HEWLETT PACKARD

# Meld Scheduling: A Technique for Relaxing Scheduling Constraints

Santosh G. Abraham, Vinod Kathail, Brian L. Deitrich*
Compiler and Architecture Research
HPL-97-39
Feburary, 1997

E-mail: [abraham, kathail]@hpl.hp.com
briand@crhc.uiuc,edu

instruction scheduling,
global scheduling,
meld scheduling,
latency constraint propagation,
instruction-level
parallel processors,
compiler optimization

Meld scheduling melds the schedules of neighboring scheduling regions to respect latencies of operations issued in one region but completing after control transfers to the other. In contrast, conventional schedulers ignore latency constraints from other regions leading to potentially avoidable stalls in an interlocked (superscalar) machine or incorrect schedules for non-interlocked (VLIW) machines. Alternatively, schedulers that conservatively require all operations to complete before the branch takes effect produce inefficient schedules.

In this paper, we present general data structures for maintaining latency constraint information at region boundaries. We present a meld scheduling algorithm for non-interlocked processors that generates latency constraints at the boundaries of scheduled regions and utilizes this information during the scheduling of other regions. We present a range of design options and describe the reasons behind our particular choices. We cover certain pitfalls and discuss how to develop an algorithm that addresses these issues. We extend the algorithm to take advantage of interlocked processors by selectively propagating latencies across region boundaries. We evaluate the performance of meld scheduling on a range of machine models on a set of SPEC 92 and Unix benchmarks. We investigate the sensitivity of the performance improvements due to changes in issue width and instruction latencies.

*University of Illinois, Urbana, Illinois

An earlier version of this report was published in and presented at the *1996 Int. Symposium on Microarchitecture (MICRO-29)*, Paris as "Meld scheduling: Relaxing scheduling constraints across region boundaries".

## 1 Introduction

Scheduling algorithms are usually oblivious of constraints coming in from blocks or regions that are already scheduled and hence may make scheduling decisions that lead to poor schedules. In a non-interlocked machine, the processor does not interlock to ensure that the inputs are available before issuing an operation. For such machines, the compiler schedules code to guarantee that an operation completes before a dependent operation issues. Within a scheduling region, the scheduler delays the issue of a dependent operation to ensure that its inputs are available. Across scheduling regions, a scheduler must ensure that certain constraints are satisfied on entry or exit of a region. For instance, one convention is to assume that on entry all operations have their inputs available. In this case, the scheduler must guarantee that all operations complete before control is transferred to another region. In contrast, a meld scheduler generates latency constraints imposed by scheduled regions, propagates constraints to the boundaries of a region to be scheduled, and translates these constraints to edge constraints recognized by the local region scheduler.

The example in Figure 1 illustrates the benefits of meld scheduling. Assume that load and arithmetic operations take four and three cycles respectively. In Figure 1(a), the conventional scheduler delays the branch to issue in cycle `7`, in order to guarantee that the load operation `e` in block `B1` completes before the branch is taken. In this example, there is an immediate use of `r6` in cycle `0` of block `B3`, but the conventional scheduler is oblivious of the schedule in block `B3` and always delays the branch. As illustrated in Figure 1(b), meld scheduling relaxes the scheduling constraint on the branch and schedules it in cycle `5` of block `B1`. When scheduling `B3`, meld scheduling recognizes the incoming constraint from the load `e` to its use, and delays the issue of operation `l` to cycle `2`. In this example, meld scheduling is able to reduce the schedule length of block `B1` without increasing the schedule length of block `B3`. Eventually, the propagation of latency constraints to successor blocks does increase the schedule length of block `B5` by `2` cycles. But the savings in the schedule length in the more frequently executed blocks more than makes up for the loss in the less frequently executed blocks. The weights on the blocks represent the profiled or estimated execution frequency of the block. In this example, meld scheduling improves weighted schedule length, a measure of overall execution time, by approximately 10%.

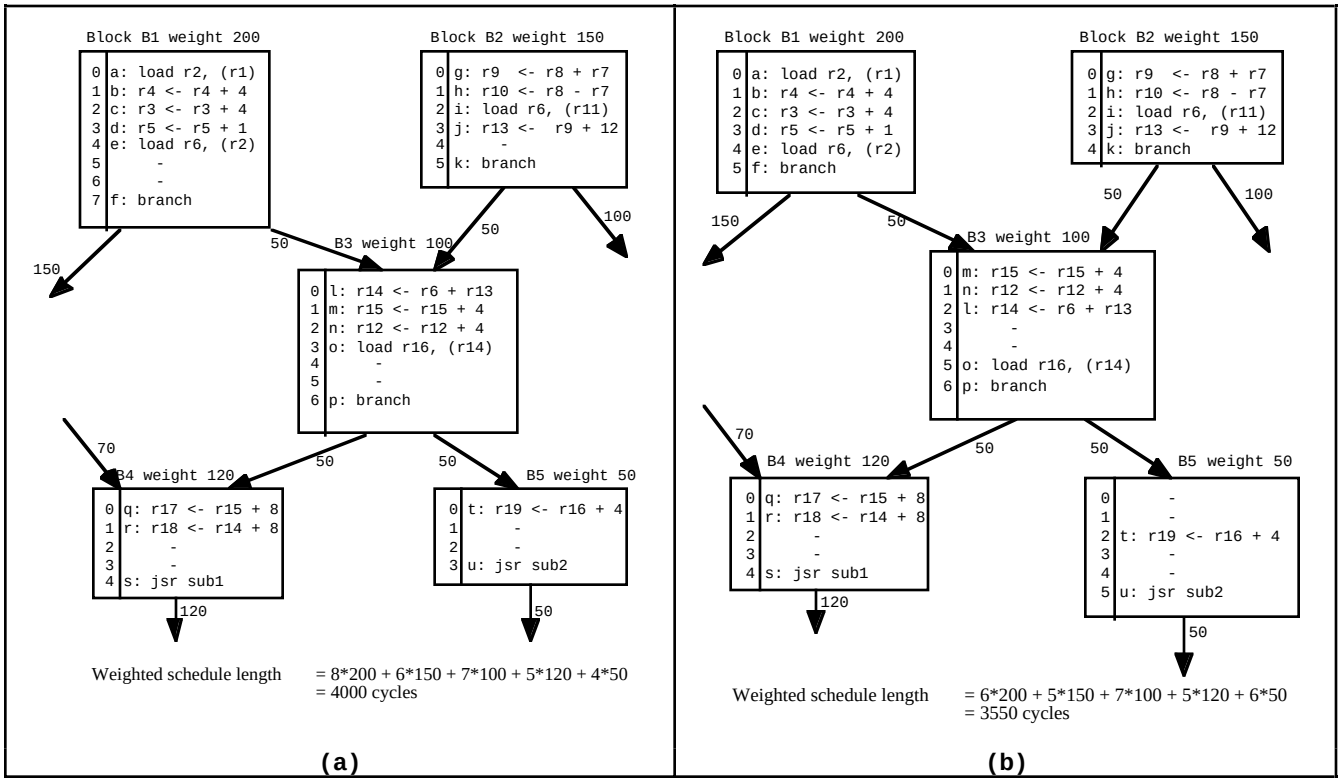



**Figure 1. Example illustrating the performance improvement of meld scheduling (a) Conventional scheduler (b) Meld scheduler**

The utility of meld scheduling for interlocked machines is less apparent. In an interlocked machine, the processor detects that inputs to an operation are not yet available using interlock hardware and delays the issue of the operation dynamically through stall cycles. For such machines, the scheduler assumes the inputs to be available at the entry and furthermore, considers branches for placement as soon as the operations generating live-out values are scheduled. Thus, the scheduler for an interlocked machine can schedule branches in cycles 5 and 4 of blocks B1 and B2, optimistically assuming that there will not be immediate uses of live-out values in successor blocks. However, if there are immediate uses of the live-out values, the processor stalls till completion of the live-out generating operations. In the example in Figure 1, the use of r6 in cycle 0 of block B3 causes stalls of 2 and 1 cycles, when the predecessor block was B1 and B3 respectively. In contrast, with meld scheduling the branches are scheduled early, but the uses of live-outs from predecessor blocks B1 and B2 are delayed in block B3 to cycle 2. Thus, an interlocking machine does not stall on the meld-scheduled code in Figure 1b. Meld scheduling is not necessary for correctness in interlocked machines, but can lead to improved overall performance through reductions in stall cycles. Meld scheduling as described in this section does not produce a net improvement over conventional scheduling for this example. We describe a variant of meld scheduling for interlocked processor that propagates latency constraints selectively to successor blocks in section 3.10.

Scheduling techniques that schedule larger regions reduce the potential for performance improvement due to meld scheduling. For instance, superblocks are composed of a linear trace of basic blocks [1]. In contrast to conventional basic block scheduling which does not propagate latency constraints to its successor blocks, superblock scheduling automatically considers latency constraints imposed by operations scheduled so far in basic blocks that are part of the superblock. Thus, superblock scheduling can provide much of the same benefits as meld scheduling in conjunction with basic block scheduling. Other global scheduling techniques [2-7] are cognizant of latencies over larger regions. Since the number of dynamic scheduling boundaries that are crossed decreases when larger scheduling regions are employed, the additional performance improvement due to meld scheduling decreases as the scheduling regions get larger. As developed in this report, meld scheduling can be applied in conjunction with almost any choice of scheduling region short of the entire function to provide additional performance improvements. Meld scheduling is complementary to schedulers that operate on individual regions, enabling higher performance for a given scheduling region or enabling the use of smaller scheduling regions for a desired level of performance.

The rest of this report is organized as follows. The next section presents the architectural model. Section 3 describes the meld scheduling algorithm. Section 4 presents experimental evaluation of meld scheduling. Section 5 describes related work and the final section summarizes the results of this work.

## 2 Architecture models

As we have discussed earlier, meld scheduling can improve the performance of both VLIW and superscalar processors. For most of this report, we use a family of non-interlocked VLIW machines based on the HPL PlayDoh architecture [8]. However, in Sections 3.10 and 4.3, we extend and evaluate meld scheduling algorithms on in-order interlocked superscalar processors.

Each machine has a set of integer, floating point, memory and branch units and is capable of issuing an instruction containing several operations in each cycle. Each instruction consists of a set of operations, where each machine operation is a RISC-style operation with source and destination operands.

We assume that functional units are fully-pipelined. Thus, operations compete for resources in the issue cycle only, and operations from different instructions (necessarily issued in distinct cycles) do not compete for resources. As a consequence, we need to maintain only latency constraints and do not need to consider resource usage constraints across region boundaries. This reduces the amount of inter-region information and simplifies constraint propagation during meld scheduling. The technique presented in this report, however, can be extended to handle resource-usage constraints for machines which do not have flow-through pipelines.

In order to investigate the effects of varying issue width and latencies, we consider a range of machine models parameterized by width and latency. The four combinations of widths labeled W1 through W4 are presented in Table 1 and the four combinations of latencies labeled L1 through

**Table 1. Number of units in various machine models**

| | Integer units | Floating point units | Memory units | Branch units |
|---|---|---|---|---|
| W1 | 1 | 1 | 1 | 1 |
| W2 | 2 | 2 | 2 | 1 |
| W3 | 4 | 2 | 2 | 1 |
| W4 | 4 | 4 | 4 | 1 |

**Table 2. Latencies of operations in various machine models**

| | Int ALU | Float Add | Int/ Float Mpy | Int/ Float Divide | Load | Store | Branch |
|---|---|---|---|---|---|---|---|
| L1 | 1 | 1 | 1 | 4 | 1 | 1 | 1 |
| L2 | 1 | 3 | 3 | 8 | 2 | 1 | 1 |
| L3 | 2 | 4 | 4 | 12 | 3 | 1 | 1 |
| L4 | 5 | 6 | 10/12 | 15/16 | 5 | 3 | 3 |

L4 are presented in Table 2. A particular machine model, W2L2, is a particular width choice (W2) and latency choice (L2) from this table, capable of issuing up to seven operations consisting of two integer, two floating-point, two load/store operations and one branch operation.

## 3 Meld scheduling algorithm

In this section, we first present the data structures used for representing inter-region constraints, then describe the overall algorithm followed by more detailed descriptions of various refinements. Figure 2 illustrates how block B3 in Figure 1 is meld scheduled and we will use Figure 2 as the running example in this section.

### 3.1 Data structures for inter-region constraints

The meld scheduler represents and transforms inter-region constraints using a set of latency maps and a distance map.

DEF: A *distance map* specifies the shortest distance in cycles from each entry to each exit of a scheduled region.

In Figure 2 on the next page, the distance map for block B1 specifies the distance from entry a to exit f to be 6 cycles. Since a basic block has only one entry and one exit, its distance map contains only one value. In a single-entry region, the distance map provides the distance from the entry to each exit. In a superblock, the distance to each exit is its *exit time*, the sum of the schedule time of the exit (branch) operation and the branch latency.

DEF: A *latency map* is a table of operand and latency dangle pairs; when queried with a particular operand, it either returns the associated latency dangle (if present in the table) or returns a default value.

Entries can be created, modified, or removed from the map and the default value of a map may be modified. The latency dangle is one of two types.

DEF: A *required dangle* of an operand specifies the number of cycles after control leaves a region through an exit (or enters a region through an entry) that an operation defining/using the operand completes.

DEF: An *available dangle* of an operand specifies the number of cycles after control enters a region through an entry (or leaves a region through an exit) that an operation defines/uses the operand.

In order to ensure that the required dangles that a scheduled region exports through its exits are honored by its successors, each unscheduled region imports required dangle constraints through its entries. In Figure 2, Block B1 exports a required dangle of 2 on r6, indicating that r6 is written two cycles into a successor block, by the load operation in flight on exit from B1. The required dangle on r6 is a constraint that unscheduled successor blocks must satisfy when they are scheduled eventually. Block B3 imports a required dangle of 2 in order to enforce the constraint that r6 must not be referenced for the first two cycles of the schedule for B3. Similarly, in order to exploit the available dangles that a scheduled region exports through its entries, each unscheduled region imports available dangles through its exits. In Figure 2, Block B4 exports an available dangle of 1 on r14, indicating there is no use of r14 till cycle 1 by operation r. The available dangles indicates the ability of a scheduled region such as B4 to absorb some of the latencies from its predecessors. Block B3 imports an available dangle of 1 on r14, permitting operations that define r14 to complete up to one cycle after control transfers out of block B3. The available dangles relaxes constraints on scheduling B3.

A particular latency map is characterized and named by three independent attributes: export/import, entry/exit, and def/use. Thus, Figure 2 shows the import entry def map, one of the latency maps for Block B3, at the top left-hand corner of Block B3. We describe these attributes next.

DEF: An *export map* of a region contains constraints that this (necessarily) scheduled region imposes on other as yet unscheduled regions.

DEF: An *import map* of a region contains constraints that other neighboring scheduled regions impose on this (necessarily) unscheduled region.

DEF: An *entry* map is at the entry of a region.

DEF: An *exit* map is at the exit of a region.

DEF: A *def map* contains constraints due to definitions of variables/operands.

DEF: A *use map* contains constraints due to uses of variables/operands.

Export maps are created after a region is scheduled and are persistent throughout the scheduling process. Figure 2 shows the latency map structures just before B3 is scheduled. Thus, blocks B1, B2, B4, which have been scheduled, have export maps at their entries and exits, even though only the export maps relevant to scheduling B3 are shown. Import maps for a region are created just before a region is scheduled and are destroyed immediately after the region is scheduled. Thus, when B3 is scheduled, only the import maps for B3 are live. B5 does not have any meld data structures because it is as yet unscheduled.

The export maps are formed by inspecting the schedules for a region. The required dangles from the export exit maps are propagated to the import entry maps. Thus, the export exit def maps of blocks B1 and B2 are used to form the import entry def map for B3. The available dangles from the export entry maps are propagated to the import exit maps. Thus, the export entry use map for B4 is used to form the import exit use map for B3. The import map constraints are translated to constraints on dependence edges that are recognized by the local scheduler. We will discuss the dependence edge constraints shown in Figure 2 later in Sections 3.6 and 3.7. Note that the def and use constraints are completely orthogonal; data flows from export def maps to import def maps and from export use maps to import use maps. Depending on the scheduling discipline and machine model, some maps may be unnecessary. For instance, in certain machine models latencies of anti- and output dependencies are always zero and consequently, required use dangles are always zero. For

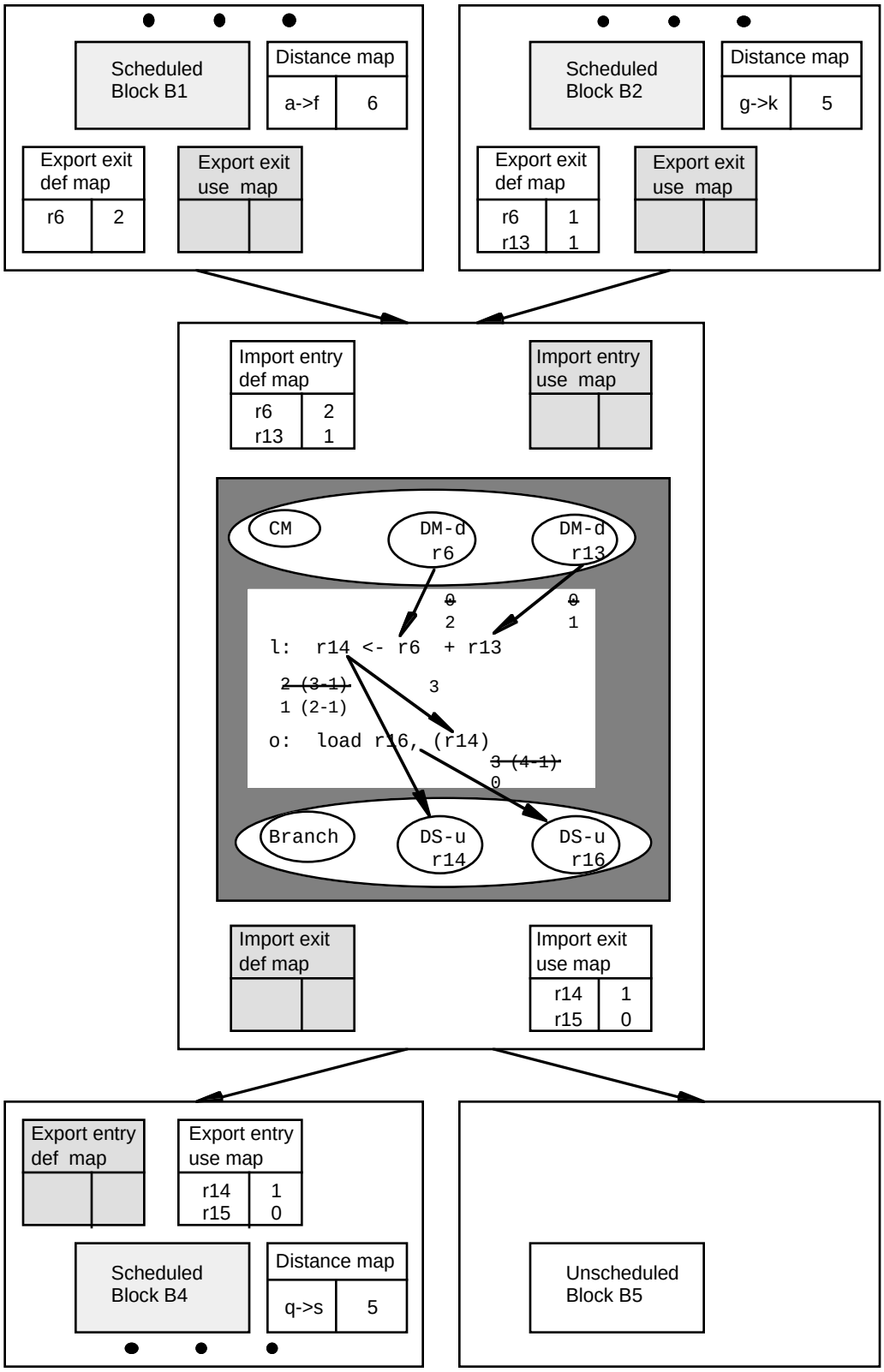



**Figure 2. Propagation of latency constraints and conversion to edge constraints**

such machines, the shaded maps in Figure 2, such as the export exit use map, can be eliminated.

The default value of an entry export map specifies the maximum available dangle for all operands. For instance, we do not allow latency dangles into procedure calls and returns and require all operations to complete before a procedure call or return. Thus, in order to ensure that all operations complete before the procedure call in cycle 4 of block B4, the incoming latency dangle must be limited to 4 or less. We enforce this constraint by setting the default value of the export entry maps of block B4 to 4. The default value of an export exit map is always set to zero.

Drawing an analogy to data-flow problems, the export maps together with the distance maps specify the transfer function for a scheduled region. Just before a region is scheduled, the constraints at each exit (up-exposed constraints) are determined by examining the entry export maps and distance maps of scheduled successors. Similarly, the constraints at each entry (down-exposed constraints) are determined by examining the exit export maps and distance maps of scheduled predecessors. The latency maps themselves form the gen sets in the data-flow problems. The distance maps indirectly form the kill functions, reducing each operand-latency pair by the distance and killing any pairs whose latency drops to zero or less.

### 3.2 Overview of the algorithm

An overview of the meld scheduling algorithm is shown in Figure 3. The scheduling regions tile the entire function and we iterate over the scheduling regions in some priority order. Once a region is picked for scheduling, we create the meld data structures for this region. We propagate the export maps from neighboring predecessor scheduled regions and build the import entry maps. We transfer the required dangles in the import entry maps to edge constraints. We propagate the export maps from neighboring successor scheduled regions and build the import exit maps. We use the available dangles in the import exit maps to relax the edge constraints to the exits. Using these augmented edge constraints, we schedule the region. After scheduling, we construct export maps for each entry and exit. We calculate the minimum entry to exit distance for each pair of entry and exit nodes and construct the distance map for this region.

```
Iterate over scheduling regions in some priority order {
    • Propagate export exit maps from predecessors and
      build import entry maps
    • Transfer required dangles from import entry maps
      to edge constraints
    • Propagate export entry maps from successors and
      build import exit maps
    • Use available dangles from import exit maps to
      relax edge constraints
    • Schedule region
    • Inspect schedule and build export maps and
      distance maps for this region
}
```

**Figure 3. Meld scheduling algorithm overview**

### 3.3 Lazy versus eager propagation

We have implicitly assumed lazy propagation of constraints implicitly; just before a region is scheduled, we propagate constraints to the region and construct its import maps. Lazy propagation collects constraints from other scheduled regions only when necessary. However, lazy propagation may visit a scheduled region many times, each time for constructing the import map for a different region. In contrast, eager propagation always keeps the import maps at the boundaries between scheduled and unscheduled regions up to date. Thus, when the meld scheduler picks a region for scheduling, it forms the region's import maps by locally combining the import maps of immediate predecessors/successors. However, after a region is scheduled, eager propagation must update the import maps of all scheduled regions reachable through other scheduled regions. Eager propagation may update a particular import map many times, each time after scheduling a different region. Thus neither lazy nor eager propagation is consistently better than the other. Lazy propagation has the advantage that the import maps are transient data structures. We believe that lazy propagation is easier to debug and maintain, because the validity of persistent data structures (export maps and distance maps) can be verified by locally examining a scheduled region. We use lazy propagation in our implementation and experiments.

### 3.4 Iteration order

In conventional scheduling, the order in which regions are scheduled does not affect the schedule lengths because the scheduler is oblivious of the schedules of other regions. In meld scheduling, a region scheduled earlier in the iteration order only needs to honor constraints from a few already scheduled regions. Therefore, in comparison to conventional scheduling, regions scheduled earlier have relatively shorter schedule lengths and regions scheduled later have longer schedule lengths. Therefore, we sort regions in decreasing profiled or estimated execution frequency order and schedule them in sorted order. In the example in Figure 1, blocks B1 and B2 have the highest execution frequency, are scheduled earlier by the meld scheduler and have shorter schedule lengths than in the conventional schedule. In contrast, block B5 has a lower execution frequency, is scheduled last, and has a longer schedule length than with the conventional scheduler.

### 3.5 Propagating latency constraints to import maps

The concept of distance is important in propagating latency constraints. The *distance* from a region exit to another region entry is the minimum number of cycles before control can transfer from the exit to the entry along any path of scheduled regions. The length of a path consisting of scheduled regions is the sum of the schedule

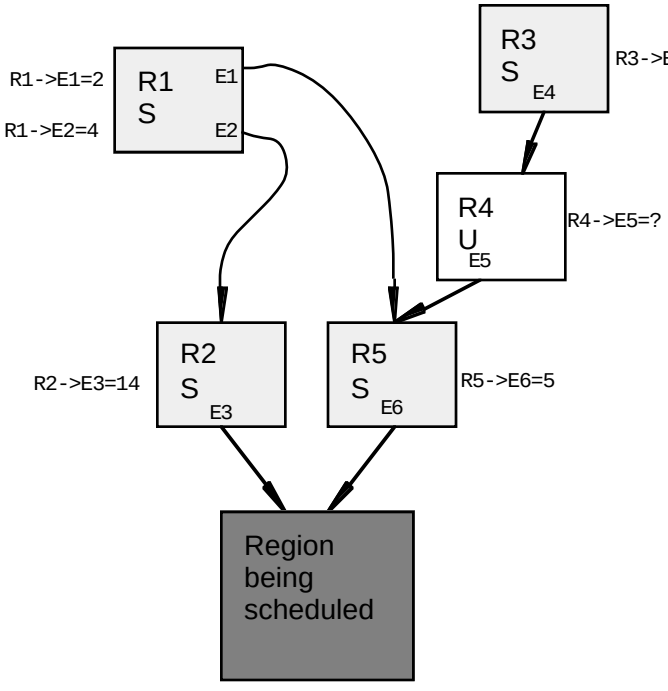
lengths of the individual region segments. The distance from a region exit to another region entry is the minimum length along all such paths. The distance determines how much a required dangle from an exit is reduced when seen at a region entry. For instance, the required dangle from a region exit of, say 10 cycles, is reduced to 4 cycles from the viewpoint of a region entry that is at a distance of 6 cycles from the exit. Similarly, the available dangle at a region exit is the sum of the dangle for the corresponding operand at a successor region entry and the distance to that entry.

Consider the computation of import entry maps at an entry of a region being scheduled. We separate this computation into two steps: first, calculating the distance from the exits of regions reachable through scheduled regions to this entry and second, composing the maps at these exits to form the import entry maps. We calculate the distance using a modified version of Dijkstra's shortest path computation algorithm (see [9]). We form a unique region exit identifier by combining the region and exit operation identifiers. In the distance table, we associate two pieces of information with each exit identifier: the current shortest path length from this exit and whether this exit has been expanded. An exit first enters the distance table in the unexpanded state. Subsequently, when we inspect the predecessors of this exit, we mark the exit as expanded.

We start the process by considering the immediate predecessor exits associated with the control-flow edges incident on the entry of the region being scheduled. Since unscheduled regions do not export constraints, we filter out exits that belong to unscheduled regions. We enter the remaining exits in the distance table as unexpanded with the distance component set to zero. In each iteration, we pick an exit with the minimum distance component among all unexpanded exits and expand it. This process terminates when there are no more unexpanded exits in the distance table.

We expand each exit by first determining the region entry for this exit. The constant `max_dangle` is set to the maximum expected dangle, typically the latency of the longest operation. The distance of this entry to the entry of the region being scheduled is the sum of the schedule length from the region entry to the exit and the distance component of the exit. If this computed distance exceeds `max_dangle`, any dangles propagated through this entry-exit pair will be absorbed before it reaches the entry of the region being scheduled. Therefore, we move on to the next unexpanded exit. Otherwise, we include the immediate predecessor exits of the entry in the distance table. If the distance table already contains the exit, the associated distance is set to the minimum of the current distance value and the computed distance through the expanded exit. On termination, the distance components in the table are the desired distances from each of the exits to the entry of the region being scheduled.

The example in Figure 4 illustrates the distance calculation scheme. The left-hand side of Figure 4 shows the control flow graph, with scheduled regions shaded and marked with an S and unscheduled regions not shaded and




| Step | Action | | |
|---|---|---|---|
| | Map | Dist. | Expanded |
| Step1 | Expand entry | | |
| | R2E3 | 0 | N |
| | R5E6 | 0 | N |
| Step2 | Expand R2E3 | | |
| | R2E3 | 0 | Y |
| | R5E6 | 0 | N |
| Step3 | Expand R5E6 | | |
| | R2E3 | 0 | Y |
| | R5E6 | 0 | Y |
| | R1E1 | 5 | N |
| Step4 | Expand R1E1 | | |
| | R2E3 | 0 | Y |
| | R5E6 | 0 | Y |
| | R1E1 | 5 | Y |

**Figure 4. Example illustrating distance calculation**

marked with an U. The right hand side of Figure 4 shows the exit that is expanded in each step and the updated distance table after each step.

In Step 1, the entry of the region being scheduled is expanded and its immediate predecessor exits are entered in the table, viz. E3 and E6, with distance of 0. In Step 2, we choose the exit E3 for expansion. We consider the immediate predecessor scheduled exits for inclusion in the table; in this case, E2. We use the available entry-exit distance map to calculate the distance of the minimum path from such exits to the entry through the exit being expanded. For exit E2, this distance is 14+0=14. In this example, we assume that max_dangle is 10. Since the distance from E2 exceeds 10, we know that no dangle through this exit can reach the entry and so we do not include E2 in the table. After marking E3 as expanded, in Step 3 the algorithm expands E6. The distance from E1 to the entry through E6 is 5 and we enter this information in the table. Exit E5 is also an immediate predecessor exit of region R5. However, since R4 is unscheduled, we ignore this exit and all reachable predecessor exits through E5, viz. E4. We do not need to consider required dangles through E4 because region R4 will eventually absorb those dangles. Once all the exits in the table are expanded, the algorithm terminates and the table contains the minimum distances to all relevant exits.

Once the distances are available, we iterate over all the exits in the distance table and apply the composition procedure shown in Figure 5 to each of the export exit maps and distances associated with the exits. As described in Figure 5, in each composition step, we iterate over the constraints in an export exit map. For each such constraint, we look up the import entry map for a corresponding constraint and update it.

After computing the import entry maps, we use a similar procedure to calculate the distances from an exit to all relevant entries and compose the export entry maps to determine the import exit maps.

```
procedure compose_import_entry_map (export_exit_map,
        distance, import_entry_map) {

    Iterate over pairs in export_exit_map {
      Set dangle_operand and required_dangle from pair;
       req_dangle_at_entry = required_dangle - distance;
      if (req_dangle_at_entry > 0 AND
          (dangle_operand is not in import_entry_map OR
            req_dangle_at_entry > current_dangle in
                                        import_entry_map)) {
        enter (dangle_operand, req_dangle_at_entry) in
            import_entry_map;
      }
    }
}
```

**Figure 5. Computing import entry map from export exit maps**

### 3.6 Latency map to dependence edge constraints

In this step, we transfer the constraints in the import maps into constraints that are recognized by the local scheduler. The entry import maps contain required dangles that place additional constraints on scheduling operations relative to the entry. Typically, these constraints are enforced by placing edges from the entry node to individual operations with latency markings indicating the minimum amount by which an operation should be delayed relative to the entry. The exit import maps relax constraints on scheduling operations relative to the exit. To begin with, each operation is constrained so that it completes before control leaves a scheduling region through an exit. If a particular operation's operands are not present in the import maps, then the constraint can be relaxed so as to merely guarantee that the operation is issued before control leaves the region. Otherwise, the constraint can be relaxed by the available dangle specified in the import maps.

For concreteness, we describe the details of transferring the import map constraints to edge constraints in the intermediate representation of the HP Labs research compiler, Elcor. In Elcor, each entry has a start node called a Control-Merge pseudo-operation (shown as CM in the example in Figure 2). Each CM may have associated Data-Merge (DM) pseudo-operations that define (DM-d) or use (DM-u) operands. An exit or branch operation, may have Data-Switches (DS) that define (DS-d) or use (DS-u) operands. The Data-Merges and Data-Switches are scheduled implicitly with the associated CM or Branch operation.

For each operand in the import entry maps, we ensure that there is a DM-d and a DM-u that define and use the associated operand respectively. In the example in Figure 2, the operands r6 and r13 appear in the import entry def map and therefore, we create DM-d operations that define the operands r6 and r13. The DM-u operations are not shown but are created. Similarly, we create DS-u operations for r14 and r15. In addition to dependence edges between real operations, we draw edges from DM operations and from/to the DS operations. Initially, the latencies on edges from the DM operations are set to zero and the latencies of edges to the DS operations are set to (latency of the operation - branch latency) to ensure that the operation completes before control leaves the region. For each constraint in the entry import maps, we set the latencies of the outgoing edges from the corresponding DM operations to the latency specified in the import maps. In the example in Figure 2, we set the latencies on edges originating from the DM associated with r6 to 2. For each constraint in the import exit maps, we reduce the latencies of the incoming edges to the corresponding DS operations by the latency in the import maps. In Figure 2, we reduce the latency on the edge from operation l to the DS-u by 1 (the value in the import exit map for r14) from 2 to 1. There is no constraint in the import exit map for the operand r16 and therefore, the outgoing edge latency on operation o is reduced to 0.

### 3.7 Generating export maps from scheduled region

After scheduling a region, we generate the export maps for that region. The export maps describe the constraints imposed by this region on other, as yet unscheduled, regions and are formed by scanning the scheduled code for this region.

In our implementation, the export entry maps are formed incrementally by iterating over all operations in the superblock and scanning all operands of each operation. For each scanned operand, we compute the cycle at which it starts reading/writing. If the operand is not present in the export entry map, we create an entry associating the operand with the computed value; otherwise, we replace the current value with the computed value if it is lesser.

The export exit maps are computed as follows. In one forward scan through the superblock, we incrementally form a pseudo exit def map and pseudo exit use map to record the end times that an operation finishes reading or writing an operand. When we reach a branch, we record its exit time (branch schedule time + branch latency). When we have scanned all operations with schedule times less than the exit time, we generate the export exit def map and export exit use map for the exit by subtracting the exit time from the end times in the pseudo maps and discarding any entries whose required dangle drops to zero or less.

### 3.8 Refinements

In this section, we describe the details of the basic algorithm, and propose efficient ways of handling certain special cases.

#### 3.8.1 Cycles in CFG

Cycles in the control flow graph (CFG) can potentially lead to incorrect schedules. Consider the example in Figure 6. Region `R2` has been scheduled and region `R3` is being scheduled. Consider the dependence carried through region `R2` from the load to its use in region `R3`. To this point, we have left unspecified whether the region being scheduled is treated as scheduled or unscheduled during the import map calculation. If region `R3` is treated as unscheduled, then it does not export constraints. The load may be scheduled in the last cycle of region `R3` and the use in cycle `0`. If the load latency is larger than the sum of the schedule length of region `R2` and the branch latency, the dependence between the load and its use may be violated.

In the above example, the load-use dependence is carried from the exit of the region being scheduled, through other scheduled regions, to the entry of the region being scheduled. In order to satisfy this dependence, a region being scheduled must export available dangle constraints through its entries. But, since it is not yet scheduled, these available dangle constraints are not yet known. We can still ensure that such cyclic dependencies are satisfied by exporting conservative available dangle constraints and setting the export entry maps and entry-exit distances to conservative initial values.

An obvious conservative approach is to assume that the region cannot absorb any incoming dangles. Accordingly, we set the export map entries to zero for all operands referenced in this region. and set the entry-exit distances to zero. We use these preliminary settings to calculate the import maps. If the region is contained in a cycle in the CFG in which all other regions are scheduled, these preliminary settings ensure that dangles from a region are not allowed to propagate back into the region through its entry. In the previous example, the use of the load appears in the preliminary export entry map of region `R3` and hence also appears in the import exit map with an allowed dangle equal to the entry-exit distance of region `R2`. Hence the load is constrained to complete before control enters region `R3` again through region `R2` and correctness is preserved.

In our implementation, we calculate less conservative preliminary export maps with the potential for better performance. After picking a superblock for scheduling, we first compute its entry import maps and transfer these constraints to corresponding dependence edge constraints. Then, we calculate the depth of each operation in the superblock. The depth of an operation is the length of the longest path from the start node to this operation. Alternatively, it is the earliest schedule time of the operation that will obey all dependence constraints and hence is a lower bound on schedule time on a finite resource machine. This depth is used in determining a conservative estimate for available dangle for the operands of the operation. Similarly, we use the depth of the branch operation to calculate preliminary entry-exit distances.

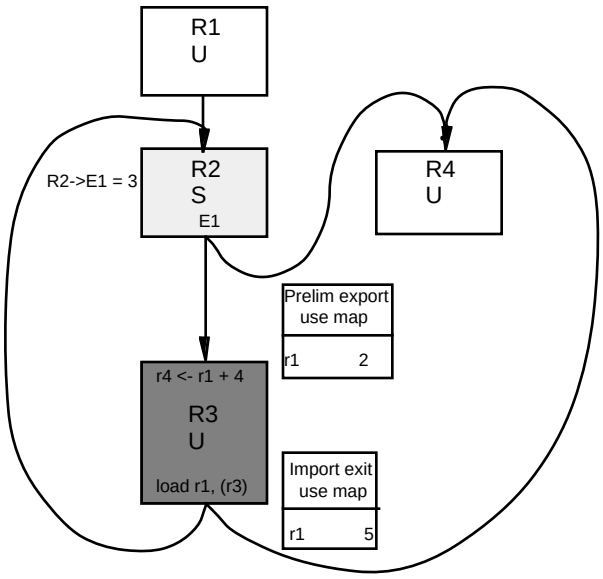



**Figure 6. Handling cycles in control-flow graph**

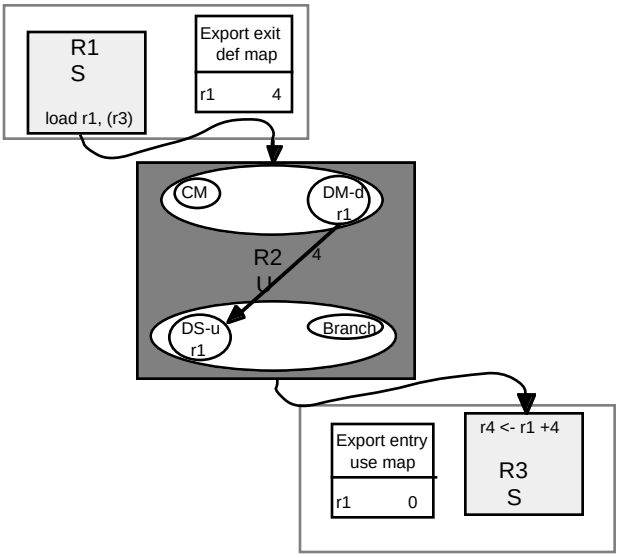



**Figure 7. Handling of pass-through dangles**

In the example in Figure 6, assume the use operation has a depth of `2`. Using this value in the preliminary export map, we calculate an import exit use map constraint of `5` on `r1`. The load may be scheduled with an outgoing dangle of up to `5`.

### 3.8.2 Pass-through dangles

In the example in Figure 7, both regions `R1` and `R3` are already scheduled and region `R2` is picked up for scheduling. Note the dependence from the load in the last cycle of region `R1` to the use in cycle `0` of region `R3`. If region `R2` is scheduled in less than 4 cycles, the dependence is not satisfied. An incoming required dangle for which there is no internal reference but an upward available dangle is referred to as a pass-through dangle. The load constraint appears in the import entry def map of region `R2` and the use constraint in the import exit use map. But the transfer of these import map constraints to edge constraints is unclear.

In our implementation, pass-through dangles are handled as follows. Data-Merges and Data-Switches are created for constraints in the entry and exit import maps. Subsequent edge-drawing connects the Data-Merge to the Data-Switch. The edge latency is set first from the entry import map; in this case to 3 (required dangle of 4 minus branch latency of 1). The edge latency is then reduced by the available dangle from the exit import map; in this case it remains at 3 because the available dangle is 0. Since the Data-Switch is implicitly scheduled with its branch, we ensure that the branch is scheduled in cycle 3 or later and the load to use dependence is satisfied in the above example.

### 3.8.3 Inter-region output and anti-dependencies

In the example in Figure 8(a), there is a definition of `r6` in the region being scheduled and an up-exposed available def dangle on `r6` of `2` cycles. The meld scheduler, as described, inserts a Data-Switch defining `r6` associated with the branch and an edge from the definition of `r6` to the Data-Switch with a latency of (divide latency (12) - branch latency (1) - available dangle (2) = 9). The insertion of this edge delays the exit branch. An alternative is to push the operation defining `r6` below the exit branch. The branch is now not delayed, potentially improving performance. A similar situation occurs with anti-dependencies. In the case of inter-region anti- and output-dependencies, the operation can be scheduled either sufficiently above or below a branch but not within a window around the branch.

Though these constraints can be handled also as resource constraints, we model these constraints somewhat imperfectly as edge constraints. We compare the depth of the defining operation and the exit operation to determine whether it is better to constrain the defining operation below or above the branch. If the defining operation has a larger depth and it is currently above the branch, we move the defining operation and all its dependent operations

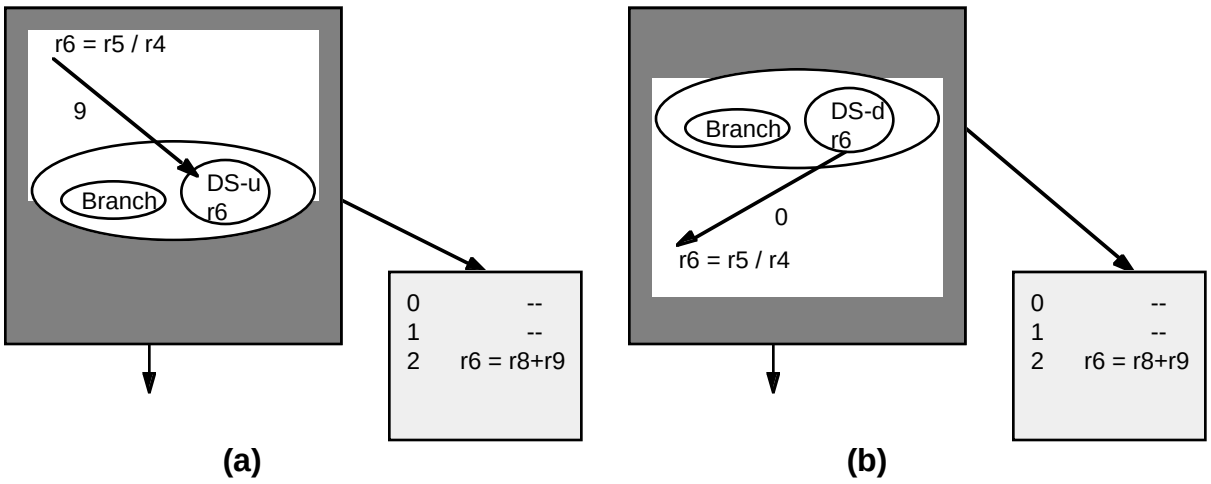



**Figure 8. Handling inter-region output dependencies**

below the branch in the sequential order. Figure 8(b) illustrates how we move the defining operation below the branch. On the other hand, if the defining operation is currently below the branch, we may choose to move it and the operations on which it is dependent above the branch. Finally, we draw an appropriate edge to satisfy the up-exposed def dangle. In Figure 8(b), we draw an edge from the branch to the definition of r6.

### 3.9 Complete algorithm

The complete algorithm for meld scheduling as implemented in Elcor, our research compiler at HP Labs, is described in Figure 9. This algorithm accommodates all the refinements described earlier in this section. Careful consideration is given in the complete algorithm to reduce the number of times the dependence graph is constructed. Our dependence graph construction step is expensive, and we do not support incremental updates to the dependence graph. The dependence graph is required before certain steps in the meld scheduler such as depth-based heuristics and transferring of map constraints to edge constraints. On the other hand, certain steps in the meld scheduler, such as pushing operations below branches or inserting Data-Merge or Data-Switch nodes require reconstructing the dependence graph.

### 3.10 Meld for in-order interlocked processors

The meld scheduling algorithm, as described so far, assumes absolutely no interlocking hardware and requires the compiler to satisfy any dangles. We now develop a meld scheduling algorithm for an in-order issue, interlocked (superscalar) processor. In our interlocking model, the processor stalls on the issue of an instruction if an instruction requires results from another issued but not completed operation and these results will not be available in time. A conventional superscalar scheduler incurs stall cycles due to inter-region dangles. In contrast, the VLIW meld scheduler as described in this report eliminates all stall cycles but increases schedule length. Our goal in developing a superscalar scheduler is to eliminate a substantial fraction of the stall cycles of a conventional superscalar scheduler while only minimally increasing schedule length.

Our adaptation of the meld scheduler for superscalars is based on the following observations. Firstly, branches are rarely mispredicted and it may be better to stall in these instances than to propagate dangles to the target regions of such branches. We refer to a control-flow edge that is predicted to be taken as a predicted edge; otherwise as a mispredicted edge. Propagating dangles through a mispredicted edge may increase the schedule length of the target region, and the target region may be more frequently executed than the mispredicted edge is taken. The alternative of not propagating dangles may cause some stall cycles when control transfers through these mispredicted edges. But this approach does not delay other paths through the target region because the schedule length of the target region is not increased. Secondly, even if stall

```
Iterate over scheduling regions in frequency order {
    • Create meld scheduling constraints data structures for region;
    • Build import entry map for the entry;
    • Construct dependence graph for the region;
    • Constrain latencies on entry edges using import entry maps;
    • Calculate depth for ops in region;
    • Set preliminary export entry map and entry-exit
      distances for the region;

    • redraw_edges = FALSE
    • Iterate over exits of region {
          - Calculate cutoff distance to limit the traversal for building import exit maps;
          - Build import exit map for this exit using the cutoff distance;
          - Handle problematic cases of inter-region anti and output dependencies as follows:
                Iterate over problematic operations {
                   Decide placement with respect to this exit using the heuristic;
                   if (operation is moved or constrained by up-exposed dangles) redraw_edges = TRUE;
     }
    • if (redraw_edges) {
          - Delete dependence edges and reconstruct dependence graph
          - Constrain entry edges using entry import maps
     }
    • Iterate over exits of region {
          - Relax latency constraints on edges to exit using import exit maps
     }
    • Schedule region
    • Sort operations in sequential order by schedule times
    • Scan and build export maps and distance maps for this region
}
```

**Figure 9. The complete algorithm for meld scheduling**

cycles are required to satisfy dangles along mispredicted edges, these stall cycles are likely to be overlapped with the misprediction penalty. The misprediction penalty is governed by the length of the entire instruction pipeline, whereas the latency dangles are proportional to the length of the pipelines of individual functional units. Most functional units have shorter pipelines than the overall instruction pipeline and dangles due to these functional units will be overlapped with the misprediction penalty. Notable exceptions are operations such as divide/square root and certain memory operations such as loading directly from second-level cache. But, these operations are not very frequent.

The superscalar meld scheduler is very similar to the original VLIW meld scheduler, except that it propagates dangles only along statically predicted-taken exits. The major changes are the control flow edge classification scheme and a change in the propagation termination tests. In our implementation, we have profile information on block execution counts and exit frequencies. Using this profile information, we classify a branch to be taken if it is taken with a probability of more than 0.5 in the profile run. Once we have classified all branches as predicted taken/not taken, we label the control flow edges from the predicted taken branches as predicted edges. The superscalar meld scheduler uses a modified termination test for latency propagation. Whereas the original VLIW meld scheduler terminated the latency propagation along a control flow edge if it encountered an unscheduled block or if the accumulated distance exceeded a threshold, the superscalar meld scheduler additionally terminates latency propagation if the control flow edge traversed is a mispredicted edge. Thus, in calculating the incoming required dangles before scheduling a block, the superscalar meld scheduler examines only the required dangles from blocks that are reachable through predicted control flow edges.

## 4 Experimental evaluation

In this section, we present experimental results obtained by our meld scheduling implementation.

### 4.1 Methodology

We use the IMPACT compiler from the University of Illinois IMPACT project to get input in aggressively-optimized superblock form [1]. The IMPACT compiler performs traditional global optimizations, unrolls loops eight times, forms superblocks and applies ILP optimizations to each superblock. The memory disambiguation information computed by the IMPACT compiler is part of the input. In addition, the input code contains profile information; each superblock is annotated with weights indicating how often each superblock is executed and how often each exit is taken. The Elcor compiler at HP Labs takes the input in superblock form, performs data-flow analyses, constructs dependence graphs and schedules each superblock. Although the Elcor compiler supports meld scheduling for both modulo scheduling and superblock/hyperblock scheduling, the experiments did not use modulo scheduling for loops.

To quantify the benefits of meld scheduling, we need a baseline model that does not propagate latency constraints. We considered two alternatives, both of which involve padding schedules with enough no-ops to ensure correct execution on non-interlocked machines. The first alternative simply allows no latency dangles from the region being scheduled. Thus, for each exit, the operations issued before the exit must also complete before the control transfers to the successor region. A problem with this approach is that it causes side-exits of a superblock to be delayed, and consequently, forces operations that are control-dependent on the branch to be also delayed. Thus, this approach increases the schedule length to other exits including the main exit from the superblock. The second alternative is to allow latency dangles through exits into successor regions but ensure that the dangles are absorbed by the *immediate* successors. This can be done as described below.

After scheduling is complete for the entire function, we inspect the immediate predecessors, calculate the maximum incoming dangle and insert a block of no-ops that absorbs this dangle. The second alternative gives better performance than the first, since no-ops are inserted once at the top of superblocks as opposed to once at each of the exits. In order to obtain a fair comparison between meld and no-meld, we use the second alternative as the baseline or *no-meld* model.

To quantify how much of the available performance improvement our technique is able to capture, we define an upper bound on the performance improvement. The upper bound is calculated by allowing regions to dangle into neighboring regions, but ignoring the dangle when calculating performance. This represents the best case in which all outgoing dangles can be exported without any attendant cost of absorbing incoming dangles. It is important to note, however, that this upper bound is unachievable in many cases on both non-interlocked and superscalar machines. We refer to the upper bound as the *ignore-dangles* model.

For each machine model, we schedule the code first using the no-meld model and then using the meld scheduler. The upper bound is computed while scheduling using the no-meld model by simply ignoring the dangles. The reported execution time improvements are based on weighted schedule lengths and not based on actual simulation. The execution time of a particular block is obtained by summing up the contributions of each of its exits. The contribution of a particular exit is the product of the number of times this exit was taken during profiling times the exit time of the exit. For a branch, the exit time is the sum of the branch's schedule time and the branch latency. This method is fairly accurate for a pure VLIW machine, because the run-time issuing of instructions tracks the compiler-generated schedule. However, it does not account for cache misses, branch mispredictions, TLB misses, etc. These other factors are expected to be similar

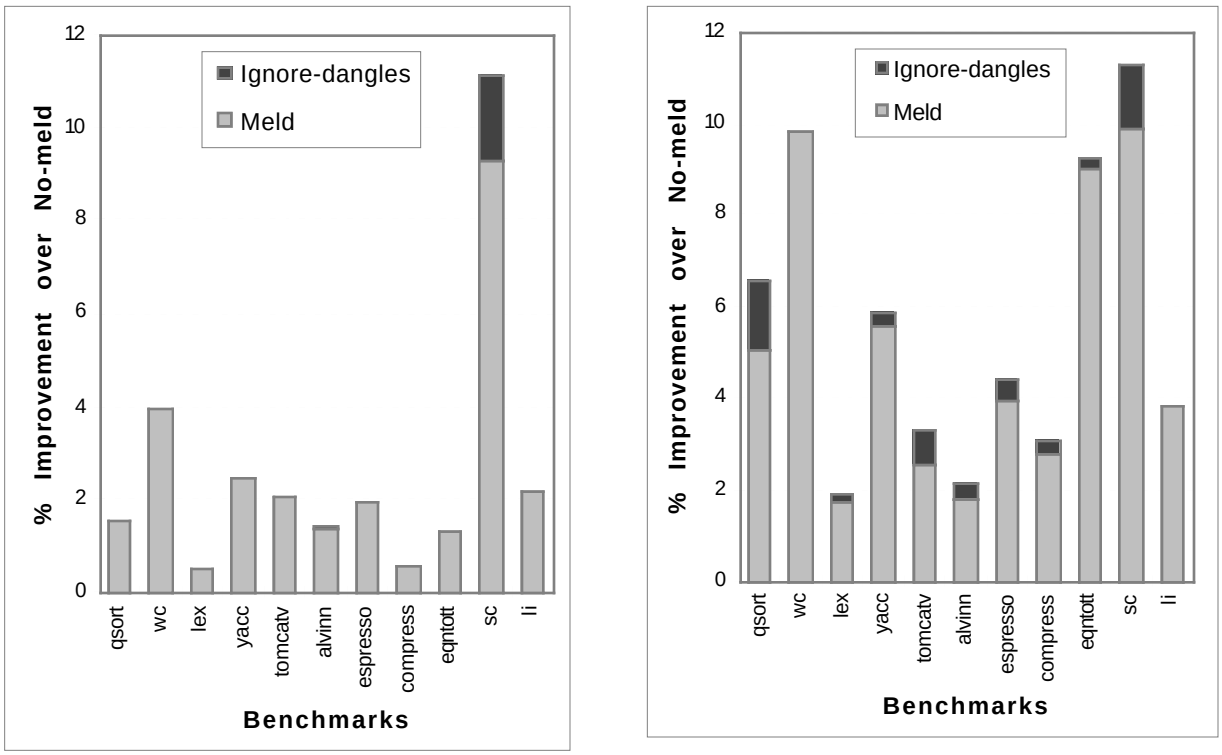



**Figure 10. Performance improvement due to meld scheduling and the upper bound for various benchmarks. (a) Machine model W2L2 (b) Machine model W2L3**

with or without meld scheduling and are not expected to affect significantly the accuracy of our evaluation of meld scheduling.

### 4.2 VLIW meld scheduler evaluation

Figure 10 shows the performance improvement over no-meld for a set of Unix and SPEC92 benchmarks on the W2L2 and W2L3 machine models described earlier in Section 2. Note that the W2 model is capable of issuing up to seven operations per cycle, consisting of two integer, two float, two memory operations and one branch. Each bar in the figure has two components. The first gray component indicates the execution time improvement of meld scheduling over the no-meld case. The second black component indicates the additional execution time improvement under the ignore-dangles model. Thus, the two components together represent the upper bound discussed earlier. Meld scheduling improves execution time between 1 and 9% for the W2L2 model and between 1.7% and 9.8% for the W2L3 model. Note that the black component in the bar graphs are very small, especially forthe W2L2 model, indicating that our meld scheduler absorbs almost all the inter-superblock latency dangles. With the longer latencies of the W2L3 model, meld scheduling gives higher execution time improvements but does not reach the (potentially unrealizable) upper bound on many of the benchmarks. For the W2L3 model, the difference between improvements due to meld scheduling and the upper bound varies from 0 to less than 1.5%.

It is interesting to compare the results for the integer benchmarks such as sc with the ones for the floating-point benchmarks such as alvinn. In both machine models, floating-point latency is higher than the integer latency. Thus, it may seem that meld scheduling should provide greater improvement on floating-point benchmarks as compared to integer benchmarks. However, floating-point benchmarks are highly loop-intensive and inter-region dangles are less of a problem, since most of the performance-critical dangles occur at the back edges. Modulo scheduling of loops [10-13] is capable of handling these dangles during scheduling. Loop unrolling provides similar benefits. In our experiments, loops were unrolled eight times reducing the impact of inter-region dangles on the overall performance. For example, the average size of the superblocks was 88 operations for alvinn and only 28 for sc. Moreover, superblocks in alvinn were executed from start to finish with a probability of 0.96, while the probability on sc was only 0.4. Thus, the dynamic number of operations executed before control transfers to another scheduling region is much larger for alvinn than for sc. Since opportunities for hiding inter-region dangles increases with the density of dynamic inter-region control transfers, meld scheduling shows better performance improvement for sc than for alvinn. Note that, for both alvinn and tomcatv, the meld scheduler absorbs almost all the inter-region latency dangles for W2L2 model and comes very close to the upper bound for W2L3 model.

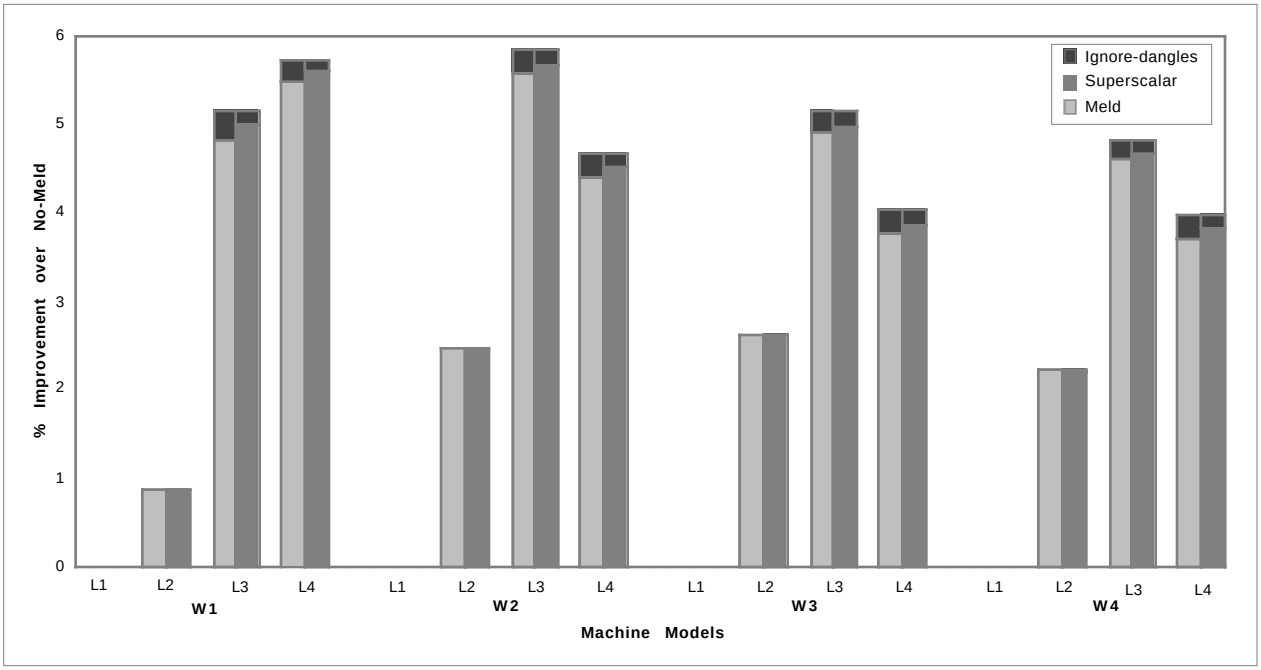



**Figure 11. Estimated performance improvement due to in-order superscalar, meld scheduling and the upper bound on various machines for the benchmark yacc**

Figure 11 shows detailed results for the yacc benchmark on all 16 machine models. We estimate the performance improvement of an in-order superscalar over the no-meld model as follows. We schedule using a conventional scheduler permitting dangles through superblock exits and ignoring dangles on superblock entries. For each control flow edge, we compute the number of stall cycles as follows. We determine the required dangles through the associated exit and the available dangles from the associated entry of the control flow edge and find the maximum difference between the required dangle and available dangle. We compute the product of the edge frequency and the stall cycles and sum over all control flow edges to obtain an estimate of stall cycles on an in-order superscalar machine. Note that this estimated performance is not based on actual simulation and does not consider the effect of overlapping stalls due to cache misses, TLB misses, branch mispredictions and other dynamic factors. Each pair of bars in Figure 11 shows the performance improvements for a particular width and latency combination. The first bar of each pair is as in Figure 10; the lower portion shows the performance improvement due to meld over the no-meld model and the upper black portion shows the remaining gap to the ignore-dangles upper bound. The lower gray portion of the second bar shows the estimated performance improvement of a superscalar machine over the no-meld model. The top black portion shows the gap between the estimated superscalar performance and the ignore-dangles upper bound. As expected, when the latencies are unity as in the L1 machine models, there are no latency dangles and hence no improvement. For the L2 models, superscalar and meld scheduling achieves the upper bound. In the other two models, L3 and L4 the gap is around 0.2%. In all the cases, estimated superscalar performance is slightly better than meld scheduling on a VLIW machine model, but by no more than a fraction of a percent.

Overall, for small latency models, the possible improvement is limited by the outgoing dangles. For wider issue widths and larger latencies, the schedule length is dominated by the critical path and the dangles become smaller relative to the schedule length. Thus, the overall improvement (as a ratio) using meld scheduling is large when latencies are large and when the available parallelism matches the machine parallelism.

### 4.3 Superscalar meld scheduler evaluation

We compared the performance of the superscalar meld scheduler described in Section 3.10 to the conventional scheduler on a set of SPEC and UNIX benchmarks on both the W2L2 and W2L3 machine models. Our performance comparisons calculate schedule length and stall cycles based on control-flow profile information and not on actual simulation. Note that the superscalar scheduler usually employs a dynamic hardware branch predictor. In cases where the dynamic predictions do not match the static predictions used by the superscalar meld scheduler, there is potential for performance loss in the superscalar meld scheduler. If the dynamic predictor correctly predicts a branch to be taken that was statically predicted not taken, there may be stall cycles incurred due to latency dangles that the superscalar meld scheduler ignored. Since our evaluation is not based on actual simulation, we effectively assume that the dynamic predictions are the

same as the static predictor, biasing the comparison slightly in favor of the superscalar meld scheduler. Recall that the conventional scheduler stalls on predicted control flow transfers. In contrast, the superscalar meld scheduler never stalls on predicted branches but potentially has longer schedule lengths. Our experiments indicate that the increase in schedule length in the superscalar scheduler was sometimes lesser and sometimes greater than the additional stalls in the conventional scheduler. Thus, the overall performance of the superscalar meld scheduler was better than that of the conventional scheduler on some benchmarks, such as `sc` and `alvinn` but worse on others such as `espresso` and `tomcatv`.

We carried out a more detailed evaluation of some cases where the superscalar meld scheduler performance was lower than the conventional scheduler. In all these cases, we found that the superscalar meld scheduler was attempting to satisfy constraints that were necessary for a VLIW machine model but could be ignored for a interlocking superscalar model. For instance, the superscalar meld scheduler scheduled code to ensure that there were no dangles through a return and that dangles through back edges are satisfied. The superscalar meld scheduler was derived from the VLIW meld scheduler and some of the VLIW constraints were not removed. Thus, our experiments did not clearly establish that the superscalar meld scheduler outperforms a conventional scheduler.

Further enhancements can be made to the superscalar meld scheduler. A first step is to ensure that the scheduler takes advantage of the interlocking hardware and ignores infrequent latency dangles, such as into calls and returns. Second, we can modify the scheduler so that inter-region constraints are treated as soft constraints which are met only if there is no increase in schedule length. Thus, the superscalar meld scheduler does not cause any increase in schedule length while attempting to reduce stalls due to inter-region dangles.

## 5 Related work

There is a substantial body of work in the area of instruction scheduling for instruction-level parallel (ILP) machines [2, 4-7, 14, 15, 1] . Most of the work, however, is directed at two related areas. The first area is the type of scheduling region, *e.g.*, trace, superblock, hyperblock, general dag, innermost loops. The motivation here is either to enlarge the scope of scheduling or to simplify compiler engineering. The second area is the actual scheduling algorithm and heuristics used within a region. Many of these scheduling techniques are developed in the context of superscalar machines [5, 6, 1]. Thus, they accurately model resource-usage and latencies within a region in order to get the best performance. But they ignore the constraints at region boundaries in the hope that any required run-time stalls will not affect the performance significantly.

In contrast, both the Multiflow Trace machine and Cydra 5 relied on their respective compilers to manage all resources and latencies. (Cydra 5 did have a latency-stalling mechanism but only for memory operations.) Consequently, the Multiflow compiler [2, 3, 14] and the Cydra 5 compiler [15] used some form of meld scheduling to ensure correctness and to get good performance. However, there is no evaluation of the benefits provided by meld scheduling over simple-minded approaches such as padding. This report generalizes the technique and quantifies the benefits of meld scheduling.

In the Multiflow compiler [3, 14], resource-usage and latency constraints at region boundaries are represented as *partial schedules*. After a trace has been scheduled, partial schedules for each entry and each exit are exported to the surrounding code. The scheduling of a trace honors some or all of the partial schedules coming from scheduled predecessors and successors. If the trace being scheduled has a scheduled predecessor at the main entry, then the predecessor's partial schedule is placed in the first few instructions of the schedule. Similarly, the partial schedule coming from a scheduled successor at the main exit is placed in the last few instructions of the schedule. After a trace has been scheduled, partial schedules at side entries are merged with the schedule. There is no special treatment given to cycles in the flow-graph. The main loop, *i.e.*, pick a trace, schedule it, export constraints, handles cycles naturally inserting appropriate compensation blocks to reconcile constraints if necessary.

In the Cydra 5 compiler [16, 15], acyclic scheduling is done at a basic-block level after a global instruction motion phase. In a basic block, instructions are scheduled from bottom to top. An unscheduled basic block honors all resource-usage and latency constraints coming from scheduled predecessors and successors. Constraints at an entry (exit) are computed by looking at all predecessors (successors), not just the immediate ones, that fall within a pre-defined number of cycles. Cycles in the flow-graph are handled as follows. The scheduler computes the constraints for the exit of the block, schedules the block except for the start node, computes constraints for the start node, and finally, places the start node in the schedule. Note that no compensation code is inserted during scheduling.

The technique presented in this report builds on the one in the Cydra 5 compiler. In contrast to both the Cydra 5 compiler and the Multiflow compiler, the technique presented in this report can accommodate general region topology and different scheduling paradigms. The constraint propagation part of the technique is capable of handling general multi-entry, multi-exit regions. The way constraints are computed and honored, by necessity, depend on the scheduler. Currently, we have implemented meld scheduling in the context of two different schedulers, modulo scheduler for loops and superblock/hyperblock scheduler for the rest of the code, and integrated them using the same framework. In contrast to the Multiflow compiler, our technique never introduces any blocks just to

reconcile constraints, since a region honors all constraints coming from its scheduled predecessors and successors. We do, however, assume flow-through pipelines and don't handle resource usage constraints across region boundaries, which both the Multiflow and Cydra 5 compilers do. This assumption reduces the amount of information and simplifies constraint propagation. The technique presented in this report, however, can be extended to handle resource-usage for machines which do not have flow-through pipelines.

## 6 Conclusions

Conventional scheduling algorithms operate without relevant information about the schedules of previously scheduled regions. In this report, we develop general data structures consisting of export maps on a per entry and per exit basis and entry-exit distance maps. We develop general and efficient algorithms for propagating this information to the entries and exits of a region being scheduled. In contrast to previous work, our meld constraint representation and propagation schemes are independent of the underlying scheduling region and scheduler.

In the context of meld-scheduling in conjunction with superblock scheduling, we identify and address a few pitfalls and additional optimizations. Latency constraints may propagate through other scheduled regions back to the region being scheduled. In the presence of such cycles, we develop techniques to detect such cases as well as to relax constraints using depth information. Latency constraints may come in from the entry and exit on an operand that is not present within the region. In this case, the scheduling of the exit must be delayed sufficiently to satisfy the constraint. In our meld scheduler, such pass-through dangles are handled in a uniform manner similar to other constraints. Upward exposed def dangles impose constraints on inter-region output- and anti-dependencies in a strict VLIW machine. Our heuristics reorder operations, if necessary, to reduce the impact of such constraints. The complete algorithm is developed with attention to space- and time-efficiency.

We define a baseline no-meld schedule in which pads of no-ops are inserted at the entry of each scheduled region to absorb incoming dangles from immediate predecessor regions. In comparison to the no-meld scheduler, our meld scheduler improved weighted schedule length, a measure of execution time, between 1% and 10% on machine models with non-unit latencies. Our meld scheduler also came close to ignore-dangles, an (unrealizable) upper bound on any meld scheduling strategy, coming within 0.2% on most machine models on yacc and within 1.5% on W2L2 and W2L3 machine models on all benchmarks.

Though the main focus of this report is on developing and evaluating a general meld scheduler, an interesting observation is that the schedule for in-order interlocked machines corresponds to the ignore-dangles schedule. Thus, the gap between ignore-dangles and meld is also an upper-bound on how much better these machines can do than a non-interlocking machine scheduled using the meld scheduler. On our benchmarks, the performance gap between the ignore-dangles upper bound and meld scheduling is small. Our detailed investigation of the yacc benchmark considered a broad range of machine models and also estimated the performance of an in-order interlocked machine. The estimated performance of the in-order interlocked machine is only slightly better than the performance of a non-interlocking machine scheduled using our meld scheduler.

## Acknowledgments

We thank Bob Rau for suggesting improvements in our cycle handling capability and in the presentation of our experimental results. We also thank all the members of Compiler and Architecture Research group at HP Labs.

## References

1. W. W. Hwu, *et al*. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing* 7, 1/2 (1993), 229-248.
2. J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* C-30, 7 (1981), 478-490.
3. J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. (The MIT Press, Cambridge, MA, 1985).
4. A. Nicolau. *Percolation scheduling: a parallel compilation technique*. Technical Report TR 85-678, Department of Computer Science, Corneal, 1985.
5. D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (1991), 241-255.
6. S. A. Mahlke, *et al*. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th Annual International Symposium on Microarchitecture* (1992), 45-54.
7. S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, Oregon, 1992).
8. V. Kathail, M. S. Schlansker, and B. R. Rau. *HPL PlayDoh architecture specification: Version 1.0*. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto CA, 1993.
9. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. (Addison-Wesley Publishing Company, Reading, MA, 1983).
10. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *Proceedings of the Fourteenth Annual Workshop on Microprogramming* (1981), 183-198.

11. M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), 318-327.
12. B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose, California, 1994), 63-74.
13. D. M. Lavery and W. W. Hwu. Unrolling-based optimizations for software pipelining. *Proceedings of the 28th Annual International Symposium on Microarchitecture* (Ann Arbor, Michigan, 1995).
14. G. Lowney, *et al*. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing* 7, 1/2 (1993), 51-142.
15. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing* 7, 1/2 (1993), 181-228.
16. S. Srivastava. Implementation of Global Scheduling. 1988, Cydrome Internal Document.