would be required to union all the definitions reaching each use. With ϕ-nodes, we are only required to union definitions reaching each ϕ-node.

8.5 The Interference Graph

The flow graph shown in Figure 2.1 is slightly misleading in its depiction of the *build-coalesce* process. The actual process is sketched more precisely here:

```
⟨allocate space for the bit matrix⟩
⟨fill in the bit matrix, accumulating the degree of each node⟩
do {
    ⟨allocate adjacency vectors based on the degree of each node⟩
    ⟨clear the bit matrix⟩
    ⟨fill in the bit matrix and the adjacency vectors⟩
    improved = coalesce();
} while (improved);
⟨free space for the bit matrix⟩
```

We allocate $\lceil n^2/16 \rceil$ bytes for the triangular bit matrix, where n is the number of live ranges (including the machine registers). When adding an edge between the two live ranges i and j to the bit matrix, we first check to see if the corresponding bit is already set. If the bit is not yet set, we set it and increment the degree of both i and j. The specific bit is indexed by $i + j^2/2$, where $i < j$. Of course, we must find the correct byte and select the specific bit.

When allocating the adjacency vector for a node n, we allocate 2 bytes for each edge incident on n. This allows room for an index to the interfering node, though not a pointer. Of course, this policy also limits our implementation to graphs of less than 65,536 nodes. In practice, this limit has not been approached (consider the examples, especially `twldrv`, in Section 8.9).

To fill in the interference graph, we must examine every basic block in the flow graph (in some arbitrary order) and walk over the instructions of the block in reverse order. As we walk over the instructions, we maintain a set `live` containing all the live ranges that are both *live* and *available* at that point in the routine. The set is initialized from the global information collected earlier.

```
for each block b in the flow graph {
   ⟨build the set live from b->liveOut⟩
   i = b->inst;
   do {
      ⟨examine instruction i updating graph and live⟩
      i = i->prevInst;
   } while (i != b->inst);
}
```

The structure `b->liveOut` is a packed vector containing the set of all values that are *live* and *avail* at the end of the block. For each value in the vector, we find the associated live range and add it to the set `live`.

Each instruction is handled by making any defined live ranges interfere with all the members of `live`. Then all defined live ranges are removed from `live` and all used live ranges are added to `live`.

```
ik = i->kind;
if (ik->details & COPY)
   deleteMember(live, find(i->parm[1]));
for (p=0; p<ik->defs; p++)
   addEdges(graph, live, find(i->parm[p]));
for (p=0; p<ik->defs; p++)
   deleteMember(live, find(i->parm[p]));
while (p < ik->uses)
   addMember(live, find(i->parm[p++]));
```

Note the use of `find` to compute the live range number from the values stored in each instruction.

Copy instructions are handled specially to avoid adding an undesirable interference between the source and destination (recall the discussion in Section 2.2.2).

We call `addEdges` to insert an entire set of interferences for a single live range. In practice, this allows some loop-invariant computations to be preserved as individual edges are added. The alternative, a loop over the members of `live` repeatedly calling `addEdge`, is much slower. In our implementation, we have two routines for adding edges to the interference graph: the first only sets bits in the bit matrix; the second sets bits in the bit matrix and adds edges to the adjacency vectors. These are passed, one at a time, as procedure parameters defining `addEdges` to a single routine that traverses the control-flow graph, recording interferences.

Handling Multiple Register Classes

Many architectures provide more than one class of register. In some cases, the classes might be distinct; for instance, the floating-point and integer registers provided by the RS/6000 are completely separate. In other cases, two classes may overlap, completely or partially. The classes are determined by the operations possible on the registers; for example, on the RS/6000, floating-point addition is only possible for operands contained in floating-point registers. On the 68020, addition is possible with the entire set of general-purpose registers; but multiplication is only possible on the data registers and only the address registers may be used with the auto-increment addressing mode. Thus, the register classes are architecture dependent and are defined by the restrictions imposed by the instruction set.

Unfortunately, it is sometimes difficult to determine the register classes defined by a particular architecture. We have no specific methodology beyond careful examination of the instruction set and addressing modes. Note though that instructions where a particular register is required (versus a limited set of registers) are handled via coalescing.

We specify the class of each live range with a small bit vector (usually contained in a single byte). For the sake of a concrete example, consider the register set provided by a combination of the Motorola 68020 and the 68881 floating-point coprocessor. There are three basic classes of register: address registers, data registers, and floating-point registers. We assign one bit to each register class: say 1, 2, and 4, respectively. Therefore, a live range that could be assigned to either an address register or a data register would have a class of 3. We also construct a table of all machine instructions, specifying the largest possible class of each register operand. Thus, the source and result of a register-to-register integer add instruction would have a class of 3, whereas the operands of a floating-point add instruction would have a class of 4. Some instructions should have unrestricted classes. For example, the source and destination of a copy instruction should have class of 7. Similarly, the source of a store instruction and the result of a load instruction should have a class of 7. This policy allows maximal freedom when coloring.

The class of each live range is determined by examining all the instructions that use it and intersecting their requirements. This is accomplished in a single pass over the routine before constructing the interference graph. Given live ranges of many classes, we can be more precise about when two live ranges interfere. We say that two live ranges interfere only if the intersection of their classes is not zero. Thus, floating-point live ranges do not interfere with integer live ranges in our example. Intuitively, this makes sense – they are not competing for the same set of machine registers.

The class of each live range is also used to guide insertion of edges between the live range and the machine registers. For example, we force the insertion of edges between a floating-point live range and all the integer machine registers. This ensures that the floating-point live range will not be assigned to an integer machine register. We add an edge between a live range and a machine register if the intersection of their classes is zero.

8.6 Coalescing

The *coalesce* phase is implemented as a single traversal of the control-flow graph, examining all the instructions in each basic block and removing copies when possible. Because some coalesces can preclude others, we visit high-priority blocks first, where priority is determined by loop-nesting depth. For each instruction `i`, we perform the following steps:

```
if (i->kind->details & COPY) {
   dst = pathCompress(i->parm[0]);
   src = pathCompress(i->parm[1]);
   if (src == dst) ⟨unlink i from the block⟩
   else (!interfere(graph, src, dst)) {
      father = min(src, dst);
      son = max(src, dst);
      union(father, son);
      ⟨update graph so that father contains all the edges from son⟩
      ⟨unlink i from the block⟩
   }
}
```

Deleting redundant copy instructions from the block is accomplished by unlinking `i` from the doubly-linked list of instructions. Combining the sets of edges is more complex. We always link the `son` to the `father`'s tree, since the first k live ranges are reserved for machine registers and should always take precedence. Initially, each node has a single adjacency vector containing all the neighbors. As nodes are coalesced, the adjacency vectors are linked together; therefore, we assume all nodes have lists of adjacency vectors. As the list is moved from `son` to `father`, we traverse each adjacency vector, checking for new edges. If an edge is new to the `father` (determined by examining the bit matrix), we increment the degree of `father` and set the appropriate bit in the bit matrix.

If any unions are performed during a pass of *coalesce,* the interference graph is rebuilt and coalescing is repeated. Of course, the lists of adjacency vectors are freed before the graph is rebuilt and a new adjacency vector of the correct size is allocated for each node. The size is determined by each node's degree.

Note the use of `pathCompress` here compared to our use of `find` during *build.* Our implementation of `find` is simply a macro that does a single indirection to find the root of the appropriate union-find tree; `pathCompress` is a recursive procedure that must search up to the root of the tree, compressing the path upon return. Since *build* only accesses the union-find structure, we compress all paths before building the graph. On the other hand, *coalesce* repeatedly modifies the structure (via `union`), exposing new opportunities for incremental path compression when accessing the union-find structure.

Uses of Coalescing

In Section 2.2.4, we claimed that coalescing served many purposes besides the obvious one of removing unneeded copy instructions. In the following sections, we describe how we use coalescing to simplify handling of parameters passed in registers and to handle the special register requirements of certain architectures.

Passing Parameters in Registers To enable separate compilation, a compiler adheres to a *calling convention,* usually established by the machine's architect. The convention will specify the location of the return address, the stack frame layout, which registers are preserved across a call, and the location of parameters and function return values. In many RISC machines, the first few parameters will be passed in specific registers and a function value will be returned in a particular register.

We handle parameters passed in registers by introducing copies between the appropriate machine registers and virtual registers. Figure 8.5 illustrates a simple example. The left column shows a small C routine and the middle column shows an ILOC implementation of the same routine. In the uppermost fragment of ILOC, three `mv` instructions are used to copy the arguments from their machine registers (in this case, `r0`, `r1`, and `r2`) into virtual registers. The second ILOC fragment implements the call to `bar`, including a copy of the argument into the correct machine register

```
int foo(int x,
        int y,
        int z) {
   return bar(x + y) + z;
}
```

```
foo: frame
     mv  r50 r0
     mv  r51 r1
     mv  r52 r2

     add r53 r50 r51
     mv  r0  r53
     bs  bar
     mv  r54 r0

     add r55 r54 r52

     mv  r0  r55
     rtn
```

```
foo: frame
     mv  r52 r2
     add r0  r0  r1
     bs  bar
     add r0  r0  r52
     rtn
```

Figure 8.5 Passing Parameters in Registers

and a second copy to recover the result of calling `bar`. The last fragment copies the expression result into the machine register used to hold function values on return. The rightmost column shows the same code after coalescing. For this example, we assume that `r2` is not preserved across procedure calls; therefore, `r52` and `r2` would interfere, preventing their coalesce. During coloring, `r52` would be assigned to a "callee-saves" register (a register preserved across call sites) or spilled (if no such register was available).

In our implementation, we do not allow machine registers to be spilled; therefore, we must be careful when coalescing a long-lived virtual register and a machine register. Without coalescing, a long-lived but little-used live range might be spilled. If coalescing is allowed, the resulting unspillable live range may constrain the interference graph, provoking many more spills. Our approach to the problem is to limit the amount of coalescing that occurs on the first pass through the *build-color-spill* loop. Specifically, we avoid coalescing copies introduced to handle incoming parameters, since these are likely to be long-lived. After the first pass introducing spill code, we allow these copies to be coalesced, assuming that the first pass spilled most of the unimportant live ranges. As a consequence, we force the allocator to run for at least two passes on every routine.

Handling Instruction Constraints Some architectures specify that certain instructions take their operands in specific registers or produce their results in a specific register. These requirements can be handled exactly like the call to `bar` in Figure 8.5.

Two-address instructions are slightly more complex. In many architectures (*e.g.*, the Motorola 68000 family), arithmetic instructions are 2-address instead of 3-address. In other words, instead of supporting a general form of $r_0 \leftarrow r_1 + r_2$, they require the result and the first operand to use the same register; *e.g.*, $r_1 \leftarrow r_1 + r_2$. In ILOC, the operands are never overwritten by the result. This policy enables easy reuse of any expression value; however, we are sometimes faced with the problem of expressing ILOC's 3-address instructions in terms of simpler 2-address instructions. Chaitin *et al.* point out that we can handle such constraints by attempting to coalesce the result live range with the appropriate operand. Furthermore, if the operation is commutative, we can attempt to coalesce with either operand. If the coalesce is successful, we have a trivial mapping to the 2-address assembly instruction. If the result cannot be coalesced with either operand, an additional copy instruction must be introduced to avoid overwriting the operands. For example,

```
add  r0  r1  r2
```
i.e., $r_0 \leftarrow r_1 + r_2$

becomes

```
mv   r0  r1
add  r0  r0  r2
```

This extra copy is introduced at the last stage of allocation, after coloring has been successfully completed.

8.7 Spill Costs

The correct computation of spill costs is important, since they control the choice of spill candidates during coloring. Chaitin describes several special cases that must be handled correctly when generating spill code (recall the discussion in Section 2.2.5); these same considerations apply when estimating spill costs. Furthermore, we must account for the possibility of rematerialization. Chaitin also points out that spilling the source or destination of a copy instruction means that the copy instruction can be deleted. Finally, if aggressive live range splitting is employed, we must account for the correct handling of split instructions (Section 6.2.2).

For each live range, we accumulate the costs due to required loads, required stores, and copies saved. Of course, the costs are simply estimates of the actual run-time costs; in our implementation, we weight the number of instructions required by 10^d, where d is the loop-nesting depth of each instruction. Additionally, we weight loads and stores by an additional factor of 2 (compared to copies, load-immediates, and add-immediates). Production implementations might weight loads more heavily, though instruction scheduling can often reduce load costs.

Our implementation of *spill costs* is organized as a single pass over the blocks in the control-flow graph, where the instructions in each block are traversed in reverse order.

```
for (r=0; r<ranges; r++) {
   range[r].loads = 0.0;
   range[r].stores = 0.0;
   range[r].copies = 0.0;
   range[r].infinite = FALSE;
}
for each block b in the flow graph {
   clearSet(needLoad);
   ⟨build the set live from b->liveOut⟩
   copySet(mustSpill, live);
   i = b->inst;
   do {
      ⟨examine instruction i updating sets and accumulating costs⟩
      i = i->prevInst;
   } while (i != b->inst);
   for all members r of needLoad
      range[r].loads += b->depth;
}
⟨summarize costs for each live range⟩
```

We maintain a set `needLoad` of live ranges that are live and have been used since the last death.[33] At any death, or the beginning of the block, we charge a load to each member of the set.

Live ranges that are live across a death are added to the set `mustSpill`. At a definition of the live range `i`, if `i` is not a member of `mustSpill`, then the definition must be "close" to all uses of `i` and `range[i].infinite` is set.

Finally, we maintain the set `live` throughout the basic block to enable detection of deaths. For each instruction `i`, we perform three basic steps:

```
ik = i->kind;
⟨handle definitions of i⟩
⟨check for deaths⟩
⟨handle uses of i⟩
```

[33] A death is the last use of a live range in a basic block.

At each definition, we must update `live` and `needLoad`. Additionally, we must check for membership in `mustSpill`.

```
for (p=0; p<ik->defs; p++) {
   r = i->parm[p];
   if (member(needLoad, r)) {
      deleteMember(needLoad, r));
      if (!member(mustSpill, r)) range[r].infinite = TRUE;
   }
   range[r].stores += b->depth;
   deleteMember(live, r);
}
```

Checking for deaths is accomplished by examining each use and testing for membership in `live`. If a death is found, we must update `mustSpill` and clear `needLoad`.

```
for (p=ik->defs; p<ik->uses; p++) {
   r = i->parm[p];
   if (!member(live, r)) {
      for all members m of needLoad {
         range[m].loads += b->depth;
         addMember(mustSpill, m);
      }
      clearSet(needLoad);
   }
}
```

The uses must be re-examined in order to update `live` and `needLoad`.

```
for (p=ik->defs; p<ik->uses; p++) {
   r = i->parm[p];
   addMember(live, r);
   addMember(needLoad, r);
}
```

After the entire routine has been examined, we are able to summarize the cost of each live range.

```
for (r=k; r<ranges; r++) {
   if (range[r].lattice == BOT)
      range[r].cost = 2.0 * (range[r].loads + range[r].stores);
   else
      range[r].cost = range[r].loads - range[r].stores;
   range[r].cost -= range[r].copies;
}
```

Note that we correctly account for the rematerialization tag for each live range. If `range[r].infinite` is set, then the live range is treated as though it had infinite spill cost and the computed cost is ignored.

This description is necessarily simplified. We have ignored the possibility of different classes of registers. This is primarily important when updating `needLoad` and `mustSpill` after discovering a death. For example, it is clear that the last use of a floating-point live range provides no new allocation opportunities for an integer live range; therefore, only floating-point live ranges should be added to `mustSpill` and deleted from `needLoad`.

The structure of *spill code* is very similar to *spill costs*. Rather than counting required loads and stores, the actual spill instructions are inserted for spilled live ranges. Similarly, the instructions defining live ranges that are both spilled and rematerializable are deleted.

8.8 Coloring

Coloring occurs in two phases: *simplify* and *select*. Each phase runs in $O(n + e)$ time, where n is the number of nodes in the interference graph and e is the number of edges.

In preparation for *simplify*, we classify each node based on its degree and its spill cost. All nodes of degree $< k$ are placed in a set `low`. Nodes of degree $\geq k$ and finite spill cost are placed in a set `high`. Nodes of high degree and infinite spill cost (recall the discussion in Section 8.7) are not included in either set.

Simplify builds a stack `s` containing all the nodes, including spill candidates. The bulk of *simplify* is a single loop:

```
loop {
   while (members(low) > 0) {
      m = chooseOne(low);
      deleteMember(low, m);
      ⟨remove m from the graph, updating low and high⟩
      push(s, m);
   }
   if (members(high) == 0) break;
   ⟨select a spill candidate m from high⟩
   deleteMember(high, m);
   addMember(low, m);
}
```

To remove a node `m` from the graph, we visit each neighbor `n` (using the adjacency vectors constructed in Section 8.5) and decrement its degree. If the new degree of `n` is $k-1$, then `n` is removed from `high` and added to `low`.

To select a spill candidate, we consider all the nodes in `high`, where `high` contains all nodes of finite spill cost remaining in the graph. We choose the node with the smallest ratio of spill cost to degree.[34] In our implementation, we simply examine all the nodes in sequence. This can be expensive if many spill candidates must be selected; additionally, it does not fit within the $O(n+e)$ time bounds for *simplify*. While other implementations are imaginable, we have not explored the problem thoroughly.

The implementation of *select* is straightforward. Each node is initially given the color k (where final colors will be in the range $0 \ldots k-1$). The body of *select* is a single loop:

```
while (!empty(s)) {
   m = pop(s);
   if (!range[m].infinite && range[m].cost <= 0.0)
      range[m].spill = TRUE;
   else {
      for (c=0; c<k; c++) used[c] = FALSE;
      for all the neighbors n of m
         used[range[n].color] = TRUE;
      used[k] = FALSE;
      c = 0;
      while (used[c]) c++;
      if (c < k) range[m].color = c;
      else range[m].spill = TRUE;
   }
}
```

In our implementation, we attempt to use colors from the set of "callee-saves" registers last. These are preserved across call sites and should be reserved as long as possible for use by long-lived live ranges. We do not preserve "caller-saves" registers across a call site; instead, we recognize that they are killed by the call and force them to interfere with live ranges that are live across the call site (recall our handling of `r52` in the example shown in Figure 8.5).

[34] Ben Chase reminded us to use "cross multiplication" when comparing two fractions.

8.9 Compile-Time Characteristics

Table 8.1 illustrates the compile-time performance of the optimistic allocator on three routines from the SPEC benchmark suite. The left-most column describes the allocator phase being measured (recall the flow graph shown in Figure 3.3). The individual phases are grouped together, where each group represents one iteration of the major *build-color-spill* loop. Thus, we see that the allocator was required to spill twice for `repvid`; on the third iteration, a coloring was discovered. On the other hand, `tomcatv` required an additional iteration to converge.

All times are given in seconds. The times were collected via calls to system-supplied timer routines, based on a 100 Hz clock, on an unloaded IBM RS/6000 model 540. The allocator was run ten times on each routine and the resulting times were averaged.

The *control-flow* row gives the time required for control-flow analysis, including construction of the forward and reverse dominator trees and the forward and reverse dominance frontiers. The times given for *renumber* include the time required for live analysis and construction of the pruned SSA graph. The *color* row gives times showing the combined cost of *simplify* and *select*. In addition to the time costs, the *spill code* entries also show the number of live ranges spilled.

The *build-coalesce* entries are more complex. The first row reports the time for the entire *build-coalesce* loop and the number of bytes required for the triangular bit matrix. The additional rows report the number of bytes required for the adjacency vectors and the current number of live ranges. Each pass of coalescing will decrease the number of live ranges and the number of edges within the graph. There is an additional row for each iteration of the *build-coalesce* loop.

The space required for the bit matrix is $\lceil n^2/16 \rceil$ bytes, where n is the number of live ranges plus the number of machine registers (in our case, 32). The adjacency vectors require 4 bytes per edge in the graph.

The *total* row includes the total time required for allocation and the largest amount of storage required at any point for the interference graph (the sum of the requirements for the bit matrix and the adjacency vectors during the initial pass of the *build-coalesce* loop).

The three routines were selected to illustrate compile-time costs over a range of sizes. The first routine is `repvid`, from the program `doduc`, with 144 non-comment lines of FORTRAN. It compiles to a `.text` size of 1284 bytes using IBM's `xlf` compiler

	repvid			tomcatv			twldrv		
phase	*time*	*bytes*	*ranges*	*time*	*bytes*	*ranges*	*time*	*bytes*	*ranges*
control-flow	.00			.00			.01		
renumber	.03			.06			.57		
build-coalesce	.17	8,696		.39	21,684		10.27	1,196,292	
		57,320	341		124,720	557		2,161,348	4,343
		38,472	235		85,560	422		1,407,220	2,967
		36,460	228		79,612	414		1,375,468	2,948
		36,168	227		77,356	411		1,361,472	2,944
					75,844	409		1,357,192	2,942
								1,355,044	2,941
spill costs	.01			.02			.16		
color	.02			.04			1.16		
spill code	.01		38	.02		80	.17		337
renumber	.02			.02			.10		
build-coalesce	.06	6,088		.09	19,744		.63	827,648	
		30,092	280		51,740	530		380,652	3,607
		26,460	242		47,076	478		365,072	3,448
spill costs	.01			.01			.07		
color	.01			.02			.14		
spill code	.01		13	.01		25	.03		78
renumber	.01			.02			.10		
build-coalesce	.03	4,696		.05	17,164		.60	758,644	
		26,404	242		47,432	792		363,104	3,452
								363,000	3,451
spill costs	.01			.01			.07		
color	.01			.02			.13		
spill code				.01		3			
renumber				.02					
build-coalesce				.05	17,296				
					47,592	494			
spill costs				.01					
color				.01					
total	.40	66,016		.89	146,404		14.19	3,357,640	

Table 8.1 Compile-Time Characteristics

with optimization. The second routine is `tomcatv`, with 133 lines and a `.text` size of 3064 bytes. The largest routine is `twldrv` from the program `fpppp`, with 881 lines and a `.text` size of 15,616 bytes.

The results in Table 8.1 illuminate a number of interesting details about the optimistic allocator.

- The very low cost of control-flow analysis illustrates the speed and practicality of the algorithm for calculating dominance frontiers [29].
- The initial pass of the *build-coalesce* process dominates the overall cost of allocation (as noted by Chaitin). In comparison, additional iterations of the *build-color-spill* loop are quite inexpensive.
- The higher cost of coloring in the first pass arises from the cost of choosing nodes to spill. While the cost of coloring is linear in the size of the graph, spill selection is $O(s \cdot n)$, where s is the number of spill choices and n is the number of nodes. With a large number of spills, this term dominates the cost of *simplify*.

We are pleased with the overall speed of the allocator. Our results appear to be slightly faster than the times reported by IBM's `xlf` compiler for register allocation and comparable to the times reported for optimization.

8.10 Discussion

The recent introduction and exploration of SSA form has been been an exciting development in an area that appeared to be well understood. Many researchers are recasting old optimizations in terms of SSA for improved clarity and efficiency; our approach to determining live ranges is one example. Furthermore, new techniques and sharper versions of old techniques are being developed; *e.g.*, our work on rematerialization.

It is difficult to recover an efficient, executable program from SSA form after extensive transformations; removing the ϕ-nodes requires insertion of many copies. Cytron *et al.* suggest using coloring to minimize the number of copies, relying on any of the common coloring-based allocation techniques [29]. This idea is not strictly accurate. Chow's approach – priority-based coloring – is inadequate since it cannot remove copies via coalescing. More to the point, *coloring* is not actually required; rather, the combination of *renumber*, *build*, and *coalesce* will remove the unnecessary copies.

Our implementation has always been entirely isolated from the optimizer; however, given closer connection to an optimizer based on SSA, a simplified implementation of *renumber* would be possible. Additionally, the structure of *renumber, build,* and *coalesce* could be modified, perhaps improving overall performance. A sketch of one possible implementation is given here:

1. Build (or inherit) the pruned SSA form. During the construction, rename to remove copies (some care must be exercised with copies involving machine registers). The copies will be effectively subsumed in the ϕ-nodes.[35]

2. Build the interference graph, handling ϕ-nodes much like copies.

3. During *coalesce,* attempt to coalesce each operand of a ϕ-node with the result.

This approach may be much faster than our current scheme since it eliminates most copies without recourse to coalescing (recall the high cost of the initial *build-coalesce* loop as shown in Table 8.1). On the other hand, this approach may be slower due to larger interference graphs or more coalescing related to ϕ-nodes. Obviously, implementation experience will help resolve these questions.

[35]Kenny Zadeck pointed out this possibility.

Chapter 9

Conclusion

Chaitin and his colleagues described the first global register allocator based on graph coloring. This thesis describes a series of improvements and extensions to their work. The improvements lead to reduced spill costs and faster code; the extensions enable wider application of the basic techniques. Additionally, we report on experimental studies measuring the effectiveness of each of our improvements. Finally, we describe many implementation details and include measurements designed to provide accurate intuitions about the time and space requirements of coloring allocators.

In the next two sections, we offer some perspective on optimization, register allocation, and graph coloring. In the final two sections, we summarize the contributions of the thesis and discuss directions for future work.

9.1 Register Allocation and Optimization

The isolation of register allocation from other parts of optimization is a simplifying separation of concerns. When there are enough registers, this simplification looks like a good decision – the individual optimizations are simpler to build and the resulting code is still good. When there are not enough registers, the assumption underlying the separation of concerns breaks down and we begin to see cases of "over-optimization" – cases where optimization causes degradation due to excessive spill code.

How many registers are enough? The answer depends on many factors: the application code, the amount of instruction-level parallelism offered by the target machine (including pipeline latency), and the speed of the CPU compared to the bandwidth of memory. Furthermore, the ability of the compiler to take advantage of the machine's resources is important. A compiler that attempts only minimal optimization will require very few registers for best results. On the other hand, those "best results" will presumably be worse than results achieved with an aggressive optimizing compiler.

Some alternatives have been studied. For example, Leverett attacks the problem of combining register allocation with instruction selection [53]. Others have explored the possibility of combining instruction scheduling (in one of several possible flavors) with register allocation, recognizing that extensive motion due to scheduling can dramatically increase register pressure [11]. Callahan *et al.* perform aggressive loop transformations while accounting for register pressure to avoid over-optimization [15].

In each of these cases, we could argue that consideration of register pressure unnecessarily complicates the optimizer; the correct solution to the problem of over-optimization is more registers. Of course, this is not helpful to those implementing compilers for existing machines; but, it will perhaps serve as a cautionary note to architects – current optimizers are only effective when the machine has an adequate register set. Machines with long pipelines, many execution units, or a relatively low bandwidth to memory will require many registers for best performance.

9.2 Register Allocation and Graph Coloring

The reduction of register allocation to graph coloring is a further simplification of the problem. Of course, since we are "simplifying" to an NP-complete problem, we may not have made much progress in the theoretical sense; but in the practical sense, the advantages have proven enormous. However, the viability of the reduction to graph coloring again depends on an adequate register set. With sufficient registers, a graph coloring register allocator offers a wonderfully clear approach to the problem. With insufficient registers, the abstraction to graph coloring will look like an over-simplification, since too much important information about the structure of the routine is lost. Therefore, it becomes interesting to consider other approaches to the problem global register allocation. Our work with live range splitting is one possibility. Other possibilities include priority-based coloring, by Chow and Hennessy [26], and the hierarchical graph coloring allocator of Callahan and Koblenz [16].

9.3 Contributions

The work described in this thesis may be divided into three parts:

1. improvements to the Yorktown allocator, working within the basic framework established by Chaitin *et al.*,
2. exploration of aggressive live range splitting, extending Chaitin's framework to enable more precise spill code, and
3. practical work, including a variety of experiments and engineering studies.

In the first category, we include improvements to the Yorktown allocator's coloring and spilling heuristics. The optimistic heuristic for coloring and spilling is perhaps the most important single contribution. By making only a small change to Chaitin's heuristic, we obtain better colorings with less spill code. The optimistic heuristic has been implemented as part of several industrial and academic compilers. Furthermore, it has influenced the design of new methods for global register allocation.

The correct handling of register pairs is a natural consequence of our use of the optimistic heuristic. The obvious difficulties of handling pairs with Chaitin's heuristic were simply extreme cases of similar, though less obvious, difficulties that arose when coloring individual registers. We believe this extension may have impact on the design of future processors. Some architectures (*e.g.*, the MIPS R2000 and the IBM RS/6000) have avoided the use of register pairs to support double-precision arithmetic. This design decision may have been influenced by the lack of an adequate method for allocating register pairs.

The need for better rematerialization became obvious during our exploration of aggressive live range splitting. Because of our experiments using SSA to support *renumber*, we were able to discover a generalization of Chaitin's approach to rematerialization. In addition to the improvement obtained in the splitting allocator, we were able to achieve improvements in the code produced by the optimistic allocator.

Our approach to live range splitting is basically an attempt to preserve some of the information ordinarily lost during the reduction to graph coloring – information that should prove useful when spilling. While our results were not entirely satisfactory, we were able to expose, and in some cases solve, several unexpected problems that seem inherent to any splitting allocator. Examples include the fully and partially redundant stores arising from splitting and spilling. Of course, the work on improving rematerialization was originally prompted by problems observed during splitting. Furthermore, the heuristics for *conservative coalescing* and *biased coloring*, though reported in connection with rematerialization, were discovered during our work on splitting.

In the third category, we include contributions stemming from our experimental implementations and comparisons of the various allocators. The experiments served many purposes. In some cases, they gave an indication of the importance of certain modifications. During our work with live range splitting, the experimental comparisons helped expose the weaknesses of different splitting heuristics – weaknesses that were not at all obvious before implementation.

Results of our practical work include:

- a methodology for comparing allocators,
- extensive experimental results showing the usefulness of the optimistic heuristic and rematerialization, and pointing out both the problems and potential profits of live range splitting,
- a limited comparison of priority-based coloring with the Yorktown allocator,
- explanations of many of the important algorithms and data structures required for efficient implementation of our allocator, and
- measurements of many phases of the allocators, designed to provide useful intuition about the time and space requirements of global register allocation via graph coloring.

9.4 Future Work

Despite our efforts, the test suite used in our experiments is small by industrial standards.[36] We need to spend considerably more time expanding and diversifying the range of test programs. Of course, as the test suite becomes larger, more extensive support for automatic testing will become mandatory.

The experimental results reported for aggressive live range splitting are conservative. We have already discovered improvements that may significantly affect our results. We also intend to continue exploring possibilities offered by other splitting heuristics. Additionally, we will search for sharper approaches to the problem of *conservative coalescing*.

We are interested in implementing the allocators described by Chow and Hennessy and by Callahan and Koblenz in the context of our compiler. This would finally allow useful comparisons between radically different approaches. Of course, such implementations and comparisons will have to be conducted carefully, perhaps with the active participation of the inventors.

Finally, we are interested in ways of overcoming the separation between optimization and allocation. We have mentioned some possibilities for avoiding over-optimization in Section 9.1. Alternatively, we could perhaps pass additional information from the optimizer to the allocator to support undoing certain optimizations instead of spilling (similar in effect to rematerialization).

[36]Our colleagues in industry describe test suites containing over one million lines of code.

Appendix A

Detailed Results

The following sections contain the raw results of our experiments comparing allocation quality (see Section 7.1). Additionally, we give calculated spill costs for each allocator on each routine. Each section contains results from a group of programs. The routines making up each program are further grouped into subsections.

Each table contains seven rows of results, with measurements for six of our experimental allocators and a row of measurements for the "huge" machine. Five columns give the number of loads, stores, copies, load-immediates, and add-immediates executed during an invocation of the routine. When a routine was executed many times during a program, we report results from the first execution only. The sixth column (*total cost*) gives a weighted total of all the measured instructions, where loads and stores are weighted twice and the other instructions are weighted once. The final column (*spill cost*) gives the difference between each allocator's total cost and the "huge" machine's total cost. This value should equal the number of cycles spent on spill code; that is, the number of cycles wasted due to insufficient registers and poor allocation.

A blank entry simply means that the result is identical to the previous row.

A.1 Forsythe, Malcolm, and Moler

The routines, transcribed from Forsythe, Malcolm, and Moler's book on computational methods [36], are included as sentimental favorites. They were among the first routines we used to test our compiler. In particular, the routines `decomp` and `solve` were always reliable test cases, exposing a remarkable number of errors in the register allocator and all phases of the optimizer.

A.1.1 fmin

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	115	45	74	3	9	406	
chaitin	270	155	26	72		957	551
optimistic	225	110		71		776	370
backtrack							
rematerial							
SSA-based	207		51			765	359
dominance	167	149	173	19		833	427

A.1.2 rkf45

fehl

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	131	46	24	34	61	473	
chaitin	176		7	29		541	68
optimistic							
backtrack							
rematerial	167		2	34		523	50
SSA-based							
dominance	175	45	4			539	66

rkf45

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	15	21	15	0	11	98	
chaitin	41	41	2			177	79
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

A.1.3 seval

seval

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	13	3	3	2	10	47	
chaitin	21	8				73	26
optimistic							
backtrack							
rematerial							
SSA-based							
dominance	21		5	4		75	28

spline

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	254	101	1	5	54	770	
chaitin	301	116	2			895	125
optimistic	298	115				887	117
backtrack							
rematerial	292	114	0	6		872	102
SSA-based	291		1			871	101
dominance	265	111	10	8	53	823	53

A.1.4 solve

decomp

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	79	49	49	19	86	410	
chaitin	224	102	16	18		772	362
optimistic	206	91	17			715	305
backtrack							
rematerial	200	87		19		696	286
SSA-based	195	82	26	22		688	278
dominance	187	82	53	21		718	308

solve

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	26	15	7	5	23	117	
chaitin	53	31	5	6		202	85
optimistic							
backtrack							
rematerial			6	5			
SSA-based			5	6			
dominance	54	32	7	5		207	90

A.1.5 svd

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	495	322	210	37	318	2,199	
chaitin	1,371	760	88			4,705	2,509
optimistic	1,166	671	147			4,176	1,977
backtrack							
rematerial	1,158		141	48		4,165	1,966
SSA-based	1,159	632	128	54		4,082	1,883
dominance	1,309	857	472	68		5,190	2,991

A.1.6 urand

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	13	10	32	4	2	84	
chaitin	17	14				100	16
optimistic							
backtrack							
rematerial							
SSA-based							
dominance			34			102	18

A.1.7 zeroin

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	79	41	51	8	8	307	
chaitin	113	63	47	136		543	236
optimistic							
backtrack							
rematerial	111		49			541	234
SSA-based	65	49	78			450	143
dominance	69	54	151	30		435	128

A.2 SPEC

A.2.1 doduc

In the SPEC benchmark suite, the program `doduc` is supplied with several data files – some for initial testing and one for actual benchmark. We used the standard benchmark data file for our tests.

arret

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	5	1	0	3	0	15	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

bilan

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	982	256	15	90	130	2,711	
chaitin	1,109	301		782	142	3,759	1,048
optimistic	1,108	301				3,757	1,046
backtrack							
rematerial	1,082	287				3,677	966
SSA-based				780		3,675	964
dominance	1,108	298				3,749	1,038

bilsla

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	7	5	0	1	1	26	
chaitin	11	9		2		42	16
optimistic							
backtrack							
rematerial				1		41	15
SSA-based							
dominance							

cardeb

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	216	155	2	15	168	927	
chaitin	214	153	0	108		1,010	83
optimistic							
backtrack							
rematerial							
SSA-based			2			1,012	85
dominance				178		1,082	155

coeray

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	34	24	6	4	9	135	
chaitin	36	22	4	6		135	0
optimistic				4		133	−2
backtrack							
rematerial							
SSA-based							
dominance			10	6		141	6

colbur

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	44	33	13	12	10	189	
chaitin	53	39	5	15		214	25
optimistic	52	38		13		208	19
backtrack							
rematerial	53	39	6			213	24
SSA-based			7			214	25
dominance	54			15		218	29

dcoera

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	60	51	11	4	1	248	
chaitin	77	48	4	12		277	29
optimistic	74	45		10		263	15
backtrack							
rematerial							
SSA-based							
dominance	70	44	9	11		259	11

ddeflu

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge		194	94	45	69	91	781
chaitin	283	211	12	133		1,224	443
optimistic	254	185	14			1,116	335
backtrack	252	183				1,108	327
rematerial	263	196	12	135		1,156	375
SSA-based	240	164	19	120		1,038	257
dominance	238	168	55	121		1,079	298

debflu

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	993	629	95	252	630	4,221	
chaitin	1,243	1,067	47	831	662	6,160	1,939
optimistic	1,048	873		801		5,352	1,131
backtrack							
rematerial							
SSA-based	1,067	902	126		646	5,511	1,290
dominance	1,046	872	196	804	658	5,494	1,273

debico

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge		699	295	1	99	149	2,239
chaitin	745	319	4	395	173	2,700	463
optimistic	744	318				2,696	459
backtrack							
rematerial	731	317	1	385		2,655	418
SSA-based				398		2,668	431
dominance	718			599		2,843	606

deseco

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	3,495	902	155	619	1,272	10,840	
chaitin	4,618	1,639	112	2,219	1,495	16,340	5,500
optimistic	4,497	1,582	140	2,029	1,470	15,797	4,957
backtrack							
rematerial	4,330	1,539		2,140	1,458	15,476	4,636
SSA-based	4,276	1,469	127	2,127	1,459	15,203	4,363
dominance	4,239	1,494	280	2,640	1,533	15,919	5,079

drepvi

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	53	15	12	15	28	191	
chaitin	117	75	4	27		443	252
optimistic	108	67				409	218
backtrack	107	66				405	214
rematerial	104	52	3	23		366	175
SSA-based	99	48	6			351	160
dominance	105	54	10	29		385	194

drigl

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	33	19	4	24	34	166	
chaitin	35	33				198	32
optimistic							
backtrack							
rematerial				23		197	31
SSA-based				24		198	32
dominance			6	23		199	33

dyeh

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	32	32	10	3	2	143	
chaitin	26	22	6			107	−36
optimistic							
backtrack							
rematerial			5			106	−37
SSA-based							
dominance	23	19	2			91	−52

ewv

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	8	10	3	2	8	49	
chaitin	8	12				53	4
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

heat

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	46	20	16	26	17	191	
chaitin	54	30	11	29		225	34
optimistic							
backtrack							
rematerial	53		10			222	31
SSA-based	52	28	11			217	26
dominance	51	26	18	30		219	28

hmoy

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	6	0	0	5	3	20	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

ihbtr

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	75	24	15	42	159	414	
chaitin	217	90	0	93		866	452
optimistic	204	77				814	400
backtrack							
rematerial	202	76		94		809	395
SSA-based	161	72	13	95		733	319
dominance	204	85	128	77		942	528

inideb

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	170	170	0	21	151	852	
chaitin	163	163		99		902	50
optimistic							
backtrack							
rematerial				97		900	48
SSA-based				74		877	25
dominance				166		969	117

iniset

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	9,466	9,466	152	168	9,340	47,524	
chaitin	9,467	9,467				47,528	4
optimistic							
backtrack							
rematerial	9,466	9,466	0	320		47,524	0
SSA-based							
dominance			1	321		47,526	2

inisla

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	9	8	0	1	1	36	
chaitin	16	16		2		67	31
optimistic							
backtrack							
rematerial		15		1		64	28
SSA-based							
dominance							

inithx

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	606	201	2	153	351	2,120	
chaitin	756	274	1	422		2,834	714
optimistic	717	259		395		2,699	579
backtrack							
rematerial	669	231		405		2,557	437
SSA-based		229	3			2,555	435
dominance			24	507		2,678	558

integr

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	477	106	12	163	192	1,533	
chaitin	611	176		281		2,059	526
optimistic				257		2,035	502
backtrack							
rematerial	567	147		273		1,905	372
SSA-based		145	27			1,916	383
dominance	529	133	63	401		1,980	447

inter

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	4	2	0	0	0	12	
chaitin	11	6				34	22
optimistic							
backtrack							
rematerial							
SSA-based							
dominance	8	4				24	12

lectur

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	289	257	41	204	340	1,677	
chaitin	310	326	8	314		1,934	257
optimistic	396	324		310		1,898	221
backtrack							
rematerial			4	259		1,834	166
SSA-based			7	235		1,822	145
dominance	282	322	18	294	328	1,848	171

lissag

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	19	11	0	2	12	74	
chaitin	24	16				94	20
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

main

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	43,914	43,919	0	5,495	43,916	225,080	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance				5,498		225,083	3

orgpar

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	29	22	5	7	18	132	
chaitin	38	3	1	10		171	39
optimistic							
backtrack							
rematerial		32	2	7		167	35
SSA-based							
dominance		31				165	33

paroi

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	709	410	32	141	488	2,899	
chaitin	1,102	598	29	462	788	4,679	1,780
optimistic	1,021	543	29		713	4,332	1,433
backtrack							
rematerial	965	540	45	514		4,282	1,383
SSA-based	950	523	41		764	4,265	1,366
dominance	1,011	568	170	545		4,637	1,738

pastem

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	600	54	219	218	345	2,090	
chaitin	680	107	219	244		2,379	289
optimistic							
backtrack							
rematerial	651	93	178	299		2,319	220
SSA-based	638	92	204			2,308	218
dominance			303	433		2,541	451

prophy

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	838	577	62	249	251	3,392	
chaitin	1,184	855	3	1,020	245	5,346	1,954
optimistic	1,059	769			244	4,923	1,531
backtrack							
rematerial		767		1,018		4,917	1,525
SSA-based	1,115	793	19	991		5,070	1,678
dominance	1,128	833	656	1,231	245	6,054	2,662

repvid

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	483	143	63	51	259	1,625	
chaitin	573	221	128	301	259	2,276	651
optimistic	560	208				2,224	599
backtrack							
rematerial	534	169	63			2,029	404
SSA-based			88			2,054	429
dominance	507	155	166	413		2,162	537

saturr

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	179	170	11	12	11	732	
chaitin	229	175	4	23	3	838	106
optimistic							
backtrack							
rematerial							
SSA-based	231			27		846	114
dominance	236	180	11			873	141

si

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	20	1	12	2	27	83	
chaitin	23	3				93	10
optimistic							
backtrack							
rematerial							
SSA-based							
dominance			13			94	11

sigma

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	12	0	0	5	0	29	
chaitin	15	3				41	12
optimistic							
backtrack							
rematerial							
SSA-based			1			42	13
dominance							

sortie

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	23	11	1	14	5	88	
chaitin	28	16				108	20
optimistic							
backtrack							
rematerial							
SSA-based	29	17				112	24
dominance	28	16	3	18		114	26

subb

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	310	248	0	78	0	1,194	
chaitin	391	215				1,290	96
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

supp

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	435	308	1	20	0	1,507	
chaitin	522	294	0	21		1,653	146
optimistic	523	295		20		1,656	149
backtrack							
rematerial							
SSA-based	520	292				1,644	137
dominance	510	282		33		1,617	110

vgjyeh

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	12	13	4	3	1	58	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

x21y21

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	42	42	0	2	46	216	
chaitin	43	43				220	4
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

yeh

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	65	73	27	5	10	318	
chaitin	92	70	14			353	35
optimistic	76	58				297	−21
backtrack							
rematerial	74	56	12	8		290	−28
SSA-based	73	54	11			283	−35
dominance	64	53	17			269	−49

A.2.2 fpppp

Since `fpppp` accepts input controlling the problem size, we arbitrarily used the number "5" which seems adequate to exercise most of the program.

aclear

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	1	16	0	1	17	52	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

d2esp

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	389	401	25	26	369	2,000	
chaitin	397	417	8	46		2,051	51
optimistic	393	413	9	45		2,035	35
backtrack							
rematerial	392			46		2,034	34
SSA-based	393	414				2,038	38
dominance	390	412	12	43		2,028	28

efill

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	124	88	3	18	43	488	
chaitin	175	124	0	20		661	173
optimistic	157	104		17		582	94
backtrack							
rematerial							
SSA-based	156	92	2			558	70
dominance	161	93	53			621	133

fmtgen

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	38	20	7	4	18	145	
chaitin		21	4			144	−1
optimistic							
backtrack							
rematerial		20	5			143	−2
SSA-based							
dominance	50	31	62	9		251	106

fmtset

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	141	89	5	11	32	508	
chaitin	142	90				512	4
optimistic							
backtrack							
rematerial	141	89	4	12		508	0
SSA-based							
dominance	139	87		19		507	−1

fpppp

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	1,930	274	0	5	0	4,413	
chaitin	2,561	379				5,885	1,472
optimistic	2,553	373				5,857	1,444
backtrack							
rematerial							
SSA-based							
dominance							

gamgen Note that the SSA-based splitter produced illegal code for `gamgen`.

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	20,052	10,859	0	23	6,834	68,679	
chaitin	20,062	10,864		74	8,836	70,762	2,083
optimistic							
backtrack							
rematerial							
SSA-based							
dominance		10,863			11,233	73,157	4,478

ilsw

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	0	1	0	0	0	2	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

intowp

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	1	0	0	1	1	4	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

lclear

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	1	26	0	1	27	83	
chaitin							0
optimistic							
backtrack							
rematerial							
SSA-based							
dominance							

main

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	129	283	7	76	134	1,041	
chaitin			4	289		1,251	210
optimistic							
backtrack							
rematerial			3	279		1,240	199
SSA-based			2			1,239	198
dominance	130	284	4	275		1,241	200

twldrv

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	42,312,412	10,400,205	171,806	819,862	1,286,276	107,703,178	
chaitin	46,867,951	12,261,427	54,111	1,372,374	1,749,739	121,434,980	13,731,802
optimistic	46,077,911	11,813,625	110,337			119,014,802	11,311,624
backtrack							
rematerial	45,965,582	11,813,259		1,483,478		118,901,236	11,198,058
SSA-based	46,606,622	11,894,122	633,610			120,868,315	13,165,137
dominance	47,848,939	13,874,276	5,168,232	3,190,025	1,724,989	133,529,676	25,826,498

A.2.3 matrix300

lbmk14

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	7	270,027	58	963	90,936	632,025	
chaitin	23	270,101	3	974		632,161	136
optimistic	22	270,100				632,157	132
backtrack							
rematerial							
SSA-based							
dominance							

prnt

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	304	90,005	303	301	1,202	182,424	
chaitin	321	90,018				182,484	60
optimistic							
backtrack							
rematerial							
SSA-based			302	302			
dominance			304				

saxpy

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	604	304	2	0	1	1,819	
chaitin	617	314				1,865	46
optimistic							
backtrack							
rematerial							
SSA-based	615	321				1,857	38
dominance			6			1,861	42

sgemm

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	1,230	1,830	1,215	2	901	8,238	
chaitin	4,842	4,535	902			20,559	12,321
optimistic	4,238	3,931				18,143	9,905
backtrack							
rematerial	3,637	3,629		301		16,636	8,398
SSA-based	2,742	3,328	1,204			14,546	6,308
dominance		3,326	915			14,253	6,015

sgemv

allocator	loads	stores	copies	ldi's	addi's	total cost	spill cost
huge	924	1,224	913	3	303	5,515	
chaitin	2,139	1,828	302			8,542	3,027
optimistic	1,535	1,822	303			7,323	1,808
backtrack	1,534	1,821				7,319	1,804
rematerial	1,535	1,822				7,323	1,808
SSA-based	1,524	1,821	305			7,321	1,806
dominance		1,822	616			7,624	2,109

A.2.4 tomcatv

allocator	*loads*	*stores*	*copies*	*ldi's*	*addi's*	*total cost*	*spill cost*
huge	352,236,895	71,609,914	218,082	181,174	39,362,624	833,455,498	
chaitin	489,075,095	104,910,820	93,905	232,173	39,464,322	1,227,853,230	394,397,732
optimistic	482,470,495	98,359,920	93,906			1,201,451,231	367,995,733
backtrack							
rematerial	476,018,795	98,309,945	140,781			1,188,494,756	355,039,258
SSA-based	488,972,895	111,238,545	192,382			1,240,311,757	406,856,259
dominance	462,659,096	72,301,942	269,282	232,273		1,109,887,953	276,432,455

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1974.

[2] Alfred V. Aho and Steven C. Johnson.
Optimal code generation for expression trees.
Journal of the ACM, 23(3):488–501, March 1976.

[3] Alfred V. Aho, Steven C. Johnson, and Jeffrey D. Ullman.
Code generation for expressions with common subexpressions.
Journal of the ACM, 24(1):146–160, January 1977.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, 1986.

[5] Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser.
Experiences creating a portable Cedar.
SIGPLAN Notices, 24(7):322–329, July 1989.
Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[6] Marc A. Auslander and Martin E. Hopkins.
An overview of the PL.8 compiler.
SIGPLAN Notices, 17(6):22–31, June 1982.
Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.

[7] John Backus.
The history of Fortran I, II, and III.
In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[8] Richard A. Becker, John M. Chambers, and Allan R. Wilks.
The New S Language: A Programming Environment for Data Analysis and Graphics.
Wadsworth and Brooks/Cole, Pacific Grove, California, 1988.

[9] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter.

Spill code minimization techniques for optimizing compilers.
SIGPLAN Notices, 24(7):258–263, July 1989.
Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[10] Hans-J. Boehm and Mark Weiser.
Garbage collection in an uncooperative environment.
Software – Practice and Experience, 18(9):807–820, September 1988.

[11] David G. Bradlee, Susan J. Eggers, and Robert R. Henry.
Integrating register allocation and instruction scheduling for RISCs.
In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, 1991.

[12] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon.
Coloring heuristics for register allocation.
SIGPLAN Notices, 24(7):275–284, July 1989.
Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[13] Preston Briggs, Keith D. Cooper, and Linda Torczon.
Coloring register pairs.
ACM Letters on Programming Languages and Systems, 1992.
To appear.

[14] Preston Briggs, Keith D. Cooper, and Linda Torczon.
Rematerialization.
SIGPLAN Notices, 27, 1992.
To appear in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[15] David Callahan, Steve Carr, and Ken Kennedy.
Improving register allocation for subscripted variables.
SIGPLAN Notices, 25(6):53–65, June 1990.
Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[16] David Callahan and Brian Koblenz.
Register allocation via hierarchical graph coloring.
SIGPLAN Notices, 26(6):192–203, June 1991.
Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

[17] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren.
A practical environment for Fortran programming.
IEEE Computer, 20(11):75–89, November 1987.

[18] Gregory J. Chaitin.
Register allocation and spilling via graph coloring.
SIGPLAN Notices, 17(6):98–105, June 1982.
Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.

[19] Gregory J. Chaitin.
Personal communication.
E-mail message, January 1992.

[20] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein.
Register allocation via coloring.
Computer Languages, 6:47–57, January 1981.

[21] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante.
Automatic construction of sparse data flow evaluation graphs.
In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[22] Fred C. Chow.
A Portable Machine-Independent Global Optimizer – Design and Measurements.
PhD thesis, Stanford University, December 1983.

[23] Fred C. Chow.
Minimizing register use penalty at procedure call.
SIGPLAN Notices, 23(7):85–94, July 1988.
Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.

[24] Fred C. Chow.
Personal communication.
E-mail message, November 1991.

[25] Fred C. Chow and John L. Hennessy.
Register allocation by priority-based coloring.
SIGPLAN Notices, 19(6):222–232, June 1984.
Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.

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

[27] Keith D. Cooper, Ken Kennedy, and Linda Torczon.
The impact of interprocedural analysis and optimization on the $\mathbb{R}^n$ programming environment.
ACM Transactions on Programming Languages and Systems, 8(4):491–523, October 1986.

[28] Ron Cytron and Jeanne Ferrante.
What's in a name? The value of renaming for parallelism detection and storage allocation.
In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[29] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck.
Efficiently computing static single assignment form and the control dependence graph.
ACM Transactions on Programming Languages and Systems, 13(4):451–490, October 1991.

[30] Andrei P. Ershov.
Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs.
Doklady Akademii Nauk S.S.S.R., 142(4), 1962.
English translation in *Soviet Mathematics* 3, 1962.

[31] Andrei P. Ershov.
Alpha – an automatic programming system of high efficiency.
Journal of the ACM, 13(1), January 1966.

[32] Andrei P. Ershov, L. L. Zmiyevskaya, R. D. Mishkovitch, and L. K. Trokhan.
Economy and allocation of memory in the Alpha translator.
In Ershov, editor, *The Alpha Automatic Programming System*, pages 161–196. Academic Press, 1971.

[33] Andrei P. Ershov.[37]
Origins of Programming: Discourses on Methodology.
Springer-Verlag, 1990.

[34] Janet Fabri.
Automatic storage optimization.
SIGPLAN Notices, 14(8):83–91, August 1979.
Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.

[35] Janet Fabri.
Automatic Storage Optimization.
UMI Research Press, Ann Arbor, Michigan, 1982.

[36] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler.
Computer Methods for Mathematical Computations.
Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[37] The modern transliteration is *Ershov*. Older variations include *Yershov* and *Eršov*.

[37] R. A. Freiburghouse.
Register allocation via usage counts.
Communications of the ACM, 17(11):638–642, November 1974.

[38] Michael R. Garey and David S. Johnson.
The complexity of near-optimal graph coloring.
Journal of the ACM, 23(1):43–49, January 1976.

[39] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer.
Some simplified NP-complete graph problems.
Theoretical Computer Science, 1:237–267, 1976.

[40] Martin C. Golumbic.
Algorithmic Graph Theory and Perfect Graphs.
Academic Press, 1980.

[41] Rajiv Gupta, Mary Lou Soffa, and Tim Steele.
Register allocation via clique separators.
SIGPLAN Notices, 24(7):264–274, July 1989.
Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[42] David R. Hanson.
Fast allocation and deallocation of memory based on object lifetimes.
Software – Practice and Experience, 20(1):5–12, January 1990.

[43] Martin E. Hopkins.
Compiling for the RT PC ROMP.
In *IBM RT Personal Computer Technology*, pages 76–82. IBM, 1986.

[44] Martin E. Hopkins.
Personal communication.
Conversation during visit to Rice, February 1991.

[45] Intel Corporation.
i860™ *XP Microprocessor*, 1991.

[46] Mark Scott Johnson and Terrence C. Miller.
Effectiveness of a machine-level global optimizer.
SIGPLAN Notices, 21(7):99–108, July 1986.
Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.

[47] John B. Kam and Jeffrey D. Ullman.
Global data flow analysis and iterative algorithms.
Journal of the ACM, 23(1):158–171, January 1976.

[48] Richard M. Karp.
Reducibility among combinatorial problems.
In Miller and Thatcher, editors, *Complexity of Computer Computations*. Plenum, 1972.

[49] Ken Kennedy.
Global Flow Analysis and Register Allocation for Simple Code Structures.
PhD thesis, Courant Institute, New York University, October 1971.

[50] Ken Kennedy, 1986.
In conversation, Kennedy tells about Cocke adjuring people: "Imagine you have so many registers you never need to spill; imagine you have 16 registers." The punchline came one day after the PL.8 compiler had been forced to spill: "Imagine you have so many registers you never need to spill; imagine you have *32* registers".

[51] James R. Larus and Paul N. Hilfinger.
Register allocation in the SPUR Lisp compiler.
SIGPLAN Notices, 21(7):255–263, July 1986.
Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.

[52] S. S. Lavrov.
Store economy in closed operator schemes.
Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki, 1(4):687–701, 1961.
English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3, 1962.

[53] Bruce W. Leverett.
Register Allocation in Optimizing Compilers.
UMI Research Press, Ann Arbor, Michigan, 1983.

[54] David W. Matula and Leland L. Beck.
Smallest-last ordering and clustering and graph coloring algorithms.
Journal of the ACM, 30(3):417–427, July 1983.

[55] Christopher Mills, Stanley C. Ahalt, and Jim Fowler.
Compiled instruction set simulation.
Software – Practice and Experience, 21(8):877–889, August 1991.

[56] Thomas P. Murtagh.
An improved storage management scheme for block structured languages.
ACM Transactions on Programming Languages and Systems, 13(3):372–398, July 1991.

[57] Brian R. Nickerson.
Graph coloring register allocation for processors with multi-register operands.
SIGPLAN Notices, 25(6):40–52, June 1990.
Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[58] Vatsa Santhanam and Daryl Odnert.
Register allocation across procedure and module boundaries.
SIGPLAN Notices, 25(6):28–39, June 1990.

Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[59] Jacob T. Schwartz.
On programming: An interim report on the SETL project, Installment II: The SETL Language and examples of its use.
Technical report, Courant Institute, New York University, October 1973.

[60] Ravi Sethi and Jeffrey D. Ullman.
The generation of optimal code for arithmetic expressions.
Journal of the ACM, 17(7):715–728, July 1970.

[61] SPEC release 1.2, September 1990.
Standards Performance Evaluation Corporation.

[62] Robert Endre Tarjan.
Testing flow graph reducibility.
Journal of Computer and System Sciences, 9:355–365, 1974.

[63] David W. Wall.
Register allocation at link time.
SIGPLAN Notices, 21(7):264–275, July 1986.
Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.

[64] Mark N. Wegman and F. Kenneth Zadeck.
Constant propagation with conditional branches.
ACM Transactions on Programming Languages and Systems, 13(2):181–210, April 1991.