Global Register Allocation Using Program Structure

by

Jason Eckhardt

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

Approved, Thesis Committee:

Keith D. Cooper, Professor, Chair
Computer Science

Ken Kennedy, Professor
Computer Science

Walid Taha, Assistant Professor
Computer Science

Rice University, Houston, Texas
April, 2005
Global Register Allocation Using Program Structure

Jason Eckhardt

Abstract

The Chaitin-Briggs approach to register allocation by graph coloring is the dominant method used in industrial and research compilers. It usually produces highly-efficient allocations, but sometimes exhibits pathological spilling behavior so that some programs execute significantly more spill operations than is necessary. This thesis examines and improves two previously proposed approaches of attacking this problem. Passive splitting attempts a lazy form of live range splitting which can substantially reduce dynamic spill count compared to Chaitin-Briggs. We incorporate program structure into the passive splitting framework to better guide splitting decisions and to place splits at infrequently executed regions of code. Also investigated is the Hierarchical Graph Coloring approach, which uses program structure during allocation. We provide an empirical evaluation of this poorly-understood algorithm, and propose some improvements.
Acknowledgements

I would like to thank my principal advisor Keith Cooper for his advice, mentorship, and support. Keith’s optimism, enthusiasm, and encouragement kept my spirits high throughout my time at Rice. Ken Kennedy and Walid Taha form the rest of my committee and provided useful advice and comments. John Mellor-Crummey employed me at one time, gave advice, and taught me much about parallel programming and high-performance computing in general. Anshu Dasgupta introduced me to the LLVM system, discussed useful algorithmic and technical details of all sorts, and his Perl sorcery made data collection for research much easier. Anshu also designed and wrote the Chaitin-Briggs allocator that served as the basis for the Callahan-Koblenz allocator implemented for this work. Tim Harvey answered numerous questions about the ILOC compiler, the Chaitin-Briggs allocator, and the original passive splitting implementation. Michael Berg, Preston Briggs, David Callahan, and Brian Koblenz of Cray answered many questions about the Callahan-Koblenz allocator. In particular, Brian patiently provided a number of very detailed examples, explanations, and code fragments regarding the more subtle aspects of their allocator. His email exchanges with me cleared up many of the mysteries left by the original Callahan-Koblenz article. Chris Lattner of the LLVM compiler group answered numerous questions about LLVM internals, making the implementation work here easier. Iva Jean Jorgensen made sure that the paperwork and administrative details required of a graduate student were understandable and completed at the proper times, probably saving me from missing more than one crucial deadline. Finally, thanks to fellow graduate students Mike Brogioli, Daniel Chavarria-Miranda, Cristian Coarfa, Yuri Dotsenko, Apan Qasem, Todd Waterman for their mentorship and interesting conversations
ranging from technical to political to philosophical.
Contents

Abstract i
List of Illustrations vi
List of Tables viii

1 Introduction 1
  1.1 Register Allocation ............................................. 1
  1.2 Overview ........................................................ 3
    1.2.1 Passive Splitting ............................................ 3
    1.2.2 Hierarchical Graph Coloring ................................. 3

2 Background 5
  2.1 Chaitin-Briggs Allocator ....................................... 5
  2.2 Deficiencies of the Chaitin-Briggs Spilling Scheme ............ 7

3 Passive Splitting 9
  3.1 Chaitin-Briggs Allocator With Passive Splitting .............. 10
    3.1.1 Overview of the Algorithm ................................ 11
  3.2 Some Improvements ............................................ 13
    3.2.1 The Improved Algorithm ................................... 16
  3.3 Experiments ................................................... 21
  3.4 Other Issues and Modifications ................................ 24
    3.4.1 Representing Call-clobbered Registers at Call Sites .... 24
    3.4.2 Two-address Instructions/Modified Input Operands ....... 25
  3.5 Summary ....................................................... 26
4 Hierarchical Graph Coloring

4.1 Overview of the Callahan-Koblenz Allocator .................. 33
4.1.1 Preferencing ............................................. 37
4.2 Experimental Comparison ....................................... 38
4.2.1 Dynamic Spill Code Comparison .............................. 39
4.2.2 Control-flow Overhead of Tiling ............................. 46
4.3 Some Modifications and Improvements ......................... 48
4.3.1 Cycles of Copies at Tile-boundaries ........................ 48
4.3.2 Spill Cost Tweaks ........................................ 54
4.4 Summary ...................................................... 56

5 Related Work .................................................. 57

6 Conclusion ...................................................... 61

6.1 Contributions ................................................ 61
6.2 Future Work .................................................. 62
6.3 Final Thoughts ............................................... 64

Bibliography .................................................... 65
Illustrations

2.1 The Chaitin-Briggs Allocator ........................................... 5

3.1 Example of passive splitting ........................................... 11
3.2 The Passive Splitting Allocator ........................................ 12
3.3 Example of improved passive splitting ............................ 14
3.4 Spill reconsideration opportunity .................................... 15
3.5 The Improved Passive Splitting Allocator ......................... 17
3.6 Algorithm for loop annotation ....................................... 18
3.7 Algorithm for computing split costs ............................... 27
3.8 Algorithm fragment for color selection ............................ 28
3.9 Algorithm for determining splits ................................... 29
3.10 Algorithm for inserting split code ................................. 30
3.11 Algorithm for placing split code ................................... 31

4.1 Example tile tree .......................................................... 34
4.2 The Callahan-Koblenz Allocator ..................................... 35
4.3 Example of CK advantage .............................................. 43
4.4 Example of CK advantage .............................................. 44
4.5 Dynamic Spill Operation Types by Percentage .................... 45
4.6 Static and Dynamic Branches ......................................... 47
4.7 Tile-boundary Copies .................................................... 49
4.8 Algorithm for basic cycle breaking with swap .................... 51
4.9  Tile-boundary Copies with Aliases .......................... 52
4.10 Decomposing Tile-boundary Copies ............................ 53
Tables

3.1 Dynamic spill operations for wave5, g271decoder, tomcatv . . . . . 23
3.2 Compile-time impact of improved passive splitting (seconds) . . . . . 24

4.1 Dynamic spill operations for SPECInt2000 and epic (billions) . . . 42
4.2 Dynamic spill operations for bzip and epic (billions) . . . . . . . . . 55
Chapter 1

Introduction

1.1 Register Allocation

Typical modern compilers operate by parsing the source text and translating it into an intermediate representation, applying a series of code-improving\textsuperscript{1} transformations to the intermediate representation of the source, and finally passing the code to a back-end which generates object code for a particular target machine. One of the last functions performed by the code generator is register allocation, which chooses the program variables that will live in the processor’s scarce, high-speed registers. It is crucial that heavily-used variables reside in registers so that the number of dynamic memory operations performed by the program is as small as possible. Moreover, register allocation is becoming increasingly important as the ratio of main memory speed to processor speed increases.

There are many methods for global register allocation, but since the mid-1980s or so, an approach based on graph coloring has been the dominant paradigm. In this approach, nodes in an interference graph represent live ranges\textsuperscript{2} of variables in the program, with an edge between a pair of nodes indicating that the two live ranges cannot occupy the same register. By assuming $k$ colors, where $k$ is the number of registers in the target machine, an allocation can be achieved by obtaining a $k$-coloring of the interference graph and then mapping each color to a machine register. Sometimes a $k$-coloring is simply not possible, in which case the graph must be

\textsuperscript{1}A pass that takes the IR as input, and outputs the same IR in a potentially more efficient form.

\textsuperscript{2}A live range is a collection of definitions that reach a common use.
modified to make it $k$-colorable. Such modifications are done by relegating certain nodes (live ranges) to memory—or spilling.

The graph coloring approach is now dominant for good reasons. First, the concept of graph coloring is well understood. Thus, casting the seemingly complicated idea of register allocation as a graph coloring problem is desirable. Secondly, unlike some other approaches to register allocation, these allocators are applicable to entire compilation units. There are no restrictions to basic blocks or other small regions. Finally, heuristic graph coloring allocators are now well-developed, efficient, and generally produce good allocations.

The best known and most widely used graph coloring register framework is that proposed by Chaitin and colleagues [9] and extended significantly by Briggs and his colleagues [6]. The framework has become known in the literature as the Chaitin-Briggs approach. While the Chaitin-Briggs allocator produces remarkably good results much of the time, it exhibits limitations in its spilling capability. Its major weakness lies in the mechanism for deciding what to spill and where to locate those spills. Thus, it may insert more spill instructions than strictly necessary to get an allocation, and sometimes these are placed in unfortunate locations so that the dynamic number of spill operations is higher than it might otherwise be.

This sometimes pathological spilling behavior arises due to a loss of information about the structure$^3$ of the original program. The allocator operates primarily on the interference graph, which only shows whether or not a pair of live ranges can be situated in the same register. But the graph gives no indication about the original control flow or high-frequency program regions. Such information can often be used, for example, to place spill operations in less frequently executed regions. The primary question that this thesis attempts to resolve is whether or not program structure can be incorporated into the Chaitin-Briggs allocator so as to better determine which values to spill and where to spill them.

$^3$Program structure refers to control-flow constructs, such as loops.
1.2 Overview

To lay the foundation for the remainder of the thesis, Chapter 2 reviews the Chaitin-Briggs allocator, and discusses some of its deficiencies, such as its spilling mechanism and cost heuristics. The technical content is divided into two main chapters (3 and 4). Chapter 3 reviews a passive live range splitting allocator, and presents improvements to the algorithm. Chapter 4 then investigates a hierarchical graph coloring allocator, and evaluates its effectiveness compared to a CB allocator. Improvements are also proposed for the hierarchical allocator. Chapter 5 surveys the literature closely related to this work. Finally, Chapter 6 wraps up and gives some final perspectives on the work.

1.2.1 Passive Splitting

Cooper and Simpson introduced a live range splitting technique called passive splitting which improves on the sometimes catastrophic spill behavior of a vanilla Chaitin-Briggs allocator [12]. By examining code produced by their allocator, we make a number of observations showing that their results can be improved. To that end, we propose and evaluate extensions to their technique that directly incorporate program structure. It will be seen that this additional structural information can provide significant reductions in spill cost.

1.2.2 Hierarchical Graph Coloring

An ambitious global register allocator introduced by Callahan and Koblenz in [7] is examined in detail in Chapter 4. While the article makes fairly convincing arguments regarding the potential performance of the algorithm, an empirical evaluation is conspicuously absent. In his doctoral dissertation, Briggs makes this statement [3]:

\textit{We are interested in implementing the allocator described by...Callahan and Koblenz in the context of our compiler. This would finally allow useful...}
comparisons between radically different approaches.

To date, no researchers have stepped up and published such an evaluation, and so more than a decade after the article was published, there is little understanding of the either the algorithm or its behavior. For example, unlike the Chaitin-Briggs allocator, we do not know of implementations of CK in any current industrial compiler—other than the one done by Callahan and Koblenz. Also, the description of CK given in the article is at a relatively high level, so that there are a number of engineering details that are not obvious. We implement the allocator side-by-side with a Chaitin-Briggs allocator in the same framework and perform an extensive comparison, filling the gap in the literature. We also propose some simple extensions to the technique.
Chapter 2

Background

All of the work in this thesis can be considered an extension of the now standard Chaitin-Briggs register allocation framework [9, 8, 5, 3, 6]. Therefore, to better understand the extensions and their place in the overall framework, this chapter presents a brief overview of the baseline Chaitin-Briggs ("CB") allocator.

2.1 Chaitin-Briggs Allocator

![Diagram of Chaitin-Briggs Allocator]

Figure 2.1: The Chaitin-Briggs Allocator

1. *Renumber:* A live range is a set of definitions of a virtual register that reach a common use of that virtual register. All virtual registers referenced in the compilation unit being considered for register allocation are examined to determine live ranges, and each live range is given a unique name. Since the original code may reuse a register many times by redefining it, this renaming allows each disjoint use of that register to be assigned to a register independently.

2. *Build:* The interference graph is stored as a triangular bit matrix representing interferences between live ranges. Each node in the graph represents a live range,
and an edge between two nodes implies they cannot share the same register. Building the graph involves iterating through each instruction in the routine (in reverse) adding interferences for a definition in the current instruction with any currently live ranges.

3. **Coalesce**: Register-to-register copies are often useless and can be removed. A copy can be removed if the source and target ranges do not interfere. This phase removes such copies and updates the interference graph by unioning the coalesced nodes together. Coalescing changes the interference graph, and one round of coalescing may uncover more opportunities, so the build and coalesce phases are performed iteratively until no more changes occur.

4. **Spill costs**: If the allocator must eventually spill some ranges to memory, it needs a metric to determine which nodes to spill. For each live range, this phase determines the estimated cost of spilling the range. The cost is the number of instructions that would be needed to spill a live range, where the cost of each instruction is weighted by its frequency of execution. This is an estimate because the frequency of execution is taken to be $10^{\text{depth}(i)}$ where $\text{depth}(i)$ is the loop nesting depth of the spill instruction $i$.

5. **Simplify**: This is the first part of a two-phase coloring process. Assuming $k$ registers, nodes (and their edges) with fewer than $k$ neighbors are repeatedly removed from the interference graph and pushed onto a stack. Such *trivially colorable* nodes will receive a color regardless of how the remaining nodes are colored. If this process *blocks*, that is, every node in the residual graph has at least $k$ neighbors, then a node is heuristically chosen (see Section 2.2) as a potential spill candidate. The node picked is the one deemed least costly to spill among the remaining nodes. The node is removed from the graph (possibly making other nodes trivially colorable), pushed onto the stack, and the process continues until the graph is empty.
6. **Select**: Once the interference graph is emptied, the second-phase of the coloring process begins. Nodes are repeatedly popped from the stack, inserted (along with their edges) back into the graph, and colors assigned to them. A color is chosen for a node that is different than all of its neighbors. If no such color is available, then the node is marked for spilling. The process completes when the coloring stack is empty.

7. **Spill code**: For any live range that was marked for spilling, the actual code must be updated to reflect this. These ranges will now live in memory, so (roughly) a store is inserted at each definition of the range and a load is inserted at each use. This method of spilling is sometimes referred to informally as the “spill everywhere” strategy, which we also adopt in the text.

The allocator also performs rematerialization, so that certain simple expressions that were marked for spilling can be recomputed rather than being stored to and loaded from memory.

### 2.2 Deficiencies of the Chaitin-Briggs Spilling Scheme

It was mentioned earlier that if the simplify step blocks, then a node must be heuristically chosen as a potential spill candidate. This is accomplished by choosing the node \( n \) that minimizes the ratio

\[
\frac{\text{cost}_n}{\text{degree}_n}
\]

where \( \text{cost}_n \) is the penalty for accessing \( n \) from memory and \( \text{degree}_n \) is the current degree of \( n \) in the graph.

While this metric is often effective, there are limitations. For example, the degree of \( n \) includes neighbors from the entire function being considered so that regional (i.e., pertaining to code sections smaller than the function) usage patterns of \( n \) are not apparent. This may cause undesirable choices for spill candidates, such as picking a live range that is used within a loop for spilling over a live range that is completely
unmentioned in a loop. Clearly this leads to more dynamic spill code operations being executed than is necessary. Others have shown that changing this spill metric can improve the quality of allocation [2]. However, no single metric has emerged as the “right” replacement for $\frac{\text{cost}}{\text{degree}}$.

The *spill code* phase has the same problem as the spill choice metric— it does not take advantage of regional usage patterns.\(^1\) So it cannot, say, spill a live range in one region while leaving it in a register for another region.

Because the interference graph is constructed only by considering whether or not two live ranges are simultaneously live, virtually all information about the *structure* of the original program code is lost. It is that loss of information that prevents the spill mechanism from performing as well as it might otherwise.

Most of the research done on graph coloring register allocation after Briggs has been targeted at improving on the limitations of the spilling approach (discussed in Chapter 5). The remainder of this thesis presents techniques for dealing with the cases when the CB spiller performs poorly by attempting to incorporate program structure into the allocation process.

\(^1\)Chaitin does use a few simple local heuristics to avoid some redundant spills, such as reloading a value only once when two uses of that value are close together. But these do not solve the underlying problem.
Chapter 3

Passive Splitting

There are a variety of ways to improve the spill behavior of CB. One general way of attacking the problem is by live range splitting—breaking certain live ranges into smaller pieces so that each piece can be allocated independently [11, 10, 14]. Such splitting may enable a graph to be colored with fewer colors, or allow more intelligent placement of spill instructions.

Briggs [3] describes an aggressive approach to splitting. In this method, live ranges are split before the coloring phase by the insertion of split operations (copies) at certain points in the program. For example, splits can be inserted at the boundaries of loops for global live ranges that span the loop. For a live range x, this allows the allocator to consider the portion of x within the loop separately from the portion outside the loop. Briggs explored a number of ways of choosing split points, such as by using loop boundaries, dominator frontiers, etc. The technique is considered aggressive in that every range live at a split point would be split. While the experiments showed significant wins for the strategy, it also showed significant losses due to excessive split copies remaining in the code. Moreover, the aggressive splitter expands the size of the interference graph significantly, with the concomitant problems in compile time and space requirements.

In an effort to obtain some of the benefits of splitting, while not incurring the drawbacks of aggressive splitting, Cooper and Simpson introduced a lazy or passive approach [12]. Their results indicate that a less aggressive approach can still significantly reduce dynamic spill operations compared to the standard Chaitin-Briggs allocator, without incurring the big losses of being too aggressive. This chapter in-
roduces some modifications that boost its effectiveness even more.

Section 3.1 provides an overview of the original passive splitting allocator. Section 3.2 discusses opportunities for improvement and presents a modified version of the algorithm. Experimental results for the new technique are given in Section 3.3. Finally, Section 3.4 discusses some implementation issues.

3.1 Chaitin-Briggs Allocator With Passive Splitting

Passive splitting ("PS") was designed to cope with some of the situations in which Chaitin's "spill everywhere" approach performs poorly. Figure 3.1a portrays one such case. Suppose for the sake of illustration that the allocator has only one register available. Since there are two conflicting live ranges $x$ and $y$, CB will completely spill one or the other. Assuming $y$ is chosen for spilling, a store of $y$ will be inserted after every definition and a load after every use, producing the code in Figure 3.1b. Unfortunately, this approach leads to a new load operation that will execute on each iteration of the loop.

Simpson observed that this undesirable situation can be avoided by splitting $x$ around $y$. That is, because $x$ is not used until the second loop, it need not occupy a register until after the first loop (after $y$ dies). This allows $y$ to occupy the register during its lifetime in the first loop. By splitting in this fashion, spill operations are placed outside of either of the loops, and both live ranges occupy a register in the frequently executed portions of their lifetimes, as shown in Figure 3.1c.

The key to Simpson's approach is solidifying the notion of when one live range can be split around another. In the example, splitting was allowed because $y$ is contained in $x$—all the uses and definitions of $y$ occur entirely between any uses or definitions of $x$. 
Figure 3.1: Example of passive splitting: (a) original; (b) spill $y$ completely; (c) split $x$ around $y$.

3.1.1 Overview of the Algorithm

Recall the standard Chaitin-Briggs allocator from Figure 2.1. Passive splitting is implemented with a small number of changes to the overall CB framework, as depicted in Figure 3.2. Items in boldface represent phases that were changed or added to CB.

In addition the the interference graph used by CB, Simpson builds a containment graph that indicates the containment relationships between any pair of live ranges. The graph is built using an algorithm nearly identical to that for building an interference graph. It is the data structure used during splitting to determine whether or not a split is feasible. The graph is actually built at the beginning of the split costs phase described below.
It was not mentioned in Simpson's original paper, but a special phase is run before register allocation to insert an otherwise empty block in the middle of any critical edge\(^1\) in the control flow graph. This is done to ensure that there is always a proper location in which to place split operations.

1. **Split costs**: For each live range \(l\), this phase determines the cost of splitting a live range \(s\) around \(l\), where the cost is the number of LOAD and STORE instructions (weighted by loop nesting depth) needed to perform the split. The underlying implementation operates similarly to the spill costs phase in CB.

2. **Select**: This phase operates as in CB, with one change. When a node \(n\) is encountered that cannot receive a color, find splits is called in an attempt to find a color for \(n\) by splitting. If splitting was successful, \(n\) will now be assigned a color and is no longer marked for spilling. If splitting was not successful, \(n\) is marked for spilling as usual.

3. **Find splits**: Invoked by select, this is the key routine that determines whether splitting can free a color for a node \(n\) that did not receive a color. Utilizing

---

\(^1\)An edge \(e\) is a **critical edge** if the block at the head of \(e\) has multiple successors and the block at the tail of \(e\) has multiple predecessors.
the containment graph and the computed split costs, it will try to either split interfering live ranges around \( n \), or split \( n \) around interfering live ranges, choosing the least costly choice. Further, the choice is only acceptable if it costs less than spilling \( n \) everywhere.

4. *Split code*: Once a split decision has been made, the actual instructions must be inserted into the code. This operates similarly to the *spill code* phase of CB. For each live range \( s \) that was split around \( l \), a \texttt{STORE} of \( s \) is inserted before every definition of \( l \), and a \texttt{LOAD} of \( s \) is inserted after every death of \( l \).

### 3.2 Some Improvements

Simpson reports good results for passive splitting, and experiments by this author confirm that PS can significantly reduce dynamic spill operations compared to CB. Even so, examination of some benchmarks reveals that even better results are achievable.

Consider the code in Figure 3.3a. Suppose that during *select* node \( t \) did not receive a color, and that *find splits* determines that splitting \( x \) around \( t \) is less costly than spilling \( t \) everywhere.\(^2\) Recall that *split code* will insert a \texttt{STORE} before every definition of \( t \) and a \texttt{LOAD} after every death of \( t \). In this example, that has the unfortunate drawback of placing all the split instructions within the loop (Figure 3.3b). But by observing that \( x \) has no reference (use or definition) within the first loop, a much better placement for the split operations is just outside the loop (Figure 3.3c). That is, the dynamic number of spill operations will be decreased because the split code is placed in less frequently executed regions of the program. In the case of a deeply nested loop, the splits can be pushed outside of more than one loop in the nest as long as there are no references to the split range in that loop.

\(^2\)The split is legal since \( x \) contains \( t \).
Figure 3.3: Example of improved passive splitting: (a) original; (b) split $x$ around $t$ (old); (c) split $x$ around $t$ (new).

Using loops to guide the splitting is convenient in that it is relatively easy to differentiate high frequency regions from those of low frequency with purely static control flow analysis. Unless a loop has a tiny trip count (e.g., zero or one), or the body of the loop is guarded by a rarely true condition, then it is a fairly safe to assume that moving split code out of a loop is better when possible.

It is also possible to use other program structure to guide the splitting, such as with conditional regions. However, unlike loops, it is not necessarily obvious by static analysis which part of a conditional executes more frequently than the other. By incorporating profile feedback from a training run, the allocator could decide how to place split code in or around conditionals. The allocator framework used in the present work does not currently use profiling feedback.
Figure 3.4: Spill reconsideration opportunity (k = 2): (a) live ranges; (b) containment graph CG; (c) live ranges after first pass (e in memory)

Note that better placement of split code might also be achievable using an approach such as partial redundancy elimination [19]. There are at least three reasons why our approach is preferred in the context of passive splitting. First, the cost of PRE is high, relative to the cost of our approach, involving large bit vectors and a separate pass. Second, PRE itself may increase register pressure, necessitating a reallocation. Finally, the passive splitter already has all the data structures and other information about live ranges necessary to make splitting decisions quickly and easily.

Another opportunity for improvement arises in select. Consider the live ranges depicted in Figure 3.4a. Suppose that the number of colors is k = 2 (\{1,2\}), the coloring stack created by simplify is \(d, c, e, a, b\) (\(d\) is the stack top), and that only \(c\) is in a loop. Now during select, \(d\) is popped off the coloring stack first and assigned color 1. Next \(c\) is popped and assigned color 2. When \(e\) is popped, it cannot be assigned a color since all colors are used by neighbors \(d\) and \(c\). At this point, find splits attempts to free a color by considering a split of \(e\) around \(c\). While the split is feasible (\(\langle e, c \rangle \notin CG\)), it is more costly than just spilling \(e\), since \(c\) is at a greater loop
depth. Thus, $e$ is marked for spilling. Continuing, $a$ is popped and assigned color 2, the only possibility. Finally, $b$ is popped but cannot receive a color. This time, however, find splits determines that a split of $d$ around $b$ is both legal ($\langle d, b \rangle \notin CG$) and less costly than spilling $b$. Thus the split is noted and $b$ is assigned color 1. At this point, the stack is empty, and once spill and split code is inserted, the next phase of allocation will build and color the resulting graph successfully (i.e., without introducing any more splits or spills).

Rather than accepting the spill of $e$ as just described, an examination of the live ranges just after the spill and split code is inserted (Figure 3.4c) reveals that we can do better. Suppose that instead of marking $e$ for spilling, it is reconsidered for allocation in the next pass. This time $e$ is a candidate for splitting around the second part of $d$ (the range starting at the load of $d$). Not only that, doing the split is less costly than spilling $e$.

The previous example shows that splitting enabled a node that was destined for spilling to be allocated to a register—but only because it was reconsidered rather than spilled in the first pass. In other words, instead of pessimistically spilling all nodes that were marked for spilling we optimistically assume that splitting (if any) has enabled one or more of them to become colorable. This seemingly simple modification to select makes a significant difference on the benchmarks used here.

3.2.1 The Improved Algorithm

Figure 3.5 depicts the improved passive allocator, with changes from the original marked in boldface. A detailed explanation of the new functionality is given next. It was implemented in the Rice Scalar Compiler Group's ILOC compiler, starting with the original passive splitting code written by Simpson.
A New Phase: Build Loop Tree

In order to make use of loops when computing spill costs and inserting split code, a new phase build loop tree constructs a convenient representation of the program. This data structure, the LoopTree, represents the hierarchical loop nesting structure of the program. Each node in the LoopTree represents a loop. A node $c$ is a child of node $p$ if the loop represented by $c$ is contained within the loop represented by $p$, and $c$ is contained in no other loops. If two disjoint loops have the same containing loop, then they will be sibling nodes with the same parent node. Each node $t$ also contains the following: $t.blocks$ is a list of all the basic blocks contained in this loop, but not any of its inner loops; $t.parent$ points to the parent node; $t.depth$ is the loop nesting depth (depth 1 is an outermost loop). The actual control flow analysis method used here to determine loops is based on DJ-graphs [21], although a number of other techniques would work as well. This pass is performed once before the main register allocation starts.

$^3$To simplify this discussion, it is assumed that all loops are reducible [22]. However, reducibility is not a requirement for the algorithm.
annotateLoopTree(t)
    t.refs ← {}
    t.spilled_on_entry[*] ← {}
    t.reloaded_on_exit[*] ← {}
    For each inner loop c of t
        annotateLoopTree(c)
        t.refs ← t.refs ∪ c.refs
    For each block b in t.blocks
        For each instruction i in b
            For each live range l defined in i
                t.refs ← t.refs ∪ l
            For each live range l used in i
                t.refs ← t.refs ∪ l

Figure 3.6: Algorithm for loop annotation

Split Costs

Before starting, split costs needs to annotate the LoopTree with additional information about virtual register usage within the loops. This information will be queried to determine the legality of pushing a split out of a loop or loops. Figure 3.6 shows the annotation algorithm. The algorithm computes (or initializes) three additional sets of information for each loop. Member t.refs is the set of all virtual registers referenced (used or defined) in the subtree rooted at t. Set t.spilled_on_entry[e] contains the name of each VR that has previously been spilled to memory on edge e before the loop is entered. Likewise, set t.reloaded_on_exit[e] contains the name of each VR that has previously been reloaded from memory on edge e after the loop is exited.

The algorithm will perform a postorder (bottom-up) traversal of the loop tree, propagating information up the tree. For each loop, all ranges which have have a reference in the current loop t are added to t.refs. After each subtree of t has been processed, its’ ref set is incorporated into the current set for t.
Simpson’s original method of performing splitting is based on the idea that for every live range \( s_i \) split across live range \( l \), \( s_i \) will be stored immediately before a definition of \( l \), and reloaded just after a death of \( l \). Thus, for some \( s_i \), the same split code will be inserted around \( l \) as any other \( s_j \) (except, of course, the name). To represent split costs then, a single quantity was stored with each live range \( l \). That quantity being the cost of splitting any \( s_i \) around \( l \).

The key to the new approach is realizing that it can be beneficial to split \( s_i \) around \( l \) in a different way than splitting \( s_j \) around \( l \). For example, a death of \( l \) may occur within a loop, while a splittable neighbor \( s_i \) has no references in the loop. At the same time, another splittable neighbor \( s_j \) has a reference in the loop. For the former case, it is desirable to insert the split operations for \( s_i \) outside the loop, while a reload of \( s_j \) would be required (as originally) just after the death of \( l \).

To model the previous scenario, a single cost stored with \( l \) is no longer sufficient. Instead, costs are stored for every neighbor \( n \) of \( l \), and split costs will compute a distinct cost for splitting \( n \) around \( l \).

Figure 3.7 shows the new method of computing costs. For every basic block in the compilation unit, split costs iterates through the instructions in reverse order and maintains a set live of currently live ranges. If either a definition or death of \( l \) is detected in the current instruction, then every neighbor of \( l \) is examined to determine a split cost for that neighbor. There are currently two options. Either the split cost will be based on the default location (i.e., just before the definition of \( l \) or just after the death of \( l \)), or the loop tree will be consulted to determine whether the split cost can be decreased by pushing the split operation out of a loop.

To determine whether pushing splittable neighbor \( s \) out is possible, the algorithm traverses the loop tree in a bottom-up fashion. It first obtains the loop tree node \( p_i \) containing \( l \), then iteratively checks whether \( s \) has any references in \( p_i \), and then moves to the parent loop \( p_{i-1} \). If \( p_{i-1} \) contains a reference of \( s \), then the iteration stops and \( p_i \) is the outermost loop that \( s \) can be pushed out of.
Select

As described earlier, it can be beneficial to reconsider spill decisions after splitting. Thus, at the end of select (Figure 3.8), the spill set is cleared if any splits were made. This is done only on the first build-color pass, in order to minimize compile-time impact. The rationale being that most splitting will happen on the first pass, and allowing reconsideration on every pass has diminishing returns. However, it may still be worthwhile to provide a command line option to allow reconsideration on every pass.

Find Splits

This phase operates very similarly to the original passive splitter, except that when considering the neighbors $n$ of $l$, it must utilize the new per-neighbor cost for each $n$.

Split Code

As seen in Figures 3.10 and 3.11, the high-level operation of split code is similar to split costs. That is, definitions and deaths of a live range $l$ are detected on a reverse pass through the instructions in a basic block. This time, however, the actual split instructions are inserted into the program.

For a live range $s$ split around $l$, if no pushing out is possible, then a LOAD of $s$ is inserted into the instruction stream just after a death of $l$ and a STORE of $s$ just before any definitions of $l$. In either case, if a split range $s$ can be rematerialized, then no STORE of $s$ is necessary and LOADs are replaced with cheaper LOAD–IMMEDIATEs.

For splits which can be pushed out of a loop, code insertion is slightly more complicated. Placing a split operation “just outside” a loop $L$ means either placing it on an edge entering $L$ from outside of $L$ (loop entry edges), or placing it on an edge exiting to a block outside of $L$ (loop exit edges). Since it is assumed that all critical edges have been split before register allocation, there is always a place to insert such operations— in the block at the head (tail) of an entry (exit) edge.
Consider the case in which $s$ is split around $m$ in loop $L$ and the STORE of $s$ is pushed out of $L$. Function placeSplit iterates over every loop entry edge $e$ of $L$, checking if $s$ is live on edge $e$. Range $s$ is live on edge $e$ if $s \in (liveOut_{e,pred} \cap liveIn_{e,succ})$. Only the $liveOut_{e,pred}$ set actually needs to be checked, though, since $s$ is known to be live throughout $L$ (it has no references in $L$ yet it interferes with $m$). If $s$ is live on $e$, then the STORE of $s$ is inserted at the end of basic block $e.pred$ (the block at the head of $e$).

Similar processing happens when a LOAD is pushed out of $L$. This time, however, the operation will be placed on the exit edges of $L$ where it is live. Liveness on an exit edge $e$ is checked by examining $liveIn_{e,succ}$.

Any time an operation for $s$ is placed outside a loop $L$ on edge $e$, $s$ is added to either the set $L.spilled.on.entry[e]$ or $L.reloaded.on.exit[e]$. Since $s$ may have been split across more than one range, the processing above would normally insert a split operation at the loop boundary for every live range $s$ was split around. Such redundant operations are suppressed by checking whether $s$ has already been spilled (or reloaded) around $L$.

3.3 Experiments

To test the effectiveness of the passive splitting improvements, we compared Simpson's original splitter to the new splitter, both of which are implemented in the ILOC compiler. This compiler targets a RISC-like instruction set called ILOC. An ILOC program is executed by a running it through an instruction set simulator as opposed to actual hardware. The compiler first processes the original program with a series of optimization passes, and gives the result to the register allocator for processing. The optimization flags used were `-r[-RD]v[-vvvvsmp]zc[-mf]dv[-vvvvsmp]zc[-f]dn`, which causes these passes to run (in order, with repetition): reassociation, value numbering, lazy code motion, constant propagation, dead code elimination, value numbering, lazy code motion, constant propagation, dead code elimination, and con-
trol flow clean-up.

Simpson’s original study benchmarked program wave5 from SPEC95. We chose the same program and also added tomcatv (SPEC92), and g271decoder (Media-Bench). For wave5 we used 25 integer and 25 floating point registers, while 13/13 and 14/14 were used for g271decoder and tomcatv, respectively. The register file size choice for wave5 was made to obtain spilling results as similar to Simpson’s original study as possible. For the other two benchmarks, we chose register file sizes so as to provoke some amount of spilling. It might also be useful to perform a sensitivity study by reducing the number of registers in increments for each benchmark, although we have not done so here.

The first three columns of Table 3.1 show the number of dynamic spill operations executed by each allocator, where each row represents one procedure from a benchmark program. Only procedures which executed any spill operations were included in the table. There were 25 routines with dynamic spills, but only the 17 in which improved passive splitting made a difference are shown in the table. The remaining columns show the percentage improvement (where percent improvement is calculated as $\frac{old-new}{old} \times 100$). In the table headings $PS$ denotes the original passive splitter, $PS^*$ is the new splitter (program structure only), and $PS^{**}$ is the new splitter with both program structure and spill reconsideration.

Adding program structure decreased the dynamic spill operation count in 11 of the 25 procedures, in some cases dramatically. Improvements ranged from 0.08% all the way up to 50% for denpt. Two procedures show small losses while 12 procedures showed no change. By using both program structure and spill reconsideration, 14 procedures improved, with three routines showing better than 49% reduction in spill overhead.

The few losses that occurred relative to $PS$ can happen for two reasons— the same reasons mentioned by Simpson regarding $PS$ compared to Chaitin-Briggs without splitting. First, all spill and split cost analysis is currently done with static
<table>
<thead>
<tr>
<th>Bench</th>
<th>Proc</th>
<th>PS</th>
<th>PS*</th>
<th>PS**</th>
<th>%imp. PS*</th>
<th>% PS**</th>
</tr>
</thead>
<tbody>
<tr>
<td>wave5</td>
<td>field</td>
<td>13494690</td>
<td>13243885</td>
<td>13309290</td>
<td>1.86</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>vslv1p</td>
<td>11934370</td>
<td>11903095</td>
<td>12117520</td>
<td>0.26</td>
<td>-1.53</td>
</tr>
<tr>
<td></td>
<td>smooth</td>
<td>8062200</td>
<td>6937280</td>
<td>6562180</td>
<td>13.95</td>
<td>18.61</td>
</tr>
<tr>
<td></td>
<td>radaf4</td>
<td>7731100</td>
<td>7480700</td>
<td>7292900</td>
<td>3.24</td>
<td>5.67</td>
</tr>
<tr>
<td></td>
<td>radb4</td>
<td>7292900</td>
<td>7105100</td>
<td>6948600</td>
<td>2.58</td>
<td>4.72</td>
</tr>
<tr>
<td></td>
<td>parmvr</td>
<td>6518815</td>
<td>6826430</td>
<td>6831315</td>
<td>-4.72</td>
<td>-4.79</td>
</tr>
<tr>
<td></td>
<td>radaf2</td>
<td>3756000</td>
<td>3756000</td>
<td>3536900</td>
<td>5.88</td>
<td>5.04</td>
</tr>
<tr>
<td></td>
<td>radb2</td>
<td>3724700</td>
<td>3505600</td>
<td>3536900</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td></td>
<td>denpt</td>
<td>2700002</td>
<td>1350002</td>
<td>1350002</td>
<td>50.00</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>getb</td>
<td>820728</td>
<td>812912</td>
<td>789440</td>
<td>0.95</td>
<td>3.81</td>
</tr>
<tr>
<td></td>
<td>init</td>
<td>504020</td>
<td>504020</td>
<td>505870</td>
<td>0.00</td>
<td>-0.37</td>
</tr>
<tr>
<td></td>
<td>energy</td>
<td>200216</td>
<td>200216</td>
<td>100696</td>
<td>0.00</td>
<td>49.70</td>
</tr>
<tr>
<td></td>
<td>putb</td>
<td>59597</td>
<td>58620</td>
<td>57643</td>
<td>1.64</td>
<td>3.28</td>
</tr>
<tr>
<td></td>
<td>slv2xy</td>
<td>900</td>
<td>925</td>
<td>825</td>
<td>-2.78</td>
<td>8.33</td>
</tr>
<tr>
<td></td>
<td>celbnd</td>
<td>880</td>
<td>720</td>
<td>720</td>
<td>18.18</td>
<td>18.18</td>
</tr>
</tbody>
</table>

|         | program | 77975318 | 74859705 | 74115007 | 4.00     | 4.95   |
| g271d   | update  | 19847517 | 19847517 | 10031360 | 0.00     | 49.46  |
|         | program | 19847517 | 19847517 | 10031360 | 0.00     | 49.46  |
| tomatv  | main    | 52483505 | 52442514 | 49831213 | 0.08     | 5.05   |
|         | program | 52483505 | 52442514 | 49831213 | 0.08     | 5.05   |

Table 3.1: Dynamic spill operations for wave5, g271decoder, tomatv (25 integer/25 float registers, 13/13, and 14/14 respectively)

estimates. These estimates cannot predict actual runtime behavior with perfect accuracy, so that spills or splits might be placed in unfortunate locations. Second, after a round of splitting/spilling, the second pass of allocation will be presented with a different interference graph. This means that simplify might make completely different decisions than it did in the first pass.

Improved passive splitting has a reasonable compile time cost, as shown in Table 3.2. PS* increases compile time between 4 and 6% over PS, while PS** increases it by 6 to 17%. Most of the extra time spent in PS** is due the the extra renumber-build-color cycle needed when spills are reconsidered.
3.4 Other Issues and Modifications

As specified in Simpson’s original article, the passive splitting algorithm is unable to function correctly for certain kinds of instruction-set architectures or compiler intermediate representations. None of these issues affect the ILOC compiler used for the original implementation, which is somewhat simplified compared to other compilers. Also, ILOC itself is generally simpler than many actual machine instruction sets. However, it is likely that industrial practitioners or other researchers using different compiler infrastructures will be impacted.

3.4.1 Representing Call-clobbered Registers at Call Sites

In some compilers, the clobbering of caller-saved registers at a call site is represented by adding extra operands to a procedure call instruction (one for each register clobbered). Each new operand is marked as a definition of the corresponding register. Under this scenario, split costs and split code will not operate properly if they try to split a range around one of these definitions. That is, the definition will signal that a STORE is needed, but since it has no later uses, no LOAD operation (to reload the split range) will be signalled.

A small modification is made so that while processing a definition of a range that is not already live, it is treated like a death. That is, a LOAD will be needed after the definition and the cost must be adjusted accordingly. The situation must be detected while processing definitions because the defined register has no use in any instruction, so that it will have never become live, and hence never seen by the code dealing with

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$PS$</th>
<th>$PS^*$</th>
<th>$PS^{**}$</th>
<th>$\frac{PS^*}{PS}$</th>
<th>$\frac{PS^{**}}{PS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>wave5</td>
<td>54</td>
<td>56</td>
<td>62</td>
<td>1.04</td>
<td>1.15</td>
</tr>
<tr>
<td>g271decode</td>
<td>0.84</td>
<td>0.89</td>
<td>0.89</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>tomcatv</td>
<td>0.48</td>
<td>0.50</td>
<td>0.56</td>
<td>1.04</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 3.2: Compile-time impact of improved passive splitting (seconds)
deaths. For example, in `calcPerNeighborCosts`, two lines are added to the "ForDef" case as shown below. An analogous, but slightly more complicated change is needed in `placeSplits` as well.

```
calcPerNeighborCosts(l, b, type)
    ...
    if type = ForDef
        l.split_costs[n].stores ← l.split_costs[n].stores+weight
        if l $\notin$ live
            l.split_costs[n].loads ← l.split_costs[n].loads+weight
    ...
```

### 3.4.2 Two-address Instructions/Modified Input Operands

Many existing instruction set architectures contain two-address instructions or other instructions that both read and write a register operand. For example, both the PowerPC and PA-RISC ISAs have load and store instructions that update the index register. For the Intel x86 series, most arithmetic instructions are two-address, overwriting one of the two inputs with the result. Such instructions present a real complication for the original splitting algorithm. Consider the following fragment in Intel x86-like psuedo-assembly code (using virtual register numbers).

```
VR1 = ....
VR2 = ....
...
ADD VR2,[EBP+32]
...
... = VR2
...
... = VR1
```

Suppose `PS` decides to split `VR1` around `VR2`. Where should it insert the store operations? Normally, that would be done at every definition of `VR2` but clearly that presents a problem here. The idea behind splitting `VR1` is to use the same register for it that `VR2` has been assigned. The second inserted store of `VR1` (before the `ADD`
instruction) would store the improper value, since it would have been overwritten by
the first write of VR2 (recall, they both will share the same physical register). There-
fore, the reload back into VR1, which would be placed after the last use of VR2 would
get the wrong value. Because the ADD both reads VR2 for its data, and writes VR2
with its result, we cannot view the write of VR2 as a new live range. In essence, the
register mentioned in a modified input must be treated as a single live range.

The situation is complicated more by the fact that chains of two-address instruc-
tions (e.g., ADD VR2, EAX; \ldots; ADD VR2, 100) using the same register can span mul-
tiple basic blocks. A solution is currently being worked on.

3.5 Summary

Simpson's passive splitting is an elegant extension to Chaitin-Briggs that successfully
attacks the "spill everywhere" problem. We presented algorithmic improvements that
increase its effectiveness even more (up to 50% fewer dynamic spill operations) by in-
corporating program structure directly into the splitting process and by reconsidering
spill decisions. The improved algorithm also maintains the elegance and simplicity
of the original, so that practitioners can consider including the method into their
Chaitin-Briggs-style allocators.
splitCosts()
    buildContainmentGraph()
    annotateLoopTree(LoopTree.root)
    For each block b
        live ← liveOut_b
        For each successor s of b
            deaths ← liveOut_b - liveIn_s
            For each m ∈ deaths
                calcPerNeighborCosts(m, s, ForDeath)
        For each instruction i in b in reverse order
            For each live range l defined in i
                calcPerNeighborCosts(l, b, ForDef)
            For each live range l used in i
                if l ∉ live
                    calcPerNeighborCosts(l, b, ForDeath)
        Update the live set

calcPerNeighborCosts(l, b, type)
    deflt_weight ← 1^{depth(b)}
    For each neighbor n of l
        if ⟨n, l⟩ ∉ containment graph
            pushout ← 0
            p ← loop(b)
            while p ≠ NIL ∧ n ∉ p.refs
                pushout ← pushout + 1
                new_weight ← 1^{p.depth-1}
                p ← p.parent
            if pushout > 0
                weight = new_weight
            else
                weight = deflt_weight
        if type = ForDef
            l.split.costs[n].stores ← l.split.costs[n].stores+weight
        else if type = ForDeath
            l.split.costs[n].loads ← l.split.costs[n].loads+weight

Figure 3.7: Algorithm for computing split costs
select()
...
while ¬coloring_stack.empty
...
    c ← pick color for name
    if c = invalid_color
        spillset ← spillset ∪ name
        if findSplits(name) = TRUE
            spillset ← spillset − name
            if pass_number = 1
                spillset ← {}
findSplits(1)
    bestCost ← range[1].cost
    splitFound ← FALSE
    For each color c
        splitOK ← TRUE
        cost ← 0
        For each neighbor n of 1 with colors[n] = c
            if ⟨n, l⟩ ∈ containment graph
                splitOK ← FALSE
            else if rematerializable(n)
                cost ← cost + l.split_costs[n].loads × rematCost
            else
                cost ← cost + l.split_costs[n].stores × storeCost +
                l.split_costs[n].loads × loadCost
            if splitOK ∧ cost < bestCost
                bestCost ← cost; bestColor ← c
                splitDir ← splitAroundName; splitFound ← TRUE
        splitOK ← TRUE
        cost ← 0
        For each neighbor n of 1 with colors[n] = c
            if ⟨l, n⟩ ∈ containment graph
                splitOK ← FALSE
            else if rematerializable(l)
                cost ← cost + n.split_costs[l].loads × rematCost
            else
                cost ← cost + n.split_costs[l].stores × storeCost +
                n.split_costs[l].loads × loadCost
            if splitOK ∧ cost < bestCost
                bestCost ← cost; bestColor ← c
                splitDir ← splitAroundColor; splitFound ← TRUE
    if splitFound = TRUE
        colors[1] ← bestColor
        if splitDir = splitAroundName
            For each neighbor n of 1 with colors[n] = bestColor
                Mark n to be split around 1
        else
            For each neighbor n of 1 with colors[n] = bestColor
                Mark 1 to be split around n

Figure 3.9: Algorithm for determining splits
splitCode()
  For each block b
    live ← liveOut_b
  For each successor s of b
    deaths ← liveOut_b - liveIn_s
  For each m ∈ deaths
    For each live range l split around m
      placeSplit(l, s, s.first inst, ForDeath)
  For each instruction i in b in reverse order
    For each live range l defined in i
      For each live range s split around l
        if ¬rematerializable(s)
          placeSplit(s, b, i, ForDef)
    For each live range l used in i
      if l ∉ live
        For each live range s split around l
          placeSplit(s, b, i.next inst, ForDeath)
  Update the live set

Figure 3.10: Algorithm for inserting split code
placeSplit(n, b, deflt_inst, type)
    pushout ← 0
    t ← NIL
    p ← loop(b)
    while p ≠ NIL ∧ n ∉ p.refs
        pushout ← pushout + 1
        t ← p
        p ← p.parent
    if type = ForDef
        if pushout > 0
            For each entry edge e of loop t
                pred ← e.pred
                if n ∈ liveOutpred ∧ n ∉ t.spilled_on_entry[e]
                    t.spilled_on_entry[e] ← t.spilled_on_entry[e] ∪ n
                    Insert STORE n at end of pred
        else if pushout = 0
            Insert STORE n before deflt_inst in b
    else if type = ForDeath
        if pushout > 0
            For each exit edge e of loop t
                succ ← e.succ
                if n ∈ liveInsucc ∧ n ∉ t.reloaded_on_exit[e]
                    t.reloaded_on_exit[e] ← t.reloaded_on_exit[e] ∪ n
                    if rematerializable(n)
                        Insert a LOAD–IMMEDIATE for n at beginning of succ
                    else
                        Insert LOAD of n at beginning of succ
        else if pushout = 0
            if rematerializable(n)
                Insert a LOAD–IMMEDIATE of n before deflt_inst in b
            else
                Insert LOAD of n before deflt_inst in b

Figure 3.11 : Algorithm for placing split code
Chapter 4

Hierarchical Graph Coloring

The previous chapter studied a passive approach to live range splitting that involved relatively little change to the overall Chaitin-Briggs framework. This chapter examines a hierarchical graph coloring approach introduced by Callahan and Koblenz [7]. Their allocator ("CK") improves on CB by incorporating program structure directly into the allocation process. Its splitting of ranges is fairly aggressive, and requires major changes to the CB framework.

The original Callahan-Koblenz article presents a fascinating approach, and makes compelling arguments about its functionality, but unfortunately no experimental evaluation was presented by the authors. That is, they only provide a relatively high-level description of the algorithm, and no comparison to a high-quality baseline allocator is made. If the Citeseer literature database is any indication, there has been wide interest in the CK article— it has been cited almost as frequently as the well-known Briggs paper [6]. However, even after more than a decade since its publication, there still has been no evaluation published in the literature.

Thus, while the Chaitin-Briggs allocator has been investigated extensively, and is implemented in practically every industrial and research compiler, there is little understanding of the CK allocator— either in terms of implementation details or in terms of a quantitative analysis compared to, say, CB. Industrial practitioners, in particular, are necessarily conservative about implementing unproven or poorly-understood algorithms in their compilers. This is especially true in the case of CK, which, as will be seen in the following sections, is significantly more complicated than the proven, easy to implement CB allocator. This chapter intends to address this gap
in the literature, and to provide researchers and practitioners with solid empirical data about the performance of this intriguing algorithm. Because CK is considered an extension to CB, and since CB is so well understood, CB is used as the baseline of comparison.

Section 4.1 provides an overview of the Callahan-Koblenz allocator. An experimental evaluation comparing the allocator to Chaitin-Briggs is presented in Section 4.2. Finally, Section 4.3 discusses some simple implementation improvements.

4.1 Overview of the Callahan-Koblenz Allocator

The Callahan-Koblenz allocator extends Chaitin's allocator by directly incorporating program structure into the allocation process. By doing so, the allocator can more intelligently decide which variables to spill, as well as determine where to place the spill code. That is, rather than taking the "spill everywhere" approach of Chaitin, Callahan-Koblenz has the potential to place spills in less frequently executed portions of the program.

Callahan-Koblenz represents the hierarchical program structure with a tile tree. Roughly, each node in the tree represents a region of code (a tile) such as a loop or conditional and each pair of tiles in the tree must either be disjoint or properly nested, one within the other. Such a tree structure isolates the high- and low-frequency code regions and provides a basis for the allocator's overall operation and spill placement decisions. Figure 4.1 shows an example control-flow graph and its corresponding tile tree, where the set \( \text{blocks}(T) \) represents all basic blocks which belong to tile \( T \), but not to any subtiles of \( T \).

Each tile boundary represents an implicit split-point of all values live at that boundary. A strength of Callahan-Koblenz lies in the ability to allocate each portion of a live range between the tile boundaries independently\(^1\). These splitpoints also

\(^1\)Independent in the sense that each portion of the live range can be allocated to a different location.
Figure 4.1: Example tile tree: (a) CFG; (b) tiles overlaid on CFG; (c) the tile tree.

become the locations where any necessary spill code for global values will be placed.

Figure 4.2 depicts the overall structure of the Callahan-Koblenz allocator. Once a tile tree has been constructed, two major passes are made over the tile tree.

Phase 1 (bottom-up):

Each tile $T$ is visited in postorder and processed independently with the goal of producing a preliminary allocation. The overall processing of each tile is similar to a Chaitin-Briggs allocator, but includes extra bookkeeping between tiles, and does not
perform coalescing\(^2\).

1. **Build and preferences**: Build the interference graph much like Chaitin-Briggs, but restricting attention to \(\text{blocks}(T)\). Moreover, unlike the standard builder, interferences are not constructed for any variable which is live across, but not referenced in the subtree rooted at \(T\).\(^3\) Preferences, such as for the source and destinations of copy instructions, are also set up at this time (see Section 4.1.1).

2. **Incorporate subtile summaries**: All subtiles of \(T\) will have already been processed, and a compact summary of their allocations stored. This information is incorporated into \(T\)'s interference graph, as well as certain types of preferences based on the subtiles' allocations.

3. **Color**: Coloring operates similarly to the Chaitin-Briggs allocator except that color choice may be influenced by preferences. Further, if a node receives a color, that color may potentially be propagated to other nodes. Except for nodes

\(^2\)See the original Callahan and Koblenz article for a discussion regarding not doing coalescing.

\(^3\)Such live ranges, which we abbreviate “LBNR”, are similar to the “delayed bindings” of [17], or the “inactive” live ranges of [3].
which must receive a particular physical register, colors assigned in this phase are “pseudo colors” in the sense that they will be re-colored with a physical register in the second phase (and there are $k$ pseudo-colors, corresponding to the $k$ physical registers).

4. **Summarize:** After $T$ is processed, a compressed representation of its’ interference graph and allocation is constructed and passed up to the parent tile. Included in the summary are all tile-global variables allocated to registers, all tile-globals allocated to memory, and *tile summary variables*. Each TSV corresponds to a set of tile-local variables that were allocated the same color, so that the local allocation is represented in a very compact form. Conflicts involving tile-locals are stored in terms of their associated TSVs.

**Phase 2 (top-down):**

Each tile $T$ is visited in preorder with the goal of providing the final assignment of physical registers. Spill code is introduced at tile boundaries to reconcile differences in each tile’s allocation.

1. **Rebuild:** Reconstruct the interference graph for $T$ directly from its summary information.

2. **Incorporate parent summaries:** Conflicts for LBNRs that were excluded in the first phase are now added to the graph for consideration, if they received a register in the parent. Certain preferences are also set up based on the parent’s allocation.

3. **Color:** A final coloring is performed, binding pseudo-colors to physical registers. As before, coloring decisions are influenced by any preferences.

4. **Summarize:** Save $T$’s allocation and preference information to be passed down to its subtiles.
5. *Spill code:* Spill code is introduced at the tile boundaries, which may not be the same tile where a particular spill decision was made. Spill instructions could be loads, stores, or register-register copies, depending on the location of a global in $T$ and its parent.

4.1.1 Preferencing

*Preferencing* is the idea that it may be desirable to assign the same arbitrary color to multiple variables, or give a single variable a particular color (usually corresponding to a physical register). By making the coloring algorithm sensitive to such preferences, the likelihood of choosing the desired color for a node is increased.

Copy removal in Callahan-Koblenz is performed by preferencing the source variable $S$ and destination variable $D$ of a copy together by adding each to the others *preference list*. The preference-guided color assignment algorithm then attempts to give the same color to $S$ and $D$. If the attempt is successful (the preference was *satisfied*), then the resulting copy is redundant and can be trivially removed. Similarly, if either $S$ or $D$ is a physical register, such as a copy generated to implement subroutine linkage conventions, we set up a *local preference*. This is different than the previous case in that a variable is preferenced to a specific physical register.

During color assignment, when a node receives a color, the color is propagated to all the nodes on its preference list as their local preference. If a node has a local preference, then the coloring mechanism will first attempt to assign that register before resorting to using another register. Furthermore, it will try to avoid assigning a color to a node if that color is preferred by uncolored neighbors.

In addition to copy removal, preferencing also plays a crucial role regarding global values at tile boundaries. That is, it is used to influence the colors that different parts of a global live range receive. Recall that tile boundaries are implicit split-points for variables live at that boundary. Because tiles are processed independently, it is important to pass around information about these variables (in the form of preferences)
so that each tile attempts to place the same global into the same register. Unlike above, these preferences are not generated in response to copy instructions. However, if they are not satisfied, then copy operations will be inserted at the boundary to resolve the differing allocations.

4.2 Experimental Comparison

Since a goal of the CK algorithm is to minimize dynamic memory references due to poor spill behavior, the primary question that needs to be addressed is to what extent it improves on the “spill everywhere” approach of Chaitin. Secondly, there is overhead involved in considering the global register allocation problem over smaller regions (i.e., tiling). Such overhead may result when putting the regions back together to form a full allocation, so that extra operations are placed on tile boundaries. Is the overhead tolerable? This section will attempt to answer both questions with a number of experiments.

Note that a comparison of the run times of each allocator is avoided. There are three reasons for this. First, this thesis is primarily concerned with the efficiency of the code generated by the compiler. Second, the CK allocator implemented here is a prototype where compile-time efficiency played a minor role. Instead, getting a faithful, working implementation was the focus. Finally, the Chaitin-Briggs algorithm has been widely implemented and re-implemented by its inventors and others, so that the writing of finely-tuned, highly-efficient CB allocators is well understood and routine. As it stands now, the prototype implementation runs between 2X-5X slower than the CB allocator. With that said, there is reason to believe that a carefully done, mature implementation of the CK allocator would have compile-time reasonably competitive with CB. There are obvious efficiency improvements to our implementation that can be made quickly, such as using fast bit vectors rather than the set type of the C++ Standard Template Library. There will be some unavoidable overhead due to inter-tile bookkeeping, and its two-pass walk over the tile tree.
Both of the CB and CK allocators described previously have been implemented in the LLVM compiler infrastructure [16]. This is a sophisticated, near production-quality compiler which includes a code generator for the Intel x86 family. LLVM applies a large set of interprocedural and intraprocedural optimizing transformations to the program before handing it to the code generator. The CB implementation follows the outline described in Briggs’ thesis very closely, with the one omission being rematerialization.

The benchmarks were executed on a machine with a single Intel Pentium 4 processor and 1 GB of main memory running Redhat Linux 9.0. The Pentium 4 processor has 8 integer registers (conventional organization) and 8 floating registers (stack organization). There are typically only 7 integer registers available for allocation, since %ESP is generally a dedicated stack pointer.

Currently, the LLVM x86 code generator has limited support for allocating values to floating-point registers. That is, LLVM is generally unable to allocate floating-point values globally across basic blocks due to complications in handling the stack-based FP register file of x86. Therefore, the allocators were evaluated on programs from the SPEC2000 integer benchmarks and the Mediabench suite. The tables and graphs in the following experiments depict results from 9 SPEC benchmarks: gzip, vpr, crafty, parser, eon, mcf, gap, bzip2, twolf, and one program from the Mediabench suite: epic.

4.2.1 Dynamic Spill Code Comparison

The spill cost and spill placement strategy for CB was discussed in Chapter 2. A more fine-grained spill strategy is used by the Callahan-Koblenz allocator. Because live ranges can be split at tile-boundaries, the allocator may choose to place a variable \( v \) in different locations for each tile that it crosses. For example, \( v \) may be allocated to a register within tile \( t \), while being relegated to memory in the parent or a subtile. The following set of equations forms the cornerstone of this strategy:
\[ LocalWeight_t(v) = \sum_{b \in \text{blocks}(t)} P(b) \cdot Ref_b(v) \]

where \( t \) is a tile, \( P(x) \) denotes the probability of executing a block or taking a control flow edge and \( Ref_b(v) \) is the number of references to \( v \) within \( b \). Assuming that allocating a register to variable \( v \) in \( t \) is profitable (see below) during the bottom-up phase, \( LocalWeight_t(v) \) is analogous to Chaitin-Briggs' \( \text{SpillCost} \) heuristic and is used, along with the degree of the node corresponding to \( v \), in a similar fashion. However, this cost is computed based only on blocks that occur strictly within tile \( t \), as opposed to the whole function. Moreover, the reference count of block \( b \) is weighted by the probability of \( b \) being executed. Note that for the purposes of this work, we use a static estimate of \( P(b) \) rather than actual profile data to ensure a fair comparison of the spill heuristics for both allocators. If \( b \) is a block, we set \( P(b) = 10^{\text{depth}(b)} \). If \( e \) is an edge emanating from a block \( b \), \( P(e) \) is computed as 1 divided by the number of outgoing edges of \( b \).

\[ Weight_t(v) = \sum_{s \in \text{subtiles}(t)} (\text{Reg}_s(v) - \text{Mem}_s(v)) + LocalWeight_t(v) \]

Overall decisions regarding whether or not a variable should be spilled are based on \( Weight_t(v) \). It is computed as a combination of \( LocalWeight_t(v) \) and various penalty costs that may arise from making certain allocation decisions with respect to the parent or children of \( t \). It may happen that the penalty outweighs the benefit of allocating \( v \) to a register, indicating that the allocator should force \( v \) into memory.

\[ Transfer_t(v) = \sum_{e \in E(t)} P(e) \cdot \text{Live}_e(v), \quad \text{where} \quad E(t) = \text{EntryEdges}(t) \cup \text{ExitEdges}(t). \]

\[ \text{Reg}_t(v) = \begin{cases} 0, & \text{if } \text{InReg}_t(v) = \text{false} \\ \min(Transfer_t(v), Weight_t(v)), & \text{if } \text{InReg}_t(v) = \text{true} \end{cases} \]
\[ Mem_t(v) = \begin{cases} 
0, & \text{if } InReg_t(v) = \text{true} \\
Transfer_t(v), & \text{if } InReg_t(v) = \text{false} 
\end{cases} \]

where \( InReg_t(v) \) is a boolean predicate which is true if \( v \) received a register in tile \( t \), and false otherwise. \( Live_e(v) \) is a predicate that indicates if variable \( v \) is live along edge \( e \).

\( Transfer_t(v) \), \( Reg_t(v) \), and \( Mem_t(v) \) represent the various penalty costs. The first corresponds to the cost due to tile-boundary spills, while the remaining two account for any penalties due to a tile and its parent choosing different locations for the same live range. If \( v \) is allocated to a register in tile \( t \), \( Reg_t(v) \) is the penalty of allocating \( v \) to memory in the parent of \( t \). Likewise, if \( v \) is allocated to memory in tile \( t \), then \( Mem_t(v) \) is the penalty of allocating \( v \) to a register in the parent of \( t \).

To see how \( Weight_t(v) \) might be negative, consider a tile \( t \) and a single loop subtile \( s \) of \( t \). Suppose on the way up the tree, a variable \( v \) was spilled in \( s \). If \( t \) were to allocate \( v \) to a register, then the quantity \( Reg_s(v) - Mem_s(v) = 0 - Transfer_s(v) = -Transfer_s(v) \). If \( Transfer_s(v) \) is greater than \( LocalWeight_t(v) \), then it is generally better to spill \( v \) in \( t \). In other words, if \( t \) allocates \( v \) to a register, it will cost more dynamic spill operations to transfer \( v \) to and from memory around \( s \) than it would to just spill \( v \) in \( t \) in the first place. When this situation arises, \( v \) will be marked so that it will not compete with other variables for a color during the coloring phase.

Table 4.1 shows the dynamic spill behavior of each benchmark for CB and CK. The column marked \( CB \) is the number of dynamic memory operations executed by the CB-compiled version of each benchmark. The CK results are broken down into the three types of spill operations that can occur. Column \( M \) is the number of dynamic memory operations executed within tile boundaries (e.g., loops). Column \( M_{TB} \) and \( C_{TB} \) are the number of dynamic memory and register-to-register copy operations executed on tile boundaries, respectively. The two additional CK columns represent the sum of all dynamic memory operations \( (M + M_{TB}) \) and the sum of all dynamic spill operations (memory operations or copies). It is useful to isolate the different
<table>
<thead>
<tr>
<th>Bench</th>
<th>CB</th>
<th>CK</th>
<th>% imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$M$</td>
<td>$M_{TB}$</td>
</tr>
<tr>
<td>gzip</td>
<td>23.49</td>
<td>17.18</td>
<td>1.58</td>
</tr>
<tr>
<td>vpr</td>
<td>9.60</td>
<td>9.21</td>
<td>0.04</td>
</tr>
<tr>
<td>mcf</td>
<td>16.37</td>
<td>3.43</td>
<td>0.52</td>
</tr>
<tr>
<td>crafty</td>
<td>83.15</td>
<td>60.78</td>
<td>4.15</td>
</tr>
<tr>
<td>parser</td>
<td>58.64</td>
<td>30.07</td>
<td>9.30</td>
</tr>
<tr>
<td>eon</td>
<td>15.58</td>
<td>15.53</td>
<td>0.06</td>
</tr>
<tr>
<td>gap</td>
<td>58.91</td>
<td>46.13</td>
<td>4.25</td>
</tr>
<tr>
<td>bzip2</td>
<td>35.12</td>
<td>26.37</td>
<td>4.31</td>
</tr>
<tr>
<td>twolf</td>
<td>65.42</td>
<td>42.84</td>
<td>13.53</td>
</tr>
<tr>
<td>epic</td>
<td>2.08</td>
<td>1.23</td>
<td>1.41</td>
</tr>
<tr>
<td><strong>MEAN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Dynamic spill operations for SPECInt2000 and epic (billions)

types of spills for CK in order to see the effects of tiling more directly. Finally, the last two columns show the percent improvement of CK over CB. In the first case, only memory operations are considered, whereas memory and copy operations are considered in the second case. This distinction was made to show how prevalent any remaining tile-boundary register-register copies were (indicating success or failure of inter-tile preferencing), and what overall impact they had on the improvements.

Overall, the benchmarks allocated with CK executed significantly fewer dynamic spill operations than those allocated by CB—up to 73% fewer on mcf. On average, 15.21% fewer spill operations were executed for CK than for CB. On the other hand, there were two losses for CK. One slight loss for eon at 0.23%, but one significant 27.9% loss for epic (more on this later).

We examined some of the benchmarks in detail (at the assembly language level) to understand choices made by each allocator, and why CK performed relatively well compared to CB. Consider the code in Figure 4.3a, which is a typical (but simplified) scenario present in many of the benchmarks. Here there are two live ranges $x$ and $t$ competing for one register, where $x$ is referenced once early, and heavily in some distant part of the program. There are a only few references to $t$ in a small portion
of the program, but they occur in a loop, making them frequently executed. Let us assume the total number of references to \( x \) exceeds those of \( t \). In the standard CB scheme, since the spill cost is calculated based on the references throughout the program, then \( x \) would get a color and \( t \) would be spilled (as in Figure 4.3b). But from the perspective of \( t \), spilling \( t \) is a poor choice, since \( x \) is never even referenced in the loop. On the other hand, the opposite choice (giving \( t \) the color) is bad too as the many references to \( x \) will now be through memory. Because CB must spill a live range entirely, one of two poor choices must be made. As mentioned earlier, CK can consider each live range in fragments, over regions of the program. Here CK splits \( x \) before and after the loop, so that the loop portion and non-loop portions are allocated independently. This allows the result seen in Figure 4.3c, where \( t \) gets the register and \( x \) gets the register (but \( x \) is allocated to memory within the loop where it has no references). Notice also that there is a tradeoff in making such a split. A
store and a load operation must be placed at loop entry and exit to make the split, which is clearly profitable here.

Figure 4.4 : Example of CK advantage: (a) Pre-colored node EAX; (b) CB: y in EBX and x spilled; (c) CK: y in EBX and x split

Another scenario seen in the benchmarks where CK tends to win is depicted in Figure 4.4a. On the Intel x86, the standard calling conventions require that a return value be placed in EAX. This physical register requirement is represented in the intermediate language explicitly, and its corresponding node in the interreference graph will be pre-colored with EAX. Suppose now that a standard CB coloring is applied to this code, and there are two registers EAX and EBX. Further, assume EBX has already been allocated to y outside the loop (but it is free within the loop). Since x and y both conflict with pre-colored node EAX, one or the other must get EBX, say y does. Thus, x must be spilled to memory since there are no more free colors, resulting in the code in Figure 4.4b. CK, on the other hand, can split at the loop boundaries to do better. As shown in Figure 4.4, y is allocated to EBX as before. But this time the portion of x outside the loop is assigned to EAX (where it is idle) and the portion within the loop is assigned to EBX (where it is idle). So for the cost of two copy
operations outside the loop, $x$ is able to live in a register throughout its lifetime.

Returning to the loss in the **epic** benchmark, it is useful to examine the breakdown of spill operation types for CK. The graph in Figure 4.5 shows the percentage of total spill operations represented by each type. For example, in the **gzip** benchmark, about 92% of all dynamic spill operations are intra-tile memory operations, while the other 8% are primarily inter-tile memory operations. Looking at the **epic** bar, it is evident that something went wrong with CK's heuristics. That is, more than half of all the dynamic spill operations are memory operations on the tile boundaries. Without looking at the code, this would seem indicate that CK did not calculate trade-offs between intra- and inter-tile spilling appropriately.
In fact, on examining the assembly code, we found just that behavior. One routine dominating execution time contains a number of triply-nested loops. In one such nest, there is heavy register pressure in the inner loop, little pressure in the middle loop, and medium pressure in the outermost loop (and the non-loop code as well). There are also a number of global values live across the entire loop nest, with references in some loops and not others. Unfortunately, for some of these globals, the constituent fragments within each loop were alternately allocated to registers and memory. That is, the outermost loop allocated $g$ to a register, the next deeper loop allocated $g$ to memory, and the inner loop allocated it to a register. Thus, at every tile boundary there are memory operations to transfer $g$ in and out of memory as appropriate. It turns out that these tile-transfers dominate the spill operation count, as seen in the graph. It would have been better to keep $g$ in the same location across more than one tile boundary. We discovered the reason for the poor trade-off decisions being made by the algorithm and incorporated some improvements. This will be discussed later in Section 4.3.2.

4.2.2 Control-flow Overhead of Tiling

A key characteristic of the tile tree construction presented by Callahan and Koblenz is that for any edge $(v_1, v_2)$ that originates from a block outside of a tile $T$ and terminates at a block in $T$ (an entry edge), $v_1$ must be a block in the parent of $T$. Similarly, tile exit edges must terminate at a block in the parent of $T$. These conditions ensure that empty spill blocks can be placed along a particular entry or exit edge of a tile to contain spill code. If all edges in the original control-flow graph do not satisfy these conditions, then the CFG will be modified by adding the appropriate basic blocks.

Typically these extra blocks fall through to their successor and thus do not result in any additional branches in the final program. However, there are cases when inserting blocks results in unavoidable branches. Consider a branch in block $v_b$ in the original program that is situated in the innermost loop of a loop nest $L$, and which
targets a block $v_i$ outside of $L$. Each loop in $L$ will have been placed into its own tile after the tree is constructed, say $t_1, t_2, \ldots, t_d$, where $v_b \in \text{blocks}(t_d)$ and $v_i$ is in the root tile $t_0$. Since the edge $(v_b, v_i)$ does not terminate at the parent tile of $t_d$, a spill block will be inserted at every tile boundary between $t_d$ and $t_0$. If any of these blocks end up with spill code after allocation, and they don’t naturally fall-through to the successor, a branch will result.

It is possible, then, that the resulting code might execute more branches than the original, potentially degrading performance. In order to quantify the actual impact, we compared the number of branch instructions executed in the resulting code from both the Callahan-Koblenz and Chaitin-Briggs allocators. Figure 4.6 indicates that the Callahan-Koblenz allocated code executes more branches than the Chaitin-Briggs.
allocated code.

On average, the Callahan-Koblenz allocator inserted around 5.3% more branches in the code. It is interesting, however, to note from Figure 4.6 that the increase in executed branches was comparatively lower: 1.6% over all benchmarks. This difference between static and dynamic branches indicates that the branches placed at tile boundaries are infrequently executed.

4.3 Some Modifications and Improvements

The allocator implemented for this work aims to be as faithful to what the inventors intended (learned through discussions with them) as possible. In addition, some improvements were made to address problems, either in code performance or architectural complications, that were encountered along the way. Some of these are described next.

4.3.1 Cycles of Copies at Tile-boundaries

In Section 4.1, it was briefly mentioned that spill code must be introduced at the tile boundary between \( t_1 \) and \( t_2 \) for any globals that reside in different locations in \( t_1 \) and \( t_2 \). That is, a global variable that is allocated to one location in \( t_1 \) must be transferred to the location it was assigned in \( t_2 \) after leaving control of \( t_1 \). These new transfer/spill instructions would be inserted on the edge (the boundary) connecting the two tiles.

Insertion of tile-boundary spills can become complicated when there are multiple register-to-register copies. Consider Figure 4.7a. For tile \( T_0 \), globals \( a \), \( b \), and \( c \) were allocated to registers \( R_1 \), \( R_2 \), and \( R_3 \) respectively. On the other hand, the same globals were assigned respectively to registers \( R_2 \), \( R_3 \), and \( R_1 \) in \( T_1 \). Because of this arrangement, copies must be inserted on the tile-boundary edge to move values around to the proper locations. If such insertion is done naively, the situation in Figure 4.7b might result. Each copy transfers one global from its current location to
the new location. Unfortunately, the third copy reads the wrong value from R1 since it was overwritten by the first copy. In fact, no matter how carefully the three copies are arranged, it is impossible to avoid overwriting one of the values before it is read.

Such cycles among copies must be broken in order to generate correct code. Callahan and Koblenz break a cycle by introducing a temporary location, and copying one of the values to the temporary before generating the copies, as in Figure 4.7c. If there happen to be unused registers on the tile-boundary, one of them can be used as the temporary location. Otherwise, a memory location must be used which results in costly loads and stores just to break the cycle of moves.

By observing that a cycle is equivalent to a permutation of the values in the registers involved in the cycle, the situation described previously can be improved—that is, given appropriate support by the target microprocessor’s instruction set. For example, some machines provide a swap RA,RB instruction which swaps the values of RA and RB. The Intel x86 machine targeted by our implementation is one such machine, it provides a swap instruction called XCHG. It turns out that such an instruction can
be used to break cycles more efficiently. Figure 4.7d shows how this would be done on our example. The resulting code is half the number of instructions as the previous cycle breaking technique, uses no temporary location, and requires no load or store instructions. An algorithm is presented next for breaking cycles in this fashion.

In order to break cycles, they first need to be detected. A spill graph is constructed such that each node is a copy instruction, and an edge \( s_1, s_2 \) implies that \( s_1 \) must precede \( s_2 \) in the final emitted code. For our example, the graph depicted in Figure 4.7e would be built. Cycles are then detected using a variant of depth-first search, and each recorded. Next, the copy instructions in each cycle are examined to determine what swap instructions must be inserted to achieve the desired permutation of values. Finally, the original cycles are removed from the spill graph. At a later point, a simple scheduler will determine a linear ordering for all the tile-boundary spill instructions based on the resulting spill graph.

Figure 4.8 shows the algorithm to break cycles. To clarify the discussion, the algorithm is slightly simplified—it assumes that all registers involved in a cycle are of the same size, and that each register class is disjoint from all other register classes (more on this later). Consider the algorithm applied to the example in Figure 4.7. It determines that the initial arrangement \( \text{CurrOrder} = [R1, R2, R3] \) and the desired arrangement \( \text{TargetOrder} = [R2, R3, R1] \). On the first iteration through the \( j \) loop, \( R1 \) is detected to be out of target position and swapped (in \( \text{CurrOrder} \)) with \( R3 \) (which is blocking the target). Since a change was made, a new instruction swap \( R1, R3 \) is inserted into the spill graph. Thus, \( \text{CurrOrder} = [R3, R2, R1] \). On the next iteration of the \( j \) loop, \( R2 \) is out of position and \( \text{CurrOrder} \) is updated to \( [R2, R3, R1] \). That update requires swap \( R2, R3 \) to be added to the spill graph. No further changes will occur, and the algorithm finishes by removing the subgraph containing the cycle from the spill graph.

Certain architectures complicate cycle breaking even more, the Intel x86 being an example. Consider the organization of one of the x86 registers \( \text{EAX} \) as shown in
breakCycles(spill_block)
    SG ← buildSpillGraph(spill_block)
    Cycles ← detectCycles(SG)

    For each cycle C in Cycles
        // Determine the current and target order of registers.
        curr ← 0
        For each copy instruction S in C
            CurrOrder[curr] ← S.source_reg
            TargetOrder[curr] ← S.dest_reg
            TargetPositionMap[S.dest_reg] ← curr
            curr ← curr + 1

        // Modify the current order until the target arrangement is
        // achieved, emitting swap instructions for each modification.
        prev_inst ← Any predecessor p of C such that p ∈ (SG-C)
        changed ← TRUE
        while changed = TRUE
            changed ← FALSE
            For each j in [0..(curr-1)]
                registerA ← CurrOrder[j]
                if registerA ≠ TargetOrder[j]
                    pos ← TargetPositionMap[registerA]
                    registerB ← CurrOrder[pos]
                    CurrOrder[pos] ← registerA
                    CurrOrder[j] ← registerB
                    new_inst ← createNewInst(swap,registerA,registerB)
                    SG.Nodes ← SG.Nodes ∪ new_inst
                    SG.Edges ← SG.Edges ∪ \{prev_inst,new_inst\}
                    prev_inst ← new_instr
                    changed ← TRUE

    Remove subgraph C from SG

    Figure 4.8: Algorithm for basic cycle breaking with swap

Figure 4.9c. This register can be referenced as an 8-, 16-, or 32-bit quantity. EAX is
the full 32-bits, AX is the upper 16-bits, and AL and AH each access 8 of the 16 upper
bits. In other words, different register classes overlap, or alias each other. A number
Figure 4.9: Tile-boundary Copies with Aliases: (a) an allocation and initial spill block; (b) a slightly different allocation; (c) an x86 register

of “cleaner” RISC architectures also have aliasing, such as the floating-point register files of HP PA-RISC and Sun SPARC.

Register class aliasing presents difficulties for cycle-breaking. Figure 4.9a shows a code fragment and its initial spill block $SB0$. Here, $a$ is an 8-bit quantity allocated to $AL$ and $DL$ in tiles $T0$ and and $T1$ respectively. Global $b$ is a 16-bit quantity allocated to $DX$ and $AX$ for the tiles. The tile-boundary spill transfer mechanism will initially insert the copies as shown in $SB0$. At first it might appear that there is no dependence cycle. But because $AL$ and $AX$ are aliases, and $DL$ and $DX$ are aliases, then there actually is a cycle. We can longer insert swaps so easily in a situation like this, though. For example, which register class will be represented in the swap instruction? In this case, it turns out we can use the class of the “largest” covering register in the set of aliases for each operand involved in the cycle, e.g. we could emit a single $XCHG DX, AX$ to eliminate the cycle. This works because the smaller accesses both happen to reference the same sub-register ($AL$ and $DL$), so that a swap of the larger containing registers ($AX$ and $DX$) puts all values in their proper locations.

However, consider the slightly different example in Figure 4.9b. This time the
same sub-registers are not being referenced (AL and DH). So simply swapping the
larger registers will not work. To solve this problem, we decompose each copy of
an aliased register into a series of copies of each smaller portion of the register. For
example, the copies in the spill graph of Figure 4.10a would be decomposed into the
copies in Figure 4.10b (and the spill graph rebuilt to reflect the new structure).

The idea behind the decomposition is to reduce the problem of cycle-breaking
among aliased register classes to the problem of cycle-breaking among disjoint register
classes, which can be solved by the basic cycle breaking algorithm presented earlier.
The cycle in SG1D for example, would be broken by emitting XCHG AL, DL. Notice
that the decomposition itself might eliminate a cycle. In Figures 4.10c and d, SG2
is decomposed into SG2D. After the graph is rebuilt, no cycle remains and the
three copies are emitted. Finally, any remaining decomposed copies are recombined
if dependences allow. For example, if the final graph contains two sub-register copies
BL→CL and BH→CH, then they will be combined into a single copy BX→CX.

Finally, it is interesting to note that the problem of copy cycles also arises in SSA
deconstruction. Briggs et al. describe the situation as it arises in the context of
SSA deconstruction and present a solution for breaking cycles [4]. Their algorithm is
different from our graph-based solution, and is more akin to a worklist-style algorithm. Like Callahan and Koblenz, they introduce a temporary location to break cycles, whereas we are able to use architectural features (such as swaps) to break cycles more efficiently. Finally, our method is more general in that it can also handle aliased register classes.

4.3.2 Spill Cost Tweaks

Recall the discussion in Section 4.2.1 regarding the poor performance of benchmark epic. The problem there stemmed from global values being unproductively transferred in and out of memory around multiple tiles crossed by the globals. Examining the original spill formulas from Section 4.2.1, it seems that $LocalWeight_t(v)$ is sometimes underestimated. That is, consider that $Weight_t(v)$ only becomes negative (signalling a disincentive to allocate $v$ to a register in $t$) when one or more subtiles of $t$ allocate $v$ to memory such that the cost of transferring around those tiles is greater than $LocalWeight_t(v)$. Further, only when that weight metric becomes negative does it affect allocation decisions—$v$ is marked for spilling before coloring begins.

But consider the opposite situation where a number of subtiles of $t$ allocate $v$ to a register. If $v$ were to be spilled in $t$, it would incur the cost of storing $v$ on entrance to each subtile, and loading $v$ on each subtile exit. Nowhere in the formulas is that sort of cost represented. For this reason, a number of unfortunate decisions are made in epic.

If the previous scenario arises, what is needed is to boost the cost of $v$ in $t$, i.e., its $LocalWeight$, so that it appears to be more expensive to spill than its tile-local references alone would indicate. Below is the new $LocalWeight$ formula that incorporates the adjustment.

$$RegAdj_t(v) = \begin{cases} 0, & \text{if } InReg_t(v) = false \\ Transfer_t(v), & \text{if } InReg_t(v) = true \end{cases}$$
<table>
<thead>
<tr>
<th>Bench</th>
<th>CB</th>
<th>CK</th>
<th>% imp. (w/C_{TB})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$M$</td>
<td>$M_{TB}$</td>
</tr>
<tr>
<td>bzip2</td>
<td>35.12</td>
<td>26.23</td>
<td>3.62</td>
</tr>
<tr>
<td>epic</td>
<td>2.08</td>
<td>1.38</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Table 4.2: Dynamic spill operations for bzip and epic (billions)

\[
\text{Adjust}_t(v) = \sum_{s \in \text{subtiles}(t)} \text{RegAdj}_s(v)
\]

\[
\text{LocalWeight}_t(v) = \sum_{b \in \text{blocks}(t)} (P(b) \cdot \text{Ref}_b(v)) + \text{Adjust}_t(v)
\]

Table 4.2 shows benchmarks that improved for CK with the tweak. The epic benchmark went from a 27.9% loss (see Table 4.1) compared to CB before the change to a 1.71% gain after. Most of the gain is due to cutting the number of memory operations on the tile boundary (column $M_{TB}$), which was the original problem noticed in the assembly code. Benchmark bzip2 improved from 10.58% over CB to 14.12% over CB. All other benchmarks showed insignificant changes.

Our experience indicates that CK seems to be far more sensitive to changes in the spill heuristics than the CB allocator. Numerous ideas have been tried, but most of them give less than stable results. That is, some benchmarks improved while other degraded. Part of the difficulty in designing formulas that work consistently is the fact that only a tile and its children (or a tile and its parent) can be considered. Recall that in the bottom-up phase when computing spill costs for uncolored tile $t$, only its children have been previously colored, not its parent. So all spill decisions are based on what happened in the children and the nature of the references in $t$. So it is difficult to always make good decisions at $t$ that won't, say, be poor with respect to an ancestor of $t$ when reached. So the above formula is one that works reasonably well given the restrictions. It might also be useful to try a best-of-3 approach as in Bernstein [2], but in the context of CK. There is still much room for experimentation.
4.4 Summary

This chapter discussed a program structure-based allocator that can significantly reduce the number of dynamic spill operations (up to 73% on our benchmarks) compared to Chaitin-Briggs. Some modifications to further improve performance were presented. It is hoped that this work fills the long-standing gap in the literature regarding CK's actual performance, and that other researchers and practitioners gain a better understanding as a result.
Chapter 5

Related Work

Chaitin and his colleagues presented the first description of a functioning graph coloring register allocator [9, 8]. The next major step was made by Briggs and colleagues [5, 3, 6] by improving on Chaitin’s base coloring algorithm (optimistic coloring) and enlarging the class of rematerialization candidates. As discussed in Chapter 2, much of the later work dealt with improving on Chaitin’s approach to spilling.

Instead of using a single spill metric, Bernstein et al. [2] tried a best-of-three approach. They attempt to reduce the chance of any one heuristic performing poorly by trying three different spill metrics and choosing the one resulting in the least spill code. While this approach can reduce heuristic noise in the spill choice, it does not address the fundamental problems of spilling everywhere.

Bergner’s interference region spilling [1] improved substantially on Chaitin’s spilling approach. Bergner observed that instead of spilling a live range completely, it is possible to spill the range partially—only where a register is not available for it. This is particularly helpful when only a small portion of two live ranges overlap. Like the passive splitter discussed in Chapter 3, Bergner compares the cost of spilling a live range everywhere against the cost of partially spilling it.

While these previous techniques improved the spilling behavior of Chaitin, they do not directly incorporate program structure into the process. Knobe and Zadeck [15] describe a structure-based allocator using the notion of a control tree, which is vaguely similar to a tile tree. This allocator is comparable to Callahan-Koblenz in that it can split live ranges around control tree nodes (e.g., loops), it can spill inside of conditionals, and its pruning of wedges is not unlike Callahan-Koblenz’s handling of
LBNRs. Unlike CK, however, wedges are pruned before the main allocation process starts. That is, a pre-pass estimates the register pressure of a region, and attempts to reduce the pressure in high-contention regions by pruning as many wedges as necessary and available. Unfortunately, no empirical evaluation of the technique is presented, though there is reason to believe it would perform in roughly the same league as CK.

Norris et al. [20] describe an allocator that operates on the program dependence graph. The PDG is also similar to the tile tree in that it describes (along with data dependence information) the hierarchical structure of a program. Like Callahan-Koblenz, they attempt to intelligently place spill code by using the PDG. In fact, their approach is remarkably similar to CK— it appears to be essentially a version of CK adapted to the PDG.

Lueh’s fusion allocator directly leverages program structure by isolating regions of high frequency, and performing simplify for those regions first before attending to lower frequency regions. Each time a region is simplified, its graph is fused (unioned) to the next region’s graph to be considered. A fusion operation is done as long as simplification doesn’t block on the unioned graph. If the fusion for a newly added region blocks, then spill code is inserted at the boundaries of the region until simplification can proceed. Eventually, fusion will have built a simplifiable graph for the entire function. One problem with this allocator is the fact that fusion spills when simplify blocks. That is, it immediately must spill a node when all remaining nodes have at least \( k \) neighbors rather than continuing to remove in an optimistic fashion. So while some steps are taken forward by incorporating program structure, one step is taken back by not maintaining an optimistic simplifier. Even with that limitation, Lueh reports improving performance over Chaitin-style allocation by an average of 8.4% on the SPEC92 benchmarks [18].

A somewhat obscure but interesting work by Zobel [23] uses program structure to guide register allocation explicitly. Zobel builds interference graphs for regions,
such as loops or conditionals, with the goal that each can be colored independently. Unlike CK and others, however, this method builds regions in such a way that after coloring, the regions can be put back together into a full program without the need for copies at the region boundaries. While the primary reason for the decomposition is to enable parallelization of the register allocation process, better allocations can also be produced because Zobel attempts to build each interference graph such that it is an interval graph. Interval graphs are noteworthy in that they can be colored optimally in polynomial time. Zobel reports that for the benchmark suite she used, most of the interference graphs produced by the technique are interval graphs. On the other hand, the benchmark suite is fairly simple by today's standards, and it isn't clear whether the same level of success would be obtained for more sophisticated programs. Unfortunately, this work carries the same criticism as a number of the others—no comparative study is performed against another high-quality register allocator.

While this thesis is primarily concerned with Chaitin-derived allocators, there are other approaches that try to deal with spilling more intelligently. Perhaps the most widely known of these is the priority-based coloring of Chow and Hennessy [10, 11]. They build a slightly more coarse interference graph and use a form of live ranging splitting to make pieces of an uncolorable live range colorable. However, they do not directly incorporate knowledge of the program structure and thus may (for example) spill in a loop unnecessarily.

The Multiflow compiler described in [17] allocates registers for traces (collections of basic blocks), with the most frequently executed traces being processed first. In addition, they delay assigning a register to ranges that are live across but never referenced (delayed bindings) in the trace until after the trace is processed. This achieves the same effect as CK's handling of LBNRs, so that no unmentioned live range is given priority over one mentioned in the trace. Within a trace, the allocator does not operate on the graph coloring paradigm at all, but much more like a typical
bottom-up, basic block register allocator (e.g., see Chapter 13 of [13]).
Chapter 6

Conclusion

Standard Chaitin-Briggs allocators may insert unnecessarily large numbers of spill instructions, and do so in high-frequency regions of the program. Such behavior means that the number of spill operations executed at runtime might be higher than it could be. This thesis attempts to address the above problems by guiding spill decisions and spill code placement with program structure. Essentially all knowledge of the program structure is lost when the allocator's primary data structure, the interference graph, is constructed. We propose and evaluate some approaches to incorporating that lost information into the allocation process.

6.1 Contributions

This thesis makes the following contributions.

Improved Passive Splitting  We presented algorithmic improvements to Simpson's passive live range splitting allocator that increase its effectiveness by up to 50%, as measured by the number of dynamic spill operations. Program structure is used to guide the global splitting of live ranges and to place split operations at infrequently executed portions of the program. We also optimistically reconsider spill decisions at certain points to potentially enable more splits and/or fewer spills. Certain issues that complicate implementation of the original passive splitting method were discussed so that other researchers and practitioners can better understand how to incorporate passive splitting into their compilers.
Hierarchical Graph Coloring—Understanding and Improvements We implemented the Callahan-Koblenz allocator in the LLVM infrastructure and presented a comprehensive evaluation of the Callahan-Koblenz allocator as compared to the baseline Chaitin-Briggs allocator. CK produced reductions in dynamic spill operations by up to 73%, though some efficiency is lost due to factors such as additional branching. This comparison fills a long-standing void in the register allocation literature, since no empirical evaluation of CK had been previously published. A number of simple improvements were also presented, one directly in response to an examination of the poor performance of the epic benchmark.

6.2 Future Work

There are several directions for future work for the CK allocator. It was seen in Chapter 4 that the heuristic cost functions can behave poorly at times since only two tiles (one colored) are generally available for inspection at any point in the bottom-up phase. This would seem to indicate that the number of tiles, and hence tile boundaries, should be decreased—but only to the point that the benefit of tiling is not significantly degraded. In epic for example, it was learned that the middle loop of a triply nested loop with little register pressure contributed to bad decisions. It would make sense to merge the tile corresponding to that loop into either its parent tile (the loop containing it) or its child tile (the loop it contains). Likewise, merging tiles for conditionals would be sensible, since conditionals often introduce a large number of tiles, and many of very small size.

The key challenge for tile-merging is determine which tiles should be merged (or alternatively, never constructed in the first place) so that the benefit of tiling (live range splitting) is maximized, but the number of heuristic failures due to excessive numbers of tile boundaries is minimized. In addition to examining program structure to guide these decisions, register pressure within each tile should be taken into account. We believe this is one of the most promising areas of study for improving
Another area for study is in simplifying the overall algorithm itself. Specifically, might there be a way to achieve the same effect in a simpler, single-pass algorithm that is currently done with the two-pass algorithm? Two-passes necessitate a substantial amount of bookkeeping between phases, rebuilding of interference graphs from summaries, etc.

As branch misprediction penalties of deeply pipelined microprocessors increase, we are motivated to reduce branching as much as possible. Though the additional branches sometimes added by CK do not dominate execution time, it is still desirable to retain only the original program branches. To that end, it would be useful to modify the algorithm so that its tile tree construction does not need to insert branches. Unfortunately, this may prove difficult because every stage of the algorithm relies on the premise that we can consider either the parent or child of a particular tile in the tree. Not allowing the insertion of branches would mean that the tile structure would not necessarily be a tree. How easily the algorithm can be adapted is an open question.

With regard to the improved passive splitting algorithm, further work includes incorporating conditional program structure into the allocator, along with the profile feedback necessary to support it. It would also be interesting to re-implement the algorithm in a production-quality compiler targeted to a relevant, mainstream microprocessor architecture, as opposed to the somewhat idealized world of ILOC. This would also permit a much wider range of applications to be tested. It would also allow a comparison of CK and passive splitting.

Finally, we believe that CK and PS can be fruitfully combined into an even stronger allocator. That is, even though tiles are generally smaller than the overall function they belong to, each can still be a large region involving spilling. It should be possible to apply passive splitting within each tile to reduce intra-tile spilling. This in turn might improve the allocation of global values, and hence reduce tile-boundary
6.3 Final Thoughts

In addition to our mission of performing basic research in compilation, the Rice Scalar Compiler Group attempts to help the industrial practitioner in understanding particular algorithms, and whether their efforts would be well-spent incorporating an algorithm into their framework. To that end, we give our thoughts on the algorithms examined in this thesis.

We have seen that both CK and PS improve substantially on the “spill everywhere” approach of Chaitin. Further, given a functioning CB allocator, incorporating PS can be done with a reasonable amount of work because it maintains the overall flow of CB. This combination of ease-of-implementation and good register allocations gives us confidence in recommending that PS be implemented in production or research compilers that already contain a CB allocator (i.e., practically all of them).

Whether to recommend the CK allocator, on the other hand, is not so obvious—at least in an industrial/production environment where compressed delivery schedules are the norm and compiler stability release-to-release is key. While it clearly improves on CB, it also carries the cost of a significantly more complicated implementation. It takes a fairly radical departure from the relatively simple CB flow (recall Figures 2.1 and 4.2), requiring multiple passes, additional bookkeeping, and the like. Debugging a CK allocator is also more difficult. No longer is there a single interference graph for a compilation unit—now each region (tile) has its own interference graph which must be colored. Further, each graph is colored in each of two phases. Failure modes are more numerous and less easily identified. For the research environment, we feel that CK is worth implementing for future experimentation. Since the technical improvements made by CK are no longer in doubt, the decision for practitioners will rest more on the ability to create enough room in the schedule to plan for at least a couple months of intensive work, if not more.
Bibliography


