PowerPC Reference Compiler Optimization Project

Design Document, Version 1.0

IBM Israel - Science and Technology Center

1 Scheduler Design

This chapter contains the design for extending GCC's instruction scheduler to interblock scheduling. The goal of this project, as well as possibilities for future work in GCC are shortly described in Section 1.1. Section 1.2 describes the existing instruction scheduler of GCC. General restrictions and requirements affecting this task are discussed in Section 1.3. We present the primary design decisions in Section 1.4. The definitions used throughout this chapter are listed in section 1.5. Section 1.6 discusses the algorithmic problems involved in interblock scheduling, as well as solutions to them. Section 1.7 presents a high level view of the new scheduling program. Finally, the algorithms for interblock scheduling are detailed in section 1.8.

1.1 Current Goal and Future Work

The goal of this project is to enhance GCC's instruction scheduling phase. The current scheduler supports intrablock scheduling only. It will be enhanced to perform interblock scheduling. The scope of this enhancement is discussed in Section 1.4.

Possible future work on GCC, aimed at enhancing performance on PowerPC as well as other processors includes:

- Global register allocation
- Software pipelining
- Instructions selection (for PowerPC)

1.2 Existing Scheduler

GCC contains a basic block (bb) instruction scheduler. The scheduler is invoked under optimization level O2 and higher. It is invoked twice: before and after register allocation. The scheduler loops on the program's basic blocks and performs backward list scheduling of each blocks. This is as opposed to most familiar instruction schedulers, which perform forward list scheduling. In backward scheduling the block is scanned from bottom to top and instructions are moved down. In forward scheduling the block is scanned from top to bottom, and instructions are moved up. Before scheduling a bb live information at the entry to the bb, as well as registers and memory data dependences, are computed. No flow information is needed. The registers' last-use information is maintained throughout the scheduling process. Line number notes containing debugging information are also preserved throughout the scheduling process.

Note: This is just a brief and incomplete description. For more information see GCC's user's manual and the source file sched.c (5000 lines).

1.3 Restrictions and Requirements

Here we discuss general restrictions and requirements that will influence both the design and the implementation.

1. Reuse of the existing code:

   In order to make using the enhanced interblock scheduler easy and natural for GCC's developers reuse the existing infrastructure and code.
2. Use the same code for intra/inter block scheduling:

   For higher maintainability of the scheduling code it is preferred that the same C code will do both.
3. Preserve global information throughout scheduling:

   To preserve the correctness of the compiled program some global information should be updated.
4. GCC is considered a fast compiler. To preserve this, implement the optimizations in the lowest possible compile time overhead.

1.4 Primary Design Decisions

Our primary design decisions are concerned with the types of new interblock motions and the major changes to the existing scheduler. Namely:

- Interblock motions that will be implemented
- Interblock motions that will NOT be implemented
- When to invoke the scheduler
- Reversing the list scheduler
- Developing a scheduler visualization mechanism

1. Interblock motions that will be implemented:

   We move instructions between blocks that are in the same 'region'. For our purposes, a region is either an inner loop or a loop-free subroutine (see definitions in Section 1.5). Two such motions will be implemented:

   - Useful

     This is the basic interblock motion. An instruction is moved from block b1 to b2 iff b1 and b2 execute under exactly the same conditions.

   - Speculative

     This motion is practically essential for being able to do interblock motions at all. After moving an instruction speculatively, it might be executed by under conditions that would not originally cause it to be executed.

2. Interblock motions that will NOT be implemented:

   - Duplicative

     Benefits from this motion are not assured, whereas its implementation is expensive in both compile-time and development effort.

   - Software-pipelining and data dependences collapsing

     Our experience shows that to benefit from software-pipelining, the scheduler should be enhanced by the ability to collapse reg-reg data dependences at compile time. This optimization turns out to be mostly important for numeric Fortran programs, which are not in the focus of this project. In addition, its implementation is expensive in development effort, compile-time, and the size of the resulting executable program. Therefore, this optimization does not fit into the framework of this project. However, it is an important optimization, and our implementation will allow for such a future extension.

3. When to invoke the scheduler?

   The current invocations of the scheduler are before and after register allocation. The first invocation is just after the "combine" optimization, and the second is after "reload". We see no reason to change this. We will replace the intrablock scheduler by an interblock scheduler capable of both inter and intra block scheduling. In the first invocation, some blocks will be subject to interblock scheduling, while the rest will be be subject to intrablock scheduling. In the second invocation, all blocks will be subject only to intrablock scheduling. This is because (1) register allocation introduces many new anti-dependences that constrain the scheduler, (2) the "important" interblock motions should have been performed in the first invocation and (3) the compile time cost of an additional invocation of interblock scheduling is high.

4. Reversing the list scheduler:

   Here we explain why forward list scheduling is more adequate for interblock scheduling than backward list scheduling.

   The natural direction to move insns from one bb to another is in the same direction that they are moved inside basic blocks. To demonstrate the disadvantage of reverse list scheduling for interblock motions, we examine two cases of speculative scheduling:

   - Case 1:

     Consider the following program fragment:

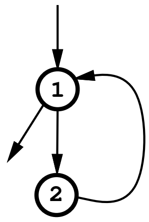
```
 C Code                      If stmnt          bb 2 code

for (i=1;i<N;i++) {            1               load r1=a[i]
  if (e) {                     | \             add r2=r1+1
    a[i]++;                    |   2           store a[i]=r2
  }                            | /
}                              3
```

If insns are moved down, speculative scheduling cannot help this program, since stores are not allowed to move speculatively. If insns are moved up, the load and the add can be moved speculatively into block 1.

- Case 2:

  Consider the following control flow graph:

  Figure 1.

  This is a loop, with block 1 as entry, back-edge (2,1), and a single exit from block 1. If insns are moved down, we cannot move anything between blocks, unless duplication is supported. If insns are moved up, we can move speculatively from block 2 to block 1.

Also note that extending forward interblock scheduling with software pipelining is feasible. We cannot say the same for backward scheduling.

In addition, forward scheduling is more natural, in the sense that it is choosing the next instruction to schedule in the same order that the processor will execute the instructions.

Because of all these reasons, we have decided to invert the list-scheduler.

5. Develop a scheduler visualization mechanism

   Currently, it is very hard to tell if the scheduler does a good job or not by examining its debugging output. (The main part of this output is the RTL after scheduling.) Although each insn in RTL is described with many characters and lines, we will try to come up with a dense representation of the scheduler work, called "visualization of the scheduling process". This is a time table, containing one column for each unit-instance, and one line for each real clock tick. This time table demonstrates the instruction assignments made by the scheduling process.

1.5 Definitions

Here we define terms that are used throughout this chapter.

1. RTL

   Register Transfer Language. GCC's intermediate representation.
2. Control flow graph

   A graph whose nodes are the basic blocks of the program, and whose edges describe the possible flow of control between these blocks.
3. Succ(b), Successor of b:

   An immediate flow successor of b. I.e., edge (b,succ(b)) exists in the control flow-graph.
4. Succ-edges(b)

   All the edges leaving from block b (including back-edges).
5. Pred(b), Predecessor of b:

   An immediate flow predecessor of b. I.e., edge (pred(b),b) exists in the control flow-graph.

   Note: Frequently, by pred(b) we mean all the immediate predecessors of b. The same holds for succ(b). E.g. pred(succ(b)) are all the immediate predecessors of all the immediate successors of b.
6. Region-to-sched

   This is used in the interblock scheduling component. Also referred to as region. An insn cannot move beyond the boundaries of its region. A region:

   - Must have a single entry.
   - Can be a subroutine without loops.
   - Can be an inner loop with single entry.
   - Can be a bb not contained in any other region.
7. Region-entry

   A block in the region. This is the single entry to the region.
8. Region-exit:

   A block not in the region, which is a successor of one of the region blocks. There can be more than one region-exit.

   Notes:

   - Back edges sometimes require special treatment in regions. Back edges are ignored, except for:
     1. Relations used "globally":
        - Succ
        - Pred

These relations may be applied outside region boundaries by the algorithm for identifying inner loops.

2. Relations applied only inside regions:
   - Succ-edges
   - PS
   - Split-edges

When dealing with the relations of reachability, succ/pred inside regions, ancestor-edges and domination, back edges are ignored and the region may be viewed as having an acyclic control flow-graph.

9. As you will see below, we use the term "block" when referring to basic blocks in the context of the control flow graph of the entire procedure. We use "bb" to refer to basic blocks that are either in the region being scheduled, or are region-exits of that region.
10. While succ() and pred() are computed for blocks, and are available at any time for every block in the procedure, the definitions below are computed for bb's in the region being currently scheduled.
11. Reachability:

    Block b2 is reachable from b1 iff there is a control flow path from b1 to b2. (By the note above, this path contains no back edges).
12. Block-probability(b):

    The probability of reaching block b from the region-entry.
13. Ancestor-edges(b):

    All edges in the region which are on paths from the region-entry to b.
14. Domination:

    Block b1 dominates b2 iff every path from region-entry to b2, must pass through b1. Every block dominates itself.
15. Duplicative:

    Motion from b2 to b1 is duplicative iff b1 does not dominate b2.
16. Post-domination:

    Block b2 post-dominates b1 iff every path from b1 to a region-exit, must pass through b2.
17. Speculative:

    Motion from b2 to b1 is speculative iff b2 does not post-dominate b1.
18. Useful:

    Motion from b2 to b1 is useful iff b2 post-dominates b1.
19. Target-block, Source-block
    - Target-block is the block being scheduled.
    - Blocks from which instructions can be moved to the target-block are the source-blocks.

20. Split-blocks(s,t):

    For speculative motions from s to t, the bbs in which life info should be checked.

21. Split-edges(s,t):

    All edges (b1,b2), such that b1 is reachable from t, s is reachable from b1 but not from b2. Back-edges are also considered here.

22. Potential-split(s), PS(s):

    All edges (b1,b2), such that s is reachable from b1, but not from b2. Back-edges are also considered here.

23. Update-blocks(s,t):

    For speculative motions from s to t, the bbs in which life info should be updated.

24. Region-latch:

    A block from which a back edge emenates.

25. Region control flow graph:

    A subset of the procedure flow graph, containing all the blocks of the region, the edges connecting these blocks, and pointers to the region-entry, region-latches and back edges in the region.

26. Extended region control flow graph:

    This is the region control flow graph, extended by the region-exit blocks and by the edges connecting the region to these blocks.

1.6 Algorithmic problems involved in interblock scheduling

When performing interblock scheduling, we have to deal with several algorithmic problems. Here we briefly describe these problems and their solutions. Detailed solutions are given in subsequent sections.

1. Control-flow graph

   Since interblock scheduling deals with moving instructions between different basic blocks, the control-flow graph relations between these blocks must be considered. The control flow graph is built for the whole procedure.

2. Identification of regions

   We defined regions as the working range of interblock scheduling. Instructions will not be moved beyond the boundaries of a region. The simplest region contains a single block. In this case, we have intrablock scheduling as a special case of interblock scheduling. The typical region will be an inner loop, but a subroutine without loops is also considered a region. The region identification process takes as input the control flow graph of the entire procedure.

3. Interblock data dependences

   The existing mechanism for computing intrablock data dependences between instructions should be extended to compute data dependences between instructions from different blocks. This computation is performed per region.

4. Interblock instruction motion analysis

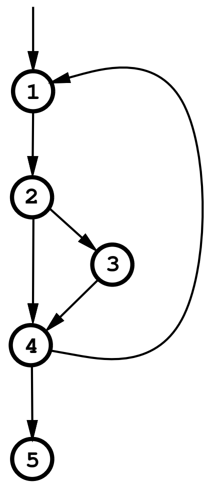
   This analysis is the core of interblock scheduling. The output from the analysis answers the following questions:

   - Is a motion possible?
   - Is a motion speculative or useful?
   - For speculative motions:
     - What is the probability?
     - In which blocks should life information be checked (split-blocks)?
     - In which blocks should life information be updated (update-blocks)?
     - For a speculative load: is this load exception-free?

The analysis of interblock instruction motions can be illustrated by the following example.

Example 1:

Figure 2.



This is an inner loop region. The loop header is b1. b1 is therefore the loop entry. The back-edge is (b4,b1), and the latch bb is b4. There is a single exit from this loop, to b5.

We now briefly describe the method of analysis using illustrations from Example 1 above. Let INSN be an instruction that is candidate to move from source block S to target block T.

- "Is a motion possible?"

  Recall that only non-duplicative scheduling is supported. Hence, in addition to requiring that S can be reached from T, we also require that T dominate S. To check this, we compute the domination relation for the region's blocks.

  In Example 1:

  - b1 dominates all other blocks.
  - b2 dominates b3 and b4.
  - b3 and b4 do not dominate any other block.
  - Possible interblock motions are:
    - b2 to b1, b3 to b1, b4 to b1.
    - b3 to b2, b4 to b2.
  - Motion from b4 to b3 which requires duplication is not allowed.

- "Is a motion speculative or useful?"

  Different treatment is needed for these two motion types. By definition, the motion is useful iff S post-dominates T (in addition to the requirement that T dominate S). So one possibility is to compute the post-domination relation between the blocks. We have chosen a more efficient method, which will be explained later.

  In Example 1, the only possible interblock motions which are speculative are "b3 to b2", "b3 to b1".

- "What is the probability?"

  For a speculative motion, we use the probability of reaching S from T for comparing with other motions. To compute this probability, we first compute the probabilities of S and T relative to the region entry, and take their ratio.

- "In which blocks should life information be checked (split-blocks)?"

  Note that for a speculative motion, there must exist a path from T that doesn't reach S. The first block in such a path, from which S cannot be reached, is called an split-block. Now, if the moved INSN defines a register REG, we must check that REG is not alive at the beginning of each such split-block.

  In Example 1, when moving speculatively from b3 to b2, we must check registers' life information in b4. In other words:

  split-blocks(b3,b2) = {b4}

  To allow efficient computation, on demand, of the set of split-blocks for a given ordered pair (S,T), we first compute a set of edges called potential-split (PS) for each block in the graph.

  Intuitively, potential-split (PS) for a block S can be thought of as:

  PS(S) = { (b1,b2) | S is reachable from b1, but not from b2 }

  In Example 1, entry=b1, and we get: PS(b1)={}, PS(b2)={}, PS(b3)={(b2,b4)}, and PS(b4)={}.

With these sets available for S and T, we can compute split-edges(S,T) by the difference PS(S) - PS(T).

In Example 1, for moving from b3 to b2, split-edges(b3,b2) = PS(b3)-PS(b2) = {(b2,b4)}, and from this we get: split-blocks(b3,b2) = {b4}.

- "In which blocks should life information be updated (update-blocks)?"

  For a speculative motion, if the moved INSN defines a register REG, we must also update that REG has become alive in certain blocks, so as to prevent moving another instruction that defines the same REG, from the above computed SPLIT blocks.

  In Example 1, when moving speculatively from b3 to b2, registers life information should be updated at the entry to b3.

- "Is this load exception-free?"

  For a speculative motion of a load instruction, there is risk of causing an "illegal-memory-access" exception that would not have occurred in the first place. Therefore, we only move loads if we know that they will not cause an exception or if we can prove that if the moved load will cause an exception, an exception will be caused by another load whether we move our load or not.

  This kind of proof is obtained by examining the load instruction, and other instructions "in the vicinity":

  - We define five classes of speculative loads: i-free, i-risky, p-free, p-risky, and m-free.
  - I-free loads are loads that are proved to be exception-free, just by examining the load insn. We put the following restriction: it is not allowed to examine the base register of the load. We can only examine the memory reference attached to the load insn. Examples for such loads are loads from TOC and loads of global data.
  - I-risky loads are loads that are proved to be exception-risky, just by examining the load insn. Again: we cannot examine the base register. Same restriction will hold for the rest of the section about exception-free loads. Examples for such loads are volatile loads and loads from shared memory.
  - P-free loads are loads for which we can prove, by examining other insns, that they are exception-free. Currently, this class is empty.
  - P-risky loads are loads for which we can prove, by examining other insns, that they are exception-risky. Currently we have two proofs for such loads. The first proof detects loads that are guarded by a compare on the base register. This proof is not affected by the scheduling process. Let load-insn be the examined load. Load-insn is P-risky iff ALL the following hold:
    - insn1 is not in the same block as load-insn
    - there is a DEF-USE dependence (insn1,load-insn)
    - compare-insn is a compare, not in the same block as load-insn
    - load-insn is reachable from compare-insn

– there is a DEF-USE dependence (insn1,compare-insn)

This proof might fail when the compare and the load are fed by an insn not in the region. To solve this, we will add to this group all indexed loads (whose address is comprised of two registers) and all loads that have no input DEF-USE dependence.

The second proof detects loads that are fed by a speculative def. This proof is affected by the scheduling process. After each speculative motion of an insn, we will record the fact that this insn was moved speculatively. The proof follows; Again let load-insn be the examined load. If there exist a DEF-USE dependence (insn1, load-insn) such that insn1 was already moved speculatively, then load-insn is p-risky.

– M-free (maybe-free) loads are all the remaining loads. They may be exception-free, but we cannot prove it.

Now, all loads in i-free and p-free classes are considered exception-free, while all loads in i-risky and p-risky classes are considered exception-risky. As for loads in the m-risky class, these are considered either exception-free or exception-risky, depending on whether we are pessimistic or optimistic. We have to take the pessimistic approach to assure the safety of our optimizations, but we will take the optimistic approach to evaluate the need for more accurate analysis.

1.7 High level view of the new scheduling program:

The structure of the new scheduling program is:

```
(1)  Compute control-flow-graph
(2)  Compute regions-to-sched
     For each region:
(3)    Compute flow-related-info
(4)    Compute data-dependence-info
       For each block:
(5)      Compute valid-source-blocks
(6)      Do list-scheduling
(7)    Update region-global-data
```

We now expand each element of the structure above, and point to differences from the original scheduling program.

1. Compute control-flow-graph:

   This is a new module in the scheduler, that is using the existing partitioning of the code into basic blocks.

   Why isn't it worthwhile to use GCC's code for computing this information?

Flow graph is built in a previous phase of GCC (called flow). But there is no easy way to ensure that this information is entirely correct when scheduling starts. In addition, we cannot reuse the code for computing the flow-graph, because:

- The representation of the flow graph is not appropriate
- The computation of the flow is not a stand-alone code.

2. Compute regions-to-sched:

   This is a new module.

   A scheduling region (region-to-sched) is an interblock scheduling range. Instructions can't move beyond the boundaries of a region. The simplest region contains only one basic block: In this case intrablock scheduling is a special case of interblock scheduling.

   When a procedure is split into scheduling regions, inner loops are identified in the procedure flow graph, using the appropriate algorithm from Section 1.8. If there are no inner loops in the procedure, the whole procedure is one big scheduling region. Otherwise, a scheduling region is either an inner loop or a basic block which does not belong to an inner loop.

   If the scheduler is called after register allocation, only intrablock scheduling is performed.

3. Compute flow-related-info:

   This is a new module.

   This information is later used by the list-scheduling algorithm to analyze instruction motions. The information includes:

   - Domination relation
   - Potential-split relation
   - Block probability

4. Compute data-dependence-info:

   This is an update to an existing module of the scheduler.

   There are two types of data dependences:

   - Register-register dependency
   - Memory-memory dependency

   With minor changes to the computation of register-register dependences we can compute interblock dependences (see algorithm below). We are looking for a similar extension to the computation of memory-memory dependences.

5. Compute valid-source-blocks:

   This is a new module.

   For a given target-block t, computes the source blocks from which insns can be moved to t. As explained above, these are the region blocks dominated by t.

   For each such source block s (dominated by t):

   - Compute split-blocks(s,t)

- If split-blocks(s,t) is not empty, then the motion is speculative, and additional information is computed:
  - Probability(s,t)
  - Update-blocks(s,t)

   Note that if the probability is "too low", s is an invalid source.

6. List-scheduling:

   This is an update to an existing module of the scheduler.

   As a consequent of our primary design decisions, we will implement a forward intrablock scheduler. The same list scheduler is also used for the interblock scheduling. It maintains a ready list for instructions whose dependences are all resolved.

   The main changes to the list scheduler are:

   - Ready list of candidates: Include instructions from all valid source blocks.
   - Choosing best instruction: Consider probabilities of speculative instructions.
   - Life information: Check before speculative motions (split blocks). Update after speculative motions (update blocks).
   - Safety of loads: Must prove for moving speculative loads.

7. Update region-global-data:

   To preserve the correctness of the compiled program, the following information should be updated:

   - Debugging information (line-notes, etc.)
   - Registers live information
   - Registers last-use information

1.8 Algorithms for the Interblock Scheduling

Here we give a high level description of the main algorithms used by our interblock scheduler. A summarizing example that follows demonstrates some of these algorithms.

The algorithms are:

A1. control-flow-graph

A2. find single-entry inner-loops

A3. block-probability

A4. domination

A5. ancestor-edges

A6. potential-split (PS)

A7. split-edges

A8. split-blocks

A9. update-blocks

A10. list-scheduling

A11. reg-reg-data-dependences

A12. mem-mem-data-dependences

1.8.1 Compute Control Flow Graph

Input: An RTL of a procedure, annotated with partitioning into basic blocks.

Output: For each block its successors and predecessors.

Method: Perform two passes through the basic blocks:

1. Map between each label and the block in which it is defined.
2. Set successors and predecessors, using the mapping built in pass 1.

1.8.2 Find Single Entry Inner Loops

General: Use depth first search traversal

Input: Control flow graph

Output: Single-entry inner-loops

Method: Employ a two pass algorithm.

In the first pass, all the headers of inner loops are found, using a dfs traversal of the control flow graph. During this traversal we compute for every block b:
—Is b is a loop header
—Is b is an inner loop header

For every block in an inner-loop we also compute the header of the inner-loop to which it belongs.

In the second pass, for each such header we locate the blocks in the corresponding inner loop, and verify that the loop has a single entry (i.e. the loop is reducible). The algorithm uses a queue of blocks. Initially, the queue contains all the region-latches (i.e. blocks b in the same loop as the header, such that an edge (b,header) exists). Then we traverse back, adding all the traversed blocks. If a traversed block s is not in the same region, then this is an irreducible loop.

1.8.3 Compute Block Probability

General:

1. Work within region boundaries.
2. Assume a single entry to the region.
3. Assign edges probabilities: If profile information available, use it.
   Otherwise:
   - Loop exit branch: Exit=10/loop=90
   - Conditional branch: 50/50
   - N-way branch: 100 div n

Input: Region control-flow-graph

Output: For every block, its probability relative to region entry

Method: The computation is based on the equation:

$$P_{block}(entry) = 100$$
$$P_{block}(b) = \sum_{q \in pred(b)} P_{block}(q) \times P_{edge}(q, b)$$

1.8.4 Compute Domination Relation

General: The implementation follows the algorithm in Aho pp. 670.
Work within region boundaries.
Ignore back edges.

Input: Region control-flow-graph

Output: For every block b, the blocks dominating it.

Method: The computation is based on the equation:

$$D(entry) = \{entry\}$$
$$D(b) = \{b\} \cup \bigcap_{q \in pred(b)} D(q)$$

1.8.5 Compute Ancestor Edges

General: Work within region boundaries.
Ignore back edges.
Used only for the computation of potential-split.

Input: region control-flow-graph.

Output: for every block, the edges on paths from the region-entry to that block.

Method:

$$ancestor\_edges(b) = \bigcup_{a \in pred(b)} ancestor\_edges(a) \cup \{(a,b)\}$$

1.8.6 Compute Potential Split

General: Work within region boundaries.
Uses the ancestor_edges relation.

Input: extended region control-flow-graph, region entry.

Output: for every block, its potential-split (PS) set of edges.

Method: The computation will be based on the equation:

$$PE(entry) = \{\}$$

$$PE(b) = [\bigcup_{a = pred(b)} PS(a) \cup succ\_edges(a)] - ancestor\_edges(b)$$

where succ_edges(a) stands for all the edges emerging from a, and '–' denotes set difference.

1.8.7 Compute Split Edges

General: These are the edges on which life information should be checked to assure that an instruction can be moved speculatively. Work within region boundaries.

Input: Target block t.
Source block s.
Region control flow graph, annotated with PS information.

Output: Split-edges for moving from s to t.

Method: The computation will be based on the equation:

$$split\_edges(s,t) = PS(s) - PS(t)$$

where '–' stands for set difference.

1.8.8 Compute Split Blocks:

General: These are the blocks in which life information should be checked to assure that an instruction can be moved speculatively. Work within region boundaries.

Input: Target block t.
Source block s.
Region control flow graph.
Split-edges relation.

Output: Split blocks for moving from s to t.

Method: The computation will be based on the equation:

$$split\_blocks(s,t) = \{b \mid \exists (a,b) \in split\_edges(s,t)\}$$

1.8.9 Compute Update Blocks:

General: These are the blocks in which life information should be updated after an instruction was moved speculatively. Work within region boundaries.

Input: Target block t.
Source block s.
Region control flow graph, annotated with PS information.

Output: Update blocks for moving from s to t.

Method: We first compute two temporary sets:

$$check\_blocks(s,t) = \{a \mid \exists\ (a,b) \in split\_edges(s,t)\}$$

$$update\_edges(s,t) = succ\_edges(check\_blocks(s,t)) - split\_edges(s,t)$$

Now we can compute:

$$update\_blocks(s,t) = \{a \mid \exists\ (a,b) \in update\_edges(s,t)\}$$

1.8.10 List Scheduling

General: Reuse the intrablock list scheduler

Input: RTL of a program, and all the data computed above.

Output: For every block b, a new ordering for b.

Method: For each target block t:

- Initiate ready list by all valid instructions with all data dependences resolved.
- Repeat until all instructions of t are scheduled:
  −Sort the ready list
  −Schedule the first (best) ready instruction
  −Add to ready list all valid instructions with all data dependences resolved.

Notes: Here we explain the details of this algorithm that are unique to interblock motions:

1. A valid instruction must belong to a valid-source-block (see "computation of valid-source-blocks" in Section (G) above).

   If the motion is speculative, then:

   - Life-information must be checked: Assume the motion is from block s to block t. If the instruction defines a register R, R should not be alive at the beginning of any block in split-blocks(s,t).
   - Some instructions are not allowed to move speculatively. This includes store instructions, floating point computation instructions that may cause exceptions, etc..
   - If the moved instruction is a load, we must prove that this load-motion is exception free.
2. The sorting of ready list now also takes into account the type of a motion (useful,speculative) and the probability associated with the motion. Also, the register pressure heuristic needs to be very careful since interblock scheduling can be more demanding in register life ranges than intrablock scheduling.
3. Scheduling an instruction: for speculative instructions, we also need to recheck live registers in split-blocks(s,t) just before the motion, and update live registers in each of the blocks of update-blocks(s,t) immediately after the motion.

1.8.11 Reg-reg Data Dependences

General: Reuse the algorithm that computes reg-reg-data-dependences inside a basic block

Input: RTL of a region, and region control-flow-graph information.

Output: For every block insn, the insns on which it depends

Method: In order to understand our extension, we first explain the existing intrablock algorithm for computing reg-reg dependences:

Scan block insns bottom-to-top. maintain two lists for registers seen: DEFs and USEs. For each insn seen, update its dependences according to the contents of the two lists, and update the lists as well.

Our extension concerns the initialization of these lists at the bottom of each block:

- For each bb, initialize two empty lists:
  –DEFs(bb) = {}
  –USEs(bb) = {}
- Scan bbs in topological reverse order (bottom to top):
  –Inside the bb, apply the current computation
  –At the end of bb (reaching bb's top):
  –For each bb1, predecessor of bb:

$$DEFs(bb1) = DEFs(bb1) \cup DEFs(bb)$$

$$USEs(bb1) = USEs(bb1) \cup USEs(bb)$$

Note: The current intra-bb reg-reg dds computation may be sophisticated, and avoid creating the (transitive) def-def edge in the case:

(1) def(R) —> (2) Use(R) —> (3) def(R)

This is legal in the case of intra-bb computation. But in our extension, it is possible that the "(2) Use(R)" and the "(3) def(R)" originate from different two bbs, not reachable from each other, that are successors of the block contacting "(1) def(R)". in this case, we must also create an edge "(1) def(R) —> (3) def(R)". Therefore, we will verify that the intra-bb reg-reg dds computation adds the def-def edge in such cases.

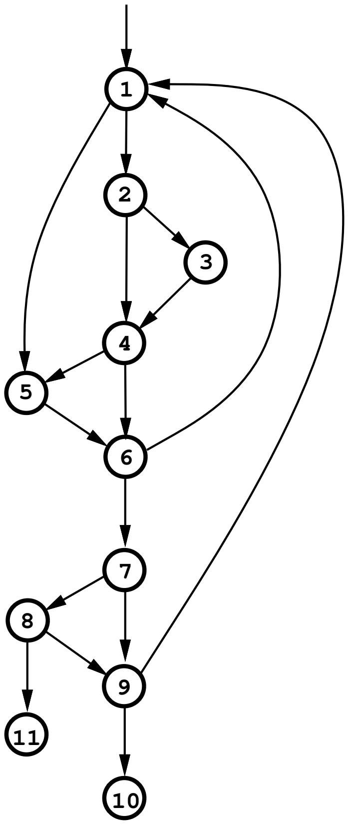
1.8.12 Mem-mem Data Dependences

We are in the process of learning this problem. We intend to extend the existing computation in a similar way to reg-reg-data-dependences.

1.8.13 Example 2: Extended region control flow graph

In the following example we demonstrate the computation of ancestor-edges, PS, dominators, split-blocks and update-blocks relations.

Figure 3



bb	dominators
1	1
2	1,2
3	1,2,3
4	1,2,4
5	1,5
6	1,6
7	1,6,7
8	1,6,7,8
9	1,6,7,9

```
bb  ancestor-edges
--  --------------
1
2   (1,2)
3   (1,2),(2,3)
4   (1,2),(2,4),(2,3),(3,4)
5   (1,2),(2,4),(2,3),(3,4),(4,5),(1,5)
6   (1,2),(2,4),(2,3),(3,4),(4,5),(1,5),(4,6),(5,6)
7   (1,2),(2,4),(2,3),(3,4),(4,5),(1,5),(4,6),(5,6),(6,7)
8   (1,2),(2,4),(2,3),(3,4),(4,5),(1,5),(4,6),(5,6),(6,7),(7,8)
9   (1,2),(2,4),(2,3),(3,4),(4,5),(1,5),(4,6),(5,6),(6,7),(7,8),(7,9),(8,9)
```

```
bb  PS
--  -----------
1
2   (1,5)
3   (1,5),(2,4)
4   (1,5)
5   (4,6)
6
7   (6,1)
8   (7,9),(6,1)
9   (8,11),(6,1)
```

```
(s,t)  PS(s)-PS(t)    Split-blocks     Update-blocks
-----  -----------    -------------    -------------
(2,1)  (1,5)          5                2
(3,1)  (1,5),(2,4)    5,4              3,2
(4,1)  (1,5)          5                2
(5,1)  (4,6)          6                5
(6,1)
(7,1)  (6,1)          1                7
(8,1)  (7,9),(6,1)    9,1              8,7
(9,1)  (8,11),(6,1)   11,1             9,7

(3,2)  (2,4)          4                3
(4,2)

(7,6)  (6,1)          1                7
(8,6)  (7,9),(6,1)    9,1              8,7
(9,6)  (8,11),(6,1)   11,1             9,7

(8,7)  (7,9)          9                8
(9,7)  (8,11)         11               9
```