RICE UNIVERSITY

Reducing the Impact of Spill Code[1]

by

Timothy J. Harvey

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

Master of Science

Approved, Thesis Committee:

Keith D. Cooper, Associate Professor
Department of Computer Science
Rice University

Linda Torczon, Faculty Fellow
Department of Computer Science
Rice University

John K. Bennett, Associate Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas

May, 1998

Reducing the Impact of Spill Code[2]

Timothy J. Harvey

Abstract

All graph-coloring register allocators rely on heuristics to arrive at a "good" answer to the NP-complete problem of register allocation, resulting in suboptimal code due to spill code.

We look at a post-pass to the allocator that removes unnecessary spill code by finding places where the availability of an unused register allows us to "promote" a spill to a register.

We explain and correct an error in Briggs' spill-code insertion algorithm that sometimes inserts an unnecessary number of spill instructions. This fix has an insignificant impact on the runtime of the compiler and never causes a degradation in runtime of the code produced.

We suggest minimizing the impact of the spill code with a small separate memory dedicated to spills and under the exclusive control of the compiler. We show an algorithm and experimental results which suggest that this hardware construct would significantly decrease the runtime of the code.

[2]This work is based in part upon work supported by the Texas Advanced Technology Program under Grant No. 003604-015 and by Darpa through Army Contract DABT63-95-C-0115.

Acknowledgments

Almost everything I have learned about compilers in the past six years is due to the (*patient*) efforts of two people:

1. Keith Cooper, who is simultaneously an advisor, employer, and, most importantly, friend

2. Preston Briggs, whose constancy of friendship is itself only equaled by his patient, unceasing mentoring

To these two people go my heartfelt appreciation; without their dedication to teaching, this work would never have happened.

Singling out people is hazardous, because it would seem to imply that others were less important – a ludicrous idea. Keith and Preston simply had the most direct impact on my knowledge of all things compilers.

I'm compelled to also acknowledge the loving support of my parents, who have always believed in the value of education.

Likewise, thanks go out to my humblingly many friends, too numerous to mention, a great number of whom have long ago completed their own graduate work and have been wondering what took *me* so long to complete mine. These poor souls were forced to listen to hours of my enthusiastic spoutings about my research[1] and deserve more than a few beers in recompense.

Unlike my human friends, my dogs, Annie, Alex, and Sophie, were more than happy to listen to me,[2] and their support cannot be discounted.

Finally, the other members of my committee, Linda Torczon and John Bennett, who have assured that if it's in here, it's right. Thanks very much.

---

[1] Most are not computer scientists, and didn't understand a word I was saying – not that *that* ever stopped me...

[2] Providing they were getting their ears scratched at the same time.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Compilers and register allocation

An optimizing compiler is a compiler that, as part of the translation process, also performs transformations on the input, with the goal of producing output that is *better*, as measured by resultant code size, speed of execution, *etc.* An optimizing compiler has three general phases:

1. the front end, which parses the input language and produces an intermediate form that is manipulable by the subsequent two phases

2. the optimizer, which performs transformations on the intermediate form

3. the back end, which transforms the intermediate form into the target language

The back end of an optimizing compiler has, as part of its job, the task of *finite resource allocation.* That is, the back end is primarily concerned with taking the intermediate form produced by the optimizer – which may reflect the target architecture's capabilities to a greater or lesser extent – and mapping it to reflect the structures of the target machine. For example, the instruction-selection pass of the back end maps the intermediate form into the actual instructions that the machine understands.

Another pass of the back end is the *register allocator*, that assigns values represented in the intermediate code to the actual physical memory on the machine. The allocator usually has two choices about where to place a value: it can either place it into a register, the fastest memory on a chip, or into main memory, which will cause accesses to that value to take longer.

Because the machine has only a limited number of registers, we can write a program that needs more registers than the machine has. This will force some number of the program's values into the slower memory. The allocator's job, then, is to decide which values are kept in registers and which values are kept in memory. Within the set of choices, it should select one that allows the program to run as fastest.

1.2 Overview

Because compilers rely on imperfect heuristics to solve the problem of register allocation, there is always room for improvement. This thesis will explore the following aspects of register allocation:

1.2.1 Spill promotion

Cooper and Simpson developed an algorithm for live-range splitting based on interference patterns [20]. This algorithm can be expensive in terms of both time and space. We look at an alternative algorithm that runs after the allocator has finished and looks for the kinds of spill mistakes that Cooper and Simpson detect. We present both the algorithm and experimental results that point to its effectiveness.

1.2.2 A fix to the spill-code insertion phase of the allocator

Briggs' dissertation gives an elegant algorithm for both computing spill costs and inserting spill code. However, an implicit assumption in the algorithm can cause the allocator to insert unnecessary spill code. The solution is a straightforward adjustment to the spill-code insertion routine that significantly reduces spilling on codes that trigger the error. This work, combined with the spill promotion work mentioned above, points to further improvements that suggest substantial benefits for a minimal increase in compiler analysis and runtime.

1.2.3 Memory allocation

To understand how better to reduce the impact of spill code, we need to understand the kind and amount of spilling inserted by the allocator. We look at the effects of applying a graph-coloring memory-allocation algorithm to the memory locations used for spilling and discuss the theoretical application of tailoring the spill code to different architectures' memory systems.

1.2.4 Compiler-controlled memory

Finally, we take a different approach to the issue of spill code. Instead of trying to reduce the amount of spill code that the allocator inserts, we propose to use a piece of on-chip memory exclusively for spill code. This memory would be as fast as cache memory, but it would be under the control of the compiler rather than the hardware.

We use the results from the memory allocation study to show that this memory space could be quite small, and we present an algorithm that the compiler could employ to utilize this space. We also present experimental results that suggest that this method would have a significant impact on a program's runtime.

Chapter 2

Background

In this chapter, we will discuss some of the context necessary for our discussion of register allocation and spill code. We discuss the underlying representation of the code with which we work. We examine the standard memory hierarchy as a motivation for this work, and we describe the graph-coloring register allocation algorithm. We present the thesis for this work, and sketch the Briggs register allocator on which we based our experiments. Finally, we present some of the history of register allocation and spill-code reduction.

2.1 The control-flow graph and intermediate representation

A compiler's front end builds the control-flow graph (CFG), a graph that encodes control-flow information in its structure. Each node in the graph is some number of instructions, and edges between nodes represent flow of control between instructions. The simplest CFG would have a node for each instruction in the program, but this would require more memory to encode this data structure than is actually needed. Any two instructions that always occur in the same order could actually be stored in the same node in the CFG. Thus, the CFG data structure encodes an ordering on two different levels: first, edges between nodes represent possible paths through the program, and second, multiple instructions in a single node are ordered linearly. A node in the CFG data structure actually represents a *basic block* – usually referred to simply as a "block" – a linear list of instructions such that if one instruction executes, they all do.

In the MSCP compiler, the instructions are represented in ILOC (pronounced, "eye-lock") in a form that looks like a generic three-address assembly code, with some constructs designed to enhance human readability. One of the notable features of ILOC is that all values are represented by an abstract data type we refer to as a *register*. Intuitively, these registers correspond to a machine's set of physical registers with one exception: whereas all machines have a fixed, finite number of registers, the number of registers in an ILOC program can be sized to demand.

ILOC allows for an arbitrary number of registers as a direct result of Backus's philosophy of separating concerns [3]. Backus's idea was that transformations in the compiler's optimizer could be thought about more clearly as a result of removing concerns about the target machine's physical limitations. Under Backus's model, the back end of the compiler is charged with understanding the limitations of the target machine and transforming the code to honor those restrictions. Thus, in the MSCP compiler, the back end has to understand how many physical registers are available and then map the greater number of registers in the ILOC program into those physical registers.

2.2 The different kinds of memory

Today's CPUs have multiple layers of memory that they can access, where the tradeoff is that the faster the memory, the smaller it is. Generally, there are four levels of memory built into a modern machine.

The first is the disk, hard drive, *etc.* This memory is extremely slow relative to the CPU, but it can be of (theoretically) unlimited size. The addresses on a disk are disjoint from the memory addresses of a particular executable. This is important, because

Faster than the disk and considerably smaller is the computer's RAM. This is moderately slow memory in relation to the CPU, but it can be very large – typically, some number of megabytes or even several gigabytes in size. This memory is

The next layer of memory is the CPU's cache. This is memory, usually located on the CPU's chip, that is reasonably fast, but relatively small, usually only a couple of hundred kilobytes. Commonly, the cache can be multi-leveled, with each level larger than the former level and, consequently, slower than the preceding level. The address space of cache memory is directly related to the RAM: the higher-order bits of a RAM address are usually used as the address of a cache entry.

Finally, the fastest memory is the small set of CPU registers. Of these four levels of memory, only the registers are under the complete control of the compiler; the other levels are controlled by the hardware. Thus, we usually use the term "memory" to refer to any level of memory except the registers. This memory is exactly as fast as the CPU itself, but it is very small – on current machines, the set of registers is usually capable of holding a maximum of sixty-four values. On almost all processors, a value cannot be used by the CPU unless it is in a register.

For example, the UltraSPARC 140 on which we ran the experiments described in this thesis has a processor that runs at 140 MHz. It has sixty-four registers, each of which holds eight bytes. The two-level cache has thirty-two kilobytes on the chip and 512 kilobytes on another chip. The total RAM is 256 megabytes, and the hard disk can store two gigabytes of data. The first-level cache runs at the same speed of the processor, while the second-level cache runs at about a third of the speed of the processor. The RAM has a 60 nanosecond access time, or just over eight processor cycles.

As another example, the SGI Origin uses a MIPS R10000 processor, which has a speed of up to 250 MHz, and requires as many as 66 cycles to retrieve a value from the RAM. The primary cache runs close to the speed of the processor, and the secondary cache's speed is 80% of the processor's speed.

Clearly, keeping a value in a register instead of in RAM is profitable. Keeping a value in cache would seem to be a reasonable alternative, but the compiler does not have control over the cache. To the compiler, a store to memory can go anywhere, subject to the state of the cache, and so the compiler must assume that a spill goes all the way to the RAM. It is even problematic if the value gets spilled into the cache. Since the cache is of very limited size, moving a value into cache often requires moving another value *out* of the cache.

The large (and growing) discrepancy between cache speed and RAM speed encourages the compiler to be very careful about utilizing the cache. Although the compiler must rely on the hardware to control the cache, the hardware's behavior is somewhat predictable, and a number of optimizations have been developed that attempt to structure the code to maximize cache performance. Because they have already run by the time the back end inserts spill code, these optimizations can easily have their improvements negated by unfortunate interaction between the cache and the back end's spill code. Thus, it is imperative to the overall performance of optimized code to reduce spill code as much as possible.

2.3 The register allocator

The phase of the back end with which we are concerned is the *register allocator*, so named because its job is to allocate the scarce resource of registers to minimize the amount of spill code inserted.

No matter how clever the algorithm used to map the abstract registers to the physical registers, there will always be cases where this is impossible. To understand this, we need to introduce the concept of *liveness*. A value is *live* in the region between the point it is defined and any point that value is used.

The number of variables live at any point in the program defines the *register pressure* at that point. That is, register pressure at a particular program point is defined as the demand for registers if all of the values live at that point were to be stored in registers. If the register pressure at some point in the program is greater than the number of registers that can hold those values, some number of the values must be stored to memory. Register allocation segregates live values into two classes: values stored in registers and values stored in memory. The latter, while live in the sense of being used at some future point in the program, do not add to the register pressure while they are stored in memory.

A simplistic algorithm for register allocation, then, would be to walk forward from the beginning of the code being compiled, assigning values to registers. For each definition of a new value, find a register that is not being used and put the new value into that register. If all of the registers are full, clear one of them by storing its value to memory. For each use of a value in some instruction, make sure that the value is already in a register, or load the value out of memory (where it must have been previously stored to make room for some other value).

This algorithm captures the idea of register allocation, but few of the myriad details. For example, the algorithm gives only one way to clear a value out of a register: save it to memory. But if the program no longer needs that value, there is no reason to keep the value in a register past the point of its last use, or its *death*.

Remember that a value's liveness is defined as the region between its definition and its uses. If we know what is live at the end of a block, we can find deaths by walking backwards through the list of instructions in the block. We start by creating a set, `currently_live`, initialized with those values live at the end of the block; the live set at the end of a block simply denotes those values that are used later in other blocks. At each instruction in the backward walk, we compare the uses in the instruction with the members of `currently_live`. If a use is not in the set, then this instruction marks the death of that value. If a value is not in the block's live-out set, it is not used after the end of the block; thus, by walking backwards through the block, the first time we find a use of a value is the last time that the value will be used not only in this block, but in all subsequent blocks. A value's death marks the point at which its register becomes free to be used by another value.

To keep `currently_live` up to date, we then add all of the instruction's uses to the set, since from this point upwards these values are live, and remove the values defined by this instruction, since we are walking backwards through the block, and a definition marks the beginning of the value's live range.

With this algorithm, we can introduce the concept of *interference*. As we perform the backward walk, we know the set of values that are live at each instruction. Normally, we move many values into and out of a particular register over the course of the program, so a particular register, $r1$, might at one time hold value v_1, and at another time hold value v_2. However, the physical register $r1$ cannot be used for both values if at any point in the program v_1 and v_2 are simultaneously live. In that case, v_1 and v_2 are said to *interfere*, and they must be stored in different physical registers.

We compute liveness for the entire program. Taking the blocks one at a time and walking backwards through each block's instruction list, we compute the entire set of interferences between all of the values, by examining which values are simultaneously live at each instruction. We can build a graph from this information, where each node in the graph represents a value, and each edge between two values represents the fact that they interfere. This graph is called the *interference graph*, and this is the graph that forms the cornerstone of a Chaitin-style register allocator.

2.4 Graph-coloring as the model for register allocation

Chaitin *et al.* proposed using the interference graph to model register allocation as a graph coloring problem. The graph coloring problem is to assign a color to each node in a graph with the restriction that two nodes connected by an edge cannot have the same color. The analogy to register allocation is direct: if we substitute "registers" for "colors," and we "color" the interference graph, then the problem becomes assigning registers to nodes with the restriction that nodes that interfere cannot share the same register. The coloring problem is not an exact match to register allocation; in the coloring problem, there are an unlimited number of colors, and the problem is to find the smallest number of colors that can be used. In register allocation, the number of "colors" – that is to say, registers – is fixed.

Intuitively, we can color a graph by taking the nodes in some order and successively applying colors (registers) to them, mindful of the restriction that no two adjacent nodes can be assigned the same color. If we find a node whose neighbors collectively use all of the available colors, we *spill* the value to memory. This is extremely simplistic; each time a value is spilled, it changes the interferences of that node in subtle ways. If, during coloring, we are forced to spill a node, we can finish the coloring, but then we must start over again by rebuilding the interference graph and coloring this new graph. In some sense, the introduction of *spill code* changes the "shape" of the code, and we have to redo all of the analysis and work. In general, we only rebuild the interference graph two or three times before we can color it without adding more spill code.

The register allocation problem, then, is not in finding the minimum number of colors for the interference graph, but, rather, inserting the minimum amount of spill code necessary to color the graph with a fixed and finite number of colors. At any point in the coloring process, there may be a number of nodes in the graph that cannot be colored. If some of these nodes are adjacent, spilling one may actually allow its neighbor to receive a color. Thus, the problem becomes one of minimizing spill code.

This is complicated by the fact that each node may have a different spill cost. Remember that nodes in the interference graph represent values, but there is no indication, for example, of how many times that value is used in the program. Since the goal of register allocation is to minimize the number of extra instructions – in the form of spill code – executed by the program, care must be taken when deciding which node to spill.

The problem of finding the smallest number of colors is NP complete [1, 39], as is the problem of finding a coloring for a fixed number of colors. Thus, register allocation must rely on heuristic solutions to solve the problem. Because the graph-coloring problem is not an exact match to register allocation, there is little in the literature on graph coloring applicable to register allocation. As a result, Chaitin *et al.*'s original work has undergone numerous reformulations, many of them resulting in significant improvements to the amount of spill code inserted.

2.5 The Briggs allocator

We will concentrate on the Briggs allocator in this thesis, because it is widely used and its implementation is clearly laid out in Briggs' own dissertation [7]. As a result, we can be confident that other researchers could expect to replicate our results.

In the Briggs allocator, there are seven steps, as shown in Figure 2.1.[1] The following is a short synopsis of each step.

- **renumber** renames values into live range names, by first converting to SSA form [22] and then unioning values that appear as parameters in a ϕ-function.
- **build** constructs the interference graph.
- **coalesce** coalesces copies. It rebuilds the interference graph and tries to coalesce more copies until no more copies are coalesced.
- **spill costs** estimates the cost of spilling each live range.
- **simplify** removes nodes from the interference graph, putting them onto a stack as they are removed. This phase is critical to the final code quality, because removing the nodes in the wrong order can cause the graph to be more difficult to color and/or increase the amount of spill code required.
- **select** chooses colors for all the live ranges on the stack created during the *simplify* phase and marks any nodes that do not receive a color as needing to be spilled.

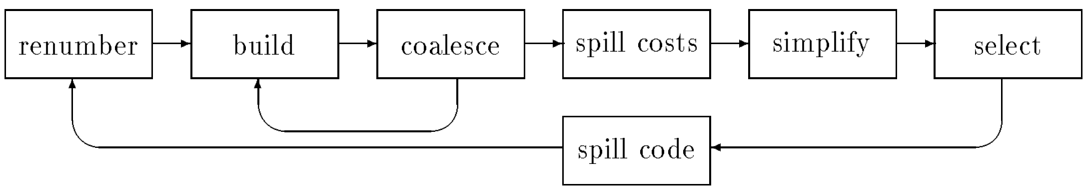

Figure 2.1 A flowgraph of the Briggs allocator

[1]This picture is somewhat simplified, but accurate for purposes of exposition.

- **spill code** inserts spill instructions to spill the nodes so identified in the **select** phase.

This is the allocator on which we ran all of the experiments in the rest of this thesis. We chose not to include some recent enhancements such as George and Appel's iterated copy coalescing [24] and Bergner *et al.*'s interference-region spilling [5], because we are not confident that they have received widespread implementation as yet. When we explain our results, we will mention how some of these enhancements might affect those results.

2.6 Previous work

Register allocation is a significant part of the compilation process. The quality of the effort put into the register allocator defines, in large measure, the quality of the code produced by the compiler. However, graph coloring as a paradigm for register allocation has two significant drawbacks. The first is that graph coloring is NP complete. Thus, we have to rely on a conglomeration of heuristics to do a "good" job of allocation. The second flaw is related to the first; the general graph coloring problem is an imperfect analogy to the problem of register allocation.

Despite this, there has been a plethora of research associated with improving the performance of register allocation. The following are some of the highlights.

Chaitin *et al.* built the first working register allocator based on graph-coloring in 1981 [16]. Chaitin published a single-author paper the following year describing the heuristics used to direct more efficient spilling than the original formulation. In 1989, Briggs, Cooper, Kennedy, and Torczon modified the coloring heuristic to be optimistic instead of pessimistic [9]. Further enhancements by this team resulted in still more improvements to spilling such as live-range splitting and rematerialization [7, 11].

More recent work of particular interest includes George and Appel's work on iterated register coalescing, in which they attempt to reduce the register pressure by folding copy operations during the **simplify** phase of the compiler [24]. Their ideas seem to be only efficacious in specific contexts, and may not be significantly helpful in general. Our implementation of their algorithm suggested that its improvements occur mainly in an environment of a high proportion of copy instructions relative to the total number of instructions in the program and using an allocator which does not have some of Briggs *et al.*'s improvements, such as rematerialization [10]. On the other hand, Bergner *et al.*'s work on interference-region spilling seems to provide substantial decreases in the amount of spill code the allocator described by Briggs inserts [5].

Some research has modified or even rejected the paradigm of graph coloring to perform allocation. In 1984, Chow and Hennessy described an algorithm they call priority-based coloring that relies on heuristically splitting live ranges [18]. 1991 saw the publication of an algorithm by Callahan and Koblenz that seeks to choose the best spills by dividing the problem into pieces based on the loop-structure of the CFG, and then combining the results of applying their algorithm to the different pieces [13]. In 1992, Proebsting and Fischer published their "probabilistic" allocation algorithm, that separates local and global allocation and seeks to minimize spill code by using their own technique for allocation – choosing which values to put into registers – and using graph-coloring for assignment – choosing specific registers to put each value already designated as receiving a register [37].

Bernstein *et al.* went further, not being satisfied with the uncertain performance of any particular heuristic [6]. Instead, they implemented three different heuristics into their allocator. At each iteration, they tested each of the three and chose the one that inserted the least spill code.

It is difficult to compare the efficacy of these techniques against each other, partly because of the immense effort required to implement each one, and partly because these techniques sometimes favor particular hardware configurations, code shape (such as George and Appel's environment of many copy instructions), *etc.*

2.7 Our experiments

In our experiments, we use a test suite of 169 routines, including code based on Forsythe *et al.*'s book on numerical methods [23], the SPEC '90 benchmarks [40], and the SPEC '95 benchmarks [41]. The SPEC '95 benchmarks have been transformed by the insertion of advisory prefetch instructions intended to improve cache behavior [35], and this transformation has the impact of increasing the register requirements in those routines. Out of the suite of 169 routines, 59 required some amount of spill code, and it is on these 59 routines that our results are generated.

The back-end of our compiler is a translator from ILOC to C code, that we then compile with Sun's `cc` compiler. This back-end design allows us to modify the parameters of the abstract target machine while instrumenting the code to give us valuable insights into the execution of the code. All codes in our test suite were extensively optimized before being applied to the various register and memory allocations in this thesis.

Our machine model is based on a 64-register architecture (32 general-purpose registers and 32 floating-point registers) with instruction costs of two machine cycles for memory operations and a single cycle for all others. This instruction-cost model comes from the example of Briggs [7]. It is a reasonable generalization of the complicated question of memory-access times and tries to average the effect of multi-level caches, scheduling to hide latency, *etc.*

In the following chapter, we examine the implementation of an idea that showed promise, only to receive moderate benefit. In Chapter 4, we show a small modification to Briggs' original spill-code-insertion algorithm that can have a sizable impact on the running time, but, because of the complexity of the problem, escaped detection until now. In Chapter 5, we examine the spill code that gets produced by the allocator and suggest some uses of that information. Finally, in Chapter 6, we explore the ramifications of giving the allocator explicit control over some portion of on-chip memory in order to reduce the runtime impact of spill code.

Chapter 3

Removing Unproductive Spills

3.1 Introduction

A fundamental flaw with graph-coloring register allocation is that the allocator makes its decisions based on a simple model of the program that provides little insight into geographical locality. That is, the interference graph encodes whether or not two values are simultaneously live, but not *how* they coexist. For example, if two values, v_1 and v_2, interfere, the interference graph cannot tell us at which points in the program they interfere.

The lack of geographical information can cause the register allocator to make incorrect decisions when choosing a value to spill. Consider the example in Figure 3.1. When two live ranges, r_1 and r_2, overlap, one of two cases can occur. In the first case, the live range of r_1 completely encloses the live range of r_2; that is, the birth of r_1 occurs before the birth of r_2, and the death of r_1 occurs after the death of r_2, and there are no uses of r_1's value within the range enclosed by r_2. In the second case, the live range of r_2 also encloses some use (including, perhaps, the last use) of r_1.

In the first case, spilling r_2 instead of r_1 does not reduce the number of r_1's interferences. Indeed, the number of interferences goes up, because spilling a value creates multiple small live ranges, each of which continues to interfere with r_1.

Cooper and Simpson propose a splitting method to handle this problem [21]. Each time the register allocator needs to spill a value, it checks to see if it would be cheaper to spill the entire live range or split that live range into smaller pieces, some of which will be able to be colored and so will not have to be spilled. Unfortunately, their algorithm also requires the use of a square bit matrix in place of the triangular bit matrix recommended by Chaitin, potentially causing a significant increase in the amount of memory required to perform allocation.

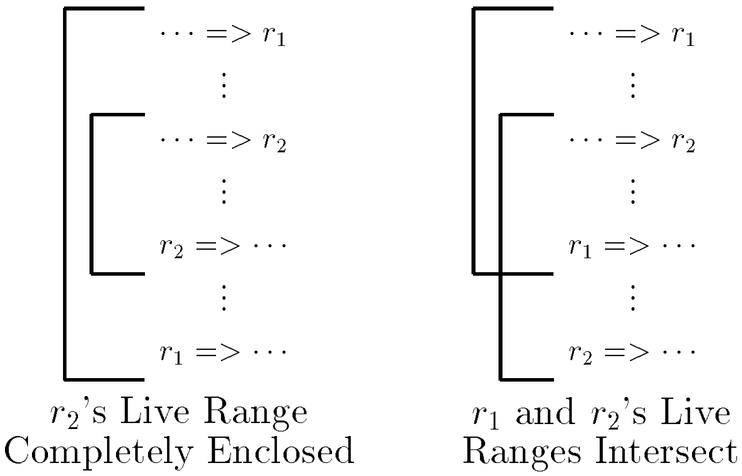

Figure 3.1 An example of how live ranges overlap

3.2 The algorithm

We propose a less aggressive transformation that would run as a follow-up to register allocation. In this method, we propose identifying values that were spilled by the allocator that can be "promoted" to a register. The algorithm, which only works on single basic blocks, is shown in Figure 3.2.

At each block, we will initialize two sets, the `currently_live` set, which tracks variables that are live as we walk backwards through the block, and the `candidates` set, which holds the list of loads from spill locations that we have encountered without yet encountering the corresponding store. At each instruction, we remove from `currently_live` the list of registers defined by the operation (since they are no longer live above this point), and add into the set the registers used in the instruction. If the size of the `currently_live` set reaches the number of registers available to be allocated, k, we clear the `candidates` set, since we will not be able to promote any of those spill instructions. If we find a store to a spill memory location and that location is in the `candidates` set, then we can try to promote the spill, as shown in the algorithm in Figure 3.3.

To promote a value from spill memory, we need to find a register that is unused between the store to the spill location and the load from it. From the previous discussion, we know that the register pressure is always at least one less than k, but

```
build pruned SSA form and def-use chains; preserve the liveness analysis
create the candidates and currently_live sets
for each block b:
    clear the candidates set
    initialize the currently_live set with live_out[b]
    for each instruction, i in a backwards walk over b:
        if sizeof(currently_live) == k:
            erase the candidates set
        if i is a load from spill memory:
            add the spill location to the candidates set
        if i is a store to spill memory:
            if the spill location is in the candidates set:
                    promote the candidate
        remove all of the definitions in i from currently_live
        add all of the uses in i to currently_live
```

Figure 3.2 Algorithm for single-block spill promotion

that does not mean that a particular register is always unused in this region. To find an unused register, we will walk forward from the store until we hit the corresponding load, keeping track of which registers are defined and used between the two spill instructions. To start, we create a new set, `available`, that we initialize with the members of `currently_live`. We then take the complement of this set, giving us the registers that are not currently in use. In a forward walk between the store and the load, we remove from `available` any register that is either defined or used. When we reach the load, anything left in `available` is a register unused between the load and the store. Now, we try to minimize the support code we will need, so we first check to see if the register stored into memory is free – if not, we will have to insert a copy instruction immediately following the store to copy the value into some other register. We next try to use the value defined by the load from the spill location – if this is a member of `available`, we will not have to insert a copy operation to replace the load, since the value will be in the correct register for subsequent use. If neither of those registers is in `available`, we choose any name and insert the appropriate code.

Regardless, the load can always be removed. If the load is the only use of the memory location defined by the store, the store, too, can be removed. We can tell this by looking at the def-use list for the memory location; if it has only one element in the list, we can remove the store.

We can modify this algorithm to run on extended basic blocks. If a block, b_j, has a single predecessor, b_i, we know that any values flowing out of b_i will reach b_j.[2] This is in contrast to the situation if b_j has multiple predecessors, say b_h and b_i. In this case, we may not be able to tell if a value in b_j flowed from b_h or from b_i. As Figure 3.4 shows, a block with multiple successors can nonetheless be part of many extended basic blocks. We can use extended basic blocks to increase the length of the instruction stream we can examine to look for spill promotion.

The expanded algorithm works in a similar fashion to the previous algorithm, except that the backward walk potentially continues through multiple blocks. Each time we enter a new block during this walk, we union together the `currently_live`

```
copy currently_live into available
complement available
for each instruction j in a forward walk starting at i:
    remove all of the uses in j from available
    if j is the load corresponding to i:
        break out of the loop
    remove all of the definitions in j from available
if the register, r, stored from in i is a member of available:
    replace j with a copy from r to the value defined by j
else if the register,s, loaded into in j is a member of available:
    add a copy from r to s immediately following i
    remove j
else pick some register, t, from available:
    add a copy from r to t immediately following i
    replace j with a copy from t to s
if j is the only use of the spill location defined by i:
    remove i
```

Figure 3.3 Algorithm for promoting the spill

[2] Instructions in a single basic block would always be part of an extended basic block if we made each instruction its own node in the CFG.

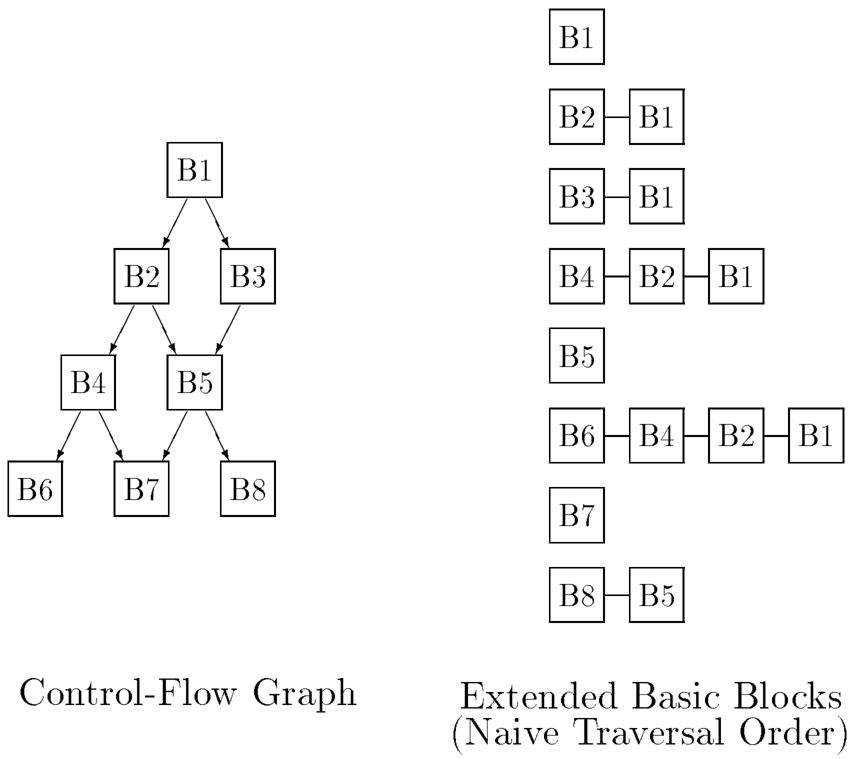

Figure 3.4 Example of extended basic blocks

set and the `live_out` set of the new block. Also, when we find a register into which we can promote the spill, we need to add that name into the `live_out` sets of all blocks encountered in the walk through the extended basic block except the starting block. Figure 3.4 shows the naive traversal order of our algorithm. As a speed improvement, we can stop the walk and move onto the next extended basic block when we leave the first block in the extended basic block and the `candidates` set clears. Further, each time we promote a spill, we remove the load from the spill location's def-use chains. If this list empties, we can then remove the store.

3.3 Experimental results

As we alluded at the end of the Chapter 1, the results of applying this algorithm were disappointing. Of fifty-nine routines into which the allocator inserts spill code, our algorithm only affected nine routines in the case of the single-block algorithm, and ten routines in the case of the extended basic block algorithm. The overall running time of the routines in all cases was reduced by less than one percent.

The numbers are shown in Figure 3.1. The first two columns of numbers are the number of cycles required just for spill operations. We estimate that each load and store of a value will require two cycles. The final column gives a percentage speedup of promotion over the allocator's code. Clearly, the extended basic block method provides greater reward; in all routines except `supp`, it provides at least a one percent improvement. In the case of `supp`, we see that the order in which the blocks are taken in the extended basic block algorithm is important – similar algorithms often benefit from attacking the blocks according to their nesting depth to offer the most opportunities to the most expensive spills.

While the overall running time of the code was only minimally affected, this experiment was not a complete failure. We have shown that the allocator *does* insert unnecessary spills that can be located and promoted, and the impact on running time may actually be more noticeable on specific machines – that is, by reducing the number of memory operations, we reduce the demand on a processor's memory system, which can have significant impact.

3.4 Future work

The primary reason why the overall running time of this program was unaffected was that, too often, we needed to copy the promoted value into a temporary register –

Routine *Name*	Original Memory Cycles	Memory Cycles After Promotion	$\frac{Improved}{Original}$
Single Basic Block Results			
twldrv	60,675,108	59,921,412	0.99
radf4	3,651,250	3,543,300	0.97
radb4	3,632,200	3,524,250	0.97
radf5	2,159,000	2,114,550	0.98
radb2	2,057,400	2,019,300	0.98
radf2	2,019,300	1,981,200	0.98
radb5	1,905,000	1,866,900	0.98
supp	744,464	738,752	0.99
decomp	252	250	0.99
Extended Basic Block Results			
twldrv	60,675,108	59,627,004	0.98
radf4	3,651,250	3,479,800	0.95
radb4	3,632,200	3,467,100	0.95
radf5	2,159,000	2,038,350	0.94
radb2	2,057,400	1,974,850	0.96
radf2	2,019,300	1,943,100	0.96
radb5	1,905,000	1,803,400	0.95
supp	744,464	740,656	0.99
decomp	252	246	0.98
rffti1	188	184	0.98

Table 3.1 Results of spill promotion

the register unused between the store and the load – and then copy it again from the temporary register into the register defined by the load. Because we are adding two single-cycle operations and usually only removing a single two-cycle memory operation, the running time of the code did not decrease.

However, we know that there is a register free across the region between the two spill instructions. We also know that we save at least one of the two copies we otherwise put in if the free register is the same one used in the store or the load instruction. Instead of copying our promoted value into the free register, we can rewrite instructions in the region between the two spill instructions to use the free register instead, when those instructions use the register used in the store instruction or loaded into by the load operation. This would have the effect of freeing up one or both of the two critical register names, saving us at least one copy – two, if the register used in the store is the same as the one defined in the load. In the worst case, we would still need to insert two copies. It would be an interesting experiment to determine if this algorithm would yield improvement over the one we implemented.

In Figure 3.5, we show an example that compares the implemented algorithm against this suggested improvement. In the best case, we save two instructions plus the load, and there should be enough opportunities to reduce the running time of the procedures in similar percentages to the reduction of memory operations we saw under the current algorithm.

$ST\ \langle SPILL \rangle\ r_1$	$ST\ \langle SPILL \rangle\ r_1$	$ST\ \langle SPILL \rangle\ r_1$
$\vdots$	$r_1 => r_2$	$\vdots$
$\cdots => r_1$	$\cdots => r_1$	$\cdots => r_2$
$\vdots$	$\vdots$	$\vdots$
$r_1 => \cdots$	$r_1 => \cdots$	$r_2 => \cdots$
$\vdots$	$r_2 => r_1$	$\vdots$
$LD\ \langle SPILL \rangle => r_1$	*(LD operation removed)*	*(LD operation removed)*
r_2 Is Unused In The Region	Store r_1 in r_2 Across The Region	Rewrite uses of r_1 In The Region

Figure 3.5 A comparison of the two spill-promotion algorithms

The problem with this idea is that it becomes more expensive to keep the live set up to date as we walk through the block(s) than under the current algorithm. Certainly, one way to manage it would be to recompute the `currently_live` set from the bottom of the block each time we modify the code, but this is potentially *very* expensive. Our intuition tells us that we could use a side data structure to record updates to the block, but it will require further work to get the implementation to be both time- and space- efficient.

We began this work attempting to find an alternative algorithm to Cooper and Simpson's live-range splitting that would offer some of the speedups of splitting without the expense of the large containment graph [21]. As a study in the opportunities for spill promotion, this work was a success, but we do not think that this idea would be a competitive alternative to the work presented by Cooper and Simpson.

In the next chapter, we examine an error in an assumption in Briggs' algorithm for inserting spill code. We will show that the solution can offer a large improvement in running time for very little effort.

Chapter 4

Improving Spill Code Insertion

4.1 Introduction

When a graph-coloring register allocator inserts spill code, it typically does something more complex than Chaitin *et al.*'s original idea of inserting a load before each use of the live range. Instead, the allocator performs a limited amount of local analysis to recognize when only a single load will service several consecutive uses. Chaitin pointed out that, when inserting spill code, there is no need to insert multiple loads before consecutive uses of the same variable if no other variables go dead between those uses [15].

Chaitin's formulation applies to whole live ranges only. In his thesis, Briggs generalized Chaitin's idea and allowed the register allocator to consider only a section of a live range contained within a single basic block [7]. Briggs describes three different opportunities to reduce the amount of spill code:

- two (or more) uses of a spilled value that are close together only require a load in front of the first use
- a use that follows closely a definition of that value does not require a reload
- if all uses are close to the definition of a value, the entire live range should not be spilled – such a live range is said to have *infinite* spill cost

The code to compute these cases is shown in Figure 4.1. There are two main sets, `need_load`, which tracks uses that require a load instruction, and `must_spill`, which identifies def-use combinations that have infinite spill cost. The second of these two is only important to computing spill costs, and we ignore it in the code and discussion. Of course, we also have the ubiquitous `currently_live` set tracking the live values as we move up through the block, as well as `spill_set`, a list of those values marked for spilling by the `select` phase of the allocator.

We iterate backwards through the list of the instructions in a block, keeping track of which registers are dead by watching when they become live – that is, when we see a use of a register that is not yet in the `currently_live` set, we know that that is the last use, or *death*, of the register. Whenever we find a death, we add loads for all of the members of the `need_load` set, clearing it afterwards.

```
for each block b in the CFG:
    clear need_load
    copy live_out of b into currently_live
    in a backwards walk of all instructions i in b:
        /* check for deaths */
        for all uses u in i:
            if u is not a member of currently_live:
                for all members m of need_load:
                    insert a load of m
                clear need_load
        /* update the various sets */
        for all defs d in i:
            delete d from need_load
            delete d from currently_live
        for all uses u in i:
            add u to currently_live
            if u is a member spill_set:
                add u to need_load
```

Figure 4.1 Briggs' algorithm for spill-code insertion

The intuition behind this is that deaths mark a lessening of register pressure. Alternatively, because we walk backwards through the code, deaths mark the point where register pressure *increases*. Thus, we can safely put off inserting any load until we see an increase in register pressure, as marked by a death.

4.2 The problem

Implicit in Briggs' formulation are the assumptions that the `need_load` set will empty its contents frequently and that there will be very few instructions in any sequence in which no computation becomes dead. In practice, this assumption usually holds. Occasionally, however, we have seen codes where values persist in the `need_load` set too long, and wasteful spilling occurs. Figure 4.2 shows an example of how this can happen.

Here, we assume that r_{31} has been marked for spilling (that is, it is a member of `spill_set`). During the backward walk, r_{31} gets added to `need_load` when we encounter the `ADD` instruction. We continue walking up through the block, but we do not find any deaths until we hit the top `MUL` instruction, when r_1 dies. So, following the algorithm, we empty the `need_load` set, inserting a load to r_{31} at this point.

The resultant code will work correctly. However, if the register allocator has to run through another iteration of spilling, the live range created by the new `LOAD` will not have infinite spill cost as we would expect, and so it is a candidate for further spilling.

r_1 dies here

```
MUL r1 r2 => r7
MUL r3 r4 => r8
MUL r5 r6 => r9
ADD r2 r31 => r10
⟨...uses of r2-r10...⟩
```

Code Before Spilling

load to r_{31} gets inserted here, but it's too far away

```
MUL r1 r2 => r7
LOAD <location> => r31
MUL r3 r4 => r8
MUL r5 r6 => r9
ADD r2 r31 => r10
⟨...uses of r2-r10...⟩
```

Code After Spilling

on the next iteration, the allocator re-spills r_{31}

```
MUL r1 r2 => r7
LOAD <location> => r31
STORE <newlocation> => r31
MUL r3 r4 => r8
MUL r5 r6 => r9
LOAD <newlocation> => r31
ADD r2 r31 => r10
⟨...uses of r2-r10...⟩
```

Code After the Next Iteration

Figure 4.2 An example showing values persisting in the `need_load` set too long

The problem arises when the load instructions inserted for the variables in the `need_load` set get too far away from the instructions in which the values are used. Under Briggs' formulation, when we finally detect a death, we empty the `need_load` set, inserting a load for every variable. Every time we spill a value, we create a number of small new live ranges to replace the original live range. Each of these live ranges ought to have infinite spill cost, because it does not make sense to spill a value that has already been spilled, as that second spilling will not be able to help the coloring of the graph. However, because the spill-code insertion algorithm can place loads of spilled values arbitrarily far from subsequent uses of those values, new live ranges are created that do not have infinite spill cost.

For example, a variable used immediately after it is defined has infinite spill cost. Spilling this value will not reduce register pressure. However, when we insert the loads for the `need_load` set, there is no guarantee that the new instructions will define their variables immediately before those variables are used.

One solution to this problem would be to keep track of all spill code as it is inserted and automatically give those values a spill cost of infinity. However, this would likely introduce the need for a new data structure, with the concomitant code necessary to maintain the data structure.

To correct this problem, we propose augmenting Briggs' algorithm with the following simple addition:

When we are emptying the `need_load` set: for each variable, v, removed from `need_load`, instead of inserting a load immediately following the current instruction, i, walk forward through the block from i, looking for the first use of v. When that use is found, insert the load of v immediately preceding the instruction containing that use. The new, short live range will have an infinite spill cost and so be barred from being spilled again.

Because a variable, v, is only added to the `need_load` set when an instruction is found that uses v, this forward walk must terminate. Since no instruction is examined more than twice during the walk – once backwards, and once going forwards while emptying `need_load` – the asymptotic complexity of the algorithm does not change.

4.3 Experimental results

As in all of the results reported in this thesis, we calculate the number of cycles required to run a routine by assuming that each memory operation takes two cycles, and all other operations take a single cycle.

In our test suite of 169 optimized procedures, only fifty-nine require the register allocator to insert spill code. When we applied the above algorithm to those fifty-nine codes, only six saw a reduction in the execution time compared to the code produced by the allocator without this improvement, and five of those six reported only a 1% decrease in execution time. However, the sixth of these cases, `jacu`, had an 8% decrease in execution time.

This improvement to Briggs' algorithm provides only minimal positive results and never produces negative results. It occasionally provides more substantial improvements. It is trivial to implement, and it does not noticeably affect the running time of the allocator. Note that all of the other experiments reported in this thesis were run with this bug-fix in place.

4.4 Future work

The improvement presented in this chapter addresses only the first of two problems that we found in spill-code insertion. While examining the output from the spill-promotion pass presented in the previous chapter, we noticed that Chaitin's notion of when to empty the `need_load` set (to use the model of Briggs' implementation) was overly conservative. We need only empty some portion of the `need_load` set if register pressure demands it. Remember that our intuitive justification for inserting a pending load of a spilled value was that deaths caused a lessening of register pressure, ensuring that we could move the load all the way backwards to that point.

However, we can be more certain if we simply use the `currently_live` set while we perform the backward walk. This allows us to track the actual register pressure, so we can move the load backwards through the block until the *total pressure* forces us to insert the load. This should have the effect of reducing the ratio of loads/uses for any particular value, and our experiments suggest that it does.

We performed a "proof of concept" experiment by using the spill-promotion code from the previous section. In that algorithm, we matched loads from a spill location to the store to the same location, monitoring the register pressure in the region between the two. In this augmented algorithm, we also looked for two loads from the same spill location and tried to promote the second of these. Note that these numbers were generated using the extended basic block algorithm.

As Table 4.1 shows, there are opportunities for improvement if we remove the superfluous load operations caused by Chaitin's conservativism. The last column of Table 4.1 shows the total runtime improvement (or degradation, as in the case of `efill`), while the number in parentheses is the ratio of cycles devoted to memory operations.

However, the correct means of handling the superfluous-loads problem is to prohibit the allocator from inserting unnecessary spill code during allocation, not to add the ability to remove the loads to our algorithm from the previous chapter. This has subtle implications in computing spill costs (and, especially, computing infinite spill costs), and we are still exploring them.

Routine Name	Original Memory Cycles	Memory Cycles After Promotion	$\frac{Improved}{Original}$
twldrv	60,675,108	56,833,000	0.94
fpppp	37,234,728	31,804,164	0.85
smooth	21,443,202	21,162,962	0.99
field	6,920,902	6,879,192	0.99
radf4	3,651,250	3,181,350	0.87
radb4	3,632,200	3,168,650	0.87
efill	2,570,138	2,333,902	0.91
radf5	2,159,000	1,778,000	0.82
radb2	2,057,400	1,835,150	0.89
radf2	2,019,300	1,809,750	0.90
radb5	1,905,000	1,619,250	0.85
supp	744,464	729,232	0.98
subb	547,680	527,520	0.96
ddeflu	25,4760	242,508	0.95
saturr	17,360	17,248	0.99
svd	1740	1,724	0.99
decomp	252	244	0.97
rffti1	188	184	0.98

Table 4.1 Results of spill promotion that include load-load promotions

In the next chapter, we will look at some theoretical benefits of cleaning up the spill code that the allocator inserts.

Chapter 5

Coloring Spill Memory

5.1 Introduction

Once the allocator has finished, we may still be able to improve the resulting allocation.

If multiple values are spilled, either on the same iteration or on different iterations of the allocator, Briggs' algorithm creates new spill locations for each value. It does not attempt to determine if the next value to be spilled can be spilled into a spill address that has already been used.

When the register allocator decides to spill two values, it might not be able to tell if the same spill location can be used to hold both values. If the two values interfere, then they obviously cannot be spilled to the same location. But remember that the allocation process requires multiple iterations of building the interference graph. Any spill locations used by a previous iteration cannot be reused under the algorithm supplied by Briggs, since spill locations are not built into the interference graph.

This problem of finding the most efficient use of spill memory is referred to as *memory allocation* by Briggs [7]. He points out that a number of factors have reduced interest in optimizing memory allocation, including ubiquitous large memories, separate compilation, and stack-based allocation, which recycles memory in a natural way. However, more emphasis is now being given to compiling for DSP chips. These chips often have concerns not found in commodity microprocessors which require readdressing the problem of memory allocation, such as limited memory environments which support, for example, auto-increment/auto-decrement addressing modes [29].

As we have said, the majority of the memory accesses cannot be analyzed by the register allocator, but the spill code that the allocator itself inserts is an exception. We can perform some analysis and optimization on the memory used for spill code.

We can think of a spill location as its own live range, with the value stored in memory instead of a register. When the interference graph is built, we can take into account these new live ranges as well, and when choosing a spill location, see if the value to be spilled interferes with any of these spill locations. If not, we can reuse a spill location for the spilled value.

We propose, however, compressing the spill memory in a separate pass after the register allocator has completed. The reason is twofold. First, the graph-coloring register allocator is an extremely complicated piece of code already, made so by the many different names that a value can take on during the course of allocation. Adding yet another level of complexity would be overly burdensome. Second, we show in our study of interference graphs [20] that any increase in the number of variables used to build the interference graph results in a nonlinear increase in the size of the interference graph and the time required to manipulate it, so it pays to move this analysis to a pass separate from the allocator itself. Note that we will only be changing the addresses in the load and store operations used for spilling, not the placement of those operations, so a post-pass will not change any assumptions made by the allocator.

The algorithm for compressing spill memory is analogous to the algorithm for allocating registers. First, each memory location is its own live range, and the interference graph is built using these as nodes. Edges between nodes, as for register values, indicate when two memory locations are in use at the same time. We then color the graph, although we are not limited to a finite number of memory locations. Instead of reducing the graph by pulling out any node with degree less than k, we push onto the coloring stack all nodes with the lowest current degree. This continues until all nodes have been removed. We then assign colors – in this case, memory locations – to the nodes. Each time we find a node for which there are no available colors, we increase the number of colors by one and assign that node the new color.

5.2 Experimental results

The register allocator inserts spill code into fifty-nine of the 169 routines in our test suite. Of these, coloring the spill memory locations as described results in a reduction of spill memory required in thirty-seven of those routines. Table 5.1 shows the amount of memory reduction. Each entry shows the number of bytes of spill memory required before and after compaction and the percentage difference between the compacted and uncompacted memory requirements. Overall, approximately 75% of the routines require less than one kilobyte of memory to hold the spills.

Routine Name	*Bytes Original*	*Bytes Final*	$\frac{Final}{Original}$	*Routine Name*	*Bytes Original*	*Bytes Final*	$\frac{Final}{Original}$
twldrv.i	12,024	10,376	0.86	saturr.i	976	520	0.53
fpppp.i	9,712	2,976	0.31	radb3.i	928	456	0.49
deseco.i	6,536	5,824	0.89	radf3.i	912	456	0.50
erhs.i	4,512	4,136	0.92	smooth.i	760	432	0.57
field.i	3,856	1,416	0.37	advbnd.i	736	528	0.72
jacld.i	3,728	3,536	0.95	radb2.i	688	376	0.55
rhs.i	3,544	2,888	0.81	ddeflu.i	688	496	0.72
parmvr.i	2,656	2,472	0.93	radf2.i	680	368	0.54
jacu.i	2,368	2,328	0.98	vslv1p.i	640	368	0.57
radbg.i	2,296	976	0.43	vslv1x.i	488	344	0.70
radfg.i	2,112	744	0.35	efill.i	480	472	0.98
supp.i	1,584	824	0.52	colbur.i	432	416	0.96
radb5.i	1,472	704	0.48	svd.i	408	304	0.75
radf5.i	1,456	696	0.48	tomcatv.i	376	368	0.98
radf4.i	1,328	640	0.48	dyeh.i	360	184	0.51
radb4.i	1,320	648	0.49	getb.i	288	256	0.89
subb.i	1,312	672	0.51	putb.i	272	240	.88
parmov.i	1,168	1,024	0.88	parmve.i	264	200	0.76
				cosqf1.i	232	224	0.97
TOTAL	73,592	49,888	0.68				

Table 5.1 Spill Memory Requirements and Compaction

5.3 Future work

This procedure is useful not only for the study of how much spill memory is actually being used (which is a result we will use in the next chapter), but also because any machine with short and long addressing modes might be able to increase the number or effect of short spill instructions – one might want to put more expensive spills in nearby memory and less expensive ones farther away, for example. In such machines, it is often the case that the instructions that access nearby memory are smaller [43]. Thus, to reduce code size, we would want to put the highest frequency spill or load from a particular address in nearby memory. Alternatively, some architectures offer a smaller, more-efficient memory operation that relies on consecutive memory accesses [29, 30]. In this case, we might want to try to group geographically close spill instructions in adjacent memory locations, perhaps by marking these nearby operations as *partners*, similar to Briggs *et al.*'s notion [11]. Even on chips that rely on a hardware-controlled cache, this might have interesting benefits, by allowing us to pack multiple spilled values into the same cache line and so reduce the number of conflicts.

Chapter 6

Compiler-Controlled Memory

Optimizations aimed at reducing the impact of memory operations on execution speed have long concentrated on improving cache performance. These efforts achieve limited success for a number of reasons, not the least of which is that the compiler has imperfect control over the contents of the cache. This is due to two factors: first, the hardware controls the cache, not the compiler, and, second, the compiler cannot always determine how memory will be accessed by the code it is compiling.

There is an exception to this rule. During the register allocation phase, the compiler must often insert substantial amounts of *spill* code; that is, instructions that move values from registers to memory and back again. Because the compiler itself inserts these memory instructions, it has more knowledge about them than other memory operations in the program.

Spill-code operations are disjoint from the memory manipulations required by the workings of the program being compiled, and, indeed, the two interfere in the cache. In this chapter, we propose a hardware solution to the problem of increased spill costs—a small *compiler-controlled memory* (CCM) to handle spilled values. We suggest working with chip designers to include a direct-mapped memory space under the exclusive control of the compiler. This memory space would be used to service the memory traffic inserted by the register allocation phase of the compiler. The instructions to access it are quite simple; they can easily fit within most instruction sets. Of course, some chips, notably DSP's, already have small, disjoint memories that can be used for this purpose; our algorithms would be immediately applicable for these architectures.

We will present two compiler algorithms to show how to exploit this memory, and we will show that the speedups achieved are sizable.

6.1 Introduction

To an outside observer, the DSP microprocessor market appears to be a breeding ground for architectural innovations. These machines change quite rapidly; new features appear with each generation. The best of these features persist; some even make it into commodity microprocessors.

Some DSP chips have a small, fast, on-chip memory that the programmer can use to improve access times. These on-chip memories are not caches; instead, they are located in a disjoint address space. This simplifies their implementation by eliminating any need for relating on-chip memory addresses to off-chip memory addresses, for associative lookup, and for automatic replacement. Instead, the designers make the programmer responsible for moving data between main memory and the on-chip memory in a timely and efficient way.

In this chapter, we present an alternative use for a small portion of this on-chip memory—as a holding place for spilled values, CCM. Our scheme has several advantages. Using CCM for spills should shorten spill latencies and let the scheduler place the load for a spilled value next to its use—speeding up execution and shortening the live range created for the spilled value. Using CCM for spills should reduce the required bandwidth between the compiler and main memory. If the system also has a cache memory, spilling to CCM should eliminate cache pollution introduced by spill operations—loads and stores that can interfere directly with the cache behavior "planned" by high-level, compiler-based transformations that exploit locality caused by regular accesses in loop nests [14, 44, 19].

We should reiterate that this work is narrowly focussed on spill-code insertion during register allocation. It has similarities to the idea of register promotion [31], and, indeed, such techniques would benefit from using CCM as well. Because of the similarity in ideas, we have chosen to call the following methods *spill promotion* – in both cases, we are taking a value from slow memory and moving it into faster memory.

While we originally conceived of this scheme as a way to make automatic use of some portion of the CCM on a DSP chip, our experiments suggest that the improvements may be large enough to justify adding a small CCM to a commodity microprocessor. To assess the reasonableness of this idea, we need to address three questions.

1. What kind of hardware support would be required?

2. How would the register allocator manage the CCM?
3. How large would the CCM need to be?

This rest of this chapter presents our vision for CCM. Section 6.2 describes our assumptions about the underlying hardware, discusses why register spilling is problematic, and suggests why a CCM is a better solution than simply using the existing cache memory. Section 6.3 describes how to build a stand-alone CCM-allocator that runs as a post-pass to the compiler, and how to fit a CCM scheme into the spill-code insertion phase of a Briggs allocator. Section 6.4 presents experimental results that show the kind of improvements that should be expected from the use of CCM and the amount of CCM required by programs in our test suite.

Finally, we must reiterate that our techniques should be directly applicable on some current DSP chips. These microprocessors and microcontrollers already have on-chip memories that are not caches; these systems expect the applications programmer to explicitly manage transfers into and out of these small, fast memories. On such systems, our results suggest a style of programming where the application programmer cedes the bottom 1 KB of on-chip memory to the compiler, which uses it to implement CCM-style spilling.[3]

6.2 Background

In this section, we look at the hardware requirements and some motivations for using the CCM. From the point of view of the compiler writer, these topics are interrelated; not only have we identified a problem that we can solve, but we also have to be able to show that the impact on the architecture is within limits of feasibility. Only in the context of these two aspects of the problem will our software solution have meaning.

6.2.1 Hardware requirements

Our scheme has modest hardware requirements.[4] The primary requirement is a small on-chip memory, the CCM. Conceptually, the CCM should sit in its own address space, accessible through special instructions that move data between the CCM and the register set. Access to CCM should be fast, with the results available for use

[3]These chips often have a single, dedicated process, so context-switching is not an issue.

[4]According to Upton *et al.*, a 1 KB cache is negligible [42].

on the next cycle. For each class of registers (*e.g.*, floating-point registers or general purpose registers), a pair of instructions

$$\texttt{spill}\ \langle \mathit{offset} \rangle,\ r_i \qquad \mathrm{CCM}[\langle \mathit{offset}]\ \Rightarrow\ r_i$$

$$\texttt{restore}\ r_j,\ \langle \mathit{offset} \rangle \qquad r_j\ \Rightarrow\ \mathrm{CCM}[\langle \text{offset} \rangle]$$

will suffice. Other applications for CCM may find uses for more complex addressing modes. For our purposes, these absolute addresses are not only sufficient, but desirable.

If the compiler can assume that the system runs only one process, the CCM can be simple and small—on the order of 1 KB. In a multi-tasked environment, the CCM should be larger and slightly more complex. To handle multiple processes, we would want to add a system-controlled base register to provide each process with its own small region within CCM. This would allow the system to avoid copying CCM contents to main memory on context switches. Still, we expect that a CCM of 16 KB to 32 KB would be more than adequate.

6.2.2 The impact of spilling on the cache

A modern microprocessor presents the compiler with a complex set of challenges. To achieve a reasonably large fraction of the processor's peak performance, the compiler must both keep multiple, pipelined functional units busy, and arrange memory accesses in an order that creates good cache locality.

- To keep the pipelines busy, the compiler must ensure that an instruction is ready to execute on each functional unit at each cycle. This requires careful instruction scheduling. It also requires that the operands of each instruction be available, in registers, at the start of the appropriate cycle.

- To improve cache locality, the compiler must reorder and rearrange loop iterations. The "heroic" transformations that "block" loops for locality introduce new overhead computations; they also rewrite some memory references into register references.

Both problems increase the demand for registers and provoke the register allocator to spill more values to memory.

Register spills are problematic for two reasons. First, they add load, stores, and address computations to the program, each of which must be scheduled, fetched, and executed. Second, they perturb the behavior of the data cache(s) and increase memory bandwidth requirements. Both effects are complicated by the fact that allocation occurs quite late in compilation; the transformations that block for cache and register locality cannot see the spills to account for their impact.

6.2.3 Why not just use the cache for spilling?

Most modern microprocessors use cache memories as tool to bridge the expanding gap between the speed of main memory and the speed of the processor [46]. At any point in time, a processor's cache holds a mapped subset of the address space that includes main memory. Hardware mechanisms adjust both the contents of cache and the mapping between cache locations and main memory locations as programs execute [26]. Cache systems combined with transformations developed by the compiler community have been reasonably effective at bridging the gap between memory speed and processor speed [33]. Unfortunately, spill code inserted in the last stages of compilation can disrupt the compiler's carefully planned sequence of memory accesses.

From a performance perspective, the cache's importance lies in one fact: a memory reference to an element already in the cache takes much less time than a reference to an element not in the cache. A reference that "hits" the cache typically completes in a single cycle, while a reference that "misses" takes five to ten cycles on a uniprocessor machine, and as long as hundreds of cycles in a distributed memory multiprocessor [28, 36, 32, 2, 4]. This difference in access time has a strong impact on the performance of individual programs. Accordingly, much recent research in compilation has been directed at techniques that improve the likelihood of references hitting in the cache. Most of this work falls into two major categories.[5]

Blocking These transformations rearrange and reorder the iterations of a loop nest in an attempt to move multiple references to a single location closer in time (*temporal locality*) or they attempt to move references to adjacent memory locations closer in time (*spatial locality*) [44, 14, 45].

[5] A third approach, *streaming* was used on the *i*860. Here, the compiler would load values from memory to registers, bypassing the cache, and then write them into an array that modeled the cache. This gave the compiler fairly precise control over the contents of cache by largely avoiding instructions that could cause a replacement. The control came at the expense of doubling the amount of data movement. [34].

Prefetching These transformations require hardware support in the form of an *advisory prefetch instruction* [13, 26]. The compiler inserts a prefetch instruction well in advance of a memory reference that it believes will miss the cache, causing the hardware to pre-load the location into cache [35, 13].

McKinley and Temam studied the actual memory behavior of a suite of scientific programs [33]. They discovered that blocking and prefetching are effective in reducing the running time of programs; at the same time, they note that there remains room for improvement.

In McKinley and Temam's study, nearly twenty-five percent of all memory references are scalar, rather than array, values. In both blocking and prefetching, the compiler analyzes the program and tries to predict the run-time behavior of the cache. The compiler bases its transformations on those predictions. The analysis and prediction techniques rely on an implicit assumption that all accesses are exposed to the analysis; they also tend to focus on repeated references to arrays. When scalar references conflict with the cache behavior "planned" by the compiler in either blocking or prefetching, they reduce the impact of those transformations.

While McKinley and Temam do not report the percentage of references that are inserted by the register allocator, we do know that every spill is a scalar reference. Other papers have addressed the issue of reducing programmer-inserted scalar references [14, 17]. In this chapter, we address the issue of reducing compiler-inserted scalar references. In some ways, these compiler-inserted references are more insidious: since spills are inserted in the final stages of compilation, the earlier transformations cannot see them, analyze them, plan for them, or eliminate them.

Our proposal for a small, fast, compiler-controlled memory would eliminate much of the unplanned disruption caused by spill code. The following sections focus on the software support needed to utilize the CCM for spilling. We present two different ways that a compiler-writer could handle CCM spilling and present experimental evidence that suggests the effect of adopting this collaborative hardware/software scheme.

6.3 Software implementation

The software support required to use CCM for spilling can take several forms. To explore our ideas, we built two different implementations: a post-pass CCM allocator that operates after "normal" register allocation, and a register allocator that directly generates CCM spills. As we will see in Section 6.4, the two approaches produce different results.

6.3.1 A post-pass CCM allocator

In the post-pass CCM allocator, the output of the register allocator is fed into the CCM allocator. The CCM allocator is a simplified Briggs allocator [7] with a few significant changes.

The CCM allocator focuses on spill locations rather than data values in registers. It runs after register allocation, so all decisions about which live ranges to keep in registers have already been finalized. Instead, the task of the CCM allocator is to discover some subset of the spilled values that can be safely and profitably relocated to CCM. Thus, instead of building SSA form for values as is typical, the CCM allocator builds an SSA form for the memory locations that hold spills. It uses addresses as symbolic names and builds the analog of live-range names from that base set of symbolic names.

In this spill-location allocator, the notion of "liveness" changes. A spill location, m, is "live" at some point, p, in a program if there exists an execution path from p to an instruction that loads from m. Thus, m is live at p if it might be loaded again after p executes. A memory location m is "defined" when the program stores a value into m. It is "used" when the program loads a value from m. The CCM allocator uses these new definitions to compute the corresponding analog of liveness information for spill locations. The liveness information is used, in turn, to build an interference graph that shows when two spill locations can and cannot share a single memory location (in main memory or in CCM).

The CCM allocator does not generate new spills. Rather, it redirects some subset of the existing spills into a size-limited CCM. When it discovers that the spills for a particular value will not fit into CCM, it simply leaves the original spill code intact. This causes the spills for that value to remain in main memory, producing a heavyweight spill rather than a CCM spill. The result is conservative, but safe.

Traditional register allocators are intraprocedural in nature—that is, they treat individual procedures as independent entities. When a traditional register allocator allocates memory to hold a spilled value, it typically places that spilled value in the activation record of the active procedure. Thus, allocation decisions for spill locations can be made independently in different procedures. With CCM, the situation changes.

Since CCM is a global resource, shared across the whole program, the CCM allocator must adopt some interprocedural conventions on the use of CCM.

Two distinct strategies make sense. First, the allocator can avoid the problem by limiting CCM spilling to values that are not live across any call site. These values will never be in CCM when another procedure is active, so their use of CCM cannot conflict with any other procedure. Second, the allocator can coordinate the use of CCM across procedures to avoid potential conflicts. This requires information about the use of CCM by other procedures in the program; this strategy makes the most sense in a post-pass CCM allocator that runs at link-time.

Our implementation of the post-pass CCM allocator can use either strategy. In the absence of interprocedural information, it adopts the conservative strategy and allocates only values that are not live across calls to CCM spill locations. If interprocedural information is available, it performs allocation in a bottom-up walk over the call graph. That is, it processes all routines reachable from procedure p before considering p. After it processes a procedure q, it records, for each call to q, the amount of CCM that q uses. When allocating for p, the CCM allocator can then use any location for a value that is not live across the call, but must use a location higher than q's high-water mark for values that are live across the call to q.

If the call graph contains cycles, corresponding to recursion, the CCM allocator behaves conservatively. It marks each procedure in the cycle as using the full CCM.

The algorithm is shown in Figure 6.1. After building the supporting data-flow analyses such as liveness and SSA for the spill memory positions, we build the interference graph. We perform coloring in a similar way to register allocation, except that when the graph contains only nodes of high degree, we simply remove the cheapest from the graph, allowing it to remain as a heavyweight spill instruction, and proceed.[6] Note that, unlike the register allocator, we only need one pass of coloring for each subroutine.

To select a specific location in CCM, the post-pass allocator uses the same algorithm that the Briggs allocator uses to pick a spill location in main memory. It starts at the beginning of CCM and tries successive locations until it finds one that will work—that is, a location not used by any interference-graph neighbor of the spilled value. Here, the "beginning" of this search space is the maximum of the CCM usage

[6]Any remaining heavyweight spill instructions should have their spill locations updated so that they are packed tightly together and, as a result, use the least memory necessary; indeed, the interference graph already built can be used to compact the memory.

of the set of subroutines across which the spilled value is live. We manage this issue by creating an array of integers that corresponds to spill locations; a pass over the code uses the live sets at subroutine calls to compute the appropriate "beginning" address for each spill location.

6.3.2 CCM allocation during spill-code insertion

To incorporate CCM spilling into the "normal" register allocator, we must make the CCM locations visible in the allocation process. The allocator assigns an abstract name and extends its data-flow analysis to include the CCM names. Thus, CCM locations appear in the interference graph alongside live ranges of program values. On the initial pass through the register allocator, the CCM locations have no interferences. At the end of the first allocation pass, the act of inserting spill code that uses CCM locations will create spans over which they are live. This, in turn, forces edges between CCM locations and live ranges into the interference graph.

The allocator ignores these edges during allocation and uses them during spill code insertion. When it goes to spill a value, edges between the node and CCM locations

---

Calculate the call graph (if necessary) [38, 12]
Conservatively mark subroutines in call-graph cycles as using all of CCM
For each subroutine, s, in a postorder walk over the call graph:
    Rewrite all spill instructions to use symbolic names
    Build liveness analysis for spill locations
    Build SSA on the spill locations
    Build live-range names
    Build the interference graph
    Calculate the cost of each live range
    Color CCM locations for each live range
    Rewrite spill instructions to spill to CCM
    Record the amount of CCM used by s

Figure 6.1 Algorithm for post-register-allocation spill promotion show the allocator which CCM locations cannot hold the spilled value. This simplifies the search for an appropriate CCM location.[7]

Adding a name space for CCM locations was easy. Managing name spaces is one of the more difficult parts of building a Briggs allocator. By comparison to the problems introduced by building SSA, forming live ranges, coalescing live ranges, and coloring the result into a small space of register and spill location names, the CCM names are simple.

1. CCM names are introduced after live ranges have been discovered. The allocator appends CCM names onto the set of live ranges. This occurs just before construction of the interference graph.

2. Interference graph construction treats CCM location names in the way that it handles register values. It uses the definition of liveness cited in the last section: a CCM location becomes live when it is stored to, and it remains live until the last load from that position. With that definition, the only real change to the interference-graph building algorithm is insertion of code that recognizes spills to CCM. CCM locations are otherwise treated exactly as register values.

3. After building the interference graph, CCM locations are ignored until the *spill* section of the allocator. Each time a value is marked for spilling under normal register allocation, we assign it to a spill location, usually based on the value of the stack pointer. We modify this by inserting a check to see if the value can be assigned to the CCM instead of being put on the stack.

4. During spill insertion, the allocator consults the interference graph to determine whether or not a suitable CCM location is available. The governing rule is: a value v cannot be spilled to CCM position m if an edge from v to m is in the interference graph. As values are spilled to CCM, the interference graph must be updated. Spilling v to CCM location m requires that we add an edge from m to each live-range neighbor of v in the interference graph.[8]

[7] Our allocator does not color spills to main memory, so the equivalent edges are not present for main memory spill locations. If the allocator runs out of spill locations in main memory, it simply extends the activation record for the current procedure. Since CCM is a fixed-size resource, the more expensive approach is warranted.

[8] Alternatively, the allocator can keep a side data structure of the CCM spills inserted in the current round of spilling. Then, the rule becomes, v cannot be spilled to m if $\langle v, m \rangle$ is in the interference

The expanded spilling algorithm is shown in Figure 6.2. This is the algorithm provided by Briggs [7]; we embolden each step that requires modification.

6.4 Experimental results

We have purposely avoided a key specification of our work up to this point in order to discuss the utility of CCM. That specification is the actual size of the memory area. In Chapter 5, we saw that fully seventy-five percent of the routines required less than a kilobyte of memory, so this seemed to be a good target for our experiments.[9] We also ran the experiments with half that amount of memory, to see if key spill promotions would be responsible for the majority of the speedup. This gives us two data points. First, it shows the relative speedup and the cost/benefit of having more CCM. Second, it shows that even with a CCM of 512K, significant speedups can be achieved.

The results are shown in Figures 6.1 and 6.2. In the first column, we show the dynamic execution costs of running each routine. The numbers in parentheses are the number of cycles used for memory operations. In the second column, we show the

---

Loop until no new spill code is added:
- **Build SSA Form (include** CCM **positions)**
- Build live-range names
- Repeat until no more coalescing possible
  - **Build the interference graph (include** CCM **positions)**
  - Coalesce copies
- Calculate spill costs
- Simplify
- **Select**
- **Spill (try to spill into** CCM **positions)**

Figure 6.2 Modified register-allocation algorithm including spill promotion

---

graph, or there is an edge $\langle v, p\rangle$ in the graph and p has already been spilled to m in this round of spilling.

[9] Note that it is the compacted values with which we are concerned, since the spill-promotion algorithms compact memory into the CCM.

relative speedup of the code after running the separate memory allocator; again, the number in parentheses is the ratio of cycles spent just in memory operations. The third column shows the relative speedup of the memory allocator that uses interprocedural information. The fourth column shows the relative speedup when running the register allocator with CCM allocation built into the spill-code generator.

Note that the post-pass can go after the scheduler (or be built into the scheduler), so that more expensive loads and stores can be given greater priority. This is an idea out of Briggs' thesis, wherein he states that the cost of loads/stores is difficult to estimate because of latency-hiding introduced by the scheduler [7].

6.5 Summary and conclusions

In this chapter, we proposed the creation of a small, fast, on-chip memory that the compiler could use for scalar memory operations inserted by the register allocator, and we showed the minor software enhancements necessary to utilize this memory space. Our proposal offers the opportunity to move scalar spill operations out of the address space containing both cache and main memory. This eliminates cache pollution introduced by spill instructions and reduces total traffic through the cache.

The hardware requirements for our CCM are simple; the amount of CCM required for real programs is modest. We showed how to incorporate the necessary compiler support in two ways: as a post-pass CCM allocator that could run at the end of compilation or at link-time, or as an integrated part of a Briggs allocator.

Using a small amount of CCM (512 to 1024 bytes) produced significant decreases in the run-time cost of register spilling. Because the amount of memory per program is so small, it makes sense to consider implementing future commodity microprocessors with a CCM with 16KB to 32KB of memory and adding a simple mechanism for context-switching it without copying it back to main memory.

This idea may find immediate application on some of the current generation DSP chips. On DSP chips with a small local memory, reserving the bottom 512 to 1024 bytes of that memory would allow the compiler to apply the techniques presented here. Many of these chips are used in applications that run a single process; in that environment, context switching is not an issue.

Routine Name	Cycle Count Without CCM	Post-Pass	Post-Pass w/ Call Graph	Integrated
decomp	825(252)	0.97(0.90)	0.96(0.88)	0.97(0.90)
svd	5,375(1,740)	0.98(0.93)	0.98(0.93)	0.98(0.93)
saturr	26,432(17,360)	0.90(0.84)	0.85(0.77)	0.90(0.84)
subb	959,280(546,000)	0.92(0.86)	0.84(0.72)	0.84(0.72)
supp	1,288,056(744,464)	0.93(0.88)	0.86(0.76)	0.86(0.77)
colbur	321,659(135,408)	0.99(0.98)	0.99(0.98)	0.99(0.98)
debflu	328,160(153,888)	1.00(1.00)	1.00(0.99)	1.00(1.00)
bilan	194,218(86,790)	1.00(0.99)	1.00(0.99)	1.00(0.99)
ddeflu	579,443(254,760)	0.99(0.98)	0.92(0.82)	0.99(0.98)
deseco	970,072(470,140)	0.97(0.94)	0.96(0.92)	0.97(0.94)
pastem	223,961(53,308)	1.00(0.99)	1.00(0.99)	1.00(0.99)
efill	4,705,095(2,570,138)	0.78(0.59)	0.78(0.59)	0.78(0.59)
fpppp	57,782,160(37,234,728)	0.95(0.92)	0.89(0.83)	0.91(0.86)
twldrv	111,128,482(60,658,016)	0.93(0.88)	0.91(0.83)	0.91(0.83)
setiv	236,731(75,692)	0.99(0.96)	0.99(0.96)	0.99(0.96)
erhs	2,873,175(1,223,036)	0.99(0.98)	0.99(0.98)	0.99(0.98)
rhs	78,222,321(38,056,812)	0.98(0.97)	0.98(0.97)	0.98(0.97)
jacld	58,617,550(34,801,900)	0.95(0.92)	0.90(0.83)	0.96(0.94)
blts	108,885,800(45,403,600)	0.99(0.98)	0.99(0.98)	1.00(0.98)
jacu	40,553,050(23,788,900)	0.93(0.88)	0.86(0.76)	0.93(0.88)
buts	109,578,600(46,621,300)	1.00(0.99)	1.00(0.99)	1.00(0.99)
denpt	5,940,008(2,070,008)	0.98(0.96)	0.98(0.96)	0.98(0.96)
parmvr	261,706,085(152,279,892)	0.99(0.99)	0.99(0.99)	0.99(0.99)
smooth	33,683,283(21,466,402)	0.97(0.95)	0.97(0.95)	0.97(0.95)
rffti1	712(188)	0.97(0.87)	0.97(0.87)	0.97(0.87)
radf5	3,803,650(2,159,000)	0.99(0.98)	0.86(0.76)	0.86(0.76)
radf4	5,718,175(3,613,150)	0.84(0.74)	0.79(0.67)	0.79(0.67)
radf2	3,362,325(2,000,250)	0.85(0.74)	0.85(0.74)	0.85(0.74)
vslv1p	21,803,385(15,174,540)	0.97(0.96)	0.97(0.96)	0.97(0.96)
radb2	3,406,775(2,038,350)	0.84(0.74)	0.84(0.74)	0.85(0.74)
radb4	5,689,600(3,632,200)	0.84(0.75)	0.79(0.67)	0.79(0.67)
radb5	3,667,125(1,905,000)	0.99(0.98)	0.89(0.80)	0.89(0.80)
slv2xy	49,928(35,492)	1.00(1.00)	0.99(0.99)	1.00(1.00)
field	12,172,573(6,821,302)	0.93(0.88)	0.93(0.87)	0.93(0.87)
init	8,877,019(3,071,802)	1.00(1.00)	0.99(0.98)	1.00(1.00)

Table 6.1 A comparison of dynamic cycle counts with 512-byte CCM

Routine Name	Cycle Count Without CCM	Post-Pass	Post-Pass w/ Call Graph	Integrated
decomp	825(252)	0.97(0.90)	0.96(0.88)	0.97(0.90)
svd	5,375(1,740)	0.98(0.93)	0.98(0.93)	0.98(0.93)
saturr	26,432(17,360)	0.90(0.84)	0.85(0.77)	0.90(0.84)
subb	959,280(546,000)	0.84(0.72)	0.80(0.65)	0.80(0.65)
supp	1,288,056(744,464)	0.86(0.77)	0.80(0.66)	0.80(0.66)
prophy	378,156(209,264)	1.00(1.00)	0.90(0.81)	1.00(1.00)
colbur	321,659(135,408)	0.99(0.98)	0.99(0.98)	0.99(0.98)
debflu	328,160(153,888)	1.00(1.00)	1.00(0.99)	1.00(1.00)
bilan	194,218(86,790)	1.00(0.99)	1.00(0.99)	1.00(0.99)
ddeflu	579,443(254,760)	0.99(0.98)	0.92(0.82)	0.99(0.98)
deseco	970,072(470,140)	0.97(0.94)	0.96(0.92)	0.97(0.94)
pastem	223,961(53,308)	1.00(0.99)	1.00(0.99)	1.00(0.99)
efill	4,705,095(2,570,138)	0.78(0.59)	0.78(0.59)	0.78(0.59)
fpppp	57,782,160(37,234,728)	0.92(0.87)	0.82(0.72)	0.86(0.79)
twldrv	111,128,482(60,658,016)	0.91(0.83)	0.91(0.83)	0.91(0.83)
setiv	236,731(75,692)	0.99(0.96)	0.99(0.96)	0.99(0.96)
erhs	2,873,175(1,223,036)	0.99(0.98)	0.99(0.98)	0.99(0.98)
rhs	78,222,321(38,056,812)	0.98(0.97)	0.98(0.97)	0.98(0.97)
jacld	58,617,550(34,801,900)	0.91(0.85)	0.82(0.69)	0.87(0.78)
blts	108,885,800(45,403,600)	0.99(0.98)	0.99(0.98)	1.00(0.98)
jacu	40,553,050(23,788,900)	0.86(0.76)	0.82(0.69)	0.82(0.69)
buts	109,578,600(46,621,300)	1.00(0.99)	1.00(0.99)	1.00(0.99)
denpt	5,940,008(2,070,008)	0.98(0.96)	0.98(0.96)	0.98(0.96)
parmvr	261,706,085(152,279,892)	0.99(0.99)	0.99(0.99)	0.99(0.99)
smooth	33,683,283(21,466,402)	0.97(0.95)	0.97(0.95)	0.97(0.95)
rffti1	712(188)	0.97(0.87)	0.97(0.87)	0.97(0.87)
radf5	3,803,650(2,159,000)	0.86(0.76)	0.86(0.76)	0.86(0.76)
radf4	5,718,175(3,613,150)	0.79(0.67)	0.79(0.67)	0.79(0.67)
radf2	3,362,325(2,000,250)	0.85(0.74)	0.85(0.74)	0.85(0.74)
vslv1p	21,803,385(15,174,540)	0.97(0.96)	0.97(0.96)	0.97(0.96)
radb2	3,406,775(2,038,350)	0.84(0.74)	0.84(0.74)	0.85(0.74)
radb4	5,689,600(3,632,200)	0.79(0.67)	0.79(0.67)	0.79(0.67)
radb5	3,667,125(1,905,000)	0.89(0.80)	0.89(0.80)	0.89(0.80)
slv2xy	49,928(35,492)	1.00(1.00)	0.99(0.99)	1.00(1.00)
field	12,172,573(6,821,302)	0.93(0.87)	0.93(0.87)	0.93(0.87)
init	8,877,019(3,071,802)	1.00(1.00)	0.99(0.98)	1.00(1.00)

Table 6.2 A comparison of dynamic cycle counts with 1024-byte CCM

Chapter 7

Summary and Conclusions

This thesis has explored the problem of spill-code-impact minimization.

In Chapter 3, we proposed simplifying the expensive analysis of Cooper and Simpson by looking for unproductive spills *after* they occurred, instead of beforehand [21]. While we managed to reduce the number of memory operations executed, the cleanup code we needed to insert negated most of the time savings, resulting in negligible overall improvements. However, we did show improvements of up to 6% in the number of cycles spent on memory operations, which could be a significant impact on codes that exert high pressure on the memory subsystem.

In Chapter 4, we presented a simple correction to Briggs' spill-code-insertion algorithm for a problem that has heretofore gone undetected. This correction requires little effort to implement, has an insignificant impact on the running time of the allocator and led to a 8% speedup on some codes.

In Chapter 5, we presented a study of the memory used for spill code. We looked at both how much memory is used in spill instructions directly from the allocator and at how much this memory can be compressed. We showed that after compression, approximately 75% of the routines in our test suite required less than one kilobyte of memory to hold all of the spill values. We used this study to help justify our suggestion for CCM, in Chapter 6. We also presented some theoretical ideas on how better to organize the inevitable spill-code memory.

We showed that a separate memory we call CCM could reduce the runtime impact of spill code. We presented three variations of algorithms to use this hardware device, and we showed that its use could provide runtime improvements of up to 22%. Using the result from the memory study as a guide, we showed that we could get these improvements for only a one kilobyte CCM and that we could achieve reasonable improvements with a CCM as small as 512 bytes.

Bibliography

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

[2] Anonymous. Performance of pentium pro and pentium ii processor/cache combinations. Technical report, ECG Technology Communications Group, Compaq Computer Corporation, May 1997.

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

[4] Bary R. Beck, David W.L. Yen, and Thomas L. Anderson. The cydra 5 minisupercomputer: Architecture and implementation. *The Journal of Supercomputing*, 7, 1993.

[5] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(6):287–295, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[6] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[7] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[8] Preston Briggs. Drawing control-flow graphs with style. Technical report, Rice University, July 1994.

[9] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989.

Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[12] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[13] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 40–52, Santa Clara, California, 1991.

[14] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994.

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

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

[17] Fred Chow, Sun Chan, Robert Kennedy an Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. *SIGPLAN Notices*, 32(6):273–286, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

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

[19] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimization. In Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, San Jose, California, 1996.

[20] Keith D. Cooper, Timothy J. Harvey, and Linda Torczon. How to build an interference graph. *Software – Practice and Experience*, page To appear, 1998.

[21] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the Seventh International Conference on Compiler Construction*, pages 174–187, Portugal, Lisbon, April 1998.

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

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

[24] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[25] Paul S. Heckbert. Ray tracing jell-o brand gelatin. *Computer Graphics*, 21(4):73–74, July 1987.

[26] John Hennesy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., second edition, 1990.

[27] Darrell Huff. *How to Lie With Statistics.* W. W. Norton and Company, Inc., thirty-eighth edition, 1982.

[28] Intel Corporation. *Pentium™ II Processor Developer's Manual*, 1997.

[29] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Storage assignment to decrease code size. *SIGPLAN Notices*, 30(6):186–195,

June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[30] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[31] John Lu and Keith Cooper. Register promotion in c programs. *SIGPLAN Notices*, 32(6):308–319, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[32] Sally A. McKee. Compiling for efficient memory utilization. In *Workshop on Interaction Between Compilers and Computer Architectures, Second IEEE Symposium on High Performance Computer Architecture (HPCA-2)*, San Jose, California, January 1996.

[33] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1996.

[34] Larry Meadows, Steven Nakamoto, and Vincent Schuster. A vectorizing, software pipelining compiler for LIW and superscalar architecture. In *Proceedings of RISC '92*, San Jose, CA, February 1992.

[35] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 62–75, Boston, Massachusetts, 1992.

[36] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1996.

[37] Todd A. Proebsting and Charles N. Fischer. Probalistic register allocation. *SIGPLAN Notices*, 27(7):300–310, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[38] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):217–226, May 1979.

[39] Ravi Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, September 1975.

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

[41] SPEC release 1.10, September 1995. Standards Performance Evaluation Corporation.

[42] Michael Upton, Thomas Huff, Trevor Mudge, and Richard Brown. Resource allocation in a high clock rate microprocessor. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994.

[43] Henry S. Warren, Jr. Static main storage packing problems. *Acta Informatica*, 9:355–376, 1978.

[44] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[45] Michael Wolfe. More interation space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989.

[46] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1), March 1995.