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.

Contents

Abstract ii

Acknowledgments iii

List of Illustrations ix

List of Tables xi

1 Introduction 1

1.1 Compilers and Optimization 1

1.2 Optimization and Register Allocation 2

1.3 Register Allocation and Graph Coloring 3

1.3.1 Minimizing Register Usage 4

1.3.2 Minimizing Spill Code 4

1.4 Overview 4

1.4.1 Improved Coloring and Spilling 5

1.4.2 Coloring Pairs 5

1.4.3 Rematerialization 6

1.4.4 Live Range Splitting 6

2 Background 7

2.1 Register Allocation via Graph Coloring 7

2.2 The Yorktown Allocator 8

2.2.1 Discovering Live Ranges 10

2.2.2 Interference 12

2.2.3 The Interference Graph 13

2.2.4 Coalescing 14

2.2.5 Spilling 16

2.2.6 Coloring 17

2.3 History 20

2.3.1 Memory Allocation 20

2.3.2 Register Allocation 21

3 Improved Coloring and Spilling 23
3.1 Problems 23
3.1.1 The Smallest Example 24
3.1.2 A Large Example 24
3.2 An Improvement 26
3.3 Limited Backtracking 29
3.4 Alternative Spill-Choice Metrics 30
3.5 Summary 32

4 Coloring Register Pairs 33
4.1 Why Are Pairs Hard to Color? 33
4.1.1 Unconstrained Pairs 34
4.1.2 Adjacent Pairs 35
4.2 The Optimistic Coloring Heuristic 38
4.3 Unaligned Pairs 39
4.4 Using Pairs for Memory Access 40
4.5 Summary 41

5 Rematerialization 42
5.1 Introduction 42
5.2 Rematerialization 44
5.2.1 Discovering Values 44
5.2.2 Propagating Rematerialization Information 45
5.2.3 Inserting Splits 46
5.2.4 Removing Unproductive Splits 47
5.3 Implementation 47
5.3.1 Renumber 47
5.3.2 Conservative Coalescing 49
5.3.3 Biased Coloring 50
5.4 Results 51
5.5 Summary 51

6 Aggressive Live Range Splitting 53
6.1 Live Range Splitting 54
6.1.1 Theoretical Difficulties 54

6.1.2 Practical Difficulties . . . 55
6.2 Aggressive Live Range Splitting . . . 55
6.2.1 Splitting . . . 55
6.2.2 Spilling . . . 56
6.2.3 Cleanup . . . 58
6.3 Implementation . . . 60
6.4 Splitting . . . 61
6.4.1 Loop-Based Splitting . . . 61
6.4.2 Splitting Based on Dominance . . . 64
6.4.3 Mechanics . . . 68
6.5 Summary . . . 69

7 Measurements and Comparisons 72

7.1 Measuring Allocation Quality . . . 72
7.1.1 Methodology . . . 73
7.1.2 Results . . . 76
7.2 Priority-Based Coloring . . . 82
7.2.1 Allocation Quality . . . 82
7.2.2 Allocation Time . . . 85

8 Engineering 93

8.1 The Internal Representation . . . 94
8.2 Set Representations . . . 97
8.3 Liveness . . . 98
8.4 Live Ranges . . . 100
8.5 The Interference Graph . . . 101
8.6 Coalescing . . . 104
8.7 Spill Costs . . . 107
8.8 Coloring . . . 110
8.9 Compile-Time Characteristics . . . 112
8.10 Discussion . . . 114

9 Conclusion 116

9.1 Register Allocation and Optimization . . . 116
9.2 Register Allocation and Graph Coloring . . . 117

9.3 Contributions 117
9.4 Future Work 119

A Detailed Results 120

A.1 Forsythe, Malcolm, and Moler 120
A.1.1 fmin 121
A.1.2 rkf45 121
A.1.3 seval 121
A.1.4 solve 122
A.1.5 svd 122
A.1.6 urand 122
A.1.7 zeroin 122
A.2 SPEC 123
A.2.1 doduc 123
A.2.2 fpppp 132
A.2.3 matrix300 135
A.2.4 tomcatv 136

Bibliography 137

Illustrations

2.1 The Yorktown Allocator 8
2.2 Renumbering 11
2.3 Effects of Coalescing 15
2.4 Coloring a Simple Graph 18

3.1 A Simple Graph Requiring Two Colors 24
3.2 The Structure of SVD 25
3.3 The Optimistic Allocator 27

5.1 Rematerialization versus Spilling 43
5.2 Introducing Splits 46

6.1 Splitting 53
6.2 Spilling Partners 57
6.3 Splitting and Spilling 58
6.4 Globally Unnecessary Stores 59
6.5 The Splitting Allocator 60
6.6 Splitting Unmentioned Live Ranges 63
6.7 Splitting at Dominance Frontiers 67

7.1 ILOC and C 75
7.2 Allocation Times 88
7.3 Observed Complexity of Priority-Based Coloring 90
7.4 Observed Complexity of the Yorktown Allocator 91

8.1 Blocks and Edges 95
8.2 Instructions 95
8.3 Instruction Kinds 96

8.4 Set Representation 98
8.5 Passing Parameters in Registers 106

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.

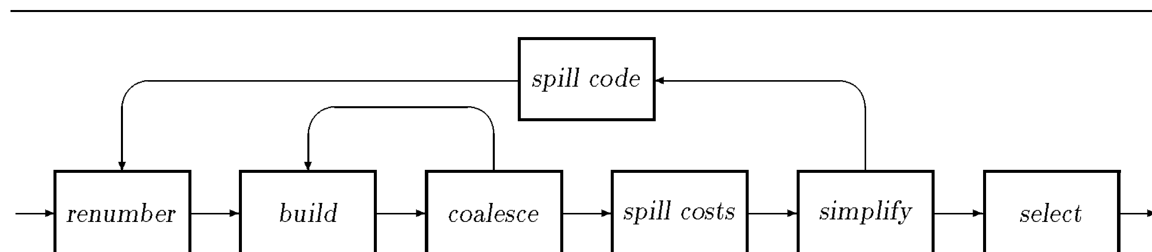

Figure 2.1 The Yorktown Allocator

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.

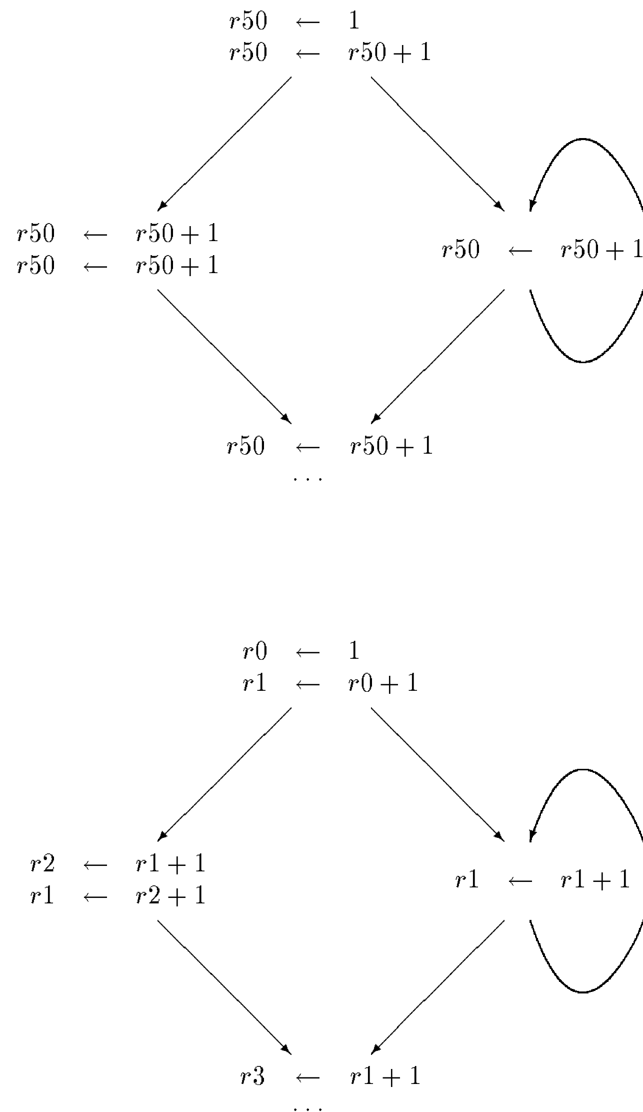

Figure 2.2 Renumbering

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.

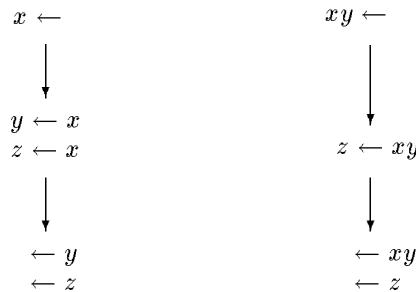

Figure 2.3 Effects of Coalescing

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.

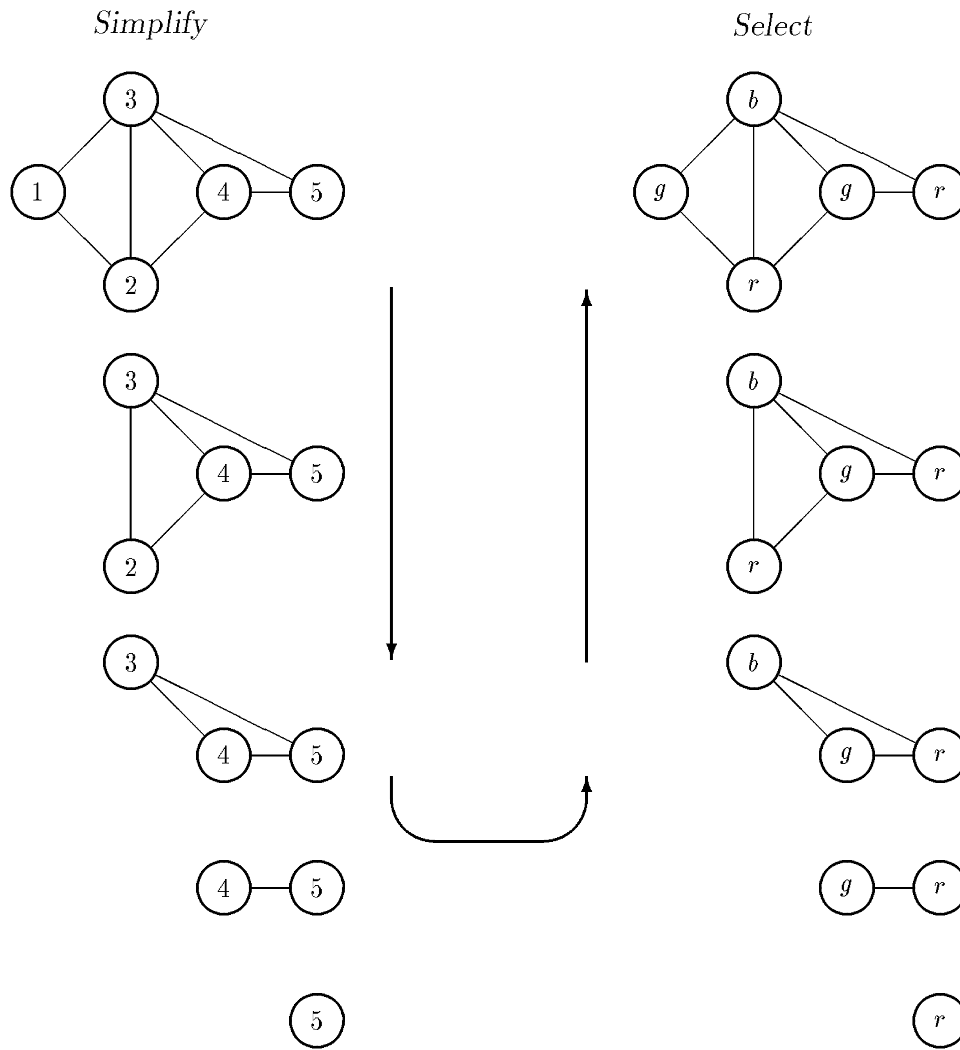

Figure 2.4 Coloring a Simple Graph

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

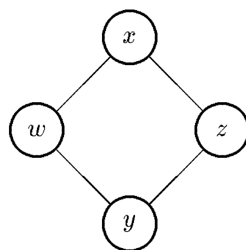

Figure 3.1 A Simple Graph Requiring Two Colors

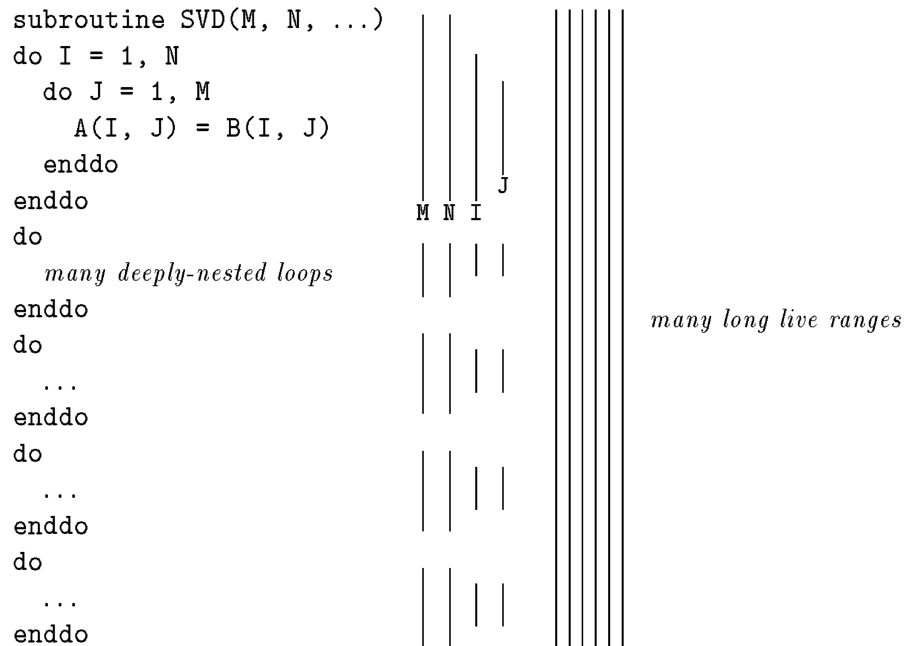

Figure 3.2 The Structure of SVD

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.

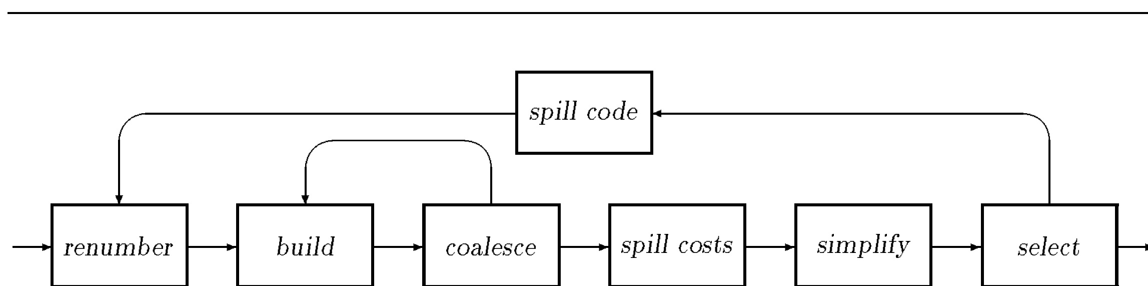

Figure 3.3 The Optimistic Allocator

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 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.

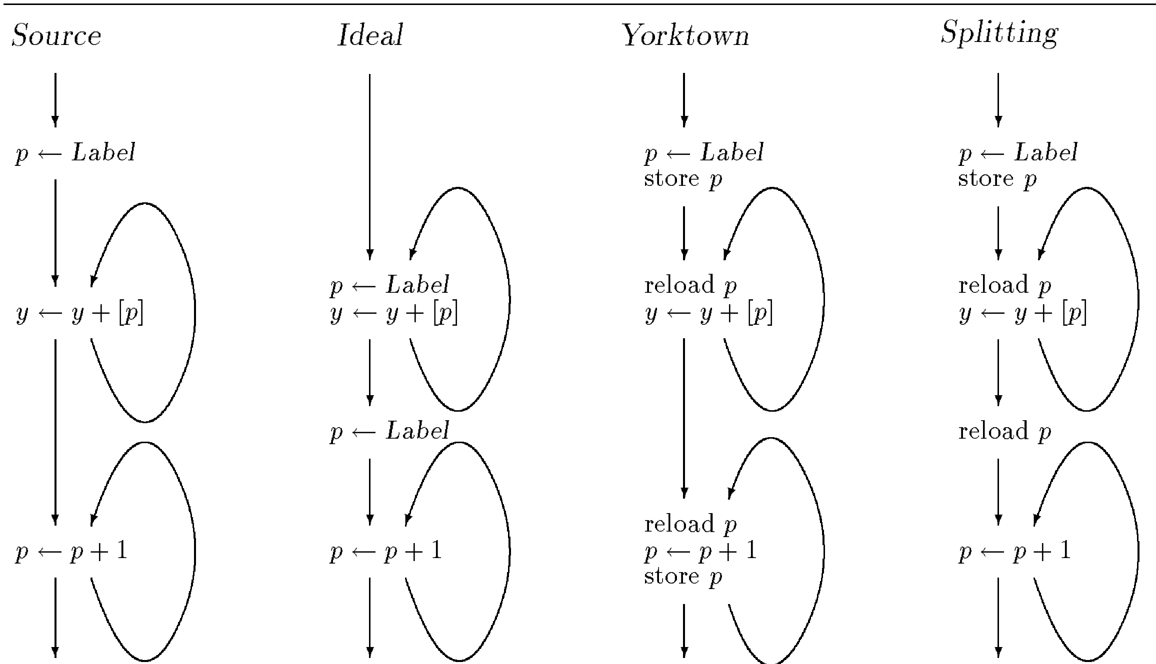

Figure 5.1 Rematerialization versus Spilling

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.
- $\bot$ *Bottom* means that the value cannot be rematerialized. Any value defined by an "inappropriate" instruction is immediately tagged with $\bot$.

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

$$
\begin{array}{ccccl}
\text{any} & \sqcap & \top & = & \text{any} \\
\text{any} & \sqcap & \bot & = & \bot \\
inst_i & \sqcap & inst_j & = & inst_i \quad \text{if } inst_i = inst_j \\
inst_i & \sqcap & inst_j & = & \bot \quad \text{if } inst_i \neq inst_j
\end{array}
$$

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 $\bot$. Values defined by a copy instruction will have their tags *lowered* to *inst* or $\bot$, 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 $\bot$.

This process tags each value is the SSA graph with either an instruction or $\bot$. If a value's tag is $\bot$, 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.

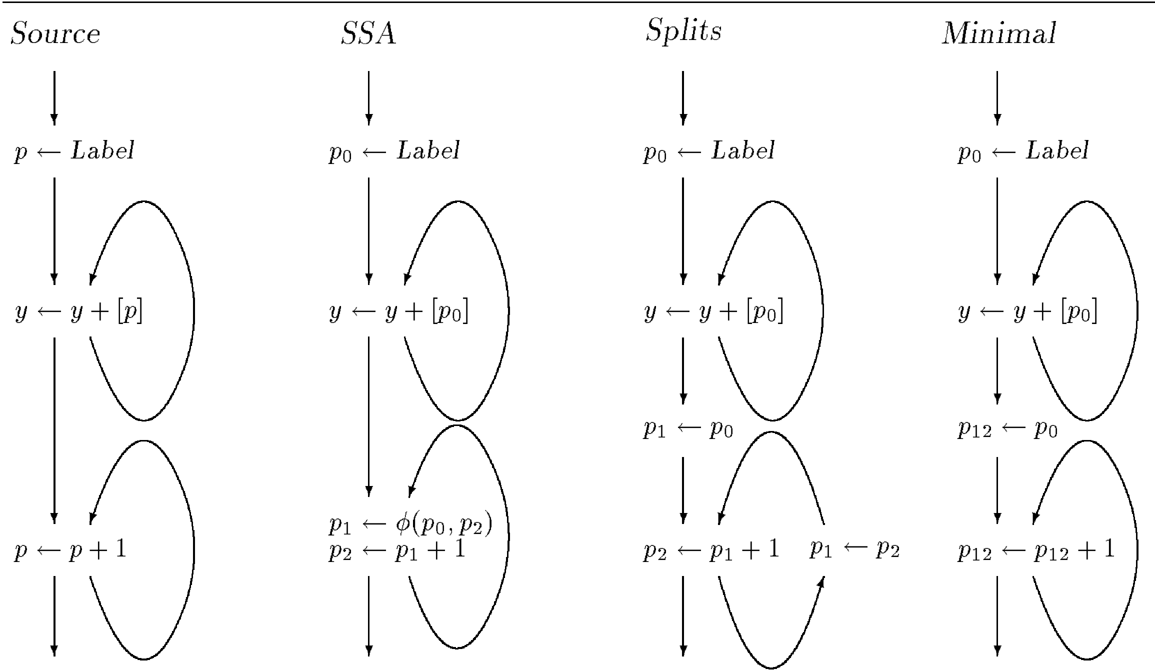

Figure 5.2 Introducing Splits

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 $\bot$) 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.

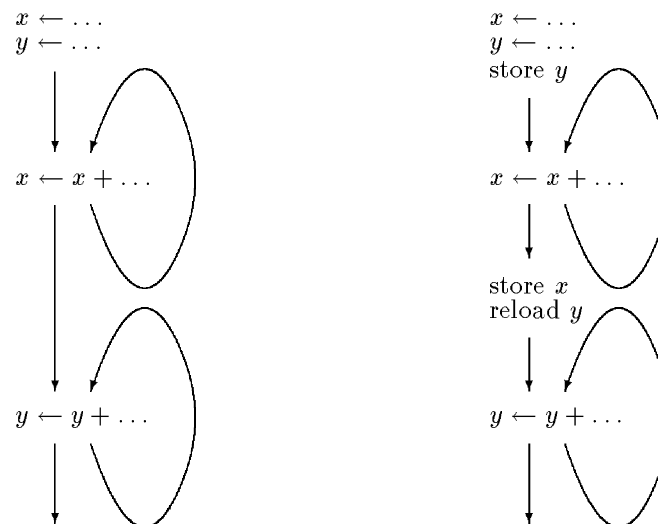

Figure 6.1 Splitting

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].

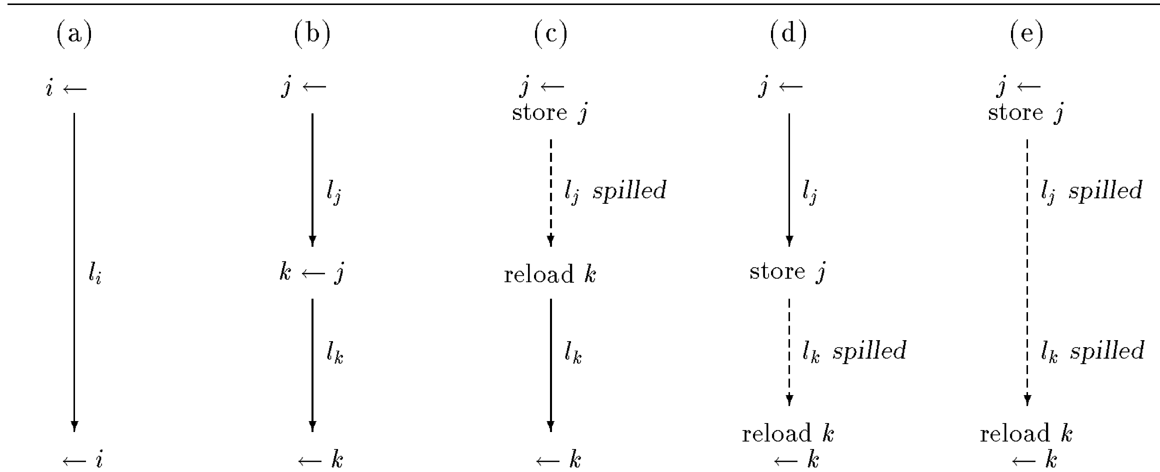

Figure 6.2 Spilling Partners

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.

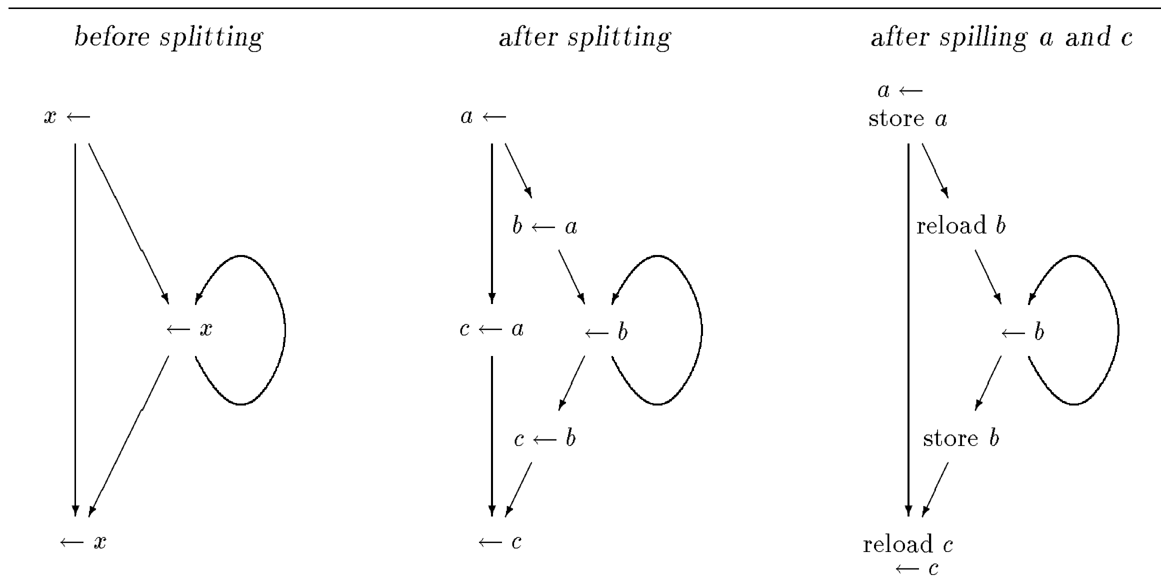

Figure 6.3 Splitting and Spilling

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.

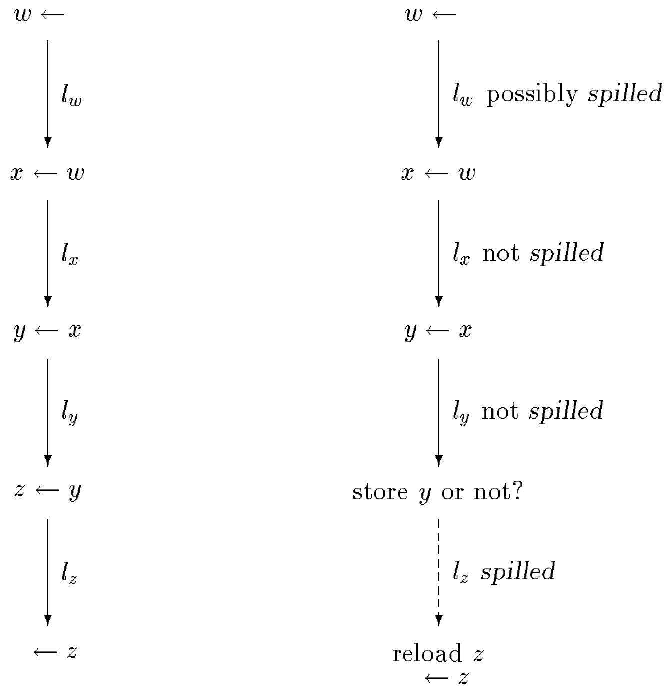

Figure 6.4 Globally Unnecessary Stores

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.

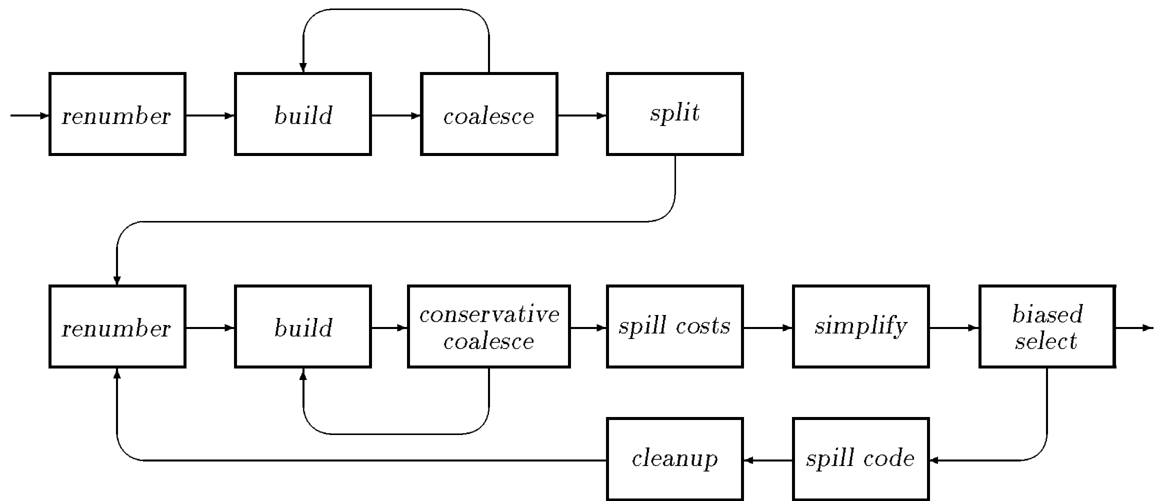

Figure 6.5 The Splitting Allocator

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

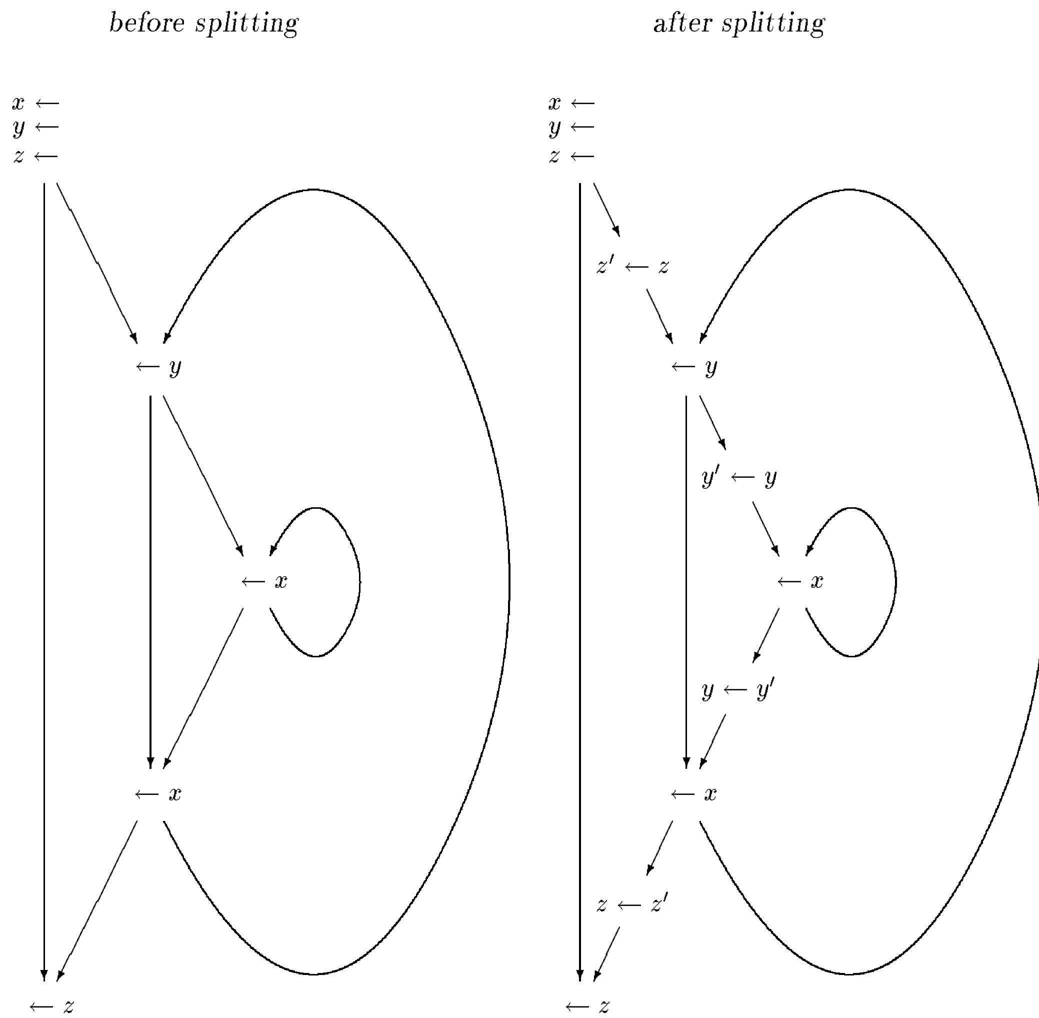

Figure 6.6 Splitting Unmentioned Live Ranges

> 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.

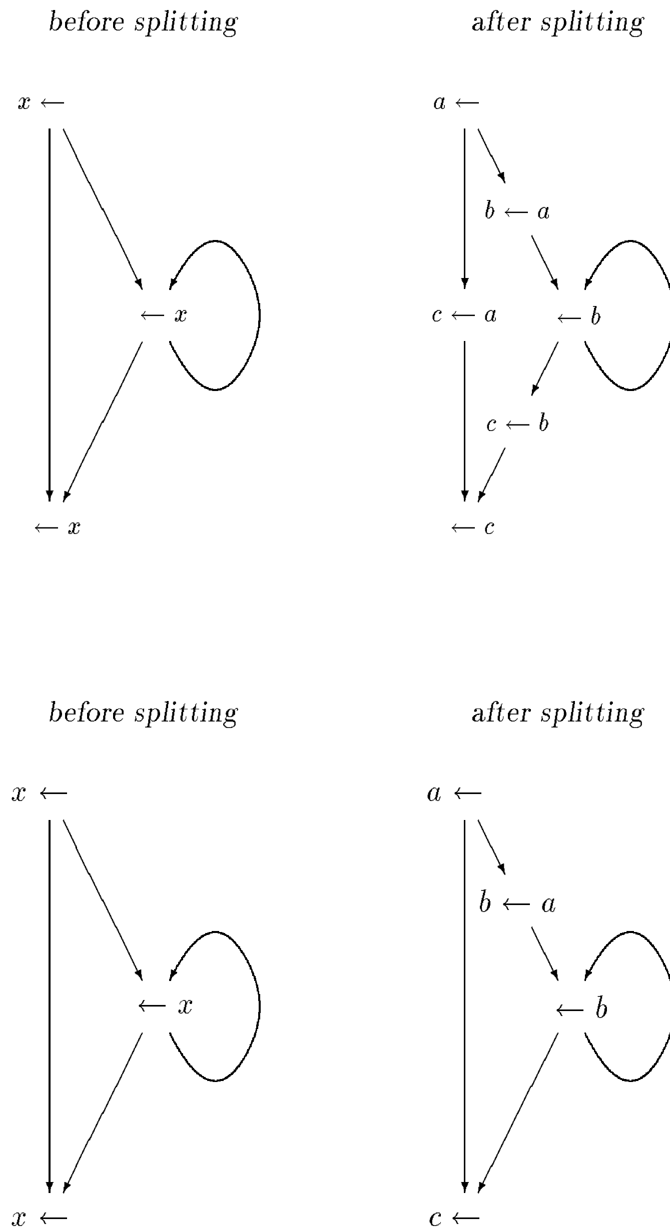

Figure 6.7 Splitting at Dominance Frontiers

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 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.