Optimal and Heuristic Global Code Motion for Minimal Spilling

Gergö Barany and Andreas Krall*
{gergo,andi}@complang.tuwien.ac.at

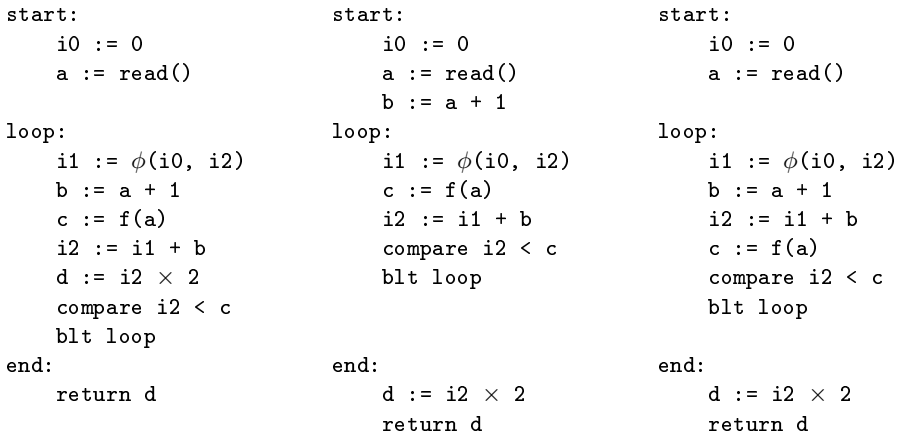
Vienna University of Technology

Abstract. The interaction of register allocation and instruction scheduling is a well-studied problem: Certain ways of arranging instructions within basic blocks reduce overlaps of live ranges, leading to the insertion of less costly spill code. However, there is little previous research on the extension of this problem to global code motion, i.e., the motion of instructions between blocks. We present an algorithm that models global code motion as an optimization problem with the goal of minimizing overlaps between live ranges in order to minimize spill code.
Our approach analyzes the program to identify the live range overlaps for all possible placements of instructions in basic blocks and all orderings of instructions within blocks. Using this information, we formulate an optimization problem to determine code motions and partial local schedules that minimize the overall cost of live range overlaps. We evaluate solutions of this optimization problem using integer linear programming, where feasible, and a simple greedy heuristic.
We conclude that global code motion with the sole goal of avoiding spills rarely leads to performance improvements because code is placed too conservatively. On the other hand, purely local optimal instruction scheduling for minimal spilling is effective at improving performance when compared to a heuristic scheduler for minimal register use.

1 Introduction

In an optimizing compiler's backend, various code generation passes apply code transformations with different, conflicting goals in mind. The register allocator attempts to assign the values used by the program to CPU registers, which are usually scarce. Where there are not enough registers, *spilling* must be performed: Excess values must be stored to memory and reloaded before they can be used. Memory accesses are slower than most other instructions, so avoiding spill code is usually beneficial on modern architectures [GYA+03]. This paper investigates the applicability of this result to global code motion directed at minimizing live range overlaps.

* This work is supported by the Austrian Science Fund (FWF) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors*, http://www.complang.tuwien.ac.at/epicopt/.



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

Fig. 1. Optimization using GCM and GCMS for a three-register processor

Register allocation and spilling conflict with transformations that lengthen a value's *live range*, the set of all program points between the value's definition and a subsequent use: *Instruction scheduling* arranges instructions within basic blocks. To maximize pipeline utilization, definitions and uses of values can be moved apart by scheduling other instructions between them, but this lengthens live ranges and can lead to more overlaps between them, which can in turn lead to excessive register demands and insertion of spill code. Similarly, *code motion* techniques that move instructions between basic blocks can increase performance, but may also lead to live range overlaps. In particular, loop-invariant code motion moves instructions out of loops if their values do not need to be computed repeatedly. However, for a value defined outside a loop but used inside the loop, this extends its live range across the entire loop, leading to an overlap with all of the live ranges inside the loop.

This paper introduces our GCMS (global code motion with spilling) algorithm for integrating the code motion, instruction scheduling, and spilling problems in one formalism. The algorithm is 'global' in the sense that it considers the entire function at once and allows code motion into and out of loops. We describe both heuristic GCMS and an integer linear programming formulation for an optimal solution of the problem.

Our algorithm is based on Click's aggressive Global Code Motion (GCM) algorithm [Cli95]. We use an example to illustrate the differences between GCM and GCMS. Figure 1(a) shows a small program in SSA form adapted from the original paper on GCM. 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 `i2` is available at the end of the loop, we can sink this multiplication to the `end`

block. Figure 1(b) 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.

This expectation fails, however, if there are not enough registers available in the target processor. Since after GCM variable `b` is live through the loop, it conflicts with `a`, `c`, and all of $\{\texttt{i0}, \texttt{i1}, \texttt{i2}\}$. Both `a` and `c` conflict with each other and with at least one of the `i` 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(c), 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. It also applies another change to the original program: The overlap between the live ranges of `b` and `c` is avoided by changing the instruction schedule such that `c`'s definition is after `b`'s last use. 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, however, that this result is specific to the limit of three registers: If four or more registers were available, GCMS would detect that unrestricted code motion is possible, and it would produce the same results as GCM in Figure 1(b).

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 spilling 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. Instead, it 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. Code motions that are not needed to avoid spilling are not performed.

The rest of this paper is organized as follows. Section 2 discusses related work in the areas of optimal instruction scheduling, optimal spilling, and integrated scheduling and register allocation techniques. Section 3 gives an overview of our GCMS algorithm. Section 4 describes our analysis for identifying all possible live range overlaps, and how to apply code motion and scheduling to avoid them. Section 5 discusses the problem of selecting a subset of possible overlaps to avoid and shows our integer linear programming (ILP) model for computing an optimal solution. Section 6 evaluates our implementation and compares the effects of heuristic and optimal GCMS to simpler existing heuristics implemented in the LLVM compiler suite. Section 7 concludes.

2 Related Work

Instruction scheduling for pipelined architectures was an early target for optimal approaches [EK91,WLH00]. The goal of such models was usually to optimize for minimal total schedule length only. However, both the increasing complexity of modern hardware and the increasing gap between processor and memory speeds made it necessary to consider register needs as well. An optimal integrated formulation was given by Chang et al. [CCK97]. More recently, Govindarajan et al. [GYA+03] concluded that on modern out-of-order superscalar processors, scheduling to minimize spilling appears to be the most profitable instruction scheduling target. Various optimal spillers for a given arrangement of instructions have also been proposed. Colombet et al. [CBD11] summarize and generalize this work.

A large body of early work in integrated instruction scheduling and register allocation [GH88,NP93,Pin93,AEBK94] aimed at balancing scheduling for instruction level parallelism against register allocation, with some spilling typically allowed. As with optimal schedulers, this trend also shifted towards work that attempted to avoid live range overlaps entirely, typically by adding sequencing arcs to basic block dependence graphs, as our GCMS algorithm also does [Tou01,XT07,Bar11].

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 [Lam88] schedules small loop kernels for optimal execution on explicitly parallel processors. Here, too, careful integration of register allocation has proved important over time [CSG01,EK12].

RASER [NP95b] performs register allocation sensitive region scheduling by first using rematerialization (duplication of computations) to reduce register pressure where needed, and then only applying global code motion operations (such as loop-invariant code motion) that do not increase register pressure beyond the number of available registers. No attempt is made to reduce register pressure by serializing registers as in our approach. RASER gives impressive improvements on machines with artificially few registers; later results on a machine with 16 registers look similar to ours [NP95a].

Johnson and Mycroft [JM03] describe an elegant combined global code motion and register allocation method based on the Value State Dependence Graph (VSDG). The VSDG is similar to our acyclic global dependence graph, 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. Our method is simpler because it tries to minimize spills without having to take a concrete number of available registers into account.

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 [BR91,ZJC03]. The notable exception is work by Winkel [Win07] 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).

3 Global Code Motion for Minimal Spilling

As an extension of GCM, our GCMS algorithm is also based on a program representation in SSA form and a global dependence graph. In SSA form [CFR+91], every value in the program has exactly one definition. Definitions from different program paths, such as in loops, are merged using special ϕ pseudo-instructions that are later replaced by register copies if necessary.

Like GCM, we build a global dependence graph with instructions as nodes and data dependences as arcs. We also add arcs to capture any ordering dependences due to side effects on memory, such as store instructions and function calls, and any instructions that explicitly access processor registers, such as moves to and from argument registers around calls. Because we will use this dependence graph to find a linear arrangement of instructions within basic blocks, we must impose an important restriction: The graph must always be acyclic to ensure that a topological ordering exists. Conveniently, in SSA form the only cyclic data dependences arise from values that travel along loop backedges to ϕ instructions. We can safely ignore these dependences as long as we ensure that the definitions of such values are never sunk out of their loops.

Our dependence graph is also used for code motion. We associate each instruction with a set of legal basic blocks as follows: Function calls, any other instructions that access memory or explicit CPU registers, and ϕ instructions are not movable, their only legal block is the block in which they originally appear. For all other blocks, we employ the concept of dominance: A block *a dominates* a block b iff any path from the function's unique entry block to b must pass through a. Following Click [Cli95], we say that an instruction is legally placed in a block b if all of its predecessors in the dependence graph are based in blocks that dominate b, and all of its successors are in blocks dominated by b. The set

of legal blocks for every instruction is computed in a forward and a backward pass over the dependence graph.

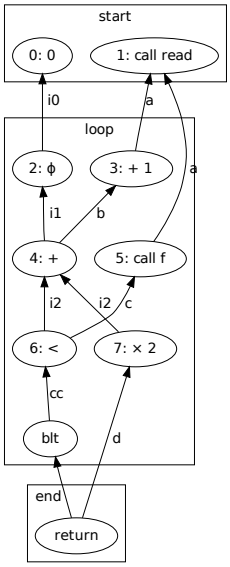


Fig. 2. Global dependence graph for example program

Figure 2 shows the global dependence graph for the example program from Figure 1. Data dependences are annotated with the name of the corresponding value. Note that the cyclic dependence of the ϕ instruction on `i2` is not shown; it is implicit, and our analysis keeps track of the fact that `i2` is live out of the `loop` block and live across the loop's backedge.

Recall that function calls, ϕ instructions, and branches are not movable in our model. Due to dependence arcs, other instructions become unmovable, too: Instructions 4 and 6 are 'stuck' between the unmovable ϕ and the branch. This leaves only instructions 3 and 7 movable into and out of the loop, but as we have seen before, this is enough to illustrate interesting interactions between global code motion and 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.

Each of these phases is discussed in more depth in the following sections.

4 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. The Figure shows the three 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

```
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(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(A, AUse, AUseBlock),
    ADefBlock \= AUseBlock,
    % There is a non-empty path from A's def to B's def...
    ADefBlock \= BDefBlock,
    cfg_forward_path(ADefBlock, BDefBlock),
    % ... and a path from B's def to A's use that does not pass
    % through a redefinition of A. That is, B is on a path from A's
    % def to its use.
    cfg_loopypath_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))).

overlapping_virtreg_pair(virtreg(A), virtreg(B)) :-
    % A and B are defined in the same block.
    virtreg_def_in(A, ADef, DefBlock),
    virtreg_def_in(B, BDef, DefBlock),
    % A has a use in a different block, so it is live out of
    % its defining block.
    virtreg_use(virtreg(A), AUse, AUseBlock),
    AUseBlock \= DefBlock,
    % B is also live out.
    virtreg_use(virtreg(B), BUse, BUseBlock),
    BUseBlock \= DefBlock,
    % There is an overlap that might be avoided if at least
    % one of these instructions were in a different block.
    Placement = [ADef-DefBlock, BDef-DefBlock,
                 AUse-AUseBlock, BUse-BUseBlock],
    record_blame(A, B, blame(placement(Placement))).
```

Fig. 3. Overlap analysis for virtual registers A and B

use. The third clause handles the case where A and B are live at the end of the same block because they are both defined there and used elsewhere.

The code uses three important auxiliary predicates for its checks:

cfg_forward_path(A, B) succeeds if there is a path in the control flow graph from block A to block B using only forward edges, i. e., not taking loop backedges into account.

cfg_loopypath_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 existing arc in the dependence graph from instruction A to 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 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. 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 and third clauses in Figure 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* 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. Table 1 shows the blame terms computed for the example program after these checks and some cleanup (removal of unmovable instructions from

Table 1. Blame terms computed for the example program, listing instruction placements and missing dependence arcs that may cause overlaps.

Pair	Invalid placements	Missing arcs	Explanation
`a`, `d`	7 in `loop`		`a` live through `loop`
`b`, `d`	3 in `start`, 7 in `loop`		`b` live through `loop` if defined in `start`
`b`, `i0`	3 in `start`		`b` live out of `start` if defined in `start`
`b`, `i2`	3 in `start`		`b` live through `loop` if defined in `start`
`c`, `d`	7 in `loop`	$7 \to 6$	order `d`'s definition after `c`'s last use
`c`, `i1`		$5 \to 4$	order `c`'s definition after `i1`'s last use
`d`, `i2`	7 in `loop`		`i2` live out of `loop` (across backedge)

placement blames). Each blame is accompanied by a brief explanation of why the live ranges would overlap if placed and arranged as stated. Pairs not listed here are found to be either non-overlapping or definitely overlapping.

The analysis discussed so far only considers pairs of values in SSA form. However, we must also consider overlaps between SSA values and explicitly named CPU registers, such as argument registers referenced in copy instructions before function calls. As such copies can occur in several places in a function, these physical registers are not in SSA form. We assume a representation in which all uses of such registers are in the same block as their definition; this allows us to treat each of these short live ranges separately, and the analysis becomes similar to the case for virtual registers.

5 Reuse Candidate Selection

After performing overlap analysis and computing all blames, a subset of reuse candidates must be selected for reuse.

5.1 Integer Linear Programming Formulation

The optimization problem we must solve is finding a nonconflicting 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.

Variables. The variables in the problem are:

- a binary variable $select_c$ for each reuse candidate c; this is 1 iff 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 iff 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 iff 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

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. However, adding weights to the second sum would easily enable us to investigate trade-offs between avoiding spills and more aggressive code motion. We intend to investigate this trade-off in future work.

Constraints. The constraints in equations (1)–(7) ensure a valid selection. 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:

$$instr_i > instr_p \tag{1}$$

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

$$\sum place_{i,b} \geq 1 \tag{2}$$

where the sum ranges over all valid blocks b for instruction i.

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

$$select_c + place_{i,b} + place_{j,b} \leq 2 + arc_{i,j} \tag{3}$$

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 invalid placement constraint ($instr_{i_1}$ in $b_1, \ldots, instr_{i_n}$ in b_n), require:

$$select_c + \sum place_{i,b} \leq n \tag{4}$$

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:

$$instr_i - instr_j > N \cdot arc_{i,j} - N \tag{5}$$

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.

Finally, we must take interactions between dependence arcs and instruction placement into account. An arc $instr_i \rightarrow instr_j$ 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

$$place_{i,b_i} + place_{j,b_j} \leq 1 \tag{6}$$

to ensure that such a placement is not selected.

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

$$arc_{i,j} + place_{i,b_i} + place_{j,b_j} \leq 2 \tag{7}$$

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

5.2 Greedy Heuristic Solution

Solving integer linear programming problems is NP-hard. While very powerful solvers exist, many problems cannot be solved optimally within reasonable time limits. We therefore complement our optimal solver with a greedy heuristic solver. This solver 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. As a small exception, we have found it useful to identify very short live ranges with a single use in the same block as the definition in the original program. We order these after all the other ranges because we have found that committing to such a very short range too early often destroys profitable code motion possibilities.

5.3 Spilling and Code Motion

Regardless of whether candidate selection was performed optimally or heuristically, GCMS finally moves on to the actual spilling phase. We use a spiller based on the PBQP formalism [SE02,HS06]. For the purposes of this paper, PBQP is simply a generalization of graph coloring register allocators [Cha82]. The difference is that nodes and edges in the conflict graph are annotated with weights modeling spill costs and register use constraints such as register pairing or aliasing.

We build a conflict graph with conflict edges both for register pairs whose live ranges definitely overlap as well as register pairs that were *not* selected by the candidate selection process. Conversely, any pair that was selected for reuse can be treated as non-conflicting, so we need not insert conflict edges for these. In fact, we want to ensure that any overlap that *can* be avoided actually *is* avoided by the register allocator. Using PBQP's edge cost matrices, we can ensure this by adding an edge with costs of some very small ϵ value between any pair of registers that we want to be allocated to different registers. The PBQP problem solver then tries to find an allocation respecting all of these constraints.

If the solver does not find a valid allocation, it returns some values to spill or rematerialize; we perform these actions and then rerun the entire algorithm on the modified function. When an allocation is found, we perform our actual code motion: In the final schedule, live ranges selected for reuse 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. For selected candidate pairs that were *not* allocated to the same register, we do not need to do anything. Thus, GCMS balances the needs of the spiller with aggressive global code motion: The freedom to move code is only restricted if this is really needed to avoid spills, but not otherwise.

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.

6 Experimental Evaluation

We have implemented optimal and heuristic GCMS in the LLVM compiler framework's backend. Since LLVM's native frontend, Clang, only handles C and C++, we use GCC as our frontend 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 and use the `-O3` optimization flag to apply aggressive optimizations both at the intermediate code level and in the backend.

The overlap analysis was implemented using SWI-Prolog, and we use CPLEX as our ILP solver. 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.

The greedy heuristic solver could in principle be based on the Prolog analysis as well, but we simply continue using our old C++ implementation. However, comparing it to the Prolog implementation of essentially the same analysis helped us considerably in finding bugs in both.

6.1 Solver Time

We ran our compiler on the SPEC CPU 2000 benchmark suite, applying the optimal GCMS algorithm on all functions of 1000 or fewer instructions (this includes more than 97 % of all functions in the suite). CPLEX was set to run with a time limit of 60 seconds of wall clock time per problem. CPLEX automatically runs as many parallel threads as is appropriate for the hardware platform and can almost fully utilize the 8 cores on our Xeon CPU, so the timeout corresponds to typically about 6 to 8 minutes of CPU time. The entire build of our 20 benchmarks with optimal GCMS takes 18 hours of wall clock time.

Figure 4 shows a scatterplot of CPLEX solver times relative to the number of instructions. Marks at or very near the 60 second line are cases where the solver reached its time limit and did not return a provably optimal result. We can see that the majority of problems is solved very quickly. It is difficult to pinpoint a general trend, although obviously solving the optimization problem for larger functions tends to take longer. Note, however, that there are some quite small functions, some even with fewer than 100 instructions, where CPLEX does not terminate within the time limit. Inspection of such cases shows that this typically happens in functions containing relatively large basic blocks with much scheduling freedom. In the ILP problems for such functions, there are many *arc* variables to consider. These affect the values of the *instr* variables, which have large domains, and we believe that this may be one of the reasons CPLEX is having difficulties in exploring the search space efficiently. Nevertheless, we are able to solve 5287 of 5506 instances (96 %) optimally, and 4472 of these (81 % overall) even within a single second.

We do not report times for the Prolog overlap analysis separately, but the overall distribution is very similar to the one in Figure 4. All blames for almost all of the functions are computed within a few seconds. Functions with a large number of basic blocks and many paths in the CFG may take longer due to the combinatorial explosion of taking all instruction placements into account. We run the overlap analysis with a 60 second time limit and fall back to the C++ heuristics if the limit is exceeded, but this only happens rarely.

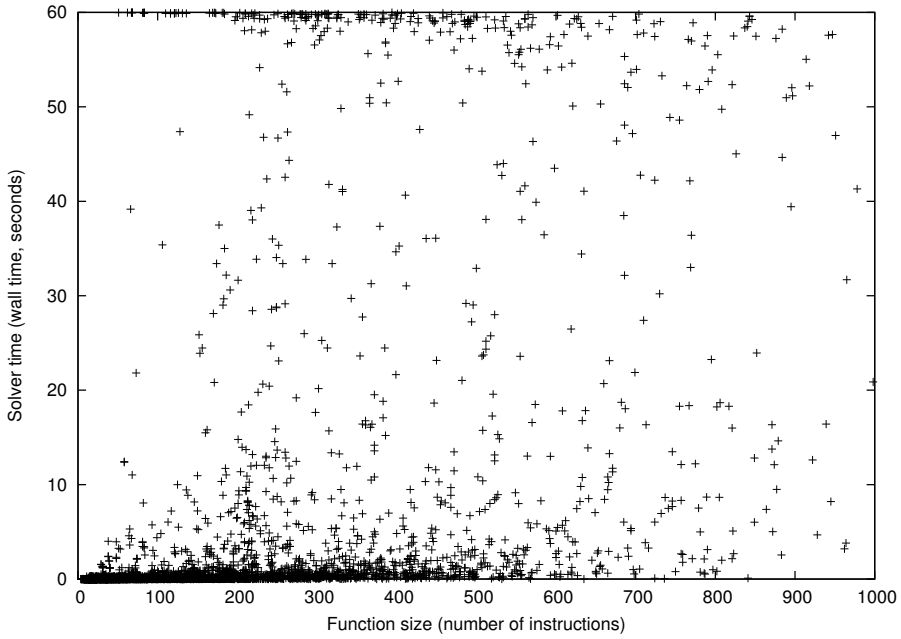


Fig. 4. Scatterplot of CPLEX solver times relative to function size

6.2 Execution Time Statistics

Table 2 shows a comparison of the execution times of our benchmark programs, compiled using five different code generation methods. Baseline is the configuration using LLVM's code motion pass which attempts to perform loop invariant code motion without exceeding a heuristically determined register usage limit. Within blocks, instructions are scheduled using a list scheduler that attempts to minimize live range lengths. The Baseline configuration is thus a good representative of modern optimizing compilers.

Heuristic GCMS is our GCMS algorithm, always using the greedy heuristic solver described in Section 5.2. Optimal GCMS uses the optimal ILP formulation for functions of up to 1000 instructions with a solver time timit of 60 seconds, as discussed above. All three configurations use the same spiller based on a PBQP formulation and using LLVM's near-optimal PBQP solver.

The 'Local' variants are also GCMS, but restricted by treating every instruction as unmovable. Thus the Local algorithm only performs instruction scheduling within the blocks LLVM chose for each instruction. We evaluate two Local variants, one with the greedy heuristic scheduler and one with the optimal ILP formulation of the candidate selection problem, as before. Each time shown in the table is CPU time, obtained as the minimum of timing five runs of each benchmark in each configuration. Additionally, the GCMS and Local variants are shown normalized to Baseline.

Table 2. Execution time statistics for SPEC CPU 2000 benchmark programs, in seconds and relative to Baseline

Benchmark	Baseline	GCMS				Local			
		Heuristic		Optimal		Heuristic		Optimal	
164.gzip	62.82	60.25	0.96	60.31	0.96	60.66	0.97	60.68	0.97
168.wupwise	60.25	59.96	1.00	60.18	1.00	60.30	1.00	60.03	1.00
171.swim	31.84	31.89	1.00	31.85	1.00	31.49	0.99	31.76	1.00
172.mgrid	50.87	52.85	1.04	52.25	1.03	53.08	1.04	51.58	1.01
173.applu	31.00	31.31	1.01	31.24	1.01	31.35	1.01	31.21	1.01
175.vpr	40.72	40.80	1.00	40.71	1.00	40.71	1.00	40.47	0.99
177.mesa	58.89	59.26	1.01	58.15	0.99	58.78	1.00	58.74	1.00
178.galgel	54.07	54.21	1.00	54.33	1.00	53.62	0.99	53.49	0.99
179.art	9.42	9.59	1.02	9.59	1.02	9.69	1.03	9.18	0.97
181.mcf	56.82	57.58	1.01	57.54	1.01	58.29	1.03	57.05	1.00
183.equake	40.81	41.42	1.01	41.14	1.01	42.42	1.04	40.72	1.00
187.facerec	80.42	81.99	1.02	85.91	1.07	82.71	1.03	80.80	1.00
189.lucas	79.48	79.35	1.00	79.24	1.00	79.14	1.00	78.88	0.99
197.parser	13.50	13.46	1.00	13.43	0.99	13.55	1.00	13.50	1.00
252.eon	13.63	13.34	0.98	13.64	1.00	13.80	1.01	13.47	0.99
253.perlbmk	25.12	24.42	0.97	24.27	0.97	26.28	1.05	24.64	0.98
255.vortex	15.32	15.42	1.01	15.68	1.02	15.35	1.00	15.45	1.01
256.bzip2	56.20	56.56	1.01	56.59	1.01	56.53	1.01	56.63	1.01
300.twolf	25.09	25.60	1.02	25.35	1.01	26.19	1.04	24.79	0.99
301.apsi	20.46	20.36	1.00	20.41	1.00	20.42	1.00	20.35	0.99
geometric mean			1.003		1.004		1.011		0.995

The results show interesting effects due to the comparison of global code motion and local scheduling. Our original research goal was to investigate whether the effect observed by Govindarajan et al. [GYA+03], that scheduling for minimal register use improves performance, also holds true for global code motion for minimial spilling. While we see performance improvements due to reduced spilling and more aggressive code motion in a few cases, in other cases performance degrades. These regressions are due to code motions that do reduce spills but at the same time move instructions to unfavorable basic blocks. We conclude that although GCMS is careful to restrict the schedule only where this is absolutely necessary to avoid spilling, such restrictions can still have an unfavorable impact on overall performance in a number of cases. On average, both heuristic and optimal GCMS produce code with comparable performance to LLVM's simpler heuristics.

This is different for purely local scheduling for minimal spilling: Here, our optimal variant is often better than LLVM's scheduling heuristic, which also has the goal of reducing spills by shortening live ranges. On average, we achieve an improvement of about 0.5 %. This local result shows that while LLVM is already close to optimal, there is still potential to improve the code produced by its state-of-the-art heuristics, and we believe that more careful global code motion

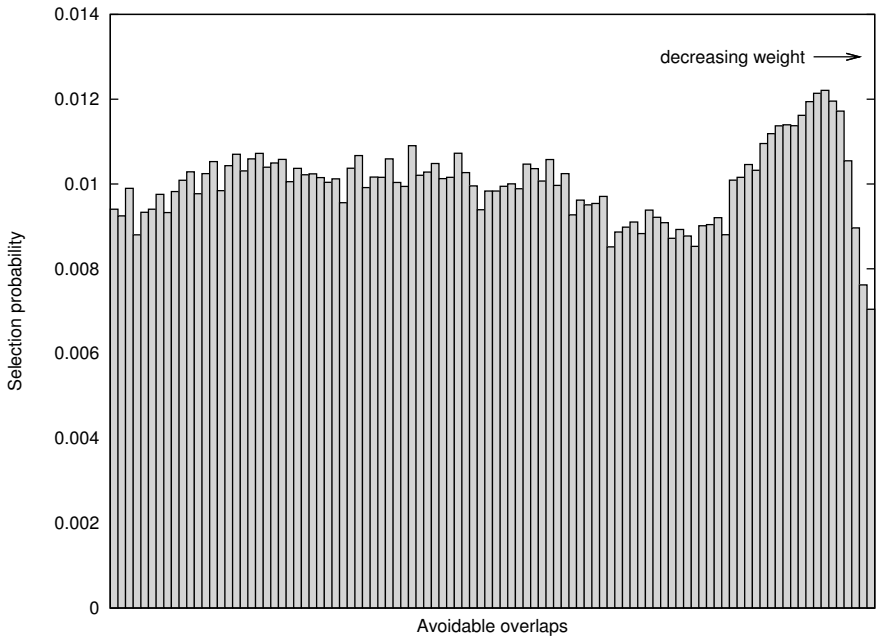


Fig. 5. Distribution of selected candidates by weight

operations can be even more beneficial. We noted in Section 5 that our optimal algorithm's objective function can easily be extended to balance avoidance of spills against freedom of code motion. As future work, we intend to evaluate this design space to find a better tradeoff between global code motion and scheduling to optimize program performance.

6.3 Comparison of Optimal Selection vs. Greedy Heuristics

The quality of our greedy heuristic solution depends on the ordering of register pairs. If we managed to select an ordering in which all the reuse candidates in the optimal solution come first, applying the greedy algorithm would produce the optimal solution. The idea behind our ordering by decreasing weight is to select the most expensive pairs, which make the biggest difference in spilling quality, as early as possible.

To gain insight into whether this is a good idea, we look at the distribution of candidates actually selected by the optimal solver. If this distribution is skewed towards candidates of larger weight, this could be a hint that our ordering by decreasing weight is a sound approach; otherwise, analyzing the distribution might suggest better alternatives.

Figure 5 shows a histogram representing the distribution of reuse candidates in the optimal solutions we found. We obtained this figure by producing a 0-1 'selection vector' for each set of candidates ordered by decreasing weight, where

a 0 entry represents 'not selected', and a 1 represents 'selected'. We divided each selection vector into 100 buckets of equal size and computed the population count (number of 1s) normalized by bucket size for each bucket. The histogram is the sum of all of these normalized vectors.

Apart from the small peak towards the lower end of the weight scale, the distribution of the selected candidates is quite even. Thus this analysis does not suggest a good weight-based ordering for the greedy analysis, although starting with a certain subset of lower-weight pairs might result in a slightly better overall selection than with our current ordering by decreasing weight.

7 Summary and Conclusions

In this paper we presented GCMS, a global code motion algorithm for minimal spilling. In GCMS, we consider all possible overlaps between live ranges in a function, taking all possible placements of instructions and schedules within basic blocks into account. From all overlaps that can be avoided, we select a profitable candidate set to avoid. These candidates can be removed from the register allocation problem's conflict set; if a candidate is chosen for reuse of a processor register, we apply the associated changes to the dependence graph that models all code motion and scheduling possibilities. However, if enough registers are available for an allocation without spilling, we do not restrict scheduling or code motion and can aggressively move code out of loops.

We evaluate both optimal and greedy heuristic solutions of the candidate selection problem. Our evaluation shows that global code motion can reduce spilling and occasionally improve program performance, but it often performs code motions that can lead to an overall performance regression. On the other hand, restricting our optimal algorithm to perform only local instruction scheduling leads to consistent improvements in performance over a state-of-the art heuristic scheduler. Finding a restricted form of GCMS that more carefully balances the needs of the spiller against aggressive code motion is future work.

References

[AEBK94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In *Proceedings of the International Conference on Programming Languages and System Architectures*, number 782 in Lecture Notes in Computer Science, pages 125–136, London, UK, 1994. Springer-Verlag.

[Bar11] Gergö Barany. Register reuse scheduling. In *9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, Chamonix, France, April 2011. Available from http://www.imec.be/odes/.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 241–255, New York, NY, USA, 1991. ACM.

[CBD11] Quentin Colombet, Florian Brandner, and Alain Darte. 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, pages 25–34, New York, NY, USA, 2011. ACM.

[CCK97] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. In *Computers and Mathematics with Applications*, 1997.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[Cli95] Cliff Click. 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, 1995.

[CSG01] Josep M. Codina, Jesús Sánchez, and Antonio González. 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, pages 175–184, Washington, DC, USA, 2001. IEEE Computer Society.

[EK91] M. Anton Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, 1991.

[EK12] Mattias Eriksson and Christoph Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1):19:1–19:24, June 2012.

[GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM.

[GYA+03] R. Govindarajan, Hongbo Yang, J.N. Amaral, Chihong Zhang, and G.R. Gao. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transactions on Computers*, 52(1):4–20, Jan. 2003.

[HS06] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In David Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, number 4228 in Lecture Notes in Computer Science, pages 346–361. Springer Berlin / Heidelberg, 2006.

[JM03] Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 1–16, Berlin, Heidelberg, 2003. Springer-Verlag.

[Lam88] M. Lam. 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, pages 318–328, New York, NY, USA, 1988. ACM.

[NP93] C. Norris and L. L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 804–813, 1993.

[NP95a] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 169–179, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[NP95b] Cindy Norris and Lori L. Pollock. Register allocation sensitive region scheduling. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 1–10, Manchester, UK, 1995. IFIP Working Group on Algol.

[Pin93] Shlomit S. Pinter. Register allocation with instruction scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 248–257, New York, NY, USA, 1993. ACM.

[SE02] Bernhard Scholz and Erik Eckstein. 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, pages 139–148, New York, NY, USA, 2002. ACM.

[Tou01] Sid Ahmed Ali Touati. Register saturation in superscalar and VLIW codes. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 213–228, 2001.

[Win07] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 43–55, Washington, DC, USA, 2007. IEEE Computer Society.

[WLH00] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 121–133, New York, NY, USA, 2000. ACM.

[XT07] Weifeng Xu and Russell Tessier. 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*, pages 113–122, New York, NY, USA, 2007. ACM.

[ZJC03] Huiyang Zhou, Matthew D. Jennings, and Thomas M. Conte. Tree traversal scheduling: A global instruction scheduling technique for VLIW/EPIC processors. In *Languages and Compilers for Parallel Computing (LCPC)*, volume 2624 of *Lecture Notes in Computer Science*, 2003.