FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrated Code Motion and Register Allocation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Gergö Barany
Registration Number 0026139

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: ao. Univ. Prof. Dipl.-Ing. Dr. Andreas Krall

The dissertation has been reviewed by:

Andreas Krall	Alain Darte

Vienna, 14th January, 2015

Gergö Barany

Acknowledgements

Many people were very patient with me over the last few years. Thanks.

Abstract

Code motion and register allocation are important but fundamentally conflicting program transformations in optimizing compiler back-ends. Global code motion aims to place instructions in less frequently executed basic blocks, while instruction scheduling within blocks or regions (all subsumed here under 'code motion') arranges instructions such that independent computations can be performed in parallel. These optimizations tend to increase the distances between the definitions and uses of values, leading to more overlaps of live ranges. In general, more overlaps lead to higher register pressure and insertion of more expensive register spill and reload instructions in the program. Eager code motion performed before register allocation can thus lead to an overall performance decrease.

On the other hand, register allocation before code motion will assign unrelated values to the same physical register, introducing false dependences between computations. These dependences block opportunities for code motion that may have been legal before assignment of registers. This is an instance of the phase ordering problem: Neither ordering of these phases of code generation provides optimal results. A common way to sidestep this problem is by solving both problems at once in an integrated fashion.

This thesis develops a fully integrated approach to global code motion and register allocation. The result is an algorithm that determines an arrangement of instructions that leads to minimal spill code while performing as much global code motion and scheduling for parallelism as possible. Based on an overlap analysis that determines all the possible interferences between live ranges when taking all possible arrangements of instructions into account, the algorithm constructs a register allocation problem such that the solution encodes code motion information to ensure a legal allocation. A candidate selection pass determines which live ranges should be considered for reuse of a processor register. The selection process includes a tuning parameter to study the trade-off between global code motion and spilling, and can be performed in a heuristic or an optimal way.

The results show that in general, global code motion should be careful to take register pressure into account. Except for rare cases where register use is low, compilers should focus on arranging code such that minimal spilling is required.

Contents

Abstract v

Contents vii

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Compiler back-end program representations 1
- 1.2 Static single assignment (SSA) form 2
- 1.3 Register allocation and spilling 3
- 1.4 Instruction scheduling 4
- 1.5 Global code motion 4
- 1.6 Phase ordering between code motion and spilling 6
- 1.7 Motivation: A simple example 7
- 1.8 Global code motion with spilling 9
- 1.9 Limits of ‘optimal’ code generation 12

2 Related Work 15
- 2.1 Instruction scheduling 15
- 2.2 Integrated scheduling and register allocation 16
- 2.3 Integrated code motion and register allocation 20
- 2.4 GCMS 22

3 Live Range Overlap Analysis 25
- 3.1 Traditional overlap analysis 25
- 3.2 Overlap analysis in the presence of code motion 26
- 3.3 Greedy overlap analysis 34
- 3.4 Handling of two-address instructions 37
- 3.5 Overlap analysis for preallocated registers 38

4 Reuse Candidate Selection 41
- 4.1 The candidate selection problem 41

4.2 Heuristic reuse candidate selection . . . 43
4.3 Optimal reuse candidate selection . . . 44
4.4 Balanced reuse candidate selection . . . 46

5 Spilling and Global Code Motion **49**
5.1 PBQP register allocation . . . 49
5.2 Spilling with reuse candidate information . . . 51
5.3 Restricted global code motion . . . 53
5.4 Final instruction scheduling . . . 54

6 Experimental Evaluation **55**
6.1 Implementation issues . . . 55
6.2 Experimental methodology . . . 58
6.3 Results . . . 59
6.4 Results of heuristic GCMS . . . 71

7 Conclusions **75**

Bibliography **77**

List of Figures

1.1 Example program with global dependence graph and control flow graph . . . 6
1.2 Optimization using GCM and GCMS for a three-register processor . . . 8
1.3 Conflict graphs for each of the program variants from Figure 1.2 . . . 9
1.4 Comparison of compiler phases using regular register allocation and GCMS . 11

3.1 Nontrivial live range overlap in a non-SSA program. . . . 26
3.2 Illustration of the overlap criterion . . . 27
3.3 Exhaustive overlap analysis for virtual registers A and B . . . 29
3.4 Greedy overlap analysis for live ranges R1 and R2 . . . 35
3.5 Some possible code motions to avoid overlap of v and w . . . 36
3.6 Limited code motion in overlap analysis for preallocated registers . . . 40

4.1 Dependence graph with with conflicting sequencing possibilities (dashed) . . 42

5.1 Sample cost vector and cost matrix for the nodes and edges of a PBQP register allocation graph . . . 49

5.2 Sample ε edge cost matrix for modeling avoidable live range overlaps 51
5.3 Conflict graph for the example program from Figure 1.2, with dashed edges representing ε edges in the PBQP graph 52
5.4 The code motion impact of allocating the example program for processors with various numbers of registers 53

6.1 Time taken by the ILP solver for problems of various sizes ($\beta = 0$) 67
6.2 Time taken by the ILP solver for problems of various sizes, detail of Figure 6.1 67
6.3 Influence of solver time on the performance of selected benchmarks 68
6.4 Influence of the β parameter on the performance of selected benchmarks . . . 70

List of Tables

3.1 Blame terms computed for the example program, listing instruction placements and missing dependence arcs that may cause overlaps. 31

6.1 Execution times of benchmark programs with various GCMS configurations, inlining threshold 225 61
6.2 Execution times of benchmark programs with various GCMS configurations, inlining threshold 450 63
6.3 Static properties of selected benchmark functions with various GCMS configurations 65
6.4 Execution time speedups of benchmark programs compiled with various GCMS configurations with heuristic candidate selection. 72
6.5 Execution time speedups of benchmark programs compiled with various GCMS configurations with heuristic candidate selection and hoisting of uses disabled. 73

CHAPTER 1

Introduction

This thesis deals with the phase ordering problem between register allocation and global code motion. This chapter recalls the definitions of these problems, introduces a motivating example of the phase ordering problem, and gives a high-level description of the integrated solution developed as part of the thesis.

1.1 Compiler back-end program representations

The problems discussed in this thesis concern program transformations in compiler back-ends. The back-end is the compiler stage invoked after the input program has been parsed and statically analyzed, and high-level optimizations have been performed. The back-end maps some machine-independent representation of the computations in the program to the instruction set of some concrete target processor. The three main phases involved here are *instruction selection*, *instruction scheduling/global code motion*, and *register allocation*.

Instruction selection chooses the target instructions to implement the abstract computations in the program. In the rest of this thesis, we assume that this has been done, and the identities of instructions will remain unchanged. The scheduling/code motion and register allocation phases are discussed below.

In what follows, it is assumed that programs consist of one or more functions, but each function is processed by the compiler back-end in complete isolation from the other functions. From the point of view of the back-end, it therefore looks as if the function currently being compiled is identical to the input program. For simplicity, we will therefore sometimes use the phrase 'the (input) program' to mean 'the function in the input program currently being processed by the back-end'.

Instructions in the program are arranged in *basic blocks* (or simply *blocks*), which are single-entry single-exit regions in the program without transfers of control within the region. That is, a transfer of control in the program may only ever jump to some

block's beginning, and instructions that transfer control may only occur at the ends of blocks. Hence, unless a CPU exception is triggered, a block's first instruction is executed whenever all the other non-branching instructions in the block are executed before a branch is reached.

Within each block, instructions are arranged in some strict linear order. Processor architectures that allow instructions to be arranged in an explicitly parallel fashion (VLIWs) are not considered here.

A function's blocks are arranged in a *control flow graph* (CFG). In this graph, there is a directed edge from each basic block to each of the blocks that it may transfer control to. There is a special entry block where execution enters the function from its caller. The entry block has no predecessors in the CFG. It is assumed that every block in the function is reachable via CFG edges from the entry block. Every block is assumed to end in some instruction that jumps to another block in the same function or that returns to the function's caller.

An edge in the CFG is *critical* if it leads from a block with more than one successor to a block with more than one predecessor. A critical edge can be split by inserting a new, empty block on the edge. The rest of this thesis assumes a program representation in which all critical edges have been split, as this creates convenient targets for global code motion.

A central notion in what follows is that of *dominance*. A basic block a *dominates* a basic block b, written $a \succeq_{\text{dom}} b$, if every path in the CFG from the entry block to b must pass through a. Dominance is reflexive, transitive, and antisymmetric, and the entry block dominates every block. Each block's dominators are themselves ordered by dominance: If $a \succeq_{\text{dom}} c$ and $b \succeq_{\text{dom}} c$, then $a \succeq_{\text{dom}} b$ or $b \succeq_{\text{dom}} a$.

A block a *strictly dominates* b, $a \succ_{\text{sdom}} b$, if $a \succeq_{\text{dom}} b$ and $a \neq b$. A block a *immediately dominates* b, $a \succ_{\text{idom}} b$, if $a \succ_{\text{sdom}} b$ and for all other dominators d of b, $d \succeq_{\text{dom}} a$. Due to these properties, the basic blocks can be arranged in a *dominator tree* in which a is a parent of b if and only if $a \succ_{\text{idom}} b$. The entry block is necessarily at the root of the dominator tree.

Dominance can be generalized from basic blocks to instructions: For a given arrangement of instructions in the program, instruction i in block b_i dominates an instruction j in block b_j if $b_i \succ_{\text{sdom}} b_j$ or $b_i = b_j$ and i appears before j in that block. In Section 1.5, this will be generalized further to a notion of dominance on the program's global dependence graph.

1.2 Static single assignment (SSA) form

As an extension of Global Code Motion (Click 1995), our algorithm is based on a representation of the program in *SSA* (static single assignment) form (Cytron et al. 1991). In SSA, every value in the program has exactly one point of definition. Definitions from different program paths, such as updates of a value in a loop, are merged by inserting ϕ pseudo-instructions. The ϕ instructions are always placed at the beginnings of basic blocks and have one operand for each of the block's predecessors, representing the flow of

a value from that predecessor. The value defined by the ϕ represents the merged value, i. e., the value that actually reaches that program point during execution of the program. The ϕ instructions are removed from the final program by allocating their operands to identical physical registers or by introducing copy instructions (Sreedhar et al. 1999).

The particular form of SSA assumed in this thesis is *strict* SSA form. This is the commonly used variant of SSA that requires every instruction that defines a value to dominate all of the uses of that value. Where this condition is not already fulfilled by the input program, i. e., where some values may be undefined along some program paths, the program is assumed to have been transformed into a state where such cases are resolved explicitly by insertion of special `undef` pseudo-instructions and appropriate ϕ instructions.

For the purposes of dominance in strict SSA form, the ϕ instructions are treated as if each use of a merged value appeared at the end of the appropriate predecessor block. These uses are dominated by the definition even if, as is common in loops, the ϕ itself appears before the definition of some value that it uses.

1.3 Register allocation and spilling

Register allocation is one of the classical optimizations in a compiler's back-end. It concerns the assignment of values (or *virtual registers*) used by a program to the processor's physical registers. Values that are not allocated to a register must be *spilled* (stored) to main memory and reloaded before use. As main memory accesses are significantly slower than register accesses, it is imperative for good performance to find an allocation that incurs the lowest possible number of executed spill and reload instructions. Spillers are typically guided by a model of *spill costs*: a static estimate of the numbers of executed loads and stores due to a spill.

A fundamental notion in register allocation is that of *liveness*: A value is live at a program point if it may be used at some later point without an intervening redefinition. A value's *live range* is the set of all program points where that value is live. Live ranges *conflict* or *overlap* if they intersect at some program point. The register allocator must ensure that at each program point, the set of all live values can be assigned to registers without conflict. If there are too many live values, some of them must be eliminated by spilling or other methods such as rematerialization, i. e., duplication of computations (Briggs, Cooper, and Torczon 1992).

The above definition of conflict can be refined further: Overlapping live ranges can be allocated to the same register if their values are provably equal. In particular, if an SSA value is defined by a copy instruction, it can be allocated to the same register as the source of the copy. This operation is known as *coalescing*. This thesis does not consider coalescing for two reasons. First, naïve coalescing can increase the number of live range conflicts, and it is not clear how to best integrate it with code motion. Second, many of the copies in target programs cannot, in fact, be coalesced because they are vital for establishing the correctness of SSA form in the presence of code motion and instructions that modify some of their input operands (see Section 3.4).

A value's live range depends crucially on the arrangement of instructions in the program. Since in the context of global code motion this arrangement is not fixed, traditional liveness analysis is not applicable to the work described in this thesis. Chapter 3 describes how overlaps between live ranges can be detected in the presence of global code motion.

The actual allocation of values to registers can be performed in one of several ways. While faster approaches exist (Poletto and Sarkar 1999), for the purposes of this work it is convenient to perform allocation on a *conflict graph*. The nodes of this graph are the live ranges in the program, and overlapping live ranges are connected by edges. Allocation is successful if a physical register can be assigned to each node such that no neighboring nodes are assigned the same register (or, more generally, overlapping registers). This approach is known as graph coloring (Chaitin 1982). The PBQP approach actually used in this work can be viewed as a generalization of graph coloring in which costs are assigned to allocations, and the overall cost is to be minimized (Scholz and Eckstein 2002; Hames and Scholz 2006).

1.4 Instruction scheduling

Instruction scheduling is the back-end phase that determines the ordering of instructions in each basic block. The schedule must respect *dependences*: Instructions may not appear before some condition is fulfilled. The most important case are data dependences between the uses of values and their definitions, i. e., values may not be used before they are defined. More general ordering dependences apply to instructions manipulating memory locations or CPU registers, where it must be ensured that values are not overwritten before the previous value is used for the last time. Dependences are usually captured in a *dependence graph*, an acyclic directed graph.

Any topological ordering of the dependence graph is in principle a legal schedule. However, not all schedules result in the same performance on modern processor architectures. On pipelined processors, instruction-level parallelism and thus performance can be improved by scheduling long computations early and at some distance from their uses, interleaving them with other computations. Scheduling can also take into account the number of different kinds of functional units in the processor and attempt not to schedule too many instructions that need the same functional unit at a time. Out-of-order execution, multiple instruction issue on superscalar processors, and register renaming performed by the CPU may to some extent alleviate bad scheduling decisions by the compiler, but scheduling can still be important in practice.

1.5 Global code motion

The motion of instructions between basic blocks in this work is based on Global Code Motion (GCM) (Click 1995), a generalization of loop-invariant code motion techniques. It is *global* in considering an entire function at once. GCM is based on a representation of the program in SSA form and includes ϕ instructions as normal nodes in the graph.

However, for the purposes of this work it is important that the dependence graph is acyclic to ensure schedulability. If we had dependence cycles, we would not be able to schedule instructions in a way that respects all dependences. In SSA form, cyclic data dependences can only occur as arcs from an instruction to a ϕ earlier in a loop. We therefore only add those dependence arcs for ϕ nodes that do not introduce such cycles. It is safe to ignore the cyclic dependences as long as we ensure that such instructions will not be sunk out of their loops.

In the graph, each instruction is associated with a list of basic blocks in which it may be placed legally. Some instructions are pinned to the blocks in which they appear originally: ϕ instructions and branches model control flow and may never be moved to another block. We also forbid moving any instruction that may have a side effect (function calls and stores), any load except from the constant pool or stack, and any instruction that uses a physical register explicitly (copies to and from argument and return registers, instructions that use condition code registers). We use arcs to fix the order of those instructions within the block that may have side-effects on memory or access physical registers. This is overly conservative, but it saves us from having to solve NP-complete problems that come up when trying to schedule live ranges for pre-assigned physical registers (Darte and Quinson 2007). We do allow reordering of loads between instructions that may store. Instructions that are not pinned to their blocks may be moved to other blocks within the constraints given by dependence arcs.

Code motion is guided by the notion of dominance: A legally placed instruction dominates all of the instructions that depend on it and is dominated by all the instructions it depends on. The earliest and latest possible blocks for all instructions can be computed in two simple passes over the graph in topological order and reverse topological order. For each instruction, the earliest block necessarily dominates the latest block, and any intervening block on the path in the dominator tree from the earliest to the latest block is also deemed valid.

Figure 1.1 shows the code, the dependence graph, and the control flow graph for a program which will be used to illustrate concepts throughout this thesis. In the dependence graph, instructions are represented as nodes, and arcs in the dependence graph are labeled with the values that flow along them. Instructions are shown within basic blocks (rectangles), but some instructions may be moved to other blocks. These instructions are annotated with a list of the names of all the blocks in which they may be placed legally.

In the CFG in Figure 1.1, the `start` block dominates `loop`, which dominates `end`. The function calls, the ϕ, the branch, and the return are not movable; all other instructions may in principle move as dependences allow. The branch depends on all arithmetic instructions except the multiplication, which is therefore the only arithmetic instruction legal for sinking out of the loop. The addition that computes `b` may be hoisted out of the loop because it only depends on values computed before the loop. The operations involving the `j` variables may not be hoisted due to the ϕ and may not be sunk due to the dependence from the branch. The cyclic dependence due to the ϕ is shown as a dashed arc, but is not present in the actual graph.

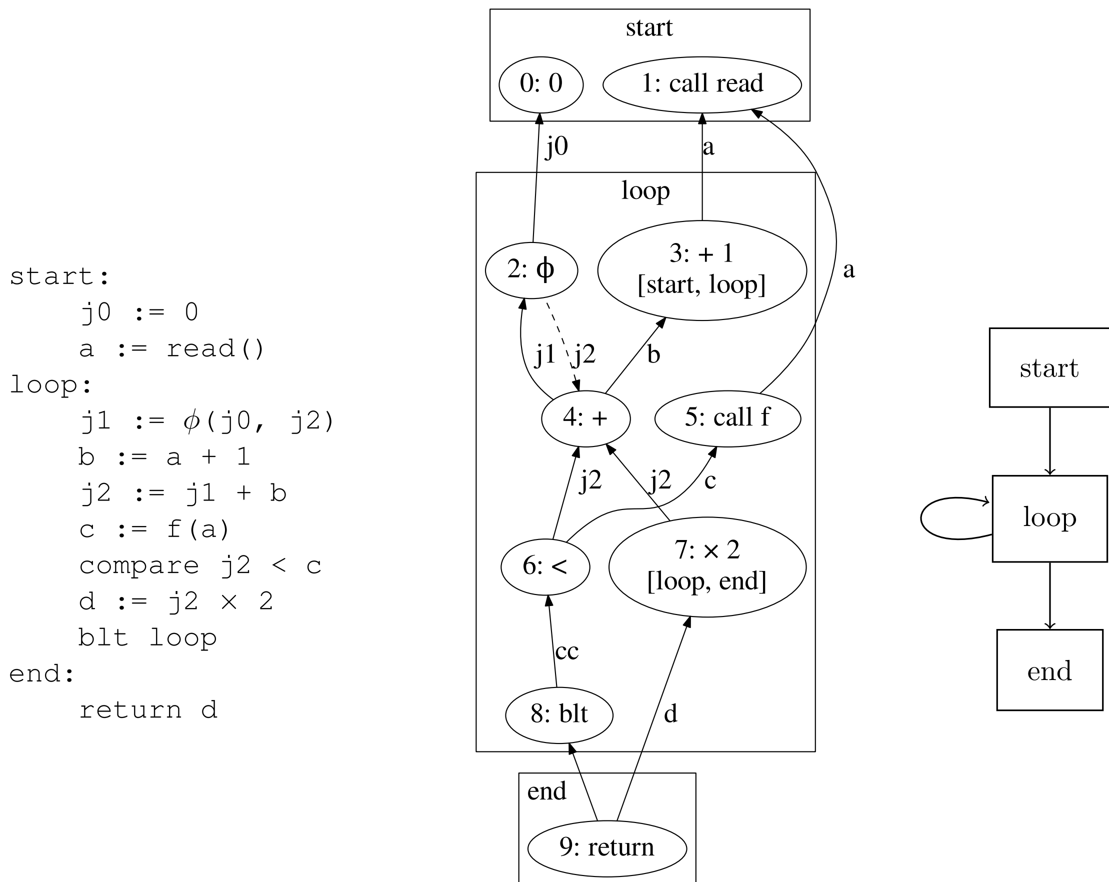


Figure 1.1: Example program with global dependence graph and control flow graph

The notion of dominance is lifted from fixed schedules of instructions to instructions in the dependence graph: An instruction i dominates an instruction j if there is a (possibly transitive) dependence from j to i, or if i's last legal block strictly dominates j's first legal block.

1.6 Phase ordering between code motion and spilling

Performing global code motion before spilling and register allocation (*prepass* scheduling) might lead to a program that requires many spills. Conversely, code motion and scheduling after register assignment (*postpass* scheduling) can be hindered by false dependences introduced by allocating unrelated values to the same physical register.

There are decomposed register allocators that separate spilling from allocation (Koes and Goldstein 2009), with the spiller inserting enough spill code to guarantee that there

is some valid allocation of the resulting program. It is possible to insert a code motion and scheduling pass after spilling but before allocation; however, care must be taken to ensure that the guarantee of a valid allocation is not invalidated by rearranging code.

Scheduling and code motion transformations may therefore attempt to balance their optimizations against the needs of register allocation, but they do not typically operate with an exact model of register usage. In previous work (see Chapter 2), many authors presented techniques to integrate local scheduling decisions within basic blocks with register allocation. In this work, we generalize such scheduling decisions to *global* code motion between blocks guided by the register allocator and based on its exact model of register demands.

The results of Govindarajan et al. (2003) suggest that, at least on modern out-of-order architectures, reducing the number of spills has more benefits than any other possible local scheduling decision. We expect this result to carry over to global code motion as well and provide experimental data that shows improvements due to our algorithm built on this premise. On embedded processors without out-of-order execution, more careful scheduling to make optimal use of the pipeline is more important. However, we believe that our transformation, which reduces the number of executed loads, is an optimization due to the removal of these expensive instructions. For this reason, our approach for integrated code motion and register allocation attempts to minimize overall spill costs while preserving as much freedom as possible for subsequent global code motion.

Our work attempts to strike a balance between the needs of register allocation and global code motion transformations. It makes use of the same graph structure to drive both spilling and code motion and can therefore handle these two important program transformations in a unified way. In our approach, spilling is performed not on a given schedule computed by a prepass scheduler, but rather a graph that represents *all* possible schedules. Some spills can be avoided by adding arcs to the graph to serialize computations. This can ensure that live ranges do not overlap, enabling the reuse of processor registers for unrelated values. The register allocator attempts to find a solution that incurs minimal spill costs but at the same time restricts the graph as little as possible. The resulting program contains a minimum of spill code but still has some freedom for aggressive code motion optimizations.

1.7 Motivation: A simple example

Figure 1.2 shows a few variants of the example program that was introduced above, adapted from the original paper on GCM (Click 1995). Figure 1.2a shows the same code as in Figure 1.1, repeated here for reference. It is easy to see that the computation of variable `b` is loop-invariant and can be hoisted out of the loop; further, the computation for `d` is not needed until after the loop. Since the value of its operand `j2` is available at the end of the loop, we can sink this multiplication to the `end` block. Figure 1.2b illustrates both of these code motions, which are automatically performed by GCM. The resulting program contains less code in the loop, which means it can be expected to run faster than the original.

```
start:
    j0 := 0
    a := read()

loop:
    j1 := φ(j0, j2)
    b := a + 1
    j2 := j1 + b
    c := f(a)
    compare j2 < c
    d := j2 × 2
    blt loop
end:
    return d
```

(a) Original function

```
start:
    j0 := 0
    a := read()
    b := a + 1
loop:
    j1 := φ(j0, j2)
    j2 := j1 + b
    c := f(a)
    compare j2 < c
    blt loop


end:
    d := j2 × 2
    return d
```

(b) After GCM

```
start:
    j0 := 0
    a := read()

loop:
    j1 := φ(j0, j2)
    b := a + 1
    j2 := j1 + b
    c := f(a)
    compare j2 < c
    blt loop

end:
    d := j2 × 2
    return d
```

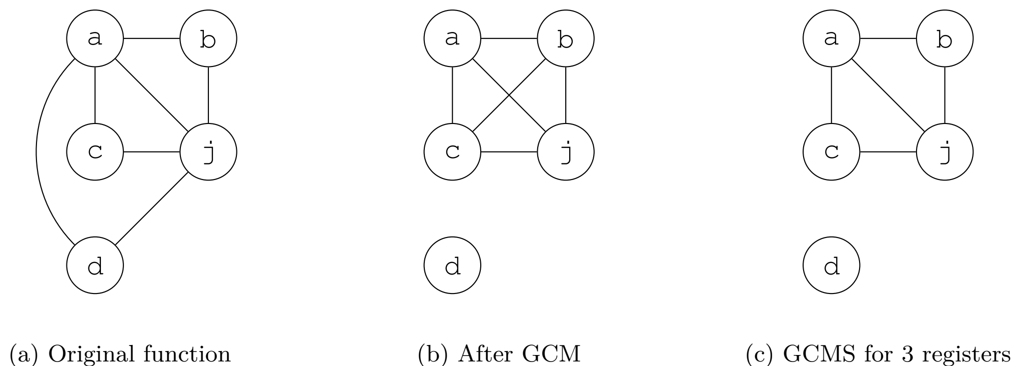
(c) GCMS for 3 registers

Figure 1.2: Optimization using GCM and GCMS for a three-register processor

This expectation fails, however, if there are not enough registers available in the target processor. Consider the conflict graphs of the original program in Figure 1.3a and the program after GCM in Figure 1.3b. For simplicity, in the conflict graphs the ϕ-related variables `j0`, `j1`, and `j2` are merged into a single variable `j`. This merging is not possible in general and not needed for the GCMS algorithm, but it simplifies the presentation in this case.

Since after GCM the variable `b` is live through the loop, it conflicts with `a`, `c`, and `j`. Both `a` and `c` conflict with each other and with at least one of the `j` variables, so after GCM we need four CPU registers for a spill-free allocation. If the target only has three registers available for allocation of this program fragment, costly spill code must be inserted into the loop. As memory accesses are considerably more expensive than simple arithmetic, GCM would trade off a small gain through loop invariant code motion against a larger loss due to spilling.

Compare this to Figure 1.2c, which shows the result of applying our GCMS algorithm for a three-register CPU. To avoid the overlap of `b` with all the variables in the loop, GCMS leaves its definition inside the loop. This ensures that a register limit of 3 can be met. However, GCMS is not fully conservative: Sinking `d` out of the loop can be done without adversely affecting the register needs, so this code motion is performed by GCMS. Note that this particular code motion does lengthen `j`'s live range, but the algorithm determines that this does not lead to unwanted live range overlaps in this case. The conflict graph for this variant of the program is shown in Figure 1.3c. Observe also that this result is specific to the limit of three registers: If four or more registers were



(a) Original function (b) After GCM (c) GCMS for 3 registers

Figure 1.3: Conflict graphs for each of the program variants from Figure 1.2

available, GCMS would detect that unrestricted code motion is possible, and it would produce the same results as GCM in Figure 1.2b.

The idea of GCMS is thus to perform GCM in a way that is more sensitive to the needs of the spiller. As illustrated in the example, code motion is restricted by the choices of the register allocator, but only where this is necessary. In functions (or parts of functions) where there are enough registers available, GCMS performs unrestricted GCM. Where there is higher register need, GCMS serializes live ranges to avoid overlaps and spill fewer values.

In contrast to most other work in this area, GCMS does not attempt to estimate the register needs of the program before or during scheduling. Such models cannot predict which spilling and allocation decisions will be made by the register allocator. This means that such algorithms can be overly conservative. For example, the IPS algorithm (see Section 2.2.1) schedules basic blocks before register allocation based on a precomputed register limit. If the register allocator spills a value that is live across a basic block but not used in that block, this amounts to increasing the register limit for that block. However, as a purely prepass approach, IPS cannot take advantage of this change introduced by the spiller.

In contrast, GCMS computes a set of promising code motions that could reduce register needs if necessary. An appropriately encoded register allocation problem ensures that the spiller chooses which code motions are actually performed. Thus GCMS's code motion model is always fully synchronized with the actual spilling and allocation decisions make by the register allocator. Code motion and scheduling restrictions that are not needed to avoid spilling are not applied.

1.8 Global code motion with spilling

Given the dependence graph and legal blocks for each instruction, GCMS proceeds in the following steps:

Overlap analysis determines for every pair of values whether their live ranges might overlap. The goal of this analysis is similar to traditional liveness analysis for register allocation, but with the crucial difference that in GCMS, instructions may move. Our overlap analysis must therefore take every legal placement and every legal ordering of instructions within blocks into account.

For every pair, the analysis determines whether the ranges definitely overlap in all schedules, never overlap in any schedule, or whether they might overlap for some arrangements of instructions. In the latter case, GCMS computes a set of code placement restrictions and extra arcs that can be added to the global dependence graph. Such restrictions ensure that the live ranges do not overlap in any schedule of the new graph, i. e., they enable *reuse* of the same processor register for both values.

Candidate selection chooses a subset of the avoidable overlaps identified in the previous phase. Not all avoidable overlaps identified by the analysis are avoidable *at the same time*: If avoiding overlaps for two register pairs leads to conflicting code motion restrictions, such as moving an instruction to two different blocks, or adding arcs that would cause a cycle in the dependence graph, at least one of the pairs cannot be chosen for reuse. GCMS must therefore choose a promising set of *candidates* among all avoidable overlaps. Only these candidate pairs will be considered for actual overlap avoidance by code motion and instruction scheduling. Since our goal is to avoid expensive spilling as far as possible, we try to find a candidate set that maximizes the sum of the spill costs of every pair of values selected for reuse.

Spilling and code motion use the results of the candidate selection phase by building a register allocation problem in which the live ranges of reuse candidates are treated as non-conflicting. The solution computed by the register allocator is then used to guide code motion: For any selected candidate whose live ranges were allocated to the same CPU register, we apply its code motion restrictions to the dependence graph. The result of this phase is a restricted graph on which we can perform standard GCM, with the guarantee that code motion will not introduce excessive overlaps between live ranges.

Figure 1.4 shows an abstract comparison of compiler back-end phases for regular register allocation and GCMS. In a typical case (as implemented in the LLVM compiler framework, for instance), instruction scheduling and loop-invariant code motion (LICM) are performed before the liveness analysis that feeds the spiller and register allocator. In our case, code motion and scheduling before spilling are not necessary; we only place instructions after spilling. Instead we build a dependence graph which is used both for code motion and for overlap analysis. Liveness analysis is replaced by two phases: overlap analysis to determine all the possibly overlapping live ranges, and selection of the set of pairs of possibly overlapping live ranges that will be presented to the register allocation phases. Spilling and register allocation proceed identically in GCMS and regular register allocation. Finally, GCMS uses the allocator's results to perform restricted global code

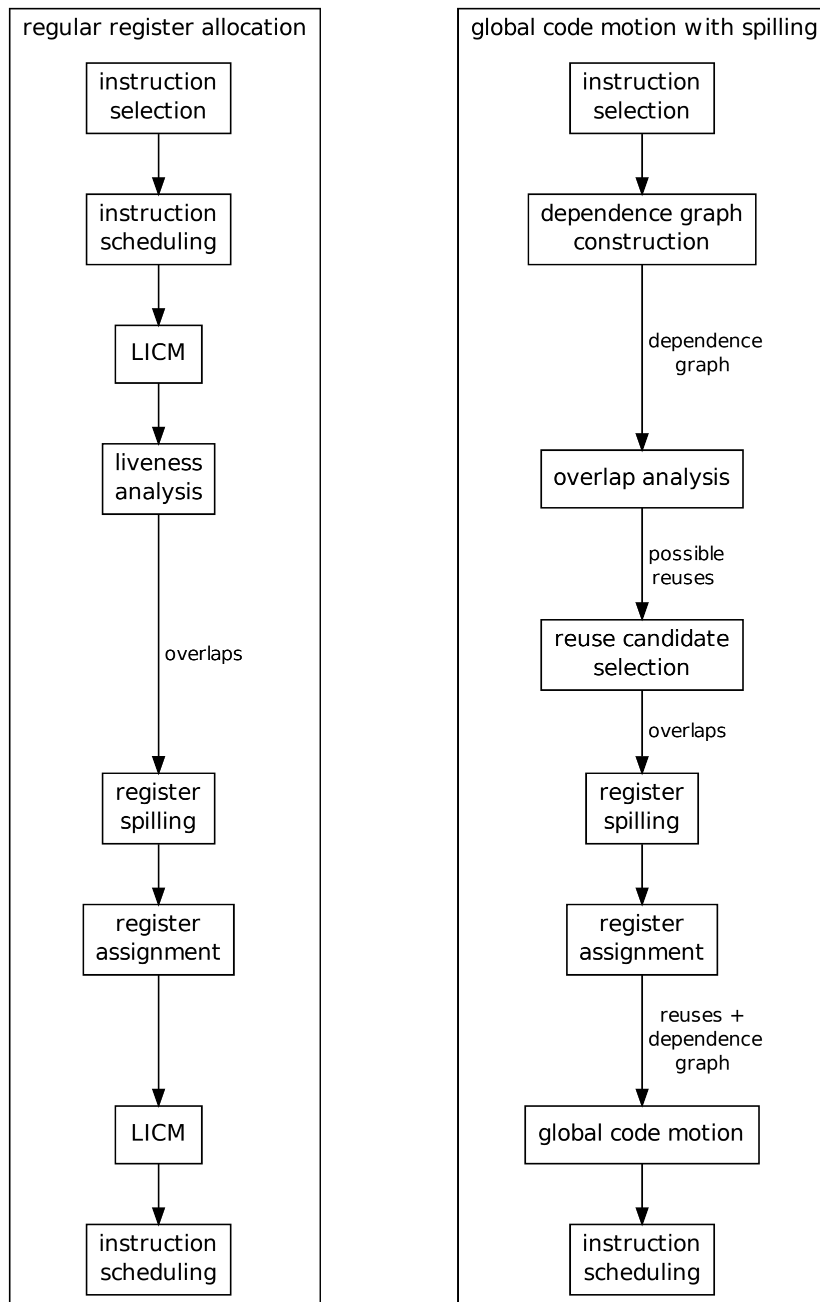


Figure 1.4: Comparison of phases in a compiler back-end using regular register allocation (left) and a back-end using integrated code motion and register allocation with GCMS (right)

motion and scheduling to determine the final arrangement of instructions in each block. A regular back-end can also follow register allocation with another round of LICM and scheduling to exploit remaining possibilities for optimization.

Each of the three main phases of GCMS is the topic of one of the following chapters.

1.9 Limits of 'optimal' code generation

The GCMS algorithm presented in this thesis formulates a specific model of integrated code generation and solves this model optimally. However, there can be cases where the generated code is not actually as fast as some other, equivalent code generated in some other way. This is due to several reasons.

First, not all data that would be needed for the generation of optimal code is available. In part, this is unavoidable due to the complexity of modern computer architectures: The execution time of a piece of code is not fixed but depends on the states of the processor pipeline and on the caches. For simple platforms, safe approximations of the hardware's worst-case behavior exist and can be exploited in code generation (Falk 2009; Lokuciejewski et al. 2010), but the exact behavior of an out-of-order processor is typically undocumented and essentially unpredictable at compile time.

Other performance-relevant data concerns the program itself. In particular, detailed basic block profiling information would allow a much more precise formulation of spill costs and the benefits of global code motion operations. However, the LLVM compiler framework does not provide this data at the required granularity: Basic block profiling data is available on the LLVM intermediate code level, but the lowering to target-specific code introduces new blocks and complex new control flow that makes it impossible in general to propagate profiling information to all points in the program. The cost model used by GCMS therefore relies on static estimates of block frequencies, which may not agree with actual dynamic frequencies.

Second, the cost model used by GCMS does not capture all aspects of spilling and code motion directly. While global code motion is guided by a cost model, the impact of artificial instruction ordering constraints on local schedule lengths is not modeled. This is based on the expectation that out-of-order execution will hide most of the associated costs (Govindarajan et al. 2003), but that is not always the case in practice. The spill cost model for candidate selection itself does not perform or predict the concrete spilling choices made by the decoupled spiller; it only minimizes the total sum of spill weights of overlapping live ranges. This is a simple and useful model, but there is no guarantee that it will always lead to the set of reuse candidates and associated schedules that absolutely minimize spilling.

Finally, some parts of GCMS rely on pre-existing, heuristic building blocks for simplicity. In particular, spilling is performed using the heuristic PBQP register allocator, and register assignment is not performed to avoid having to deal with out-of-SSA transformation. As the results in Chapter 6 show, both of these choices can sometimes cause paradoxical behavior in practice. Using an optimal formulation of spilling and register assignment with GCMS may be possible in theory. This would, however, lead to

much longer solver times. Additionally, the spilling problem itself is not fully solved yet: Even the careful optimal formulation of spilling in SSA form by Colombet, Brandner, and Darte (2011) leaves open the problem of optimal coalescing of the operands and results of ϕ instructions.

CHAPTER 2

Related Work

This chapter summarizes the most relevant existing research into instruction scheduling and its relation to register allocation. General overviews of both topics are standard material in compiler textbooks (Aho, Lam, et al. 2006; Cooper and Torczon 2004; Muchnick 1997). These textbooks traditionally present scheduling and register allocation as separate topics without considering integrated approaches.

2.1 Instruction scheduling

In the absence of instruction pipelining, the ordering of instructions within basic blocks does not matter as long as all dependences are satisfied and the register use is not so large that it would cause spilling. Very early work in code generation thus focused on minimizing the number of instructions emitted and the number of registers needed. Under the unrealistic assumption of only evaluating expression trees, not DAGs, the simple Sethi-Ullman numbering algorithm produces code that is minimal in both register use and total number of instructions (Sethi and Ullman 1970). Solving the same problem for DAGs is NP-complete, but the Sethi-Ullman algorithm can be generalized to a heuristic for DAGs (Aho, S. C. Johnson, and Ullman 1977).

With the advent of pipelined architectures, scheduling became more important in order to exploit instruction-level parallelism[1]. Rymarczyk (1982) gives an early tutorial on programming for pipelined systems.

Finding a minimal-length schedule for a pipelined processor is NP-complete in general and is typically solved using heuristic list scheduling (Hennessy and Gross 1982; Gibbons and Muchnick 1986). List scheduling is an iterative algorithm that traverses a basic block DAG and emits instructions one by one, respecting dependences and various other

[1] Instruction-level parallelism is often abbreviated ILP. In this thesis, the acronym ILP is used for *integer linear programming*, and instruction-level parallelism is spelled out in full or abbreviated as ‘parallelism’ when needed.

constraints. At each point, there are typically several instructions that are ready to be scheduled. The next instruction is selected heuristically based on considerations of latency, the length of the path to the end of the block, or availability of processor resources. In processors without pipeline interlocks, no instruction may be ready at a given time because all unscheduled instructions must wait for some data or resource to become available. In such cases, the scheduler must emit a no-op instruction. The criteria for selecting the next instruction to be scheduled may be captured formally in a rank function that assigns an integer-valued rank to each instruction in the DAG (Palem and Simons 1990). Ranks are usually precomputed immediately prior to scheduling. The list scheduler always selects an instruction of maximal rank among the ready instructions to be scheduled next, possibly using additional properties to break ties. For processors with complex pipelines, it can make sense to adapt the rank function on-the-fly depending on the amount of parallelism present in the code (Ertl and Krall 1992).

Besides heuristic list scheduling, there is also a body of work on optimal approaches. These can be based on formulations such as constraint logic programming (Ertl and Krall 1991), integer programming (Wilken, Liu, and Heffernan 2000), or enumeration with pruning (Shobaki and Wilken 2004). As with list scheduling, the goal of such models is usually to optimize for minimal total schedule length only.

2.2 Integrated scheduling and register allocation

Arranging instructions for minimal schedule length on a pipelined processor tends to lengthen live ranges and thus increase the number of live range overlaps. For example, the typical examples for list scheduling in the classic papers are based on hiding the latencies of load instructions by scheduling loads next to each other. However, this means that the loaded values take up registers earlier than they might be needed. Integrated approaches to scheduling and register allocation try to ensure that this kind of operation to exploit parallelism is only performed as long as it does not lead to excessive spilling.

The following sections mostly describe notable heuristic approaches to the problem. A recent survey article (Castañeda Lozano and Schulte 2014) discusses integrated combinatorial approaches.

2.2.1 Integrated prepass scheduling

Integrated prepass scheduling (IPS) was the first attempt to perform scheduling while considering register use (Goodman and Hsu 1988). IPS is a modified list scheduler that integrates a standard latency-oriented list scheduler with another scheduler based on the Sethi-Ullman algorithm that aims at minimizing register usage. IPS runs before register allocation.

IPS keeps track of the number of available registers at each program point; this value is initialized before scheduling each block based on the results of a global liveness analysis. For each value that is used within the block, the number of uses is also computed. IPS assumes a basic block in single-assignment form. Instructions are initially scheduled as in

regular list scheduling. Each value's use count is decreased for each scheduled instruction that uses it. The number of available registers is decreased for each value defined by the last scheduled instruction, and increased for each value for which the last scheduled instruction was the last use.

Whenever the number of available registers is below a threshold (such as 1), IPS chooses the next instruction based on the second selection function. If possible, it chooses an instruction that frees a maximal number of registers. If no such instruction is available, IPS prefers an instruction that continues evaluating an expression that has been partially evaluated already. Otherwise, it chooses arbitrarily. IPS can thus sometimes exceed the given register limit, which may cause spilling later on.

IPS is run before a final register allocator which may insert spill code where IPS did not succeed in finding a schedule that keeps within the available register limit. Simulations show that IPS outperforms both prepass and postpass scheduling in terms of the number of processor cycles executed in the final program.

2.2.2 DAG-driven register allocation

DAG-driven register allocation was introduced together with IPS (Goodman and Hsu 1988). It approaches the problem from the other direction first: Rather than scheduling the DAG while keeping register allocation in mind, DAG-driven register allocation performs allocation first and tries to minimize the false dependences introduced in the process.

DAG-driven register allocation is based on measuring the width of the DAG, which is the number of registers that would be needed if all parallelism were exploited by scheduling, and the height of the DAG, which is the length of the longest path. While the width of the DAG exceeds the number of available registers, false dependences are introduced to reduce parallelism. Each such operation may increase the DAG's height by making the critical path longer.

The allocator visits nodes in the DAG in topological order and assigns registers. For each instruction it prefers reusing a register that was used by one of its predecessors in the DAG since this does not add an extra dependence. If dependences must be added, the choice is guided by instructions' earliest issue times and earliest finish times, computed from the DAG. The allocator tries to limit the growth in the DAG's height by ensuring that the instructions with the highest earliest finish times reuse registers from instructions with the lowest earliest issue times.

Simulation results show that DAG-driven register allocation performs similarly to IPS, possibly slightly worse on highly pipelined machines and slightly better on others. The differences disappear with larger numbers of CPU registers.

The idea of DAG-driven register allocation was later generalized to global register allocation (Ambrosch et al. 1994). In dependence-conscious coloring, a graph-coloring global register allocator is informed by a collection of DAGs for the program's basic blocks. Whenever a live range is to be assigned to a CPU register, the allocator computes the impact of each possible assignment on the schedules. The allocator chooses a register that minimizes the increase in schedule lengths due to false dependences. Preliminary results

for this approach showed that it appeared effective at exploiting scheduling freedom if register pressure was not excessive.

2.2.3 RASE

The RASE (Register Allocation with Schedule Estimates) algorithm (Bradlee, Eggers, and Henry 1991) decomposes register allocation into local and global allocation for live ranges that are only live within one block or across block boundaries, respectively. Global live ranges are allocated by a standard register allocator based on graph coloring. Local live ranges are allocated during a basic block scheduling pass that must meet a given register limit.

The register limits for basic blocks are computed by the global allocator as well, based on scheduling information computed beforehand. Based on two trial scheduling passes, one without a register limit and one with a very low register limit, RASE estimates a function that quantifies how each block's schedule length would increase for any possible register limit. Nodes representing these changes in costs, weighted by estimated block frequencies, are added to the global register allocation problem. The register limit for each block is the number of nodes representing that block in the allocation problem that have been assigned a color.

The final scheduler within each block is similar to IPS except that it may never exceed its register limit; spill code is inserted by the scheduler if necessary. Overall, RASE is found to generate code that is very similar to IPS, with a possible slight advantage on very large basic blocks.

2.2.4 Parallel interference graph algorithm

Pinter (Pinter 1993) describes an algorithm based on the coloring of a parallel interference graph. This coloring gives a register allocation that does not introduce false dependences, so a simple postpass scheduling pass can exploit maximal parallelism. The basic algorithm considers one block at a time.

The parallel interference graph is constructed as follows. A node is created for each instruction, and any two instructions connected by a path in the dependence graph are connected by an edge. This graph is further enhanced with an edge between any two instructions that may not execute at the same time on a superscalar machine due to resource conflicts. For example, if there is only a single floating-point unit, all floating-point computations are connected by edges. The resulting graph captures all the pairs of instructions that may not execute in parallel. Its complement is then the graph connecting all pairs of instructions which may be scheduled in parallel; no false dependences should be introduced between these pairs. The union of this parallelism graph (in which instructions are represented by the live ranges they define) with the live range interference graph is the parallel interference graph that is to be colored. By construction, if a valid coloring exists, it does not introduce false dependences between instructions that may execute in parallel.

If no coloring can be found, some live ranges are spilled. Pinter does not present an experimental evaluation of this algorithm.

The parallel interference graph shares interesting similarities with the conflict graph that GCMS uses for integrated register allocation and code motion (see Section 5.2). However, the trade-offs are completely opposite: GCMS attempts to avoid spilling even if that means introducing false dependences, while Pinter's algorithm would rather spill than accept a false dependence, even if it might not disturb the schedule much.

2.2.5 Norris and Pollock's algorithm

The BMW algorithm of Norris and Pollock (Norris and Pollock 1993) is also similar to the local scheduling part of GCMS. It starts out with constructing a global register interference graph that captures all the possible live range overlaps if all possible schedules are considered. The number of interferences is then reduced by removing those that correspond to the 'least likely' scheduling decisions. The appropriate arcs are added to the dependence graph to ensure that these scheduling decisions cannot be taken. These arcs that are not likely to change the final schedule are determined by examining the scheduler's rank function on the DAG: Nodes are considered unlikely to be scheduled after nodes of lower rank. During register allocation, additional arcs are added if necessary to further reduce the need for registers.

Several variants of the basic algorithm are evaluated on a simulator. The results show that the best variant always outperforms IPS. The reasons for this are not discussed, but it appears that this algorithm might lead to less spilling than IPS.

2.2.6 URSA

The URSA (unified resource allocation) algorithm (Berson, Gupta, and Soffa 1993; Berson, Gupta, and Soffa 1999) treats registers and functional units in VLIW processors in a unified way and computes schedules with the intention of reducing excess resource requirements. The demand for resources (registers or functional units) is computed on a dependence DAG by decomposing it into *allocation chains*, which are sequences of instructions that can reuse some resource. Where the demand is too high, it is reduced by inserting dependence arcs or by spilling.

Crucially, the spilling performed by URSA and ILS, a variant of IPS developed for purposes of comparison with URSA (Berson, Gupta, and Soffa 1999), spills only parts of live ranges. (The authors refer to this as live range splitting.) This is compared to a variant of IPS that uses the simpler 'spill everywhere' approach in which entire live ranges are spilled to memory and reloaded before each use. The comparison of IPS, ILS, and URSA shows that both URSA and ILS vastly outperform IPS in simulations, especially for small numbers of registers. However, the improvement of URSA over ILS is much more modest. Thus it seems that a large part of the improvement due to URSA is not due to its complex resource allocation technique but rather due to sophisticated spilling.

2.2.7 Register saturation

The URSA approach was studied further from the point of view of register saturation (Touati 2001), which is the maximal register need over all schedules of a given DAG. The register saturation can be computed exactly by finding the maximal antichain of a special dependence graph associated with a killing function that maps each live range to one instruction which is its last use. However, finding a maximizing killing function is NP-complete.

The study of register saturation uncovered cases where URSA's approach to computing the register demand from the dependence graph appears to underestimate the actual needs over all schedules. Heuristics for computing and reducing the register saturation by adding dependences are given and found near optimal through comparison with optimal solutions obtained by integer linear programming. However, the relationship to global register allocation is not studied. Later work in register saturation improves the heuristics by serializing not just individual live ranges, but entire sets of live ranges at a time (Xu and Tessier 2007).

2.2.8 CRISP

The CRISP (combined register allocation and instruction scheduling) algorithm (Motwani et al. 1995) makes an interesting generalization to the integrated approaches discussed above. While older approaches tended to try to balance instruction-level parallelism against spilling and accept some spills, over time newer algorithms aimed more and more at eliminating spilling as far as possible (Govindarajan et al. 2003). In CRISP, this trade-off is captured in two model parameters that can be varied to study the problem space.

The heuristic CRISP scheduler is based on a standard list scheduler with a rank function. Using a standard rank function γ_S and a special register rank function γ_R that is meant to capture live range lengths, scheduling is performed with the new combined rank function $\gamma = \alpha\,\gamma_S + \beta\,\gamma_R$ with the parameters $\alpha, \beta \in [0, 1]$ and $\alpha + \beta = 1$. Simple experiments on random DAGs show that setting $\beta = 1$ (i. e., focusing on reducing live range lengths) is effective in reducing the number of spills versus aggressive list scheduling.

In the context of this thesis, this work is mostly notable because GCMS uses a broadly similar model to balance global code motion and spilling (see Section 4.4.2).

2.3 Integrated code motion and register allocation

All of the work mentioned above considers only local instruction scheduling within basic blocks, but no global code motion. At an intermediate level between local and global approaches, software pipelining (Lam 1988) schedules small loop kernels for optimal execution on explicitly parallel processors. Here, too, careful integration of register allocation has proved important over time (Codina, Sánchez, and González 2001; Eriksson and Kessler 2012).

The following sections discuss the more notable (partly) global integrated allocators in more detail.

2.3.1 RASER

RASER (Norris and Pollock 1995b) performs register allocation sensitive region scheduling. It is based on a region scheduling algorithm which uses a program dependence graph of hierarchically nested program regions. The algorithm estimates the amount of parallelizable computations in each node and tries to move code from regions with too many parallel computations to regions where the parallelism is not as high as the machine can accommodate.

RASER extends this by also computing the register demands of regions after applying IPS to schedule them. In regions with excess register pressure, RASER attempts to reduce the number of live values by duplication of computations. If possible and profitable (because it would not increase register pressure), the definition of each value used in such a region is duplicated before each use. The number of live values is updated on the fly.

After reducing register pressure below the limit wherever possible, RASER still allows normal code motion between regions, but only if they will not raise the register pressure above the register limit. Like many other approaches, RASER gives impressive improvements on machines with artificially few registers; later results on a machine with 16 registers are much more limited and more similar to ours, up to 3 % at most (Norris and Pollock 1995a).

2.3.2 VSDG algorithm

N. Johnson and Mycroft (2003) describe an elegant combined global code motion and register allocation method based on the Value State Dependence Graph (VSDG). The VSDG is similar to the acyclic global dependence graph used by GCMS, but it represents control flow by using special nodes for conditionals and reducible loops (their approach does not handle irreducible loops) rather than our lists of legal blocks for each instruction. The graph is traversed bottom-up in a greedy manner, measuring 'liveness width', the number of registers needed at each level. Excessive register pressure is reduced by adding dependence arcs, by spilling values, or by duplicating computations. Unfortunately, we are not aware of any data on the performance of this allocator, nor the quality of the generated code.

The concept of liveness width is similar to Touati's 'register saturation', which is only formulated for basic blocks and pipelined loops. It is natural to try to adapt this concept to general control flow graphs, but this is difficult to do if instructions may move between blocks and into and out of loops. It appears that to compute saturation, we would need to build a detailed model of where each value may be live, and this might quickly lead to combinatorial explosion. GCMS is simpler because it tries to minimize overlaps without having to take a concrete number of available registers into account.

2.3.3 Machine learning

Lokuciejewski et al. (2010) use machine learning to derive heuristics for loop-invariant code motion. Based on 73 different features such as instruction type, number of values live into, live out of, defined, and used in basic blocks, and loop nesting level, various models for moving code are trained on a benchmark set. The quantity to be optimized is the programs' worst-case execution time (WCET), which is estimated using standard tools. The authors find that their best model can sometimes substantially reduce WCET when compared to naïve loop-invariant code motion, with an average improvement of 4.6 % on a processor with 16 general-purpose registers.

Unfortunately, the authors do not give any indication which program features appear most relevant for deciding whether to move any particular loop-invariant computations.

2.3.4 Partly global approaches

Many authors have worked on what they usually refer to as global instruction scheduling problems, but their solutions are almost invariably confined to acyclic program regions, i. e., they do not perform loop invariant code motion (Bernstein and Rodeh 1991; Zhou, Jennings, and Conte 2003). The notable exception is work by Winkel (Winkel 2007) on 'real' global scheduling including moving code into and out of loops, as our algorithm does. Crucially, Winkel's optimal scheduler runs in two phases, the second of which has the explicit goal of limiting code motion to avoid lengthening live ranges too much. Besides considerable improvements in schedule length, Winkel reports reducing spills by 75 % relative to a heuristic global scheduler. In contrast to our work, Winkel compiled for the explicitly parallel Itanium processor, so his reported speedups of 10 % cannot be meaningfully compared to our results on our out-of-order target architecture (ARM Cortex-A9).

2.4 GCMS

The GCMS algorithm presented in this thesis evolved in several steps. The initial phase was *register reuse scheduling* (Barany 2011). This algorithm was already based on the idea of computing all possible schedules and their impact on live range overlaps, and letting the PBQP register allocator's results determine how to modify the dependence graph to ensure that the final schedule is valid for the chosen allocation. Register reuse scheduling was a purely local approach. Candidate selection was performed using the greedy heuristics also used by GCMS. Simulator results showed that this algorithm was successful at reducing spills versus LLVM's baseline heuristics, which also try to schedule for minimal register pressure.

GCMS (Barany and Krall 2013) then arose as the generalization of register reuse scheduling to global code motion. Based on the same principles as register reuse scheduling, the overlap analysis was lifted to the global dependence graph. GCMS allows both optimal and heuristics candidate selection and fully global code motion along paths in the dominance tree. The results for GCMS showed very slight speedups over LLVM. In

retrospect, this was at least in part due to inaccuracies in the overlap analyses implemented at the time. Additionally, the optimal solver was ony applied to relatively small functions (up to 1000 instructions) and with a low time limit (60 seconds per instance). This thesis reports better results because these shortcomings have been resolved.

CHAPTER 3

Live Range Overlap Analysis

This chapter discusses one of the two major challenges in integrating global code motion with register allocation: the problem of analyzing which live ranges might overlap in the program if the final arrangement of instructions is not known. After a discussion of traditional overlap analysis on a fixed program, a novel way of analyzing overlaps in the presence of code motion is presented.

3.1 Traditional overlap analysis

The live ranges of two values v and w overlap at a program point if they are both live at that point. Compilers traditionally compute the sets of all program points where each value is live; these sets are variously called *live ranges* or *live intervals*.

Since values are live at all points where their current value might be used in the future, live ranges can be computed using a simple data flow analysis that proceeds backwards (i. e., against the direction of control flow). In this analysis values become live at each use, and liveness is propagated up towards the point of definition, computing fixed points for loops (F. Nielson, H. R. Nielson, and Hankin 1999). The analysis is applicable to all programs; faster liveness analysis algorithms are also available specifically for programs in SSA form (Boissinot et al. 2011).

Live ranges overlap if and only if their intersection is non-empty. For a function containing n values, an interference-graph based register allocation algorithm such as PBQP must check $O(n^2)$ pairs of live ranges for overlap. These checks should therefore be very fast in practice. Common representations for live ranges include bit vectors with one bit per program point where the value is live, or sets of intervals of such program points. Alternatively, the liveness analysis only retains information on which basic blocks each value is live out of. An interference graph can then be reconstructed in an additional backward pass over each block (Cooper and Torczon 2004, section 13.5).

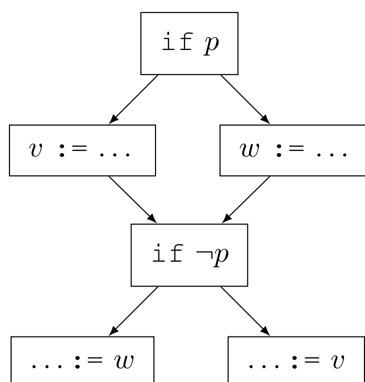


Figure 3.1: Nontrivial live range overlap in a non-SSA program.

3.2 Overlap analysis in the presence of code motion

The traditional methods described above all work on program representations in which the arrangement of instructions is fixed. They are therefore not directly suitable for integrating register allocation and code motion: The information computed by traditional liveness analysis would be invalidated by changes to the ordering or placement of instructions.

We must therefore develop an alternative formulation for the analysis of overlaps between live ranges in SSA form that does not refer to explicit live-out sets or sets of live program points.

3.2.1 Characterization of overlaps in SSA form

Recall that one of the defining characteristics of (strict) SSA form is that a value v's definition always dominates all of its uses. Dominance means that any path from the function's start to a use of v must pass through the definition. Every point of v's live range is on some path that leads to a use of v, after the path has passed the definition. In other words, an SSA value's definition dominates all points of its live range.

As a consequence, in SSA form, the live ranges of values v and w overlap only if v's definition dominates w's definition or vice versa: Assume the live ranges intersect at some program point p. By the above, $v_{\text{def}} \succeq_{\text{dom}} p$ and $w_{\text{def}} \succeq_{\text{dom}} p$, but since any point's dominators are strictly ordered by dominance, either $v_{\text{def}} \succeq_{\text{dom}} w_{\text{def}}$ or $w_{\text{def}} \succeq_{\text{dom}} v_{\text{def}}$ must hold.

Assume that the live ranges overlap and that without loss of generality $v_{\text{def}} \succeq_{\text{dom}} w_{\text{def}}$. Then w's definition is part of v's live range since there is a program path that leads from v_{def} to w_{def} and further via p to some use of v. Cooper and Torczon (2004) give this latter property ('[v and w] interfere if one is live at the definition of the other') as the definition of live range interference in a setting that implicitly assumes that the program is in SSA form, without further justifying this definition as was done here.

This property considerably simplifies the overlap analysis for programs in SSA form because cases as illustrated in Figure 3.1 cannot occur in SSA form. In this example,

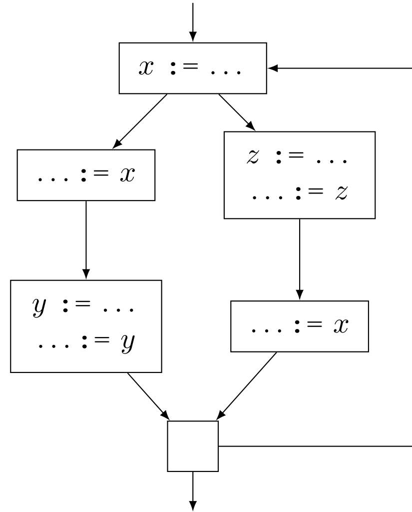


Figure 3.2: Illustration of the overlap criterion for SSA programs. The live ranges x and z overlap, but neither overlaps with y.

the live ranges of v and w overlap although neither live range contains an instruction mentioning the other one. In SSA form, the block merging the execution paths would have to contain ϕ instructions defining new variants of v and w, which would therefore overlap. Based on these observations we can give the following lemma that characterizes when values overlap without referring to liveness analysis.

Lemma (Overlap criterion for SSA form under code motion)**.** *Let v, w be SSA values in a program represented as a global acyclic dependence graph with GCM placement information. The live ranges of v and w overlap in exactly those legal arrangements of instructions where $v_{\text{def}} \succeq_{\text{dom}} w_{\text{def}}$ and there is a path from w_{def} to some use of v that does not pass through v_{def} (or vice versa).*

Proof. Assume first that the live ranges of v and w do overlap. By the arguments above, in SSA form this means that without loss of generality $v_{\text{def}} \succeq_{\text{dom}} w_{\text{def}}$ and v is live at w_{def}. But this means that there is a program path from w_{def} to some use of v without an intervening redefinition of v.

For the other direction, assume again without loss of generality that $v_{\text{def}} \succeq_{\text{dom}} w_{\text{def}}$ and there is a path from w_{def} to v_{use} without a redefinition of v. Then v is live at w_{def} since that point is on a path from v's definition to a use without an intervening redefinition. Furthermore, every value is live at its point of definition. Thus v and w are both live at w_{def}, i. e., their live ranges overlap. □

Figure 3.2 illustrates some aspects of this criterion. There are three live ranges x, y, and z, and thus pairs of live ranges to check for overlaps. Consider first the case of x and y. The definition of x dominates the definition of y, so part of the overlap criterion

is satisfied. However, all paths from y's definition to any use of x must first pass through the loop's back edge and thus through the definition of x. Thus there is no overlap between the live ranges x and y.

There is an overlap between x and z because the entire overlap criterion is satisfied: The definition of x dominates the definition of z. Further, any program path continuing from z's definition must pass through the use of x in the following block. Finally, it is easy to see using the overlap criterion that the live ranges y and z do not overlap since neither live range's definition dominates the other's definition.

The overlap criterion allows us to talk about overlapping live ranges in terms of the relative arrangements of just a few important instructions. The conditions for the overlap of live ranges can be checked using just a few reachability queries on the control flow graph and the global acyclic dependence graph. Thus the analysis of live range overlaps can be performed even in the presence of code motion, which precludes general liveness analysis.

3.2.2 Exhaustive overlap analysis

An optimal solution to GCMS requires us to consider all possible ways in which a pair of values might overlap. That is, we must consider all possible placements and orderings of all of the instructions defining or using either value. To keep this code simple, we implemented this part of the analysis in Prolog. This allows us to give simple declarative specifications of when values overlap, and Prolog's built-in backtracking takes care of actually enumerating all configurations.

The core of the overlap analysis, simplified from our actual implementation, is sketched in Figure 3.3 on page 29. The Figure shows the two most important cases in the analysis: The first clause deals with the case where values ('virtual registers') A and B might overlap because A's use is in the same block as B's definition, but there is no dependence ensuring that A's live range ends before B's definition. The second clause applies when A is defined and used in different blocks, and B's definition might be placed in an intervening block between A's definition and use.

The code uses a few important auxiliary predicates for its checks:

cfg_dominates(A, B) succeeds if basic block A dominates block B in the control flow graph.

cfg_path_notvia(A, B, C) succeeds if there is a path, possibly including loops, from A to B, but not including C. We use this to check for paths lacking a redefinition of values.

no_dependence(A, B) succeeds if there is no arc in the dependence graph from instruction A to B (which would cause A to be scheduled after B), but it could be added without causing a cycle in the graph.

If all of the conditions in the clause bodies are satisfied, a possible overlap between the values is recorded. Such overlaps are associated with 'blame terms', data structures

```
overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % B is defined by instruction BDef in BDefBlock, A has a use in the same
    % block.
    virtreg_def_in(B, BDef, BDefBlock),
    virtreg_use_in(A, AUse, BDefBlock),
    % A's use is not identical to B's def, and there is no existing
    % dependence from B's def to A's use. That is, B's def might be between
    % A's def and use.
    AUse \= BDef,
    no_dependence(BDef, AUse),
    % There is an overlap that might be avoided if B's def were scheduled
    % after A's use by adding an arc.
    Placement = [AUse-BDefBlock, BDef-BDefBlock],
    record_blame(A, B, blame(placement(Placement), no_arc([BDef-AUse]))).

overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % A and B have defs ADef and BDef in blocks ADefBlock and BDefBlock,
    % respectively.
    virtreg_def_in(A, ADef, ADefBlock),
    virtreg_def_in(B, BDef, BDefBlock),
    % A has a use in a block different from its def.
    virtreg_use_in(A, AUse, AUseBlock),
    ADefBlock \= AUseBlock,
    % A's def dominates B's def...
    cfg_dominates(ADefBlock, BDefBlock),
    % ... and there is a path from B's def to A's use that does not pass
    % through a redefinition of A. That is, B's def is on a path from A's
    % def to its use.
    cfg_path_notvia(BDefBlock, AUseBlock, ADefBlock),
    % There is an overlap that might be avoided if at least one of these
    % instructions were in a different block.
    Placement = [ADef-ADefBlock, BDef-BDefBlock, AUse-AUseBlock],
    record_blame(A, B, blame(placement(Placement))).
```

Figure 3.3: Exhaustive overlap analysis for virtual registers A and B

that capture the reason for the overlap. For any given pair of values, there might be several different causes for overlap, each associated with its own blame. An overlap can be avoided if all of the circumstances captured by the blame terms can be avoided.

There are two kinds of blame. First, there are those blames that record arcs missing from the dependence graph, computed as in the first clause in Figure 3.3. If this arc can be added to the dependence graph, B's definition will be after A's use, avoiding this overlap. Alternatively, if these two instructions are not placed in the same block, the overlap is also avoided. The second kind of blame concerns only the placement of instructions in basic blocks, as in the second clause in Figure 3.3. If all of the instructions are placed in the blocks listed in the blame term, there is an overlap between the live ranges. If at least one of them is placed in another block, there is no overlap—at least, not due to this particular placement.

As mentioned before, we use Prolog's backtracking to enumerate all invalid placements and missing dependence arcs. We collect the associated blame terms and check them for validity: If any of the collected arcs to put a value v before w can not be added to the dependence graph because it would introduce a cycle, then the other arcs for putting v before w are useless, so all of these blames are deleted. Blames for the reversed ordering, scheduling w before v, are retained because they might still be valid.

Even after this cleanup we might end up with an overlap that cannot be avoided. For example, for the pair `a` and `b` in the example program, the analysis computes that instruction 3 defining `b` may not be placed in the start block because it would then be live out of that block and overlap with `a`'s live-out definition; but neither may instruction 3 be placed in the loop block because it would be on a path from `a`'s definition to its repeated use in the loop. As these two blocks are the only ones where instruction 3 may be placed, the analysis of all blames for this pair determines that an overlap between `a` and `b` cannot be avoided.

Blame terms can be expressed as simple formulas of propositional logic over variables of the form $place_{i,b}$, meaning that some instruction i is placed in basic block b, and $arc_{i,j}$, meaning that instruction j occurs before instruction i in the same block because there is an arc $i \to j$ in the dependence graph.

Expressed in this logical formalism, we have the following overlap condition for the pair $\langle \mathtt{b}, \mathtt{c} \rangle$:

$$overlap_{\mathtt{b},\mathtt{c}} = place_{def_{\mathtt{b}},\mathtt{start}} \lor \left(place_{def_{\mathtt{b}},\mathtt{loop}} \land \neg arc_{def_{\mathtt{c}},use_{\mathtt{b}}} \right)$$

The first disjunct states that these two live ranges overlap if `b`'s definition is placed in the `start` block, since it is then live through the loop where `c` is defined. The second conjunct states that even if `b`'s definition is in the `loop` block, an extra ordering constraint is needed. Note that in Figure 1.1 there is no dependence path between instructions 4 (the use of `b`) and 5 (the definition of `c`). If the overlap is to be avoided, an arc must be added to ensure that in any legal schedule for this basic block, the use killing `b` precedes `c`'s definition.

As another example, the overlap between `j` and `d` was removed by ensuring that `d` may not be defined inside the loop. Otherwise `d`'s definition would be on a path from

Table 3.1: Blame terms computed for the example program, listing instruction placements and missing dependence arcs that may cause overlaps.

Pair	Invalid placements	Missing arcs	Overlap formula
a, d	i7 in loop		$place_{\mathtt{i7,loop}}$
b, c	i3 in start		$place_{\mathtt{i3,start}}$
b, c	i3 in loop	$5 \to 4$	$place_{\mathtt{i3,loop}} \wedge \neg arc_{\mathtt{i5,i4}}$
b, d	i3 in start, i7 in loop		$place_{\mathtt{i3,start}} \wedge place_{\mathtt{i7,loop}}$
b, j0	i3 in start		$place_{\mathtt{i3,start}}$
b, j2	i3 in start		$place_{\mathtt{i3,start}}$
c, d	i7 in loop	$7 \to 6$	$place_{\mathtt{i7,loop}} \wedge \neg arc_{\mathtt{i7,i6}}$
c, j1		$5 \to 4$	$\neg arc_{\mathtt{i5,i4}}$
d, j2	i7 in loop		$place_{\mathtt{i7,loop}}$

the definition of j2 across the loop block's end back up along the loop edge to the ϕ use of j2. The corresponding formula is:

$$overlap_{\mathtt{j,d}} = place_{def_{\mathtt{d}},\mathtt{loop}}$$

Table 3.1 shows all the blame formulas for the avoidable live range overlaps in the running example after some cleanup (removal of unmovable instructions from placement blames). Instructions (in) are numbered as in the dependence graph in Figure 1.1. Pairs not listed here are found to be either non-overlapping or definitely overlapping. Note that there are two entries in the table for the pair $\langle \mathtt{b}, \mathtt{c} \rangle$ because, as discussed above, there are two different placements of instructions that lead to an overlap between these two live ranges.

3.2.3 Correctness of exhaustive overlap analysis

The intention of exhaustive overlap analysis is to enumerate, given a global dependence graph with legal code motion information, all the possible arrangements of instructions under which a pair of live ranges may overlap. The following lemmas establish that the analysis as described above achieves this goal.

The statements and proofs of these lemmas make some simplifying assumptions about the program. These do not necessarily hold in real programs, but these corner cases are handled in the actual implementation of the analysis. The assumptions are that each instruction defines at most one virtual register and that every virtual register has at least one use. The analysis as presented here also does not mention ϕ instructions. It assumes a program representation in which each operand of a ϕ is represented by a pseudo-use at the end of the appropriate predecessor block. This allows a simple, uniform presentation of the algorithm.

Note that the analysis is asymmetric. An overlap between two live ranges v and w may sometimes be avoidable in two ways, by scheduling all of v before w's definition or all

of w before v's definition. It is useful to keep the blame sets for these two options separate. The analysis therefore applies to ordered pairs $\langle v, w \rangle$ of live ranges and computes blames under the assumption that v's definition may dominate w's definition in some final schedule.

For any two live ranges v and w, the analysis is potentially run twice: Once for the ordered pair $\langle v, w \rangle$ if an arc from w's definition to v's definition may be added to the dependence graph (ensuring that v's definition dominates w's definition), and once for the pair $\langle w, v \rangle$ if an arc from v's definition to w's definition can be added.

Lemma (Soundness of exhaustive overlap analysis)**.** *If exhaustive overlap analysis produces a blame term for virtual registers A and B, then that blame term describes a legal arrangement of instructions in the given program in which the live ranges of A and B overlap if A's definition is forced to dominate B's definition.*

Proof. There are two cases to consider, one for each of the clauses of the analysis. The first clause applies if some use of A is in the same basic block *BDefBlock* as B's definition (and these are not the same instruction). The blame describes a schedule in which there is no dependence from B's definition to the use of A, i. e., a schedule in which B's definition precedes A's use. If A's definition is forced to dominate B's definition, B is thus defined at a point where A is live, so the live ranges overlap.

The second clause produces a blame describing a case where A's definition is in a block which dominates the block containing B's definition, and there is a (possibly empty) path from that block to some block containing a use of A that does not pass through A's definition. This directly establishes the overlap criterion. □

Lemma (Completeness of exhaustive overlap analysis)**.** *If there is a legal arrangement of instructions in the given program in which the live ranges of A and B overlap, then exhaustive overlap analysis applied to both of the ordered pairs $\langle A, B \rangle$ and $\langle B, A \rangle$ produces a blame term for virtual registers A and B that describes this arrangement.*

Proof. Assume that A and B overlap according to the overlap criterion and (without loss of generality) $A_{\mathrm{def}} \succeq_{\mathrm{dom}} B_{\mathrm{def}}$.

Consider the case where A's definition and a use that fulfills the overlap criterion are in the same basic block. Then B's definition must be in the same block between these two instructions: There are no dependence arcs forcing B's definition before A's definition or after A's use. The first clause of the analysis applies and produces a blame term that captures this arrangement of instructions.

Otherwise, A's definition is in a different block from a use that fulfills the overlap criterion. The second clause of the analysis applies. The checks for dominance and a path without a redefinition of A are guaranteed to succeed due to the overlap condition, so a blame term describing this arrangement of instructions is produced. □

3.2.4 Complexity of exhaustive overlap analysis

Using the overlap criterion allows the exhaustive overlap analysis algorithm to only enumerate those features of program schedules that make a relevant difference to the

live ranges it is currently analyzing. Therefore, while the total number of possible arrangements is exponential in the size of the program, the algorithm can still analyze all possible overlaps in polynomial time.

In particular, the complexity of the algorithm on a function containing n instructions can be bounded as follows:

Algorithm step	Complexity
for each pair $\langle a, b \rangle$:	$O(n^2)$
for each legal block for a's def:	$O(lb(n))$
for each legal block for b's def:	$O(lb(n))$
for each use u of a:	C (average case)
for each legal block for u:	$O(lb(n))$
check overlap criterion	$O(n^2)$
total	$O(n^4\, lb(n)^3\, C)$

The individual parts of the analysis can be explained as follows. First, the number of live ranges in the function is $O(n)$, so any register allocator based on a conflict graph representation, such as graph coloring and PBQP, has a base complexity of at least $O(n^2)$ because it has to consider each pair of live ranges. This is the cause for the corresponding entry in the first line of the table.

In the following lines, $lb(n)$ denotes the number of legal blocks for some instruction in the program. Since in GCM instructions can only move to blocks that dominate, or are dominated by, their original block, this means that each instruction is constrained to a single path in the dominator tree. Thus $lb(n)$ is bounded by the depth of the dominator tree. It is possible to construct programs with a linear dominator tree of depth $O(n)$, although in practice, due to branching in real programs, it typically appears to be $O(\log n)$. Furthermore, almost all instructions in real programs are considerably constrained in their actual code motion freedom. The only instructions that can actually move freely are ones without dependences on any other instruction; these are only those instructions that load constants into registers. For all other instructions, the number of legal blocks appears to depend only very weakly on n. On average in real programs, we observed $lb(n) \leq 3$ for almost all instructions.

The constant bound C on the average number of uses of each value is a property of the instruction set (the maximum number of register operands of any instruction) and is independent of program size. If the program consists of n instructions, each defining a value and using at most C operands, the total number of uses is at most $n\,C$, and the average number of uses per value is C. In practical instruction sets, $C \leq 3$ typically holds.

Finally, checking the overlap criterion for a given placement of instructions amounts to checking a dominance query, which can be done in (amortized) constant time, and performing one depth-first search in the control flow graph or up to two depth-first searches in the dependence graph. This step is clearly $O(n^2)$. However, the intermediate and final results of each search can be cached and reused in future searches, which means

that the full quadratic cost does not have to be paid every time the overlap criterion is checked.

Overall, applying the observations above allows us to conclude that the average case complexity of the algorithm is between $O(n^2)$ (with a larger constant than traditional conflict graph construction) and $O(n^4)$.

In practice, the Prolog implementation of the analysis is able to analyze functions containing hundreds of instructions within seconds and thousands of instructions within a few minutes. Caching in the Prolog implementation could be improved further by more engineering, which should result in some speedup of the analysis.

3.3 Greedy overlap analysis

Sometimes the exhaustive overlap analysis does more work than necessary; rather than finding all possible ways of avoiding an overlap between live ranges, one might only be interested in a single way. This is the case for the heuristic selection procedure to be discussed in Section 4.2.

A greedy analysis can search the program for a way to avoid overlaps between live ranges and commit to each such avoided overlap by immediately modifying the program, thus restricting future code motion possibilities. The core of this analysis, sketched as Python-like pseudocode in Figure 3.4, achieves this by identifying 'bad paths' in the program and eliminating them, if possible, by sinking definitions, hoisting uses, and adding dependence arcs. The bad paths are just those identified in the overlap criterion above: Paths from one live range's definition to one of its uses that contain the other live range's definition. Whether hoisting of *R1*'s uses is allowed is controlled by the externally specified *allowUseHoisting* parameter. This operation is conditional because it may lengthen other live ranges too much.

The *schedulableBefore* function is called for each ordered pair of live ranges that might be allocated to the same physical register. It determines whether it is possible to arrange the program such that live range *R1* cannot include, nor be preceded by, *R2*'s definition. In the code, the *firstBlock* and *lastBlock* functions return, respectively, the first and last legal basic blocks for the given instruction, while the *sink* function eliminates the current first legal block and returns the next one.

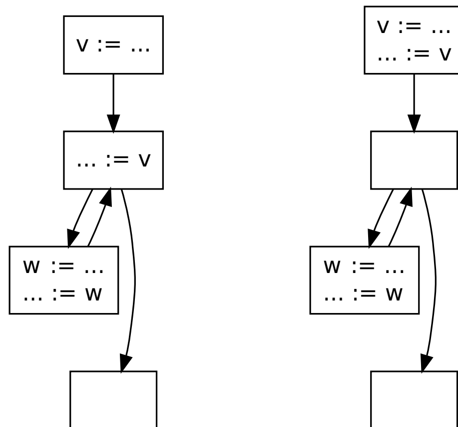
The *forwardPath* function returns *True* if and only if there is a path using only forward edges in the control flow graph between the given nodes. The path may be empty, i. e., *forwardPath(b, b)* is *True* for all b. The *loopyPath(a, b, not_via)* function returns *True* if and only if there is a path using any (forward or back) edges from a to b that does *not* pass through the *not_via* block. This function only returns *True* for non-empty paths, and returns *True* for the case $b =$ *not_via* if there is a path from a to b. It is used to determine the overlap condition established above: Does *R2*'s definition lie on a path from *R1*'s definition to a use without an intervening redefinition?

Finally, the *okBefore(a, b)* function checks whether the dependence graph allows to schedule a before b, i. e., there is not already a dependence path from a to b, and *add-*

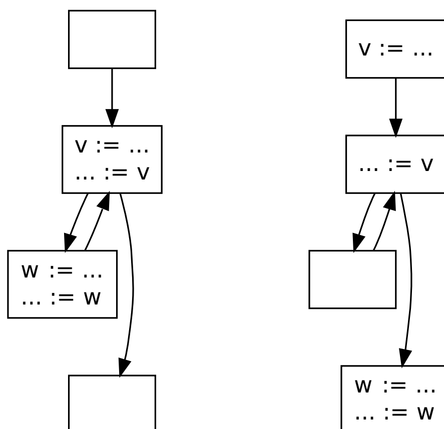
```
def schedulableBefore(R1, R2):
    # Check if an overlap between R1 and R2 can be avoided. If the live ranges may interfere, make a
    # greedy attempt to put R2's live range after R1's live range. First, check if R2 must be before R1.
    R1DefBlock = firstBlock(R1.Def)
    R2LastDefBlock = lastBlock(R2.Def)
    if ((R2LastDefBlock != R1DefBlock and forwardPath(R2LastDefBlock, R1DefBlock)) or
        (R2LastDefBlock == R1DefBlock and not okBefore(R1.Def, R2.Def))):
        # R2 is defined before R1 in the control flow graph or dependence graph.
        return False
    # Constrain placement of R2's definition so that it cannot be before R1's definition.
    R2DefBlock = firstBlock(R2.Def)
    while R2DefBlock != R1DefBlock and forwardPath(R2DefBlock, R1DefBlock):
        R2DefBlock = sink(R2.Def)
    if allowUseHoisting:
        # Try to hoist uses of R1 before the definition of R2.
        for (R1Use, R1FirstUseBlock) in R1.FirstUsesAndBlocks:
            if not badPath(R1DefBlock, R1FirstUseBlock, R2DefBlock):
                R1UseBlock = lastBlock(R1Use)
                while badPath(R1DefBlock, R1UseBlock, R2DefBlock):
                    R1UseBlock = hoist(R1Use)
    R1LastDefBlock = lastBlock(R1.Def)
    # For each use of R1, scheduled as late as possible, identify bad paths according to the overlap
    # criterion and try to fix them by sinking R1's or R2's definition.
    for R1UseBlock in R1.LastUseBlocks:
        # Sink R1's definition if that could avoid the overlap.
        if not badPath(R1LastDefBlock, R1UseBlock, R2DefBlock)):
            while badPath(R1DefBlock, R1UseBlock, R2DefBlock)):
                R1DefBlock = sink(R1.Def)
        # Similarly, sink R2's definition if possible and necessary.
        if not badPath(R1DefBlock, R1UseBlock, R2LastDefBlock):
            while badPath(R1DefBlock, R1UseBlock, R2DefBlock):
                R2DefBlock = sink(R2.Def)
        else:
            # R2 cannot be sunk to any block that avoids this path.
            return False
    # If R2's definition and some use of R1 might be placed in the same block, add a
    # dependence graph arc that ensures that the use is before the definition.
    for R1Use in R1.Uses:
        if forwardPath(firstBlock(R1Use), R2DefBlock) and not addDependence(R2.Def, R1Use):
            # dependence arc cannot be added
            return False
    # If we got here, we succeeded in placing R2 after R1's live range (or
    # on an independent path).
    return True

def badPath(R1DefBlock, R1UseBlock, R2DefBlock):
    # Determine whether the given blocks for R1's definition, a use of R1, and R2's
    # definition are on a 'bad' path as defined by the overlap criterion.
    if (R2DefBlock != R1DefBlock and forwardPath(R2DefBlock, R1DefBlock)):
        # R2's definition is before R1's definition.
        return True
    if (R2DefBlock != R1UseBlock and forwardPath(R2DefBlock, R1UseBlock)):
        # R2's definition is on a straight path between R1's definition and the use.
        return True
    if (R1DefBlock != R1UseBlock and loopyPath(R2DefBlock, R1UseBlock, not_via = R1DefBlock)):
        # R2's definition is on a path that reaches R1's use without an intervening redefinition of R1.
        return True
    # Otherwise, R2's definition does not interfere with R1's live range.
    return False
```

Figure 3.4: Greedy overlap analysis for live ranges R1 and R2



(a) Live ranges overlap (b) Hoist v out of loop



(c) Sink v into loop (d) Sink w out of loop

Figure 3.5: Some possible code motions to avoid overlap of v and w

Dependence(a, b) adds a new dependence arc from a to b if possible and returns *False* otherwise.

These auxiliaries are used by *schedulableBefore* to first sink *R2*'s definition to a point that is not before *R1*'s definition if possible, starting from their earliest possible placements. This is followed by iterating code motion steps corresponding to the operations illustrated in Figure 3.5. First, uses may be hoisted to earlier blocks as in Figure 3.5b. In the

algorithm in Figure 3.4, this is the operation that is guarded by the *allowUseHoisting* parameter. Then, for each use of *R1* (placed as late as still possible), the algorithm tries in turn to sink *R1*'s and *R2*'s definitions as in Figures 3.5c and 3.5d if these code motions can eliminate a 'bad path' for the use. Finally, if all of these control flow restrictions could be enforced, *R2*'s definition might still end up in the same block as one of *R1*'s uses. In this case, we add dependence arcs between the instructions to ensure that the use ends *R1*'s live range before *R2* is defined. The correctness of the algorithm is essentially due to the fact that the repeated checks for bad paths encode exactly the overlap criterion. If the algorithm finds no arrangement of instructions without bad paths, it returns *False*, meaning that it cannot rule out an overlap of the live ranges.

The analysis is greedy in the sense that if *schedulableBefore* succeeds for a pair of live ranges, that result is committed, and no alternative code motions are tried. If *schedulableBefore(R1, R2)* fails, the analysis retries by calling *schedulableBefore(R2, R1)*. Whenever *schedulableBefore* returns *False*, its code motion decisions (sinking or hoisting instructions, adding dependence arcs) are undone before moving on the the next pair of live ranges to analyze.

Revisiting the running example of Figure 1.1 for values `b` and `c` in terms of the pseudocode algorithm, we see that `b`'s definition has to be sunk into the loop to avoid an overlap. The path from its earliest definition in the `start` block to its use in `loop` is 'bad' with respect to `c`'s definition in `loop`: Execution can go around the loop without a redefinition of `b` between a definition of `c` and another use of `b`. Sinking the definition into the loop removes this bad path. Afterwards, a dependence arc from the call instruction has to be added to ensure that `c` is defined strictly after `b`'s use. The overlap between `j` and `d` can be removed by ensuring that `d` may not be defined inside the loop.

3.4 Handling of two-address instructions

Some machine language instructions are not directly suitable for an SSA-based program representation. This is the case for instructions that use the same register operand as both a source and a destination operand; i. e., they modify one of their inputs. Many arithmetic instructions in the x86 instruction set have this property. For example, the x86 instruction `ADD EAX, EDX` will add the values in the input registers `EAX` and `EDX` and store the new value in `EAX`, overwriting the input value. Other instruction sets have fewer such two-address instructions, although ARM's conditional move instruction `MOVCC` is an important exception: Depending on the value of the condition code register, it will either overwrite its destination register with the value in the source register or leave it unchanged.

A naïve representation of such instructions with virtual registers would not be in SSA form. Consider the following program fragment:

```
ADD v, ...          // define v
CMP ...             // set condition code register
MOVCC v, #0         // conditionally move 0 into v
```

This program is not in SSA form since it has two definitions for the same virtual register v. One solution (implemented in LLVM) is to add an extra virtual operand to two-address instructions to separate the use of the operand from its redefinition. For the purposes of register allocation, an extra constraint is recorded that the virtual use and the virtual definition operand must be allocated to the same CPU register. The example thus becomes:

```
ADD v, ...              // define v
CMP ...                 // set condition code register
MOVCC w, v, #0          // conditionally move v or 0 into w
```

This fragment is now in SSA form, and the instruction is annotated with metadata instructing the register allocator to assign both v and w to the same CPU register. Further complications arise, however, if v has several uses: Since the two-address instruction kills v's live range, it must be the last one of the uses. This puts a constraint on code motion. We could handle this either by restricting code motion to enforce this constraint, or by making sure that such instructions always refer to operands that have no other uses. We choose to do the latter by always introducing a copy of the operand to be modified before a two-address instruction:

```
ADD v, ...              // define v
CMP ...                 // set condition code register
MOV u, v                // copy v to new virtual register u
MOVCC w, u, #0          // conditionally move u or 0 into w
```

Other uses of v are now unaffected by any changes the MOVCC instruction makes to its operand u/w. The downside is that for code using a large number of two-address instructions, the added copies will cause a large increase in code size and thus also in the complexity of GCMS overlap analysis. Thus this choice is probably not the best option for an architecture such as x86, but it works well for ARM, where these instructions do not appear too frequently.

3.5 Overlap analysis for preallocated registers

The discussion so far has only considered virtual registers. In real programs, some instruction operands are preallocated to certain CPU registers. In particular, function call instructions implicitly use argument registers and define return registers. They are usually surrounded by instructions that define these arguments by copying values into them, or that copy the return registers to virtual registers. Similarly, at the start of each function its own arguments are copied from predetermined registers, and at each return predetermined registers may be written to.

Some architectures have other kinds of preallocated registers as well. A prominent example is the x86 integer division instruction DIV which (in the 32-bit case) always

reads the dividend from the concatenation of the registers EDX and EAX and always produces the quotient in EAX and the remainder in EDX.

A live range overlap between a preallocated register p and a virtual register v means that the register allocator may not assign v to p. We must therefore analyze such overlaps in order to be able to build a precise and correct register allocation problem.

Since instructions using or defining preallocated registers can occur multiple times in a single function or even a single basic block, these registers have multiple definitions and are thus not in SSA form. Recall, however, that we assume an input program representation in which a definition of a preallocated register is always in the same basic block as all of its uses. This allows us to treat each preallocated register p as a collection of short individual live ranges. A virtual register v overlaps with p if it overlaps with any of its individual live ranges.

We can therefore use the same analysis that is used to analyze overlaps between the live ranges of two virtual registers. In practice, however, many preallocated registers have so many individual live ranges that the number of possible ways of avoiding an overlap would lead to combinatorial explosion. In our implementation we therefore restrict the analysis of virtual-preallocated overlaps based on the input program's prepass schedule. The algorithm proceeds as follows. Let B be the original basic block containing the virtual register v's definition v_{def}. Let C be a candidate basic block into which v_{def} might be moved by global code motion. There are three cases to consider:

- $C \succeq_{\mathrm{dom}} B$ and $C \neq B$: only consider blames that would schedule v_{def} after all live ranges of p in C
- $C = B$: if in the prepass schedule p's definitions $p_1, \ldots, p_i$ precede v_{def} and v_{def} precedes p's definitions $p_j, \ldots, p_n$, only consider blames that schedule v_{def} after p_i and before p_j
- $B \succeq_{\mathrm{dom}} C$ and $C \neq B$: only consider blames that would schedule v_{def} before all live ranges of p in C

In other words, preserve the prepass schedule's relative ordering of v_{def} to p's live ranges in v_{def}'s original block or whichever other blocks it may be moved to. Figure 3.6 illustrates these restrictions, with all legal positions for v's definition shaded in gray. In the prepass schedule, v's definition is placed in the third block between two live ranges of the preallocated register p. In the two blocks preceding the original location of the definition, v's definition may only occur after all of p's live ranges; in the block after the original definition, v may only be defined before p's live ranges.

This restriction has the property that if there is no overlap between v and p in the prepass schedule, a blame is produced that allows us to avoid an overlap 'in the same way' even when code motion is enabled. However, this analysis is not complete, i. e., other ways of avoiding the overlap might not be recognized. For some final schedules of the program, the analysis might conclude conservatively that v and p overlap even if that is not actually the case. We have not observed this to be a problem in practice.

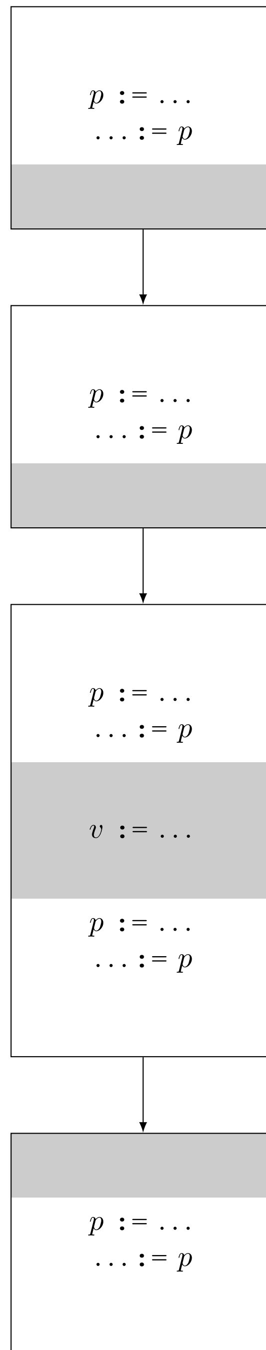


Figure 3.6: Limited code motion in overlap analysis for preallocated registers. To avoid combinatorial explosion, the definition of the virtual register v may only be placed in the shaded areas relative to the live ranges of the preallocated register p.

CHAPTER 4

Reuse Candidate Selection

The second major challenge in combining global code motion with register allocation is the problem of avoiding the generation of invalid programs. This chapter explains how the register reuse opportunities identified by the live range analysis may conflict with each other, making it necessary to select a subset of all possible reuses. Both a heuristic and an optimal combinatorial solution for this problem are given. These solutions are then extended to include a factor for balancing the freedom of code motion against the amount of spilling.

4.1 The candidate selection problem

The overlap analysis described in the previous chapter identifies pairs of values with avoidable live range overlaps. Applying the appropriate code motion operations, GCMS can ensure that both values can be assigned to the same processor register. That is, one value can reuse the register previously used by the other value; for this reason, such a pair of values is called a *possible reuse*.

Out of all the pairs identified as possible reuses due to avoidable overlaps, we need to select a maximal non-conflicting set of candidates. Not all of these reuse pairs are compatible.

4.1.1 Conflicting reuses within basic blocks

Consider the dependence graph fragment in Figure 4.1, adapted from the first paper on integrated scheduling and spilling (Goodman and Hsu 1988). Some subcomputations are independent and could be scheduled in either order: The register for the value v computed by instruction 2 could be reused for the value w defined by instruction 4 by adding an ordering arc from instruction 4 to instruction 3, the use of v. However, the opposite direction would also be legal, sequencing w strictly before v by adding an arc $2 \rightarrow 6$.

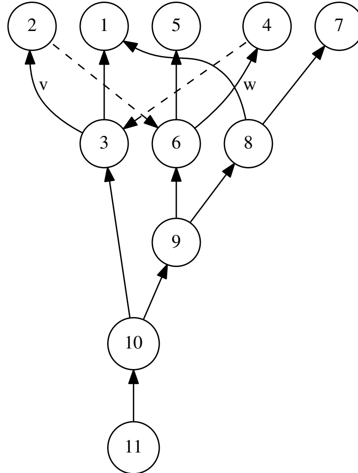


Figure 4.1: Dependence graph with with conflicting sequencing possibilities (dashed)

Both of these possibilities for reusing a register, illustrated as dashed arcs to be added to the dependence graph, would be found by GCMS's exhaustive overlap analysis. However, the two scheduling decisions cannot be taken at the same time as they would introduce a cycle. Real programs also generate cycles that are more general, i. e., that involve two pairs of different values, not just the two permutations of a single pair of values as in this example. Potential cycles can also be larger, i. e., involve more than two arcs that might be added to the dependence graph.

GCMS must ensure that the register allocator never selects reuses (by allocating certain pairs of values to the same registers) that would require the introduction of such a cycle in the dependence graph.

4.1.2 Conflicting reuses through global code motion

Another possible way for reuses to conflict is by global code motion. We have seen that live range overlaps can be avoided by sinking or hoisting instructions. However, GCMS cannot apply both hoisting as in Figure 3.5b and sinking as in Figure 3.5d to the same instructions for some live range.

Again, GCMS must be able to ensure somehow that reuses selected by the register allocator never cause conflicting, unsatisfiable constraints on global code motion.

4.1.3 Candidate selection to prevent candidate conflicts

GCMS is designed to use a regular PBQP register allocator which has no knowledge of code motion, the associated constraints, and the possible conflicts between these constraints. Therefore, in order to use such an allocator, the register allocation problem

itself must be posed in such a way that no valid solution would lead to a conflict between selected reuses.

We must therefore select a subset of all avoidable overlaps as the set of *reuse candidates*. Candidates should be selected to maximize the total weight of reused pairs (that is, the sum of their associated spill costs) in order to minimize the total weight of the overlaps that remain and might lead to spills.

Each candidate is associated with a set of dependence arcs and code motion restrictions. In what follows, let S denote the set of all of these restrictions from all candidates. Some subsets $C \subseteq S$ are *conflicting* sets: They impose incompatible instruction placement restrictions or introduce cycles in the dependence graph.

We say that a conflicting set C is *minimal* if there is no subset $C' \subset C$ that is also a conflicting set. Let $E = \{C \subseteq S \mid C \text{ is a minimal conflicting set}\}$. Then the problem of selecting a maximal set of reuse candidates without conflicts amounts to identifying a *maximal independent set* in the hypergraph $H = (S, E)$, i. e., a maximal set $M \subseteq S$ that does not include any conflict set $C \in E$. It is open whether the maximal independent set problem for hypergraphs can be solved in polynomial time; the best known algorithms for the general case run in quasi-polynomial time (Eiter, Makino, and Gottlob 2008).

In practice, as the number of register pairs is so large and the identification of minimal conflicting sets is not trivial, we do not explicitly construct the hypergraph and apply a specialized algorithm. We instead solve the entire selection problem directly using either a simple greedy heuristic or by modeling it as an integer linear programming (ILP) problem.

4.2 Heuristic reuse candidate selection

The heuristic reuse candidate selection algorithm was already sketched in Section 3.3, which introduced the greedy variant of the overlap analysis.

In heuristic selection, we simply apply the greedy analysis to each pair of live ranges. This selection algorithm inspects reuse candidates one by one and commits to any candidate that it determines to be an avoidable overlap. Committing to a candidate means immediately applying its instruction placement constraints and dependence arcs; this ensures that the candidate will definitely remain avoidable, but it restricts freedom of code motion for subsequent candidates.

Due to this greedy behavior, it is important to process candidates in an order that maximizes the chance to pick useful candidates early on. Since a live range's spill weight is a measure of how beneficial it is to keep the live range in a register, we want to avoid as many overlaps between live ranges with large weights as possible. We therefore order our candidates by decreasing weight before applying the greedy solver. Thus for the most critical pairs of live ranges we have comparatively larger freedom to apply hoisting or sinking to avoid overlaps. Due to the greedy destructive changes the analysis makes to the global dependence graph and the legal placements of instructions, the possibilities to avoid overlaps between later, cheaper pairs are restricted more and more.

4.3 Optimal reuse candidate selection

The optimization problem we must solve is finding a non-conflicting set of reuse candidates with maximal weight, where the weight is the sum of the spill costs of the two values. That is, of all possible overlaps, we want to avoid those that would lead to the largest total spill costs. We model this as an integer linear program and use an off-the-shelf solver (CPLEX) to compute an optimum.

The optimization problem is built based on the blame formulas computed by the live range overlap analysis.

4.3.1 Problem variables

The variables in the problem are:

- a binary variable $select_c$ for each reuse candidate c; this is 1 if and only if the candidate is selected
- a binary variable $place_{i,b}$ for each legal block b for any instruction i occurring in a placement constraint in any blame; this is 1 if and only if it is legal to place i in b in the optimal solution
- a binary variable $arc_{i,j}$ for any dependence arc $i \to j$ occurring in any blame; this is 1 if and only if the arc must be present in the optimal solution
- a variable $instr_i$ for each instruction in the program, constrained to the range $0 \leq instr_i < N$ where N is the total number of instructions; these are used to ensure that the dependence graph for the optimal solution does not contain cycles

4.3.2 Objective function

We want to maximize the weight of the selected candidates; as a secondary optimization goal, we want to preserve as much freedom of code motion as possible for a given candidate selection. The objective function is therefore

$$\text{maximize } \sum_c w_c \, select_c + \sum_i \sum_b place_{i,b}$$

where the first sum ranges over all candidates c, w_c is the weight of candidate c, and the second sum ranges over all placement variables for instructions i and their legal blocks b. In our problem instances, there are typically considerably more candidate selection variables than placement variables, and the candidate weights are larger than 1. Thus the first sum dominates the second, and this objective function really treats freedom of code motion as secondary to the avoidance of overlaps.

4.3.3 Constraints

The constraints in equations (4.1)–(4.7) ensure a valid selection.

Legality constraints

First, we give the constraints that model the structure of the existing dependence graph. We need this to detect possible cycles that would arise from selecting an invalid set of arcs. Therefore, we give a partial ordering of instructions that corresponds to dependences in the graph. For each instruction i with a direct predecessor p, the following must hold:

$$instr_i > instr_p \tag{4.1}$$

Next, we require that all instructions must be placed in some legal block. For each such instruction i:

$$\sum place_{i,b} \geq 1 \tag{4.2}$$

where the sum ranges over all valid blocks b for instruction i. The inequality ensures that if an instruction may appear in several different blocks without affecting the live range overlap weight, the solver preserves code motion freedom by not committing to just a single one of the blocks.

Selection constraints

We can now proceed to give the constraints related to selecting a reuse candidate. For a candidate c and each of the arcs $i \to j$ associated with it, require

$$select_c + place_{i,b} + place_{j,b} \leq 2 + arc_{i,j} \tag{4.3}$$

to model that if c is selected and both i and j are placed in some common block b, the arc must be selected as well. For each overlap formula of the form $place_{i_1,b_1} \wedge \cdots \wedge place_{i_n,b_n}$, require:

$$select_c + \sum place_{i,b} \leq n \tag{4.4}$$

This ensures that if c is selected, at least one of these placements is *not* selected.

If an arc is to be selected due to one of the candidates that requires it, ensure that it can be added to the dependence graph without causing a cycle. That is, we want to formulate the condition $arc_{i,j} \Rightarrow instr_i > instr_j$. If N is the total number of instructions, this constraint can be written as:

$$instr_i - instr_j > N \cdot arc_{i,j} - N \tag{4.5}$$

If $arc_{i,j}$ is selected, this reduces to $instr_i - instr_j > 0$, i. e., $instr_i > instr_j$. Otherwise, it is $instr_i - instr_j > -N$, which is always true for $0 \leq instr_i, instr_j < N$. These constraints ensure that the instructions along every path in the dependence graph are always topologically ordered, i. e., there is no cycle in the graph.

Arc placement constraints

Finally, we must take interactions between dependence arcs and instruction placement into account. An arc $instr_i \to instr_j$ means that $instr_j$ may not be executed after $instr_i$

along a program path, so it is not valid to place $instr_j$ into a later block than $instr_i$. Therefore, for all arcs $instr_i \rightarrow instr_j$ in the original dependence graph where $instr_i$ may be placed in some block b_i, $instr_j$ may be placed in block b_j, and there is a non-empty forward path from b_j to b_i, require

$$place_{i,b_i} + place_{j,b_j} \leq 1 \tag{4.6}$$

to ensure that such a placement is not selected.

Similarly, for every selectable arc $arc_{i,j}$ and an analogous invalid path:

$$arc_{i,j} + place_{i,b_i} + place_{j,b_j} \leq 2 \tag{4.7}$$

That is, selecting an arc means that we also ensure that the placement of instructions respects the intended ordering.

4.4 Balanced reuse candidate selection

The heuristic and optimal candidate selection methods above both assumed that the main objective of integrated global code motion and register allocation should be to minimize register pressure in order to avoid spills. It is possible, however, that this is not always the best choice. In particular, there may be cases where some spilling can be tolerated in exchange for more aggressive code motion. For example, global code motion out of a loop might increase register pressure and force the spilling of some values, but those might only be values that are never used in the loop and thus only cause spill code outside of it. In other cases, increased register pressure due to aggressive code motion might cause only cheap rematerializations rather than more costly spills.

We therefore introduce a parameter β to capture the trade-off between global code motion and spilling as follows: For $\beta = 0$, we want GCMS to behave as described above, i. e., to avoid as many overlaps as possible, possibly restricting code motion. For $\beta = 1$, we want GCMS to perform global code motion as aggressively as possible (as in Click's GCM algorithm) and only avoid overlaps by scheduling instructions within basic blocks. For any other β value between 0 and 1 we want GCMS to behave in a mixed mode, avoiding some expensive spills by restricting global code motion, but not restricting it for cheap spills. Regardless of the value of β and the influence it has on the global code motion part of GCMS, within each basic block GCMS still schedules to minimize overlaps.

The following sections describe how the β parameter can be embedded in the heuristic and optimal selection methods.

4.4.1 Heuristic balanced candidate selection

Extending the heuristic selection algorithm is simple: We apply the algorithm as before, but only for part of all the possible reuses. After considering a certain fraction of the possible reuses, the heuristic is interrupted, and normal GCM is applied to the current

state of the program. GCM will move all instructions that are still available for code motion to the latest legal block in the shallowest loop nest. We then constrain all instructions to remain in these blocks, i. e., we disable further global code motion, and then resume execution of the heuristic GCMS algorithm. For the remaining pairs, only local instruction scheduling is applied to minimize live range overlaps.

The β parameter is used to specify at which point in the process the intervening GCM step should be applied: At $\beta = 0$, we never apply GCM and allow GCMS to avoid overlaps as far as possible; this is identical to the unmodified GCMS algorithm. At $\beta = 1$, GCM is applied before considering any possible register reuse pairs. This is equivalent to simply applying aggressive GCM before register allocation and scheduling and then forbidding further global code motions.

In general, in the balanced heuristic approach GCM is applied after considering a fraction of $1 - \beta$ of all possible reuses. For example, for $\beta = 0.25$, we apply GCMS to the first 75 % of possible reuses and then perform GCM before proceeding to the remaining 25 %. That is, as in the original heuristic, we prefer avoiding overlaps between pairs of high spill weights, but at some point we trade this off against freedom of global code motion.

4.4.2 Optimal balanced candidate selection

Optimal balanced selection is based on the same ILP model as was used before. The objective function is changed to include the β parameter. The constraints describing the conditions under which live ranges overlap do not need to change.

The new objective function is

$$\text{maximize } (1 - \beta)\, S + \beta\, P$$

where

$$S = W_S \sum_c w_c\, select_c$$

is the term that models benefits in spill cost due to avoided live range overlaps, and

$$P = W_P \sum_i \sum_b w_b\, place_{i,b}$$

captures freedom of code motion.

The new weights w_b denote the benefit of placing an instruction in block b, computed from the block's loop nesting depth. To compute it, first set f_b to the block's statically estimated execution frequency based on an exponential function of the nesting depth. For more deeply nested blocks, this means that f_b is a higher cost of placing an instruction in that block, and conversely we want w_b to be a lower benefit. This can be achieved by defining $F = \max_b f_b$ and setting $w_b = F - f_b$ for each block b. Apart from adding these w_b weights, the new objective function is the same as the old one, weighting the first sum by $(1 - \beta)\, W_S$ and the second by $\beta\, W_P$.

In order to make it meaningful to talk about intermediate values of β, we must ensure that the impacts of terms S and P are equal if they are weighted equally. That is, the

weights W_S and W_P are chosen such that the two subproblems are weighted equally at $\beta = \frac{1}{2}$, i. e., we choose a nontrivial solution to the equation

$$\frac{1}{2} W_S \sum_c w_c = \frac{1}{2} W_P \sum_i \sum_b w_b.$$

In practice we do this by setting $W_P = 1$, which yields

$$W_S = \frac{\sum_i \sum_b w_b}{\sum_c w_c}.$$

The uses of β in the formulation ensure that at $\beta = 0$ the algorithm only attempts to minimize spilling, while at $\beta = 1$ spilling is of no concern and the solution maximizes the freedom of moving code to the cheapest possible blocks. In practice, either of these extreme values is undesirable because it would mean that part of the problem is completely ignored: At $\beta = 0$ we still want to allow any code motions that do not cause additional spills, and at $\beta = 1$ we still want to allow local scheduling to avoid some spills, but without impacting the freedom of code motion among blocks. We therefore choose a small ε and replace β values of 0 and 1 by ε and $1 - \varepsilon$, respectively.

CHAPTER 5

Spilling and Global Code Motion

This chapter describes how the results of the overlap analysis and candidate selection are used during spilling and for performing restricted global code motion based on the results produced by the register allocator.

5.1 PBQP register allocation

Register allocation based on the PBQP (partitioned boolean quadratic programming) problem was introduced by Scholz and Eckstein (2002). It is based on a graph representation similar to the one used in graph coloring register allocators, but with cost vectors and matrices associated with nodes and edges, respectively.

5.1.1 The PBQP model

Figure 5.1a shows an example of a cost vector for a node. Nodes represent live ranges, and the cost vector has entries representing all the possible allocation options for a live range. In this example, the live range may be spilled (*sp* option) or allocated to one of

	sp	R1	R2	R3
c	0	0	0	0

(a) Sample cost vector for a live range

	sp	R1	R2	R3
sp	0	0	0	0
R1	0	∞	0	0
R2	0	0	∞	0
R3	0	0	0	∞

(b) Sample cost matrix for a conflict edge

Figure 5.1: Sample cost vector and cost matrix for the nodes and edges of a PBQP register allocation graph

the three processor registers R1 to R3. Each entry in the vector holds the costs for the given allocation option. The spill option has some spill cost c associated with it, while allocation to CPU registers is typically treated as free. (On modern x86 processors, some instructions require special prefix bytes in order to be able to address some registers. Since this means that allocation to some registers causes larger code, these registers could theoretically be penalized in the cost vector.)

Cost matrices on edges connecting nodes can model various kinds of allocation constraints, but the most important case is that of conflicts between overlapping live ranges. Figure 5.1b shows such a conflict matrix. The allocation options for the two adjacent nodes are listed along the two dimensions of the matrix, and a cost is given for each pair of options. In the case of overlapping live ranges, allocating both to the same register is forbidden, which is modeled by infinite costs in the matrix. Allocating the live ranges to different registers, or spilling one or both, does not incur additional costs due to the conflict matrix.

Given a PBQP graph of n nodes with cost vectors $\boldsymbol{c}_i$ and a cost matrix C_{ij} for each edge, the goal of register allocation is to select exactly one option for each node such that overall costs are minimized. Each node is associated with a selection vector $\boldsymbol{x}_i \in \{0,1\}^{D_i}$, where D_i is the number of allocation options for that node. Subject to the constraint $\boldsymbol{x}_i^\top \cdot \mathbf{1} = 1$ for all $1 \leq i \leq n$, i. e., exactly one of the entries must be selected for each node, the objective function to be minimized is (Hames and Scholz 2006):

$$\sum_{1 \leq i \leq n} \boldsymbol{x}_i^\top \cdot \boldsymbol{c}_i + \sum_{1 \leq i \leq j \leq n} \boldsymbol{x}_i^\top \cdot C_{ij} \cdot \boldsymbol{x}_j$$

Solving PBQP optimally is an NP-complete problem in general. Various good-quality heuristic solvers are available (Scholz and Eckstein 2002; Hames and Scholz 2006; Buchwald, Zwinkau, and Bersch 2011).

5.1.2 Spilling in the PBQP model

If the PBQP solver does not find a valid allocation of all values to registers, the spill option will be selected for some of the nodes. The corresponding values are then spilled and the register allocation process is repeated for the new version of the program. In practice, almost all functions can finally be allocated after a few rounds of spilling.

Unfortunately, our experiments show that the use of multiple rounds of spilling can sometimes lead to paradoxical results. As will be discussed in in Chapter 6, there are cases where a schedule with lower overall overlap weight ends up spilling more. This can happen because each round of spilling works in isolation and cannot predict whether every choice it makes actually lowers register pressure at critical points in the program. Spilling a value that is defined or used at a program point with excess register pressure does not reduce register pressure at that particular point since a register is still needed to hold the value at that point. Thus in affected cases, the first round of spilling typically spills several values which later turn out to have been spilled uselessly, without having made progress towards a feasible allocation. This appears to be an inherent limitation of multi-round PBQP spilling that is present in all implementations.

	sp	R1	R2	R3
sp	0	0	0	0
R1	0	ε	0	0
R2	0	0	ε	0
R3	0	0	0	ε

Figure 5.2: Sample ε edge cost matrix for modeling avoidable live range overlaps

For spilling, the current implementation of GCMS uses the simple 'spill everywhere' model. In this approach, a store instruction is inserted immediately after the spilled value's definition, and a load is generated before each use. As a slight improvement over the completely naïve method, adjacent uses share a single reload instruction, and a use adjacent to the definition does not need a reload. It is well known that more sophisticated spilling models generate better code (Colombet, Brandner, and Darte 2011). In absolute numbers, GCMS would therefore benefit from a more complex spiller. However, the evaluation in Chapter 6 always uses this same spiller and only changes the arrangements of instructions in the underlying programs. Thus the comparison of different GCMS configurations is valid, even if the absolute performance of the benchmarks is not optimal.

In our implementation, rematerialization is chosen instead of spilling for live ranges whose definitions are identified as rematerializable by the underlying LLVM compiler framework. These are only those instructions that load constants into registers, either from a constant pool or from an immediate operand. As with spilling, a more sophisticated model would improve the absolute performance of code generated by GCMS, but for the purposes of this thesis, it suffices that all the tested configurations use the same spilling and rematerialization approach.

5.2 Spilling with reuse candidate information

Spilling in GCMS builds on the PBQP formalism by including special edges for reuse candidates that were chosen by one of the selection processes detailed in Chapter 4. Such candidates are represented as edges in the PBQP graph with ε cost matrices as shown in Figure 5.2. The ε parameter is some very small positive value (orders of magnitude below the spill costs appearing in cost vectors). The intention of these edges is to ensure that code motion restrictions are minimized: If the two values that make up a reuse candidate are allocated to the same register, it must be ensured that their live ranges do not overlap in the final arrangement of instructions in the program. Hence, if enough registers are available, it is better if such pairs of values are allocated to different registers to retain full freedom of code motion. The PBQP solver will choose an allocation that minimizes the number of ε entries and thus the number of code motion restrictions.

Pairs of values that the overlap analysis has identified as definitely overlapping are represented in the PBQP problem by normal conflict edges. Conservatively, avoidable overlaps that were not selected by candidate selection must also be represented as conflicts. Figure 5.3 shows the conflict graph for the example program used before.

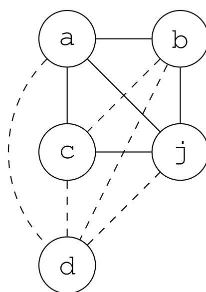


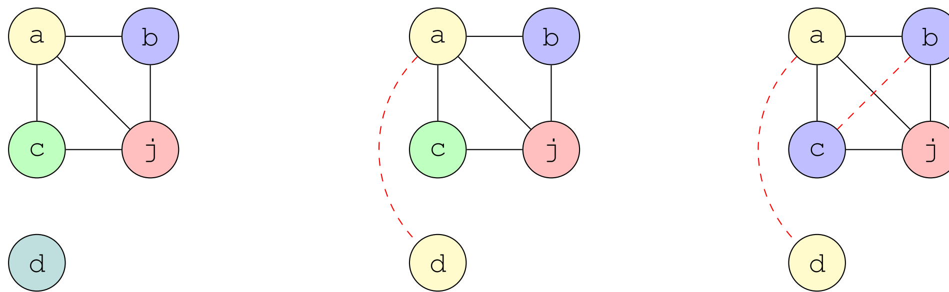
Figure 5.3: Conflict graph for the example program from Figure 1.2, with dashed edges representing ε edges in the PBQP graph

Solid edges are definite conflicts, while dashed edges represent avoidable overlaps. This high-level structure of the graph is machine independent and depends only on the results of overlap analysis and candidate selection. Refining the conflict graph into a PBQP register allocation graph integrates information on the target machine's register file: The cost vectors on nodes and cost matrices on edges refer to allocation options to concrete processor registers. The solid edges of the abstract conflict graph are instantiated to conflict matrices as shown in Figure 5.1b, while the dashed edges are instantiated to ε matrices as in Figure 5.2.

Note that since this GCMS conflict graph integrates all the possible live range overlaps over a large number of schedules, node degrees tend to be considerably larger than in the conflict graph for any one concrete schedule. This increase in node degrees is entirely due to ε edges. While the larger degree is not a problem in theory, it does pose a problem for the heuristic PBQP solvers that are used in practice. These heuristics are able to apply provably optimal reduction rules to iteratively remove nodes of degree 1 and 2 from the graph, thus decreasing other nodes' degrees; heuristic steps are only needed for nodes of higher degree. This means that the PBQP register allocation problem for the original allocation problem in Figure 1.3a can be solved provably optimally by a heuristic solver while the GCMS variant in 5.3 cannot. Thus a heuristic solver must somehow be made aware of the special meaning of ε edges.

The current implementation of GCMS handles this issue as follows. Since ε edges are only needed for code motion purposes, spilling can actually proceed without them. Thus the initial PBQP problem does not contain these edges at all. When spilling is complete, the ε edges are added to the problem and the solver is applied again to find a valid assignment that also respects as many ε edges as possible. In order to make it easier to find an assignment, the heuristic solver is modified to ignore ε edges when calculating the node degree for the purpose of selecting the next node to apply a reduction to. At this point, some spill-free assignment should theoretically always exist, but occasionally the heuristic solver determines that more spilling would be needed. In this case, ε edges for the affected live ranges are removed from the problem and the process is repeated.

Spilling and allocation with reuse candidate information otherwise proceed in one or more rounds exactly as in PBQP register allocation without code motion. However,



(a) Allocation for 5 registers

(b) Allocation for 4 registers: an ε edge is violated

(c) Allocation for 3 registers: two ε edges are violated

Figure 5.4: The code motion impact of allocating the example program for processors with various numbers of registers

where regular allocation only needs to repeat simple liveness analysis after spilling, the GCMS variant must run the entire overlap analysis and candidate selection process to rebuild the PBQP problem for the next round.

For simplicity, the current GCMS implementation performs only spilling but not register assignment; that is, after a spill-free solution has been found, virtual registers are not rewritten to CPU registers although the PBQP solution gives a valid assignment. The reason for this is to avoid tricky corner cases in out-of-SSA transformation. The actual assignment of registers is left to one of LLVM's existing register allocators. In some rare cases, this allocator does not find a valid allocation for the program and has to insert some more spill code.

5.3 Restricted global code motion

After the last round of spilling, the result returned by the PBQP solver is an assignment of values to processor registers. As explained above, this assignment is not used to rewrite the instructions. Nevertheless, it must still be ensured that it is, in principle, a valid allocation for the program after code motion. That is, live ranges selected for reusing a register may not overlap.

We therefore inspect all selected candidate pairs to see if they were allocated to the same CPU register. If so, we must restrict code motion and add ordering arcs to the dependence graph as specified by the pair's blame term. Otherwise, such restrictions need not be added, and code motion freedom is retained.

Figure 5.4 illustrates this process for the example program. Three different allocations represented as colorings of the conflict graph in Figure 5.3 are shown, for 5, 4, and 3 registers, respectively. For 5 registers (Figure 5.4a), an allocation assigning each live range to a different register is trivial to find. No restrictions of code motion and scheduling are needed in this case.

For 4 registers (Figure 5.4b), an allocation without spilling can be found, but the value `d` must be allocated to the same register as one of the other values. In the example, this was chosen to be the same register as for `a`. The ε edge connecting these two nodes is violated, so some code motion restriction must be applied. Consulting Table 3.1 (page 31) for the corresponding information computed by the overlap analysis, we see that these values overlap if and only if the instruction defining `d` is placed in the `loop` block. Forbidding this placement ensures that no matter what other code motion operations are performed, this register assignment remains valid for the final program. This is the only restriction that is needed, all other code motion operations are allowed. In particular, it is legal to hoist the definition of `b` out of the loop.

In the case of only 3 registers in Figure 5.4c, more severe restrictions on code motion are needed. As before, the results of the overlap analysis tell us that the instruction defining `d` may not be placed in the `loop` block. Since `b` must be allocated to the same register as `c`, hoisting of `b`'s definition must be forbidden, thus the instruction defining `b` may not be placed in the `start` block. Additionally, Table 3.1 requires a dependence arc from `c`'s definition to `b`'s use to ensure a legal schedule.

These examples show how GCMS adapts to the number of available processor registers: A single model of live range overlaps is translated by the register allocator into both a register assignment and a set of code motion restrictions. If enough registers are available, no restrictions on code motion are needed. This is in contrast to pessimistic prepass approaches that attempt to minimize register pressure without knowing the program's exact register need. In the case of GCMS, the register need is determined by the register allocator. If the register need is too high in some schedules but not others, this is reflected in the model's ε edges, which in turn identify the code motion and scheduling restrictions that are needed to produce a valid allocation and a corresponding valid schedule.

After applying all the needed constraints, we simply perform unmodified GCM on the resulting restricted dependence graph: Instructions are placed in their latest possible blocks in the shallowest loop nest. This keeps instructions out of loops as far as possible, but prefers to shift them into conditionally executed blocks.

5.4 Final instruction scheduling

After instructions have been placed in blocks and scheduling arcs have been added to the dependence graph, the final schedule for each block must be determined. At this point, the restrictions in the graph ensure that no scheduling decision can cause any more spilling. Scheduling can therefore fully focus on maximizing instruction-level parallelism in the program.

The scheduler used by GCMS is based on a simple model of instruction latencies extracted from processor manuals. The dependence graph is then scheduled using a standard list scheduler with a rank function based on the *critical path*, the longest path through the dependence graph (Cooper and Torczon 2004): Instructions are scheduled by always choosing one with maximal remaining path length.

CHAPTER 6

Experimental Evaluation

This chapter describes how the algorithms described earlier were implemented in the LLVM compiler framework. The algorithms are then evaluated by applying the compiler to a standard benchmark suite in various configurations.

6.1 Implementation issues

This section describes some details of the implementation that were left unspecified up to this point.

6.1.1 Implementation setting

Optimal and heuristic GCMS with and without balancing is implemented in the LLVM compiler framework's back-end. Since LLVM's native frontend, Clang, only handles C and C++, we use GCC as our front-end and the Dragonegg GCC plugin to generate LLVM intermediate code from GCC's internal representation. This allows us to apply our optimization to Fortran programs from the SPEC CPU 2000 benchmark suite as well. Unfortunately, our version of Dragonegg miscompiles six of the SPEC benchmarks, but this still leaves us with 20 benchmarks to evaluate. We generate code for the ARM Cortex-A9 architecture with VFP3 hardware floating point support. Programs are compiled for Linux and statically linked against the GNU C library. We use the `-O3` optimization flag to apply aggressive optimizations both at the intermediate code level and in the back-end. All of a benchmark's modules are linked before this optimization to enable interprocedural optimizations on the whole program, and in particular to enable inlining between functions that appear in different source files.

The exhaustive overlap analysis was implemented using SWI-Prolog, and we use CPLEX as our ILP solver. For GCMS without balancing or with $\beta = 0$, whenever CPLEX times out on a problem, we inspect the best solution it has found up to that point; if its overlap weight is lower than the weight in the prepass schedule, we use this

approximation, and otherwise fall back to the prepass schedule. For GCMS with $\beta > 0$, this decision cannot be guided by spill costs since in the $\beta > 0$ case higher spill costs than in the prepass schedule may be tolerable. In this case, any approximate solution found by CPLEX is acceptable if it times out. In either case, if CPLEX times out without having found a feasible solution, the prepass schedule is used. The default timeout of $2^{11} = 2048$ seconds is sufficient to ensure that CPLEX finds some feasible solution for every instance in the benchmark set.

The greedy heuristic solver could in principle be based on the Prolog analysis, but in practice an older implementation in C++ is used. Comparing the two implementations of essentially the same analysis was very useful for identifying bugs in both.

6.1.2 Spill cost model

The spill costs for live ranges are based on the model of statically estimated block frequencies f_b mentioned in Section 4.4.2. For a block b of loop nesting depth d_b, LLVM estimates the frequency as:

$$f_b := \left(1 + \frac{100}{d_b + 10}\right)^{d_b}$$

According to a comment in the LLVM source code, this formulation was chosen over a simpler variant like 10^{d_b} to avoid overflowing single-precision floating point numbers when computing frequencies for blocks in deeply nested loops.

Traditionally, the spill cost for a virtual register is computed by summing the estimated frequencies of the basic block each use appears in (because in general a reload may have to be inserted before each use). When using GCMS, this is not directly applicable since in the presence of global code motion, it is not clear in which basic block a use will finally be placed. Thus GCMS computes the largest possible spill cost by summing the maxima of the estimated frequencies of the blocks each use may appear in:

$$c_v := \sum_u \max \{f_b \mid \text{use } u \text{ of } v \text{ may appear in block } b\}$$

6.1.3 Spill costs in candidate weight computation

Another adjustment is needed for the spill costs of live ranges that are preallocated to CPU registers, and for live ranges that reload and immediately use a previously spilled value. Such live ranges are typically assigned infinite spill cost since it is impossible to shorten their live ranges by spilling. This cost model is sufficient for register allocation, but it must be adjusted for use in GCMS.

Candidate selection defines the weight of a candidate (v, w) as the sum of the costs of the two live ranges: $c_v + c_w$. However, infinite costs are too restrictive for code motion: Such pairs have infinite weight if one of the members has infinite spill costs, regardless of the other member's costs. Such pairs would completely dominate all other candidates even if the finite-weight member of the pair were cheap to spill.

It is therefore more useful to define a more general function for computing combined candidate weight:

$$weight(c_v, c_w) = \begin{cases} \infty & \text{if } c_v = \infty \text{ and } c_w = \infty \\ S\,c_v & \text{if } c_w = \infty \\ S\,c_w & \text{if } c_v = \infty \\ c_v + c_w & \text{otherwise} \end{cases}$$

For the ARM target, the experimentally determined value for the scaling constant of $S = 15$ seems to work well. For less regular architectures, it may make sense to choose relatively larger values of S if the live ranges v and w belong to a register class with a relatively small number of processor registers.

Since infinite numbers cannot be represented in CPLEX, any remaining infinities (for candidates where both live ranges have infinite spill weight) are replaced by a number that is orders of magnitude larger than the largest finite candidate weight.

6.1.4 Restriction of speculation

When asked to minimize spilling, GCMS has a tendency to speculate code, i. e., to move instructions out of conditionally guarded blocks. This can happen if a value is defined before a branch but only used if one of the paths is taken. GCMS may determine that it would be useful to avoid the overlap of this live range with intervening code, in particular with the computations involved in evaluating the branch condition. However, placing the use before the branch condition means speculating that instruction.

Speculation can sometimes be useful, but preliminary experiments showed that in most cases this transformation is bad for overall performance. The avoided overlap usually does not make up for the cost of having to compute the speculated instruction more often than necessary.

Therefore all the experiments reported below constrain global code motion by GCMS to forbid speculation. Where according to GCM instructions may (in principle) move along a contiguous segment of the dominator tree, GCMS forbids certain intermediate blocks. Only hoisting or sinking out of or into loops is allowed, but not more general motion into our out of conditionals within loops. Formally, for each loop in which an instruction may appear, GCMS only allows the latest block in the loop which would be legal according to GCM. Additionally, the block in which LLVM placed the instruction is always allowed.

6.1.5 Baseline compiler

The execution time measurements below compare programs compiled with GCMS against the LLVM baseline. In principle, this baseline is the program computed by LLVM's heuristics for code motion and for instruction scheduling to reduce register pressure. However, GCMS uses a different scheduler after spilling, and in blocks where no spills are needed, we might therefore observe spurious effects that are entirely due to differences in how the pipeline is utilized. In practice, GCMS's scheduler is better than LLVM's in

some such cases and vice versa, so using different schedulers adds a random disturbance to the measurements.

This disturbance can be avoided by redefining the baseline in a careful manner that uses GCMS's scheduler without affecting the spills that would be inserted for LLVM's heuristic schedule. This is done by identifying all the pairs of live ranges (that may be allocated to the same register) that do not overlap in the prepass schedule. Then GCMS's scheduler is run with the constraint that all such non-overlapping registers must remain non-overlapping. This is a strong restriction of scheduling freedom, but it still allows GCMS to rearrange some instructions without affecting spilling at all. The baseline programs in the experiments reported below are the ones produced by this scheduling pass based on LLVM's prepass schedule.

6.2 Experimental methodology

As mentioned above, 20 of the SPEC CPU 2000 benchmarks are compiled correctly by the baseline compiler. However, some of these benchmarks show large statistical variations in their execution times, varying by 2 % or more around the mean. Such variations make reliable comparisons difficult or impossible, so a further four benchmarks had to be excluded. This leaves 16 benchmark programs involved in the final evaluation.

The SPEC CPU suite offers various sizes of input instances for its benchmarks. Since a very large number of different configurations must be tested, all runs reported below were performed using the medium-sized test inputs rather than the largest possible input sets, which would have taken considerably longer.

Since we want to study the trade-off between spilling and global code motion, and there is no reason to assume that this trade-off is the same for every function in every program, we study individual functions in isolation. We used profiling to identify the three hottest functions for each of our benchmark programs. Where a hot function did not offer any possibilities for global code motion, we moved on to the next hottest one. Each of the benchmarks is then compiled with GCMS applied to only one such candidate function, for various values of the β parameter.

Each program configuration is executed 18 times and the CPU times collected. The three longest runs of each configuration are excluded since they may occasionally be tainted by statistical outliers. Of the remaining 15 data points for each program, the mean and standard deviation are shown in the tables below.

Some programs are entirely or almost entirely unaffected by GCMS because their hot functions contain no loops at all, or no relevant amounts of code that can be moved into our out of loops. The tables omit programs where all configurations of GCMS are within 1 % of the baseline's execution time.

For the remaining programs, the tables show speedups in percent of the baseline's execution time (slowdowns are shown as negative percentages). The statistical significance of the difference of each configuration to the baseline was tested using a two-tailed unpaired t-test at a significance level of $p < 0.05$. For statistically significant differences, the speedup percentage is shown in bold type in the tables.

6.3 Results

The following sections investigate the effect of GCMS on benchmark execution time and code size, looking at local scheduling vs. global code motion, the heuristic vs. the optimal solver, and various values of the β parameter ranging from 0 to 1.

6.3.1 Benchmark execution times

Tables 6.1 and 6.2 show comparisons of GCMS against the baseline versions of benchmarks. The difference between the tables is in the inlining threshold: Table 6.1 uses LLVM's default inlining threshold of 225 (a unitless number), while the data in Table 6.2 was collected after doubling the threshold to 450. The reason for more aggressive inlining was to expose larger functions containing more loops and possibly more code that can be moved between loops.

Description of the data

Each benchmark in the tables is identified by its name and the name of the function that GCMS was applied to in the first two columns. The third column (% time) shows the fraction of the total execution time of the benchmark spent in that particular function, as determined by profiling. The following columns give execution times as the mean, standard deviation, and speedup versus the baseline configuration. All times refer to the entire benchmark, not just the time spent in the particular function (which would be impossible to measure precisely). For example, Table 6.1 shows that GCMS with $\beta = 0$ applied to the `resid_` function in 172.mgrid is able to speed up the entire benchmark by about 4.2 %. However, as the benchmark spends only about 53 % of its time in this function, this means that the function itself was sped up about twice as much, or about 8 %. Such computations cannot be assumed to be very precise, however, since the precision of the profiling data is not known.

Besides the baseline, timings are given for six different configurations: The 'pin' column shows the special case of GCMS where each instruction is pinned to the basic block in which LLVM placed it. In other words, in this configuration, *global* code motion is disabled and only local instruction scheduling (aimed at minimizing spilling) is used. The remaining columns show the timings for GCMS with the β parameter ranging from 0 to 1 in increments of $\frac{1}{4}$.

Occasionally the 'pin' configuration results in the exact same program as the baseline, or one or more of the GCMS variants are identical to each other. In such cases, the table contains '=' signs to signal that the value in that place is identical to the one in the previous column.

Discussion

Examining the data in Table 6.1, we can make the following general observations:

- For the most part, the 'pin' configuration is very close to the baseline; in three cases, it is considerably better. These results are to be expected since the LLVM baseline heuristics already aim at scheduling for minimal register pressure. As the results of Section 6.3.2 will confirm further, the number and weight of spilled values is identical or very close in most of these cases. In other words, LLVM's local scheduling heuristics are already very good, and often produce schedules that minimize overall live range overlap cost.

- Similarly, the GCMS results at $\beta = 0.0$ are mostly close to or better than the baseline. In two cases, marked by '=' signs, the generated programs are even completely identical. Again, this means that LLVM's heuristics for global code motion, which try to avoid excessive register pressure, work quite well in practice.

- In general, performance decreases as the value of the β parameter increases. In almost all cases, the performance at $\beta = 0.0$ is greater than the performance at $\beta = 0.25$, which in turn is almost always greater than the performance at larger values of β. In general terms, this agrees with the expectation that spills are expensive and that avoiding them by appropriate code motion is more beneficial than hoisting simple computations to less frequently executed program points at the expense of increasing register pressure.

A few benchmarks deviate from these general patterns to some extent. First, the `zgemm_` function in 168.wupwise shows a very small but statistically significant slowdown of 0.1 % due to purely local scheduling. Inspection of the static data recorded by the compiler shows that while the GCMS overlap analysis and optimal solver reduce the overall live range overlap cost by about 10 %, the heuristic PBQP spiller spills one more value in the 'pin' configuration than the baseline (32 spilled values rather than 31), at a marginally higher total spill cost. This appears to be a side effect of using a heuristic spiller after optimal candidate selection rather than the considerably more complicated option of integrating spilling in the ILP model. However, the effect is very small.

The `dctdxf_` function in 301.apsi also shows a slowdown with purely local scheduling in the 'pin' configuration. The effect appears large, but it is not statistically significant due to the relatively large variation in the measurements as can be seen from the standard deviation. In this case the GCMS variant spills the same number of values as the baseline, but at a slightly lower total spill cost. This means that a schedule was found that minimized the spill costs at the expense of adding false dependences. In this particular case the false dependence appears to lead to a schedule that is much worse at exploiting the CPU pipeline. With global code motion enabled and $\beta = 1$, the slowdown disappears. In this configuration, some limited code motion is performed, but the number of spilled values remains the same as in the baseline, at a slightly higher total spill cost. This variant does not suffer from the false dependence problem.

Finally, the `resid_` function in 172.mgrid also shows a large upswing in performance at $\beta = 1$ after a steady decline for growing values of β. This is a particular case in which an additional spill actually leads to a speedup; it is discussed in detail in Section 6.3.5.

Table 6.1: Execution times of benchmark programs with selected functions compiled with various GCMS configurations. Times are shown with means and standard deviations in seconds and speedups as percentages. Statistically significant changes vs. the baseline ($p < 0.05$) are shown in bold. Entries marked '=' are identical to the previous column. Inlining threshold 225.

					β				
Benchmark	Function	% time	baseline	pin	0.0	0.25	0.5	0.75	1.0
164.gzip	`longest_match`	49 %	56.10 ± 0.07	56.04 ± 0.04	56.13 ± 0.05	56.29 ± 0.05	61.00 ± 0.03	62.00 ± 0.04	63.30 ± 0.06
				0.1 %	−0.1 %	**−0.4 %**	**−8.7 %**	**−10.5 %**	**−12.8 %**
164.gzip	`deflate`	17 %		55.35 ± 0.04	55.34 ± 0.03	56.16 ± 0.03	56.04 ± 0.03	56.05 ± 0.04	=
				1.3 %	**1.4 %**	**−0.1 %**	**0.1 %**	**0.1 %**	**0.1 %**
168.wupwise	`zgemm_`	54 %	59.60 ± 0.08	59.67 ± 0.06	59.79 ± 0.09	60.35 ± 0.69	61.10 ± 0.09	61.31 ± 0.10	61.39 ± 0.10
				−0.1 %	**−0.2 %**	**−1.3 %**	**−2.5 %**	**−2.9 %**	**−3.0 %**
172.mgrid	`resid_`	53 %	55.99 ± 0.28	53.71 ± 0.79	53.64 ± 0.77	54.35 ± 0.73	54.36 ± 0.71	54.58 ± 0.77	52.44 ± 0.74
				4.1 %	**4.2 %**	**2.9 %**	**2.9 %**	**2.5 %**	**6.3 %**
172.mgrid	`psinv_`	28 %		53.76 ± 0.69	53.97 ± 0.71	54.60 ± 0.70	=	=	55.04 ± 0.67
				4.0 %	**3.6 %**	**2.5 %**	**2.5 %**	**2.5 %**	**1.7 %**
173.applu	`buts_`	27 %	28.08 ± 0.05	28.05 ± 0.18	28.03 ± 0.18	28.43 ± 0.16	28.35 ± 0.14	28.33 ± 0.16	28.48 ± 0.15
				0.1 %	0.2 %	**−1.2 %**	**−1.0 %**	**−0.9 %**	**−1.4 %**
173.applu	`blts_`	21 %		28.13 ± 0.16	28.14 ± 0.16	28.43 ± 0.16	28.40 ± 0.15	28.41 ± 0.16	28.38 ± 0.15
				−0.2 %	−0.2 %	**−1.2 %**	**−1.1 %**	**−1.2 %**	**−1.1 %**
175.vpr	`try_route`	39 %	23.81 ± 0.09	23.87 ± 0.17	23.96 ± 0.18	23.90 ± 0.18	24.16 ± 0.16	24.15 ± 0.13	=
				−0.2 %	**−0.6 %**	−0.4 %	**−1.4 %**	**−1.4 %**	**−1.4 %**
179.art	`train_match`	80 %	9.73 ± 0.07	=	=	10.10 ± 0.28	10.35 ± 0.29	10.25 ± 0.30	=
				0.0 %	0.0 %	−3.8 %	**−6.4 %**	**−5.4 %**	**−5.4 %**
183.equake	`main`	38 %	48.32 ± 0.31	48.32 ± 0.74	48.36 ± 0.75	52.30 ± 0.77	52.28 ± 0.77	51.45 ± 0.83	52.82 ± 0.80
				0.0 %	0.1 %	**−8.2 %**	**−8.2 %**	**−6.5 %**	**−9.3 %**
256.bzip2	`fullGtU`	48 %	57.08 ± 0.14	=	=	=	60.44 ± 0.17	=	=
				0.0 %	0.0 %	0.0 %	**−5.9 %**	**−5.9 %**	**−5.9 %**
301.apsi	`dctdxf_`	8 %	20.98 ± 0.12	21.24 ± 0.63	21.13 ± 0.47	21.18 ± 0.53	21.21 ± 0.56	21.24 ± 0.59	21.03 ± 0.38
				−1.3 %	−0.7 %	−1.0 %	−1.1 %	−1.2 %	−0.2 %

The discussion above was based on static program properties computed by the compiler, namely the number and total cost of spilled values. These data (for an inlining threshold of 450) are presented in Section 6.3.2 below.

Impact of more aggressive inlining

Table 6.2 for GCMS with the inlining threshold doubled to 450 is broadly similar to Table 6.1 and for the most part shows the same interesting functions. Changing the inlining threshold has an impact on some functions, especially some smaller ones, by inlining code into them and increasing their share of total execution time. This is the case, for instance, for `deflate` in 164.gzip and `main` in 183.equake. Other functions appear to be mostly unchanged. It is interesting to note, comparing the baseline times, that this much inlining slows several programs down.

The general trends are similar in both tables, although several are more pronounced in Table 6.2. For example, the `try_route` function in Table 6.1 shows a relatively gently sloping slowdown as β increases. In Table 6.2, the function is similarly close to the baseline at $\beta = 0$, but considerably slower at larger values of β.

The `Perl_sv_setsv` function from 253.perlbmk is new in Table 6.2. After showing a downwards performance trend for increasing β values, it displays a sudden upswing at $\beta = 1$. This case is also explored in more detail in Section 6.3.5.

The 301.apsi benchmark disappears in Table 6.2: Inlining changes the program sufficiently to make the scheduling anomaly disappear that led to the slowdowns shown in Table 6.1. However, the `train_match` function in the 179.art benchmark now shows a similar problem. The amount of spilling is the same in the $\beta = 0$ configuration as in the baseline and the 'pin' configurations, but the optimal model's results cause conservative code motion which results in a program that is slightly worse overall.

The following sections list more data and describe additional experiments. Unless noted otherwise, all of these refer to runs with the larger inlining threshold of 450.

6.3.2 Static spill statistics

Table 6.3 shows data collected by the compiler on the amount of spilling generated by the different variants of GCMS. This data is useful for cross-referencing with the performance data presented above. It allows us to check that increasing the β parameter to make global code motion more aggressive tends to lead to more spilling. The comparison with performance numbers can show whether increased spilling seems to correspond to lower performance.

A number of the entries in the table are marked with asterisks (*). These mark the cases where the final register allocator that follows GCMS's spilling and code motion passes did not find a valid allocation and had to insert additional spill code. Recall from Section 5.2 that the current implementation of GCMS performs spilling but not out-of-SSA transformation, and that therefore it cannot apply the register assignment that the allocator found. After GCMS is done, LLVM's standard passes perform out-of-SSA

Table 6.2: Execution times of benchmark programs with selected functions compiled with various GCMS configurations. Times are shown with means and standard deviations in seconds and speedups as percentages. Statistically significant changes vs. the baseline ($p < 0.05$) are shown in bold. Entries marked '=' are identical to the previous column. Inlining threshold 450.

					β				
Benchmark	Function	% time	baseline	pin	0.0	0.25	0.5	0.75	1.0
164.gzip	`longest_match`	49 %	55.90 ± 0.05	55.90 ± 0.06	55.95 ± 0.05	55.91 ± 0.06	60.97 ± 0.06	60.71 ± 0.07	63.36 ± 0.06
				0.0 %	**−0.1 %**	−0.1 %	**−9.1 %**	**−8.6 %**	**−13.4 %**
164.gzip	`deflate`	28 %		55.92 ± 0.06	56.01 ± 0.09	55.92 ± 0.05	56.82 ± 0.06	57.76 ± 0.09	57.35 ± 0.04
				−0.1 %	**−0.2 %**	−0.1 %	**−1.7 %**	**−3.4 %**	**−2.6 %**
168.wupwise	`zgemm_`	53 %	60.11 ± 0.06	60.28 ± 0.07	60.26 ± 0.07	60.75 ± 0.05	61.91 ± 0.05	61.95 ± 0.06	61.93 ± 0.06
				−0.3 %	**−0.3 %**	**−1.1 %**	**−3.0 %**	**−3.1 %**	**−3.0 %**
172.mgrid	`resid_`	53 %	56.28 ± 0.14	53.11 ± 0.39	53.05 ± 0.37	53.79 ± 0.33	53.80 ± 0.33	53.84 ± 0.29	51.82 ± 0.29
				5.6 %	**5.7 %**	**4.4 %**	**4.4 %**	**4.3 %**	**7.9 %**
172.mgrid	`psinv_`	28 %		53.14 ± 0.69	53.35 ± 0.26	54.00 ± 0.21	53.26 ± 0.14	53.27 ± 0.19	54.42 ± 0.28
				5.6 %	**5.2 %**	**4.1 %**	**5.4 %**	**5.3 %**	**3.3 %**
173.applu	`buts_`	27 %	28.02 ± 0.07	27.93 ± 0.07	27.94 ± 0.06	28.33 ± 0.06	28.27 ± 0.07	28.26 ± 0.07	28.39 ± 0.07
				0.3 %	**0.3 %**	**−1.1 %**	**−0.9 %**	**−0.8 %**	**−1.3 %**
173.applu	`blts_`	20 %		28.05 ± 0.07	28.06 ± 0.07	28.35 ± 0.06	28.30 ± 0.04	28.31 ± 0.06	28.30 ± 0.05
				−0.1 %	−0.1 %	**−1.2 %**	**−1.0 %**	**−1.0 %**	**−1.0 %**
175.vpr	`try_route`	74 %	23.64 ± 0.05	23.67 ± 0.08	23.67 ± 0.08	24.3 ± 0.06	24.4 ± 0.07	24.68 ± 0.07	=
				−0.1 %	−0.1 %	**−2.8 %**	**−3.2 %**	**−4.4 %**	**−4.4 %**
179.art	`train_match`	82 %	9.56 ± 0.08	=	9.74 ± 0.19	9.78 ± 0.07	10.03 ± 0.07	9.92 ± 0.06	9.98 ± 0.19
				0.0 %	**−1.8 %**	**−2.3 %**	**−4.9 %**	**−3.7 %**	**−4.4 %**
183.equake	`main`	40 %	50.37 ± 0.24	48.41 ± 0.56	49.37 ± 0.60	51.76 ± 0.60	52.84 ± 0.57	53.38 ± 0.56	52.35 ± 0.57
				3.9 %	**2.0 %**	**−2.8 %**	**−4.9 %**	**−6.0 %**	**−3.9 %**
253.perlbmk	`Perl_sv_setsv`	8 %	23.62 ± 0.06	23.64 ± 0.04	23.62 ± 0.04	23.87 ± 0.03	24.98 ± 0.03	24.96 ± 0.02	23.62 ± 0.04
				−0.1 %	0.0 %	**−1.1 %**	**−5.7 %**	**−5.7 %**	0.0 %
256.bzip2	`fullGtU`	48 %	59.15 ± 0.10	=	=	=	62.90 ± 0.11	=	=
				0.0 %	0.0 %	0.0 %	**−6.3 %**	**−6.3 %**	**−6.3 %**

transformation, some coalescing, and final register allocation using the default PBQP allocator.

As the table shows, this final allocation usually succeeds without any additional spilling. However, in some cases and for various reasons (too optimistic coalescing or the inherent limitations of a heuristic allocator), no valid assignment can be found, and LLVM must insert more spill code. The number and weights of extra spills are not shown in the table. Due to the out-of-SSA transformation and coalescing performed by LLVM, these numbers refer to live ranges that may have more than one point of definition, which makes them uncomparable to our numbers based on SSA values. However, for the most part, the additional spills are a few cheap ones that only modify small details of the program. Nevertheless, the numbers suggest that an implementation of GCMS for production use should include full out-of-SSA transformation and register assignment to avoid this kind of problem.

As expected, increasing the value of the β parameter tends to gradually increase the number of spills as well as the total spill weight. There are some exceptions to this general trend, such as the case of `deflate` in 164.gzip: Here the $\beta = 1$ variant spills less than $\beta = 0.75$. This is caused by the fact that PBQP spilling and register allocation proceeds in multiple rounds with imperfect communication between them. In this particular example, the first round at $\beta = 1$ does, in fact, spill more than the first round at $\beta = 0.75$. Some more spills follow in subsequent rounds. However, at some point in the process, the $\beta = 0.75$ variant spills values that are not helpful in reducing the register pressure; as discussed in Section 5.1.2, this can always happen in multi-round PBQP allocation because the PBQP solver cannot predict the impact of its decisions on future rounds. The move from 'pin' to $\beta = 0$ for `blts_` in 173.applu is similar: This should not generate more spilling, but again, an early round of spilling makes suboptimal decisions and generates some unnecessary spills.

The third major example is the step from $\beta = 0.75$ to $\beta = 1$ on the `Perl_sv_setsv` function in 253.perlbmk. Here, again, the first rounds of spilling operate as expected, with $\beta = 1$ causing more expensive spilling (although the total number of spilled values is lower). In later rounds it turns out that some of the spills at $\beta = 0.75$ did not contribute to reducing register pressure. In the case of this benchmark, the performance penalty due to this problem, which is inherent in multi-round PBQP spilling, is especially large. In almost all other cases this situation does not occur.

The table also sheds light on some unexpected properties of the experimental data from Table 6.2. For example, despite the general trend of decreasing performance with increasing values of β, on the `longest_match` function in 164.gzip the $\beta = 0.75$ configuration performs better than $\beta = 0.5$. The static numbers show that both of these generate the same amount of spill code. However, the $\beta = 0.75$ variant performs more aggressive global code motion and can thus achieve slightly better performance. A similar pattern is visible for the `psinv` function in 172.mgrid. However, in most cases there is a monotonous increase in spilling that corresponds to mostly monotonous slowdowns in Table 6.2 for increasing values of β. Finally, the `fullGtU` function in 256.bzip2 shows a peculiar pattern: Neither the baseline nor any of the GCMS configurations generate any

Table 6.3: Static properties of selected benchmark functions with various GCMS configurations. The values shown are the number and total spill weight of spilled values. Asterisks indicate cases where LLVM inserted additional spills after out-of-SSA transformation.

				β				
Benchmark	Function	baseline	pin	0.0	0.25	0.5	0.75	1.0
164.gzip	`longest_match`	25	25	25	30	31	31	35
		13073.8	13073.8	13073.8	18784.4	22003.3	22003.3	46548.7
164.gzip	`deflate`	43	43	40*	48	65	71	68
		4310.1	4310.1	2678.5*	5149.2	16651.8	31569.8	29194.4
168.wupwise	`zgemm_`	31	32	29*	49*	58*	61*	61*
		44820.0	45670.0	39850.0*	23960.0*	47530.0*	50140.0*	50140.0*
172.mgrid	`resid_`	51	43	41*	47	47	47	48
		196831.1	144138.5	141652.8*	164889.2	164889.2	164889.2	171437.8
172.mgrid	`psinv_`	50	43	41*	47	47	47	55
		190263.3	144157.4	141671.7*	164908.2	164908.2	164908.2	238011.0
173.applu	`buts_`	44	44	48	67	67	67	68
		1163612.0	1163612.0	1260661.0	6939027.0	7230599.0	7230599.0	7274564.0
173.applu	`blts_`	48	48	54	76	75	76	76
		1077341.8	1077341.8	1198109.8	6173274.8	6103359.8	6115630.8	6115630.8
175.vpr	`try_route`	55	55	53*	69	99	110	110
		2291996.0	2291996.0	2271044.0*	7419622.0	15453493.0	14983923.0	14983923.0
179.art	`train_match`	18	18	18	33	52	53	53
		213839.9	213839.9	213839.9	7023765.2	25393869.2	25216719.2	25216719.2
183.equake	`main`	139	140	141*	211	169*	181*	182*
		481465.6	483377.8	482710.0*	3463830.0	1374410.0*	1393487.0*	1395102.0*
253.perlbmk	`Perl_sv_setsv`	5	5	5	37	43	43	32
		20.0	20.0	20.0	12400.0	18120.0	18120.0	16740.0
256.bzip2	`fullGtU`	0	0	0	0	0	0	0
		0.0	0.0	0.0	0.0	0.0	0.0	0.0

spill code, but Table 6.2 shows a large slowdown at $\beta \geq 0.5$. The reasons for this are discussed in Section 6.3.5.

6.3.3 Solver time

Figures 6.1 and 6.2 show the time taken by CPLEX when solving the integer linear programming problems involved in compiling the benchmark programs' hot functions. Figure 6.2 is an enlargement of the area highlighted by the dotted box near the origin of Figure 6.1 (the aspect ratio is not preserved).

The data shown are for an inlining threshold of 450 and for $\beta = 0$. For larger values of β, solver times tend to be shorter because in these cases the optimal solution selects fewer candidates and the solver does not have to explore as many scheduling decisions. Each line in the plots corresponds to the successive rounds of spilling for a particular function. That is, the first point of a line plots solver time against the function's size in instructions before spilling, the second point shows solver time after one round of spilling, and so on. The solver timeout for these experiments was set to 2048 seconds of wall time.

The plots show that while in very general terms solver time tends to increase with program size, there is very high variance. In the extreme case near the upper-left corner of Figure 6.1, the solver times out on a function of only 387 instructions (`buts_` in 173.applu), while near the lower-right corner it solves instances optimally for a function more than ten times that size (`Perl_sv_getsv` in 253.perlbmk) within less than a tenth of the time limit. Instances for functions up to several hundred instructions in size are solved within less than a second. A large function of initially 2105 instructions, `main` in 183.equake, takes a total of 5 rounds of which 3 time out, but the results in Table 6.2 show that nonetheless this results in a good solution.

Each round of spilling increases the number of instructions in the function monotonously. This can be expected to increase the solver time in general, and it is indeed reflected in the plots for several instances. However, in many cases, the solver time decreases dramatically despite the increase in code size: Although there are more live ranges and more instructions to schedule, the individual scheduling decisions often become simpler since spill reload live ranges have fewer uses than the original live ranges.

6.3.4 Impact of solver time on benchmarks

Nontrivial solver times raise the question how useful it is to let the solver run to completion until a provably optimal solution is found. CPLEX and similar solvers are based on the branch and bound method and typically find a sequence of intermediate solutions of increasing quality before some solution is finally proved optimal. It is interesting to investigate the quality of these intermediate solutions, which can in theory be found very quickly.

In practice, intermediate solutions are often not found immediately because CPLEX applies a pre-solving step before starting the branch and bound phase. The pre-solver simplifies the problem, which speeds up the subsequent search. However, it can sometimes

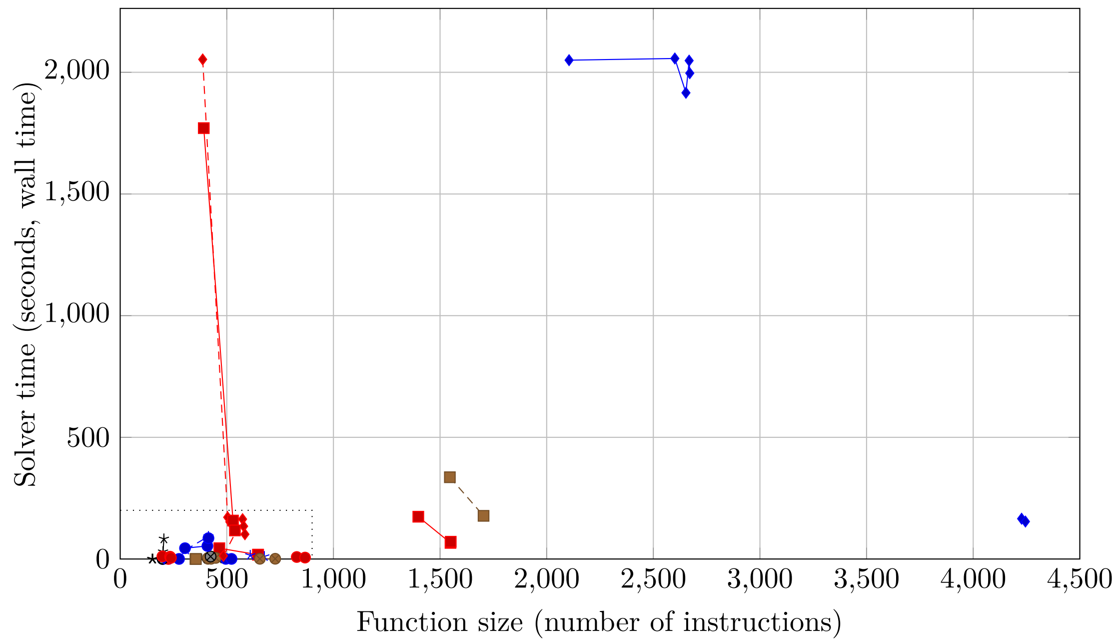


Figure 6.1: Time taken by the ILP solver for problems of various sizes ($\beta = 0$)

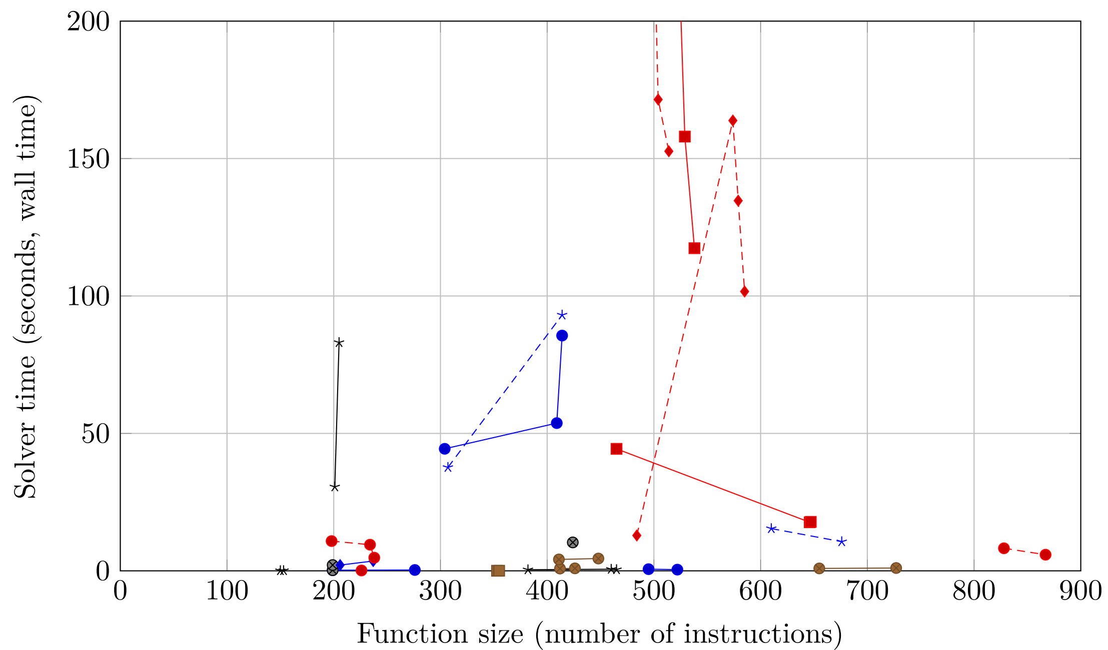


Figure 6.2: Time taken by the ILP solver for problems of various sizes, detail of Figure 6.1

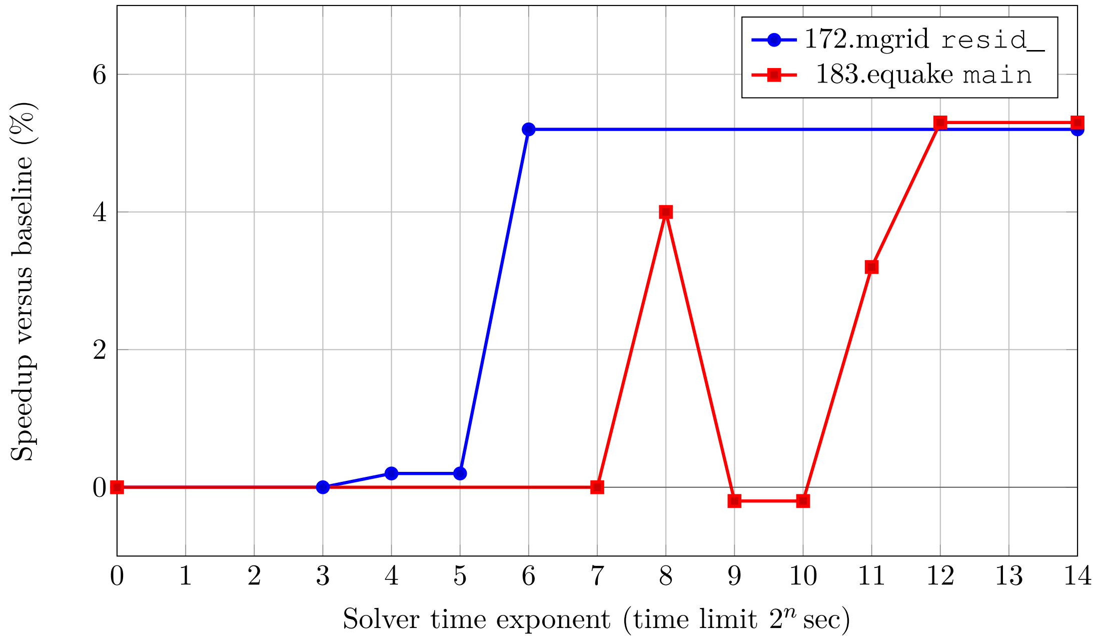


Figure 6.3: Influence of solver time on the performance of selected benchmarks, for solver times from $2^0 = 1$ sec to $2^{14} = 16384$ sec

take a very long time and lead to a timeout without having found any intermediate solution at all.

Varying the solver's time limit allows us to inspect intermediate results that increase in quality as the time limit increases. If the basic assumption of GCMS holds, namely that a better solution of the optimization problem leads to less spilling and better performance, then the performance of the benchmark problems should also increase accordingly.

Figure 6.3 displays the results of varying the solver's time limit and observing the resulting changes in execution time. For meaningful results, this experiment was applied only to functions that needed nontrivial solver times and that produced measurable improvements in benchmark execution speed at $\beta = 0.0$. The selected functions were therefore `resid_` from 172.mgrid and `main` from 183.equake. The solver timeout was varied exponentially, using the powers of 2 from $2^0 = 1$ sec to $2^{14} = 16384$ sec.

The first solutions for `resid_` from 172.mgrid are found at a time limit of $2^4 = 16$ seconds. At this limit, a solution is found for the first round of GCMS and PBQP allocation, but not for the subsequent rounds. This results in a very slight improvement over the baseline. All three rounds are solved optimally starting at a limit of $2^6 = 64$ seconds, leading to a significant speedup.

The case of the `main` function in 183.equake is more complicated as it does not show the expected monotonous improvement in execution time for increasing time limits. At a time limit of $2^8 = 256$ seconds, an intermediate solution is found for the first round but for not subsequent rounds. At $2^9 = 512$ and $2^{10} = 1024$ seconds, intermediate

solutions for more rounds are found, and less spilling is generated than in the previous case. However, in these two cases, the heuristic PBQP register assignment with ε edges fails (see Section 5.2). GCMS must therefore remove some of the ε edges from the assignment problem, which means introducing false dependences that lead to a schedule that is worse than necessary. This is simply an artifact of the imperfect heuristic PBQP solver. Scaling the time limit further, GCMS spills less and does not run into the ε edge assignment problem again. Provably optimal solutions for all rounds are found at a time limit of $2^{14} = 16384$.

This data set is very limited, but it supports the underlying assumptions of GCMS, namely that increasingly good solutions of the optimization problem lead to less and less spilling, and that this in turn can improve program performance. However, the heuristic PBQP register allocator is a potential weak point in the overall process of code motion, spilling, and register allocation.

6.3.5 Detailed impact of β parameter

The data shown in Tables 6.1 and 6.2 is somewhat coarse-grained. A number of benchmarks show seemingly abrupt changes from one β value to the next, but it is not visible in the tables whether these are gradual changes over intervening β values or sudden jumps from one β setting to a very slightly larger one. Building and measuring variants of all the benchmarks with very fine-grained scaling of β would take a very long time, but in Figure 6.4 this has been done for three of the benchmarks that exhibit interesting patterns. These three functions were selected because they show interesting patterns in Table 6.2, i. e., not simply a linear trend. In this plot, β is scaled in steps of 0.05. Where preliminary results showed large changes around certain points, even more fine-grained scaling in steps of 0.01 was used.

The `resid` function in 172.mgrid shows an upwards performance trend from $\beta = 0.75$ to $\beta = 1$ in 6.2. In Figure 6.4 we can see that this is in fact a single large step at the very end of the scale, from $\beta = 0.99$ to $\beta = 1$. Inspection of the generated code that this change introduces an additional value to be spilled (47 spilled values rather than 46). The two variants do not differ in terms of global code motion, but the extra spill changes the register pressure in a large, hot basic block in a loop. The spilled value has only one use in this block, so a single reload must be inserted. This incurs some costs on every loop iteration. However, the reload and use occur quite early in the block, and after the use an additional CPU register is freed for use in the rest of the block. This allows the PBQP register allocator to find a register assignment that causes fewer false dependences due to register reuse, and the GCMS scheduler can find a more aggressive schedule with higher instruction-level parallelism. Thus this is a case where an extra spill, even accompanied by a reload instruction inside a hot loop, turns out to be beneficial. This is an exception to the general trend observed in most of the other benchmarks (and even in this function, with $\beta \in [0, 0.99]$) that more spilling leads to lower performance.

The `Perl_sv_setsv` function in 253.perlbmk shows several interesting changes. First, there is a relatively large step from the $\beta = 0$ configuration, which performs identically to the baseline, to $\beta = 0.01$, which shows a slowdown of about 1.6 %. In this

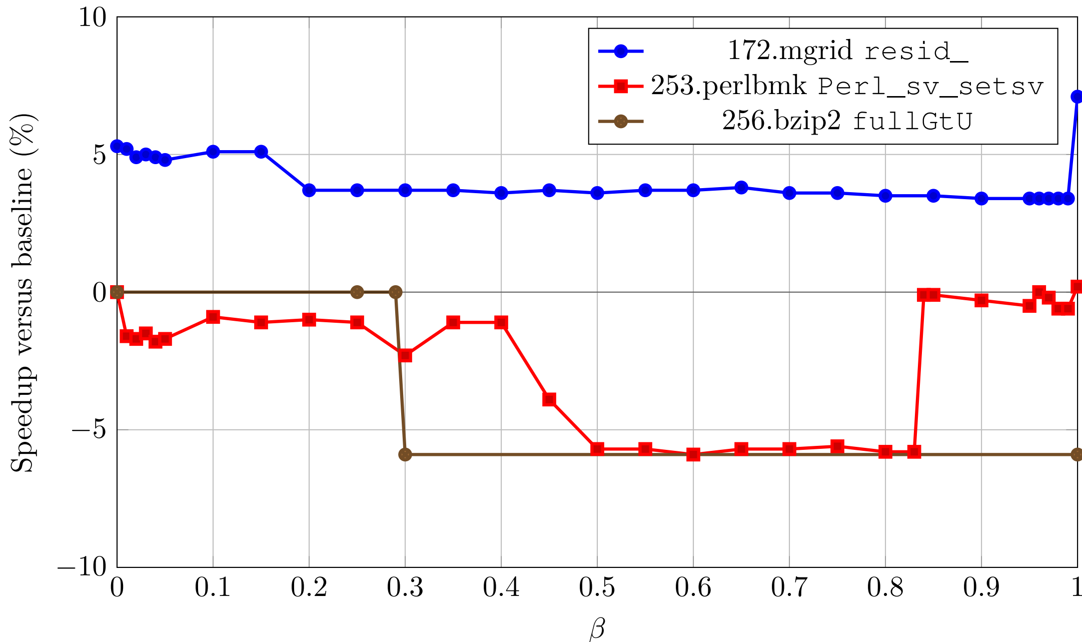


Figure 6.4: Influence of the β parameter on the performance of selected benchmarks

particular function, the very small change in β is enough to cause a large change in the number of spilled values, from 5 to 33. While each of the additional spills is relatively cheap in itself, this does cause a large number of reload instructions to be inserted in the program. Taken together, these have a measurable negative impact on program performance.

This is counteracted later with a large improvement in performance when moving from $\beta = 0.83$ to $\beta = 0.84$. This step corresponds to a large difference in the programs that are generated. Analyzing the generated code showed that this is the point at which the problem of multi-round spilling discussed in Section 6.3.2 disappears: At $\beta \geq 0.84$, the first round spills slightly more, but the overall choice of values to spill is more useful.

Finally, Figure 6.4 shows that the abrupt change in the performance of `fullGtU` in 256.bzip2 is due to a single slight change in β from 0.29 to 0.3. At this point, GCMS decides to hoist a loop invariant move-immediate instruction out of its loop. This increases register pressure. While no spilling in the narrow sense is needed (see the 0 entries for this function in Table 6.3), the increased register need causes the allocator to use one more callee-saved register. LLVM's prologue/epilogue insertion pass must therefore make sure that this extra register is saved on function entry and restored at function exit. As this particular function is both called very frequently and has a large number of early return sites, the associated costs add up and cause a large slowdown. Additionally, the slightly different allocation causes an extra register-to-register copy instruction to be inserted in the function's entry block. This may lead to issues with code alignment and

instruction cache misses, although this cannot be verified as the target platform does not provide performance counters.

6.4 Results of heuristic GCMS

The following sections discuss the heuristic candidate selection method from Section 4.2, with the extension to the use of the β balancing parameter as discussed in Section 4.4.1.

6.4.1 Compile times

The compile time impact of heuristic GCMS is negligible. The largest of the functions of interest is `Perl_sv_setsv` with 4227 instructions before the first round of spilling; on this function, the two rounds of spilling using GCMS's PBQP spiller, but using only a standard liveness analysis without code motion, complete in 1.9 seconds. Using heuristic GCMS with $\beta = 0$, this grows to 5.0 seconds, which is tolerable when attempting aggressive optimization of a very large function.

On all other hot functions under test, total allocation time including GCMS was within 2 seconds, and less than 1 second in almost all cases.

6.4.2 Benchmark execution times

Table 6.4 shows an evaluation of the heuristic candidate selection method. The format is similar to Tables 6.1 and 6.2, but is simplified in that only the changes versus the baseline are shown, not absolute times. As before, a function is included in the table if some configuration differs from the baseline by at least 1 %. Statistically significant differences (t-test, $p < 0.05$) are shown in bold. As before, a '=' entry means that the value is identical to the previous column because the generated benchmark executables are identical.

The table contains a 'pin' configuration, as before: GCMS restricted such that instructions may not move from the block where LLVM placed them, only local instruction scheduling is allowed. Further, there are two different sets of GCMS configurations with the β parameter varying from 0 to 1. This table follows the algorithm laid out in Figure 3.4 with the *allowUseHoisting* parameter set to *True*.

Examining the 'pin' column, we can see that, as with the optimal solver, heuristic GCMS is mostly close to the baseline performance. On the two functions from the 172.mgrid benchmarks where the optimal solver performs particularly well, the heuristic also improves performance, although by a smaller amount. Unsurprisingly, the heuristic can also sometimes make bad choices, but the resulting slowdowns are quite small.

Moving on to the β columns, the large number of '=' entries is conspicuous. Recall that the balancing heuristic works by first applying heuristic GCMS to a fraction $1 - \beta$ of the possible reuse pairs, then performs GCM and from that point on does not allow any further global code motion, only local scheduling. For most benchmarks the variants generated for all β values from 0 to 0.75 are identical. This means that the available

Table 6.4: Execution time speedups of benchmark programs compiled with various GCMS configurations with heuristic candidate selection.

			β				
Benchmark	Function	pin	0.0	0.25	0.5	0.75	1.0
164.gzip	`longest_match`	**0.7** %	**0.7** %	=	=	=	**−0.2** %
164.gzip	`deflate`	**−0.1** %	**−0.2** %	=	=	=	**−1.6** %
172.mgrid	`resid_`	**1.7** %	**1.9** %	=	=	=	**1.7** %
172.mgrid	`psinv_`	**1.0** %	**0.9** %	=	=	=	**1.1** %
173.applu	`buts_`	**−0.3** %	**−1.0** %	=	=	**−1.0** %	**−1.1** %
175.vpr	`try_route`	**−0.3** %	0.1 %	=	**0.2** %	=	**−4.1** %
179.art	`train_match`	0.0 %	0.0 %	=	=	=	**−1.4** %
183.equake	`main`	0.2 %	**4.0** %	=	=	=	**4.6** %
256.bzip2	`fullGtU`	0.0 %	0.0 %	=	=	=	**−6.5** %

amount of global code motion is usually exhausted very quickly, after analyzing a relatively small fraction of all possible reuse pairs. After analyzing the first few pairs, the greedy heuristic has chosen a single legal basic block for almost all instructions, so a parameter value $\beta < 1$ has almost no effect. This is only different for two benchmark functions (`buts_` in 173.applu and `try_route` in 175.vpr) which have more freely movable code and can therefore exploit intermediate β values.

At $\beta = 1$, the available code motion freedom is exploited by applying aggressive GCM before considering the possible reuses. In all benchmarks, this places some instructions in different blocks than the $\beta = 0$ heuristic that avoids expensive live range overlaps by moving code. In many, but not all cases, the code generated with $\beta = 0$ and the code generated with purely local scheduling after LLVM's placement of code both perform better than with $\beta = 1$. Again, this shows that in general, avoiding spills is a better code generation strategy than aggressive global code motion, even if followed by a local scheduling step that tries to reduce spilling.

6.4.3 Benchmark execution times without hoisting of uses

Since most of the global code motion potential is used up by the first few possible reuse pairs, the heuristic GCMS approach is sensitive to bad choices made early on that destroy later potential to generate good code. This can be especially critical if the hoisting of uses is enabled. This operation, illustrated in Figure 3.5b, may or may not be beneficial overall: Hoisting a use of v out of a loop avoids the one particular live range overlap illustrated in that example, but it might lengthen the live range of a value defined by the instruction using v. This can lead to a larger total amount of spill code. Table 6.5 therefore evaluates a variant of heuristic GCMS where this operation is disabled. That is, the *allowUseHoisting* parameter in the algorithm from Figure 3.4 is set to *False* for these runs. This variant may only sink, but not hoist, code to avoid live range overlaps.

Table 6.5: Execution time speedups of benchmark programs compiled with various GCMS configurations with heuristic candidate selection and hoisting of uses disabled.

		β				
Benchmark	Function	0.0	0.25	0.5	0.75	1.0
164.gzip	`deflate`	**−0.3** %	=	=	=	**−1.6** %
173.applu	`buts_`	−0.1 %	=	=	**−0.2** %	**−1.1** %
183.equake	`main`	**5.4** %	=	=	=	**4.6** %

This change only rarely makes a difference. In the majority of cases, the exact same code is generated whether or not hoisting of uses is enabled. Table 6.5 only shows the three benchmark functions where disabling hoisting does generate different code. The effect on `deflate` from 164.gzip is negligible, but the other two benchmarks improve markedly compared to the results in Table 6.4. On `buts_` in 173.applu, a significant 1 % slowdown turns into an insignificant 0.1 %, while on `main` from 183.equake, an impressive speedup of 4 % is improved further to 5.4 %.

Note that at $\beta = 1$, GCM is applied and further code motion is forbidden. Thus at this setting it makes no difference whether hoisting of uses is allowed. The executables are identical, and the measurements of a single set of runs are shown in the $\beta = 1$ columns of both Table 6.4 and Table 6.5.

Overall, it appears that disabling code hoisting is the better choice. This again supports the thesis that reducing spilling is a good general objective for code generation. Further, the results show that heuristic GCMS can achieve significant speedups on suitable programs. Since the greedy heuristic's compile time costs are negligible, this algorithm could be a practical choice for a realistic compiler.

CHAPTER 7

Conclusions

This thesis presented GCMS (global code motion with spilling), an algorithmic framework for combining the disjoint code generation phases of *global code motion* (including local instruction scheduling) and *register allocation*.

This integration is interesting because these compilation phases pursue different goals. Global code motion and scheduling tend to move a value's definition and its uses apart, in order to minimize the number of executions of some of these instructions, or in order to exploit the processor's pipeline parallelism. Register allocation attempts to assign the values of unrelated, non-overlapping computations to processor registers without exceeding a predefined limit of available registers. Register allocation can thus introduce false dependences which limit the applicability of code motion and scheduling; code motion and scheduling, on the other hand, cause overlaps between values' live ranges, making allocation more difficult.

GCMS is based on a novel analysis that computes all the possible ways in which live ranges in a function can overlap, taking all the possible global code motions and local schedules into account. By exploiting the properties of strict SSA form, all possible live range overlaps and the code motion constraints under which they can be avoided can be enumerated efficiently. Given this data, GCMS builds a special kind of register allocation problem based on the PBQP formalism in which definitely overlapping live ranges are represented differently from avoidable live range overlaps. Avoidable overlaps are associated with code motion constraints describing how to ensure that live ranges are non-overlapping. With this formulation, a solution of the problem without spills yields both a valid register allocation and a set of code motion constraints that must be applied to the program to ensure the legality of the allocation.

Before the register allocation problem can actually be built, another problem has to be overcome: Not all of the code motion constraints identified by the analysis can be applied at the same time. Certain sets of constraints would introduce cyclic dependences or contradictory placements of instructions in basic blocks. Identifying a legal subset of constraints is the *candidate selection* problem. Various approaches to candidate selection

were discussed in this thesis: heuristic vs. optimal; local vs. global; and including a parameter β that captures the trade-off between minimizing spilling and preserving maximal freedom of code motion.

GCMS was implemented in the LLVM compiler framework and compared against LLVM's standard heuristics, which attempt to shorten live ranges in order to minimize the amount of spill code inserted in the program. The target platform was ARM Cortex-A9, a modern out of order processor. The evaluation led to the following general observations:

- In general, global code motion operations that attempt to reduce live range overlaps ($\beta = 0$ in parameterized GCMS) lead to less spill code and higher program performance than aggressive global code motion operations that attempt to reduce the number of times instructions are executed ($\beta = 1$).
- Nevertheless, in certain individual cases, spilling a value can result in much better performance. This can happen if a spill is relatively cheap but it lowers the register pressure enough to allow an instruction schedule that is better at exploiting instruction level parallelism. GCMS is able to produce such schedules because its formulation and solution minimize false dependences if enough registers are available. Other commonly used heuristics aimed at minimizing spilling, such as the ones used by LLVM, do not have this property.
- The results of the heuristics used by LLVM to minimize spilling are of high quality and often close or even identical to the optimal solution computed by local or global GCMS. However, in some cases both the optimal and the heuristic solutions to GCMS outperform LLVM by a wide margin, suggesting areas where improvement might be possible.
- Optimal GCMS sometimes shows behavior that appears paradoxical, but such cases can be explained as artifacts of the underlying heuristic PBQP register allocator or the lack of out-of-SSA transformation in the current implementation. A fully optimal implementation of integrated GCMS candidate selection, spilling, and register assignment is a goal for future work.

In light of these findings, GCMS appears to be an interesting, principled approach to code generation. While optimal GCMS suffers from compile times that make it impractical for general use, heuristic GCMS is fast and offers a unified, tunable algorithm for global code motion, instruction scheduling, and register allocation.

Bibliography

Aho, Alfred V., S. C. Johnson, and J. D. Ullman (1977). "Code Generation for Expressions with Common Subexpressions". In: *J. ACM* 24.1, pages 146–160. ISSN: 0004-5411. DOI: 10.1145/321992.322001. URL: http://doi.acm.org/10.1145/321992.322001 (cited on page 15).

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811 (cited on page 15).

Ambrosch, Wolfgang, M. Anton Ertl, Felix Beer, and Andreas Krall (1994). "Dependence-Conscious Global Register Allocation". In: *Proceedings of the International Conference on Programming Languages and System Architectures*. Lecture Notes in Computer Science 782. London, UK: Springer-Verlag, pages 125–136. ISBN: 3-540-57840-4 (cited on page 17).

Barany, Gergö (2011). "Register Reuse Scheduling". In: *9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*. Available from http://www.imec.be/odes/. Chamonix, France (cited on page 22).

Barany, Gergö and Andreas Krall (2013). "Optimal and Heuristic Global Code Motion for Minimal Spilling". In: *CC 2013 - International Conference on Compiler Construction*. LNCS 7791. Springer (cited on page 22).

Bernstein, David and Michael Rodeh (1991). "Global instruction scheduling for superscalar machines". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, pages 241–255. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113466. URL: http://doi.acm.org/10.1145/113445.113466 (cited on page 22).

Berson, David A., Rajiv Gupta, and Mary Lou Soffa (1993). "URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures". In: *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., pages 243–254. ISBN: 0-444-88464-5 (cited on page 19).

Berson, David A., Rajiv Gupta, and Mary Lou Soffa (1999). "Integrated Instruction Scheduling and Register Allocation Techniques". In: *LCPC '98: Proceedings of the*

11th International Workshop on Languages and Compilers for Parallel Computing, pages 247–262. ISBN: 3-540-66426-2 (cited on page 19).

Boissinot, Benoit, Florian Brandner, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello (2011). "A Non-iterative Data-flow Algorithm for Computing Liveness Sets in Strict SSA Programs". In: *Proceedings of the 9th Asian Conference on Programming Languages and Systems*. APLAS'11. Kenting, Taiwan: Springer-Verlag, pages 137–154. ISBN: 978-3-642-25317-1. DOI: 10.1007/978-3-642-25318-8_13. URL: http://dx.doi.org/10.1007/978-3-642-25318-8_13 (cited on page 25).

Bradlee, David G., Susan J. Eggers, and Robert R. Henry (1991). "Integrating register allocation and instruction scheduling for RISCs". In: *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. Santa Clara, California, United States: ACM, pages 122–131. ISBN: 0-89791-380-9. DOI: http://doi.acm.org/10.1145/106972.106986 (cited on page 18).

Briggs, Preston, Keith D. Cooper, and Linda Torczon (1992). "Rematerialization". In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. PLDI '92. San Francisco, California, United States: ACM, pages 311–321. ISBN: 0-89791-475-9. DOI: http://doi.acm.org/10.1145/143095.143143. URL: http://doi.acm.org/10.1145/143095.143143 (cited on page 3).

Buchwald, Sebastian, Andreas Zwinkau, and Thomas Bersch (2011). "SSA-Based Register Allocation with PBQP". English. In: *Compiler Construction*. Edited by Jens Knoop. Volume 6601. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 42–61. ISBN: 978-3-642-19860-1. DOI: 10.1007/978-3-642-19861-8_4. URL: http://dx.doi.org/10.1007/978-3-642-19861-8_4 (cited on page 50).

Castañeda Lozano, Roberto and Christian Schulte (2014). "Survey on Combinatorial Register Allocation and Instruction Scheduling". In: *CoRR* abs/1409.7628. URL: http://arxiv.org/abs/1409.7628 (cited on page 16).

Chaitin, G. J. (1982). "Register allocation & spilling via graph coloring". In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. SIGPLAN '82. Boston, Massachusetts, United States: ACM, pages 98–105. ISBN: 0-89791-074-5. DOI: http://doi.acm.org/10.1145/800230.806984. URL: http://doi.acm.org/10.1145/800230.806984 (cited on page 4).

Click, Cliff (1995). "Global code motion/global value numbering". In: *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. PLDI '95, pages 246–257. ISBN: 0-89791-697-2. DOI: http://doi.acm.org/10.1145/207110.207154. URL: http://doi.acm.org/10.1145/207110.207154 (cited on pages 2, 4, 7).

Codina, Josep M., Jesús Sánchez, and Antonio González (2001). "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors". In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. PACT '01. Washington, DC, USA: IEEE Computer Society, pages 175–184. ISBN: 0-7695-1363-8. URL: `http://dl.acm.org/citation.cfm?id=645988.674300` (cited on page 20).

Colombet, Quentin, Florian Brandner, and Alain Darte (2011). "Studying optimal spilling in the light of SSA". In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. CASES '11. Taipei, Taiwan: ACM, pages 25–34. ISBN: 978-1-4503-0713-0. DOI: `10.1145/2038698.2038706`. URL: `http://doi.acm.org/10.1145/2038698.2038706` (cited on pages 13, 51).

Cooper, Keith D. and Linda Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann Publishers. ISBN: 978-1-55860-699-9 (cited on pages 15, 25, 26, 54).

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck (1991). "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Trans. Program. Lang. Syst.* 13.4, pages 451–490. ISSN: 0164-0925. DOI: `http://doi.acm.org/10.1145/115372.115320` (cited on page 2).

Darte, Alain and C. Quinson (2007). "Scheduling Register-Allocated Codes in User-Guided High-Level Synthesis". In: *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 140–147. DOI: `10.1109/ASAP.2007.4429971` (cited on page 5).

Eiter, Thomas, Kazuhisa Makino, and Georg Gottlob (2008). "Computational aspects of monotone dualization: A brief survey". In: *Discrete Appl. Math.* 156.11, pages 2035–2049. ISSN: 0166-218X. DOI: `10.1016/j.dam.2007.04.017`. URL: `http://dx.doi.org/10.1016/j.dam.2007.04.017` (cited on page 43).

Eriksson, Mattias and Christoph Kessler (2012). "Integrated Code Generation for Loops". In: *ACM Trans. Embed. Comput. Syst.* 11S.1, 19:1–19:24. ISSN: 1539-9087. DOI: `10.1145/2180887.2180896`. URL: `http://doi.acm.org/10.1145/2180887.2180896` (cited on page 20).

Ertl, M. Anton and Andreas Krall (1991). "Optimal instruction scheduling using constraint logic programming". In: *Programming Language Implementation and Logic Programming*. Volume 528. Lecture Notes in Computer Science (cited on page 16).

Ertl, M. Anton and Andreas Krall (1992). "Instruction Scheduling for Complex Pipelines". In: *Proceedings of the 4th International Conference on Compiler Construction*. CC '92. London, UK, UK: Springer-Verlag, pages 207–218. ISBN: 3-540-55984-1. URL: `http://dl.acm.org/citation.cfm?id=647471.727284` (cited on page 16).

Falk, Heiko (2009). "WCET-aware Register Allocation Based on Graph Coloring". In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. San Francisco, California: ACM, pages 726–731. ISBN: 978-1-60558-497-3. DOI: `10.1145/1629911.`

1630100. URL: http://doi.acm.org/10.1145/1629911.1630100 (cited on page 12).

Gibbons, Philip B. and Steven S. Muchnick (1986). “Efficient Instruction Scheduling for a Pipelined Architecture”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’86. Palo Alto, California, USA: ACM, pages 11–16. ISBN: 0-89791-197-0. DOI: 10.1145/12276.13312. URL: http://doi.acm.org/10.1145/12276.13312 (cited on page 15).

Goodman, J. R. and W.-C. Hsu (1988). “Code scheduling and register allocation in large basic blocks”. In: *ICS ’88: Proceedings of the 2nd international conference on Supercomputing*. St. Malo, France: ACM, pages 442–452. ISBN: 0-89791-272-1. DOI: http://doi.acm.org/10.1145/55364.55407 (cited on pages 16, 17, 41).

Govindarajan, R., Hongbo Yang, J.N. Amaral, Chihong Zhang, and G.R. Gao (2003). “Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures”. In: *IEEE Transactions on Computers* 52.1, pages 4–20. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1159750 (cited on pages 7, 12, 20).

Hames, Lang and Bernhard Scholz (2006). “Nearly Optimal Register Allocation with PBQP”. In: *Modular Programming Languages*. Edited by David Lightfoot and Clemens Szyperski. Lecture Notes in Computer Science 4228. Springer Berlin / Heidelberg, pages 346–361. URL: http://dx.doi.org/10.1007/11860990%5C_21 (cited on pages 4, 50).

Hennessy, John L. and Thomas R. Gross (1982). “Code Generation and Reorganization in the Presence of Pipeline Constraints”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. Albuquerque, New Mexico: ACM, pages 120–127. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582166. URL: http://doi.acm.org/10.1145/582153.582166 (cited on page 15).

Johnson, Neil and Alan Mycroft (2003). “Combined code motion and register allocation using the value state dependence graph”. In: *Proceedings of the 12th international conference on Compiler construction*. CC’03. Warsaw, Poland: Springer-Verlag, pages 1–16. ISBN: 3-540-00904-3. URL: http://dl.acm.org/citation.cfm?id=1765931.1765933 (cited on page 21).

Koes, David Ryan and Seth Copen Goldstein (2009). “Register Allocation Deconstructed”. In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’09. Nice, France: ACM, pages 21–30. ISBN: 978-1-60558-696-0. URL: http://dl.acm.org/citation.cfm?id=1543820.1543824 (cited on page 6).

Lam, Monica S. (1988). “Software pipelining: an effective scheduling technique for VLIW machines”. In: *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. PLDI ’88. Atlanta, Georgia, United States:

ACM, pages 318–328. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54022. URL: http://doi.acm.org/10.1145/53990.54022 (cited on page 20).

Lokuciejewski, Paul, Marco Stolpe, Katharina Morik, and Peter Marwedel (2010). "Automatic Selection of Machine Learning Models for WCET-aware Compiler Heuristic Generation". In: *4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART'10)* (cited on pages 12, 22).

Motwani, Rajeev, Krishna V. Palem, Vivek Sarkar, and Salem Reyen (1995). *Combining Register Allocation and Instruction Scheduling*. Technical report. Stanford University. URL: http://infolab.stanford.edu/pub/cstr/reports/cs/tn/95/22/CS-TN-95-22.pdf (cited on page 20).

Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-320-4 (cited on page 15).

Nielson, Flemming, Hanne Riis Nielson, and Chris Hankin (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540654100 (cited on page 25).

Norris, Cindy and Lori L. Pollock (1993). "A scheduler-sensitive global register allocator". In: *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 804–813. ISBN: 0-8186-4340-4. DOI: http://doi.acm.org/10.1145/169627.169839 (cited on page 19).

Norris, Cindy and Lori L. Pollock (1995a). "An experimental study of several cooperative register allocation and instruction scheduling strategies". In: *Proceedings of the 28th annual international symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, United States: IEEE Computer Society Press, pages 169–179. ISBN: 0-8186-7349-4. URL: http://dl.acm.org/citation.cfm?id=225160.225190 (cited on page 21).

Norris, Cindy and Lori L. Pollock (1995b). "Register allocation sensitive region scheduling". In: *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*. PACT '95. Limassol, Cyprus: IFIP Working Group on Algol, pages 1–10. ISBN: 0-89791-745-6. URL: http://dl.acm.org/citation.cfm?id=224659.224668 (cited on page 21).

Palem, Krishna and Barbara Simons (1990). "Scheduling Time-critical Instructions on RISC Machines". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: ACM, pages 270–280. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96737. URL: http://doi.acm.org/10.1145/96709.96737 (cited on page 16).

Pinter, Shlomit S. (1993). "Register allocation with instruction scheduling". In: *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. Albuquerque, New Mexico, United States: ACM, pages 248–257.

ISBN: 0-89791-598-4. DOI: http://doi.acm.org/10.1145/155090.155114 (cited on page 18).

Poletto, Massimiliano and Vivek Sarkar (1999). "Linear scan register allocation". In: *ACM Trans. Program. Lang. Syst.* 21 (5), pages 895–913. ISSN: 0164-0925. DOI: 10.1145/330249.330250. URL: http://portal.acm.org/citation.cfm?id=330249.330250 (cited on page 4).

Rymarczyk, James W. (1982). "Coding Guidelines for Pipelined Processors". In: *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems.* ASPLOS I. Palo Alto, California, USA: ACM, pages 12–19. ISBN: 0-89791-066-4. DOI: 10.1145/800050.801821. URL: http://doi.acm.org/10.1145/800050.801821 (cited on page 15).

Scholz, Bernhard and Erik Eckstein (2002). "Register allocation for irregular architectures". In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems.* LCTES/SCOPES '02. Berlin, Germany: ACM, pages 139–148. ISBN: 1-58113-527-0. DOI: http://doi.acm.org/10.1145/513829.513854. URL: http://doi.acm.org/10.1145/513829.513854 (cited on pages 4, 49, 50).

Sethi, Ravi and J. D. Ullman (1970). "The Generation of Optimal Code for Arithmetic Expressions". In: *J. ACM* 17.4, pages 715–728. ISSN: 0004-5411. DOI: 10.1145/321607.321620. URL: http://doi.acm.org/10.1145/321607.321620 (cited on page 15).

Shobaki, Ghassan and Kent Wilken (2004). "Optimal Superblock Scheduling Using Enumeration". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 283–293. ISSN: 1072-4451. DOI: http://doi.ieeecomputersociety.org/10.1109/MICRO.2004.27 (cited on page 16).

Sreedhar, Vugranam C., Roy Dz-ching Ju, David M. Gillies, and Vatsa Santhanam (1999). "Translating out of static single assignment form". In: *Static Analysis Symposium.* Lecture Notes in Computer Science 1694. Springer Verlag, page 849 (cited on page 3).

Touati, Sid Ahmed Ali (2001). "Register Saturation in Superscalar and VLIW Codes". In: *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 213–228. ISBN: 3-540-41861-X. URL: http://www.springerlink.com/content/t8gk0y1fwkmd457w/ (cited on page 20).

Wilken, Kent, Jack Liu, and Mark Heffernan (2000). "Optimal instruction scheduling using integer programming". In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.* PLDI '00. Vancouver, British Columbia, Canada: ACM, pages 121–133. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349318. URL: http://doi.acm.org/10.1145/349299.349318 (cited on page 16).

Winkel, Sebastian (2007). "Optimal versus Heuristic Global Code Scheduling". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.*

MICRO 40. Washington, DC, USA: IEEE Computer Society, pages 43–55. ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.10. URL: http://dx.doi.org/10.1109/MICRO.2007.10 (cited on page 22).

Xu, Weifeng and Russell Tessier (2007). "Tetris: a new register pressure control technique for VLIW processors". In: *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. San Diego, California, USA: ACM, pages 113–122. ISBN: 978-1-59593-632-5. DOI: http://doi.acm.org/10.1145/1254766.1254783 (cited on page 20).

Zhou, Huiyang, Matthew D. Jennings, and Thomas M. Conte (2003). "Tree Traversal Scheduling: A Global Instruction Scheduling Technique for VLIW/EPIC Processors". In: *Languages and Compilers for Parallel Computing (LCPC)*. Volume 2624. Lecture Notes in Computer Science (cited on page 22).