Journal of Instruction-Level Parallelism 1 (2000) 1-6 Submitted 4/00; published 5/00

Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs

Jay Bharadwaj JAY.BHARADWAJ@INTEL.COM
Kishore Menezes KISHORE.MENEZES@INTEL.COM
Intel Corporation
3600 Juliette Lane
Santa Clara, CA 95052 USA

Chris McKinsey CHRIS.MCKINSEY@MOTOROLA.COM
*Star*Core Technology Center*
2100 Riveredge Parkway, Suite 600
Atlanta, GA 30328 USA

Abstract

The IA-64 architecture is rich with features that enable aggressive exploitation of instruction-level parallelism. Features such as speculation, predication, multiway branches and others provide compilers with new opportunities for the extraction of parallelism in programs. Code scheduling is a central component in any compiler for the IA-64 architecture. This paper describes the implementation of the global code scheduler (GCS) in Intel's compiler for the IA-64 architecture. GCS schedules code over acyclic regions of control flow. There is a tight coupling between the formation and scheduling of regions. GCS employs a new path based data dependence representation that combines control flow and data dependence information to make data analysis easy and accurate. This paper provides details of this representation. The scheduler uses a novel instruction scheduling technique called Wavefront scheduling. The concepts of wavefront scheduling and deferred compensation are explained to demonstrate the efficient generation of compensation code while scheduling. This paper also presents P-ready code motion, an opportunistic instruction level tail duplication which aims to strike a balance between code expansion and performance potential. Performance results show greater than 30% improvement in speedup for wavefront scheduling over basic block scheduling on the Itanium microarchitecture.

1. Introduction

The IA-64 architecture relies on the extraction of instruction-level parallelism (ILP) in software. One philosophy of the architecture is to enable the compiler to expose the parallelism in programs, thereby simplifying the hardware implementations. The architecture provides many features that the compiler can employ in accomplishing this task. The IA-64 architecture [1] provides support for control and data speculation, allowing the compiler to reduce the impact of memory latency by breaking control and memory dependence barriers. Full predication support is also available that allows removal of branches to transform a control dependence to a data dependence on the compare controlling the branch. In effect, predication enables elimination of branches and their associated mispredictions. Such architectural support comes with the challenge for the compiler to use these features judiciously. Instruction scheduling plays a major role in the usage of such features to extract ILP.

Several instruction scheduling techniques have been described in the literature to perform scheduling across basic block boundaries. Trace scheduling [2] [3], superblock and hyperblock scheduling [4] [5] operate on regions known as traces, which consist of a contiguous set of basic blocks. In contrast, treegion scheduling [6] uses a decision-tree subgraph of a program's control flow graph as a scheduling region. Ebcioglu's VLIW scheduling [7] arranges instructions in the form of a decision tree. Bernstein & Rodeh describe global instruction scheduling using a program dependence graph PDG [8] as a framework [9]. In most of these techniques, control-flow transformations are used to simplify regions with side-entries or a merge in control flow. Hyperblock scheduling uses predication to eliminate control-flow within the region. Tail duplication, which increases code size, is used to eliminate side entries into the region. Trace scheduling allows side entries into the scheduling region but only allows straight line code within the region. The *global code scheduler* (GCS) in Intel's compiler for the IA-64 architecture, allows arbitrary acyclic control flow within the scheduling scope referred to as a scheduling region. It also enables code scheduling across inner loops by abstracting them away through nesting. There is no restriction placed on the number of entries into or exits from the scheduling region. In order to enable effective and simplified data flow analysis in the presence of control flow a path based dependence representation has been developed that combines data dependence and control flow within a region. This representation is described in Section 3.

Most scheduling techniques find it difficult to make good decisions on the generation and scheduling of compensation code. This problem is addressed by *GCS* using *wavefront scheduling* and *deferred compensation*. The global code scheduler schedules along all the paths in a region simultaneously. The *wavefront* is a set of blocks that represents a strongly independent cut set of the region. Instructions are only scheduled into blocks on the wavefront. The wavefront can be thought of as the boundary between scheduled and yet to be scheduled code in the scheduling region. Section 4 describes this technique.

Control flow in program code can make the task of code motion difficult and complicated. Trace scheduling requires bookkeeping for valid code motion, while superblock scheduling uses tail duplication of entire blocks to avoid such bookkeeping. Tail duplication of blocks, especially in the absence of profile information, is not always practical due to the code expansion and harmful cache and TLB effects it causes. In *GCS*, tail duplication is done at the instruction level and is referred to as *P-ready code motion*. An instruction is duplicated based on a cost and profitability analysis. Section 6 details this approach. Aggressive movement of code results in groups of branches at the tail end of the scheduling region. Using the support for multiway branches in the IA-64 architecture, these branches can be scheduled to execute together. To do this effectively *GCS* keeps track of block layout decisions and makes the appropriate changes to the block order in the process of scheduling the region as described in Section 7 .

2. Scheduling Regions

The *global code scheduler* is capable of scheduling code in any subgraph of the control flow graph provided the subgraph is acyclic. A cyclic subgraph is treated as an acyclic subgraph by ignoring the backedge for scheduling purposes. Though *GCS* allows for regions with multiple entries and exits, as a result of our region picking heuristics, the regions most commonly encountered have a single entry. In order to permit code motion across regions

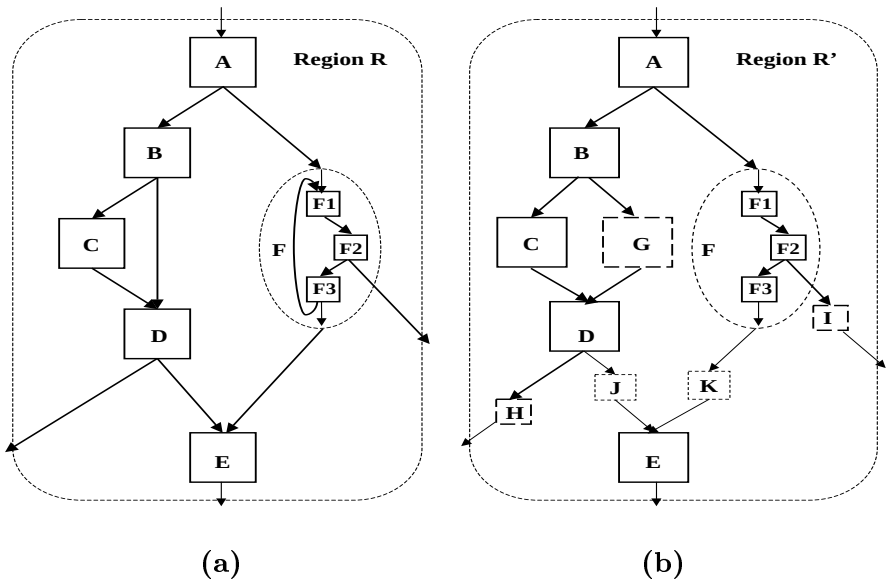


Figure 1: (a) While scheduling region R, code may be moved across nested region F but not into or out of F. (b) The region R with *JS blocks* and *interface blocks* added.

(such as inner loops) *GCS* maintains a region hierarchy, and the region of code being scheduled may contain nested regions representing already scheduled code. Figure 1 is an example of a scheduling region formed by *GCS*. As shown in Figure 1(b) before scheduling begins all *JS (join-split) edges* [10] are removed from the scheduling region. A *JS edge* is a control flow graph edge that emanates from a *split* node, i.e. a node with multiple successors, and ends at a *join* node, i.e. a node with multiple predecessors. *JS edges* are removed by adding an empty block (referred to as a *JS block*) between the *split* node and the *join* node. The removal of *JS edges* is required to enable the wavefront scheduling technique described later in this paper. It is not always simple to remove a *JS edge* while keeping the control flow graph functionally correct. One example is a *join* node with a label associated with it, which has multiple predecessors that branch to it indirectly. In case of such instances, *GCS* creates a fictitious *JS block* that is put in at the beginning of the scheduling process, and deleted at the end. The scheduler is constrained from inserting any code into this block, thereby guaranteeing that it can be removed at the end of the scheduling process. The scheduling algorithm may require blocks at certain region exits and entries to schedule compensation code in. Blocks called *interface blocks* are provided for this purpose at *side entries* and *side exits*. A *side entry* is a node in the region that has at least one predecessor that lies within, and at least one predecessor that lies outside the scheduling region. Similarly a *side exit* is a node in the region that has at least one successor that lies within, and at least one successor that lies outside the scheduling region. In Figure 1(b), blocks G, J, and K are *JS blocks*, and, H, and I are *interface blocks*.

In *GCS*, region picking and scheduling are strongly coupled. The high level driver for region picking and scheduling is described by Algorithm 1. Region picking commences from the innermost loop. After a region is picked and scheduled, all or part of the region is grouped into one or more nodes and nested. In general, all scheduled blocks are nested.

Algorithm 1 Region Formation and Scheduling.

```
for each loop inner to outer
    while unscheduled blocks remain
        Pick a Region R
        Schedule Region R
        Nest well scheduled blocks
    endwhile
endfor
```

However, we may choose not to include blocks with poor resource usage so that these blocks are scheduled again as part of another region. The nested nodes now contain final versions of scheduled code. This cycle of region formation and scheduling continues until the entire routine becomes one nested region or, in other words, is completely scheduled.

When scheduled blocks are nested, the data flow information within them is summarized and stored. One component of this information describes the memory references within the nested node. *Live-in* or *live-out* values referenced within the nested node, and their latencies, also form part of the summary. When scheduling the region containing the nested node, the data flow summary information helps determine correctness of code motions across the nested node. This information is also needed to respect latencies for dependencies crossing the nested node boundary.

3. Path Based Dependence Representation

Data dependence representation is a central component of any code scheduler. A well-designed representation increases the efficiency of the scheduler and can enable aggressive code motion. This section describes the data structures and their use in data and memory dependence representation [11] as implemented in *GCS*.

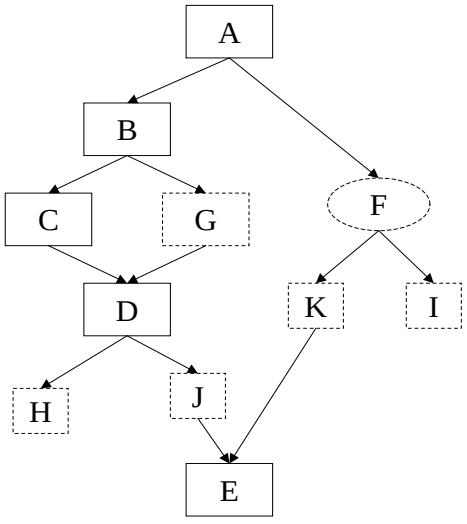
3.1 Path Vectors

The number of distinct control flow paths through an acyclic region is finite. In the worst case the number of paths in a region can grow exponentially with the number of blocks. *GCS*'s region picking heuristics keep the number of paths in the region below a specified threshold. This control over the number of paths enables usage of bit vectors to represent boolean properties on a per path basis. We define a *path vector* to be a bit vector in which each bit maps to a unique path in the region. A *path vector* can be used to represent a subset of all the paths in the region.

A variety of path vectors are used in *GCS*. Some path vectors are associated with blocks and encode properties of blocks. For instance, the path vector *BPV(n)* encodes all the paths that flow through block n. Other path vectors encode properties of instructions and are consequently associated with instructions. The *def-use path vector DUPV(w,r,v)* represents the set of control-flow paths along which the value v is written by w and read by r. Similarly, a *speculation path vector* can be used to indicate the control-flow paths along

which an instruction is speculated, while a *not-ready path vector* indicates the control-flow paths along which an instruction is not ready to be scheduled.

Figure 2 depicts the paths through a scheduling region and illustrates the computation of the path vector *BPV(n)*. Control flow relationships, such as dominance/post-dominance, control equivalence, disjointness etc, can be determined by performing bitwise operations on this path vector. For example, in Figure 2 blocks B and D are control equivalent since *BPV(B)* equals *BPV(D)*, or in other words the set of paths through B is the same as the set of paths through D. It can also be deduced that block A either dominates or post-dominates block E since *BPV(A)* is a superset of *BPV(E)*. Algorithm 2 enumerates the paths through an acyclic region and computes the block path vectors. For each node n, Step 1 calculates the number of distinct subpaths that can be constructed originating from node n and terminating at an exit node. In Step 2, the total number of paths through the region is computed. It is assumed that all region entries are strongly independent, a condition that is satisfied in scheduling regions due to the addition of *interface* blocks. In Step 3, the path vector bits are first distributed among the region entry blocks so that the paths through any entry map to contiguous bits in the bit vector. Next this contiguous set of bits is partitioned among the successors of the entry block, and this is repeated in topological fashion. When the algorithm terminates, the paths through any block n with p predecessors gets mapped to p subsets of *NP(n)* contiguous bits each. The complexity of function *PV_Partition()* called in Step 3 is proportional to the size of the bit vector it operates on. *PV_Partition()* is called once per edge. Hence the complexity of this algorithm is $\mathrm{O}(E) * \mathrm{O}(P)$, where E is the number of edges, and P is the number of control flow paths in the graph.



Path Legend:
P0: ABCDH
P1: ABCDJE
P2: ABGDH
P3: ABGDJE
P4: AFKE
P5: AFI

Block Path Vector (BPV):

	P5	P4	P3	P2	P1	P0
BPV(A) = {	1	1	1	1	1	1 }
BPV(B) = {	0	0	1	1	1	1 }
BPV(C) = {	0	0	0	0	1	1 }
BPV(D) = {	0	0	1	1	1	1 }
BPV(E) = {	0	1	1	0	1	0 }
BPV(F) = {	1	1	0	0	0	0 }
BPV(G) = {	0	0	1	1	0	0 }
BPV(H) = {	0	0	0	1	0	1 }
BPV(I) = {	1	0	0	0	0	0 }
BPV(J) = {	0	0	1	0	1	0 }
BPV(K) = {	0	1	0	0	0	0 }

Figure 2: Example region showing paths, and *block path vectors*. Node F is a nested node, blocks G, J, and K are *JS blocks*, and H, and I are *interface blocks*.

Algorithm 2 Enumeration of paths, and computation of block path vectors in a scheduling region.

```
// Step1: Calculate number of distinct subpaths NP(n)
//        originating at each node n
for each node n in reverse topological order
    NP(n) = 0
    for each successor s of n
        NP(n) += NP(s)
    endfor
    if (n is an exit node)
      NP(n)++
    endif
    BPV(n) = {00..00}
endfor
// Step 2: Calculate total number of paths through the region
num_total_paths = 0
for each entry node e
    num_total_paths+ = NP(e)
endfor
// Step 3: Compute block path vectors
k = 0
for each entry node e
    BPV(e) = Bit vector comprising of k low order
             zero bits followed by NP(e) one bits,
             followed by remaining high order zero bits
    k+ = NP(e)
endfor
for each node n in topological order
    k = 0
    for each successor s of n
        // Partition every pattern of NP(n) contiguous ones in BPV(n) into
        // k zeroes followed by NP(s) ones followed by NP(n) -k- NP(s)
        // zeroes. The result of this partition is put into pv, and BPV(n)
        // is left unchanged.
        PV_Partition(pv, BPV(n), k,
                     NP(n),
                     NP(s),
                     num_total_paths)
        BPV(s) = BPV(s) ⋃ pv
        k+ = NP(s)
    endfor
endfor
```

3.2 Data Dependence Representation

In a register based intermediate representation, instructions that read or write a register v can be partitioned into two sets, a set of readers, *Readers(v)*, and a set of writers, *Writers(v)*. An instruction may be a member of both sets if it both reads and writes the register. The data flow dependence between any member w of *Writers(v)* and a member r of *Readers(v)* is represented as a *def-use path vector DUPV(w,r,v)*. This path vector is defined to be the set of control flow paths along which the value v written by w is read by r. An example is depicted in Figure 3. Given a virtual register v, Algorithm 3 illustrates how to compute the *def-use path vectors DUPV(w,r,v)* for all r, w in *Readers(v)* and *Writers(v)* respectively. This algorithm uses preceedence relationships between instructions. For the purposes of this algorithm, an instruction is said to preceed another only if the preceedence relationship can be established in the acyclic control flow, i.e. if they both lie on some common path. The algorithm first assigns into *DUPV(w,r,v)*, all paths on which both w and r are executed. It then removes those paths on which there is an intervening write of v by another instruction thereby yielding the actual flow dependence paths. Only flow dependence information is used in *GCS*. Anti and output dependence information is not necessary since it can be inferred as described in subsection 3.3. This reduces the storage space required, and the cost of keeping the dependence graph up to date while scheduling.

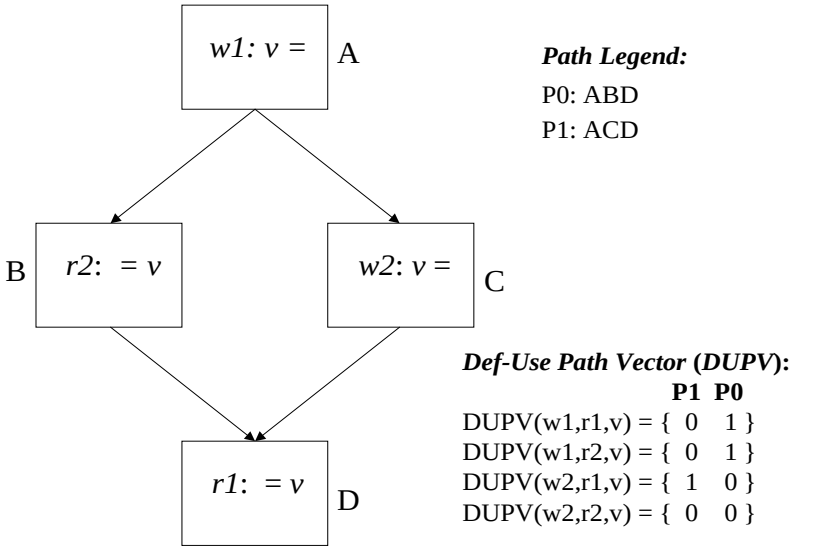


Figure 3: Example to illustrate def-use path vectors.

The use of *def-use path vectors* partitions the data dependence graph into independent subgraphs based on registers, i.e. *DUPV(w,r,v)* is only affected by changes to instructions belonging to *Writers(v)*, or *Readers(v)*. This enables efficient local updates of the dependence information. For example, if an instruction I is moved from one block to another, then we only need to recompute the dependences associated with the virtual registers it accesses. For each virtual register x that it accesses, all *DUPV(w,r,x)* need to be recomputed where w

Algorithm 3 Computation of DUPV(w,r,v), given v.

```
for each w in writers(v)
    for each r in readers(v)
        if (w preceedes r)
          dupv(w,r,v) = BPV(Block(w)) ⋂ BPV(Block(r))
        endif
    endfor
endfor
for each w1 in writers(v)
    for each w2 in writers(v)
        if (w1 preceedes w2)
          for each r in readers(v)
              if (w2 preceedes r)
                dupv(w1,r,v) = dupv(w1,r,v)
                               − BPV(Block(w2))
              endif
          endfor
        endif
    endfor
endfor
```

belongs to *Writers(x)* and r belongs to *Readers(x)*. As seen from Algorithm 3, the cost of a local update to the dependence graph for a virtual register v is $O(N_W^2 * N_R)$ where N_R is the number of readers and N_W the number of writers of v. The use of distinct register names for unrelated lifetimes reduces the number of readers and writers, consequently reducing the complexity of the dependence graph update. Another advantage of the virtual register based partitioning of the dependence graph is that it allows a lazy approach to updating it. When an instruction I is moved from one block to another, the affected def-use path vectors are not recomputed immediately. Instead the def-use path vectors associated with the virtual registers referenced by I are marked *stale*. Any subsequent access to dependence information marked *stale*, triggers an automatic update of those def-use path vectors. This avoids unnecessary updates due to multiple invalidates that may occur between uses of the dependence information.

3.3 Renaming

Most schedulers maintain anti and output dependence information. This is done primarily to detect if moving an instruction violates such a dependence and if renaming would be required to eliminate the dependence. Maintaining such information implies additional complexity since the information would need to be continually updated. Our code scheduler does not maintain information for anti and output dependences. Instead the path vectors *DUPV(w,r,v)* which encode flow dependences are used to glean this information on a demand-driven basis. To detect an anti or output dependence violation, the instruction

I involved in the code motion is assumed to have already moved to the new location. For each register v that is written by I, the path vector *DUPV(I,r,v)* is recomputed for all $r \in Readers(v)$. A change in *DUPV(I,r,v)* indicates that an anti or output dependence is violated by the code motion, and renaming is required. The process of violating an anti dependence creates a flow dependence. Similarly when an output dependence is violated, a flow dependence is changed (unless the result is dead, in which case the output dependence violation does not matter). Hence by just looking for changes to flow dependences caused by a prospective code motion, the need for renaming can be determined.

Furthermore, it is only necessary to look for changes to flow dependences on the instruction I being moved to conclude that no dependences have been violated. This is because any change caused to a flow dependence on any other writer J of v, must be accompanied by a change in *DUPV(I,r,v)*. If the code motion of I causes any change in a flow dependence on another instruction J, then *DUPV(J,r,v)* must have changed for some reader r. This can only be so if a new path now exists in *DUPV(J,r,v)* that did not use to, or if a path that used to exist has been removed from *DUPV(J,r,v)*. The only way a new path p could now exist in *DUPV(J,r,v)*, is if instruction I used to be the sole writer of v that lay between J and r on path P before, and the code motion of I causes it to no longer do so. In this case this path p should have belonged to *DUPV(I,r,v)* before but must no longer do so, which implies that *DUPV(I,r,v)* has changed. The only way a pre-existing path P could no longer be in *DUPV(J,r,v)* is if the code motion of I causes it to now lie between J and r on this path P, whereas no writer of v used to. This implies that path P must now belong to *DUPV(I,r,v)* whereas it did not use to. Therefore any change in *DUPV(J,r,v)* must be accompanied by a change to *DUPV(I,r,v)*. The pseudo-code to compute the need to rename is provided in Algorithm 4.

Algorithm 4 Computation to determine if code motion requires renaming. RecomputeDupv() recomputes DUPV(I,r,v) assuming I is moved to *target_block* for all $r \in$ Readers(v).

```
for each result v of instruction I
    new_dupv = RecomputeDupv(I,v,target_block)
    for each reader r of v
        if (old_dupv(I,r,v) ≠ new_dupv(I,r,v))
          return True
        endif
    endfor
endfor
return False
```

3.4 Memory Dependence Representation

Memory dependences are represented separately from the register-based dependences. For memory dependences flow, anti, and output dependences are computed and stored. To take

advantage of the data speculation capabilities in the IA-64 architecture, we associate with each memory dependence edge, the probability that the memory references interfere.

4. Wavefront Scheduling And Deferred Compensation

One of the weaknesses of most cross block scheduling techniques is their inability to be judicious regarding the scheduling of compensation code. In these techniques, when evaluating a candidate instruction for scheduling that requires compensation copies in other blocks, there is no notion or measure of how desirable the compensation is to the other block where it is required. In some techniques compensation is allowed only when the block where the compensation is needed has not already been scheduled. *Wavefront scheduling* and *deferred compensation* [12] try to address these issues.

4.1 Wavefront

Given an acyclic region, *JS edges* are eliminated and *interface blocks* are added, as described in Section 2, to form a *scheduling region*. We define a *wavefront* in the *scheduling region* as a strongly independent cut set that partitions the *scheduling region* into three parts:

- nodes above the wavefront
- nodes on the wavefront
- nodes below the wavefront

The wavefront is a strongly independent cut set, implying that every path in the region passes through exactly one node on the wavefront. Therefore the wavefront nodes collectively dominate all the nodes below the wavefront, and collectively post-dominate all the nodes above the wavefront. This property guarantees that when scheduling a candidate instruction I originally in block B_k in the region into block B_w on the wavefront, compensation code can be inserted entirely into blocks on the wavefront. Figure 4 shows an acyclic *scheduling region* and all the possible wavefronts in it.

Given a wavefront W, and a block B in it, if B has multiple successors and we replaced B on the wavefront with its successors, the result would still be a wavefront. The removal of *JS edges* before wavefront scheduling guarantees that every successor of B has B as its sole predecessor. Hence each path that flows through B also flows through exactly one successor, and every path that flows through any successor also flows through B. This proves that the resulting set of nodes is still a wavefront.

Given a wavefront W and a block B with a single successor B_{succ}, all predecessors of B_{succ} must lie on or below the wavefront since every path flows through a node on the wavefront. Again, the removal of *JS edges* before wavefront scheduling guarantees that each of the predecessors of B_{succ} has only B_{succ} as a successor. Therefore the paths that flow through *Pred(*B_{succ}*)* are exactly the paths that flow through B_{succ}. If all predecessors of B_{succ} lie on the wavefront and if we replaced all predecessors of B_{succ} with B_{succ}, the result would still be a wavefront since each path in the region will still flow through exactly one node on the resulting block set.

Due to the addition of interface blocks the set of entry blocks forms a strongly independent cut set, and hence is a wavefront. Similarly the set of exit blocks also forms a

wavefront. A wavefront consisting of the entry blocks can be moved down topologically over the region by replacing any block with multiple successors, by its successors, or by replacing any collection of one or more blocks on the wavefront that form the complete set of predecessors to a single common successor block, by the successor block. The wavefront can always be advanced in this manner until the wavefront comprises all the exit blocks. In other words, this process of advancing the wavefront can never deadlock. A deadlock is only possible if there is no block on the wavefront with multiple successors, and no block on the wavefront with a successor all of whose predecessors are on the wavefront. Let us number the blocks topologically and let B_b be the topologically first block that lies below the wavefront. All predecessors of B_b must lie on the wavefront as otherwise B_b would not be topologically the first block that lies below the wavefront. Hence a deadlock can never occur. The method of advancing the wavefront described above guarantees that there exists at least one wavefront which passes through any given block in the region.

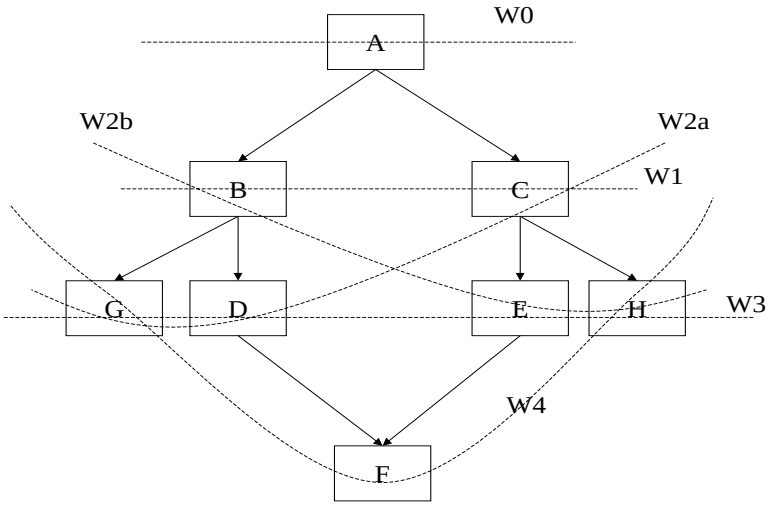


Figure 4: Wavefront advancement in top-down scheduler.

4.2 Wavefront Scheduling

Before wavefront scheduling begins all blocks contain only unscheduled code. During the scheduling process, instructions are scheduled only into blocks on the wavefront. A separate list of candidate instructions is built and maintained for each block on the wavefront. The candidate list for a block B may contain instructions originating from B, from blocks topologically below B, or from blocks topologically above B. Scheduling a candidate originating from a block topologically below B constitutes *upward motion*. Similarly, scheduling a candidate originating from a block topologically above B constitutes *downward motion*. After a selected instruction candidate is scheduled, the necessary bookkeeping is done to keep track of compensation needs. Assuming top down scheduling, the wavefront starts out as the set of all entry blocks. As instructions are scheduled into blocks on the wavefront, these blocks contain both unscheduled code originating from them, as well as code scheduled into them. When code scheduling into a block B is completed, it is declared *closed*. This

implies that any unscheduled instructions from B or from any block topologically above it will be scheduled into a block on a subsequent wavefront. The scheduler declares a block *closed* only after examination of the correctness and profitability of downward code motion of the unscheduled instructions originating from or above it. Once block B is *closed*, an attempt is made to advance the wavefront across it. If successful, block B and possibly some other *closed* blocks are replaced with one or more new blocks. Block B now lies above the wavefront and represents a fully scheduled block. Thus the wavefront advances down the region until it finally passes through all the exit nodes in the region.

In Figure 4, the wavefronts are numbered to show the advancement of the wavefront in a top-down scheduling scheme. Note that there can be multiple alternatives to how the wavefront is advanced as shown in the figure by W2a and W2b. Algorithm 5 provides a method for advancing the wavefront. The algorithm ensures that only closed nodes or nodes in which scheduling is complete are moved above the wavefront.

Algorithm 5 Advancing the wavefront. Closed nodes are nodes which are candidates for moving above the wavefront.

```
// Compute in "del_nodes" the set of nodes to be moved
// above the wavefront.
del_nodes = Set of closed nodes on wavefront
for each node n in del_nodes
    if (n has exactly one successor s)
      for each predecessor p of s
          if (p ∉ del_nodes)
            del_nodes = del_nodes − {n}
            break
          endif
      endfor
    endif
endfor
for each node n in del_nodes
    Wavefront = Wavefront −{n}
    Wavefront = Wavefront ⋃ SuccessorsOf(n)
endfor
```

4.3 Deferred Compensation

The control flow paths in the scheduling region R along which an instruction I needs to execute can be deduced from the original position of I and it's data flow properties. This set of control flow paths can be represented as a path vector which we call the *compensation path vector* of the instruction *CPV(I)*. When I is scheduled in a block K that does not fully dominate the block from which I originated (or in other words *BPV(K)* is not a superset of *CPV(I)*), we need to generate compensation copies of I to be scheduled elsewhere. Rather than creating such copies and associating them with other blocks immediately, we record

the compensation needs in *CPV(I)*. This is done by scheduling a copy *I'* of I in block K, and updating *CPV(I)* to $(CPV(I) - BPV(K))$. The actual generation of the compensation copies is deferred until they are actually scheduled. The instruction I left behind in it's original block now represents all the compensation copies that will be generated whether it be one copy or many. When the last compensation copy is to be scheduled in a block C, *BPV(C)* will be a superset of *CPV(I)*, signaling that this is the last compensation copy. At this time instead of scheduling a *copy* of I, the instruction I itself is scheduled in C. Figure 5 illustrates this process. Delaying compensation code generation allows scheduling freedom for the compensation (the destination of the compensation is not dictated *a priori*). Scheduling multiple copies of an instruction on a path through the region is avoided as shown in the next subsection. Compensation for downward code motion below splits, is handled similarly.

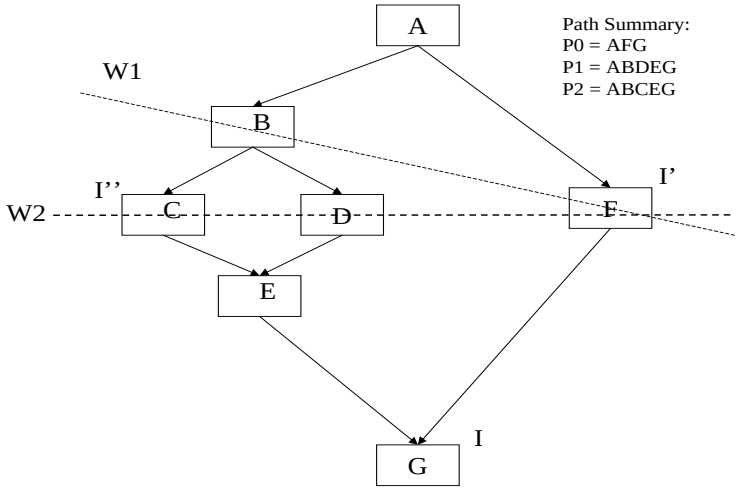


Figure 5: Copy I' of I (originating from block G) is scheduled first in F. CPV(I) is set to {110} and compensation code has to be scheduled on or before E. Next, copy I" is scheduled in C on wavefront W2. CPV(I) becomes {010} and compensation has to be scheduled on or before D. Until compensation is scheduled in D, the wavefront cannot be advanced since D cannot be *closed*.

4.4 Advancing Wavefront in the Presence of Deferred Compensation

In *GCS*, information about the topologically last blocks where an instruction needs to be scheduled is maintained for each instruction (including those representing *deferred compensation*). The constraint on how late a deferred compensation can be scheduled is imposed in order to avoid generating multiple copies of an instruction on a path through the region. For example, in Figure 5, when copy *I'* of instruction I originating from G is scheduled in F, the compensation copies represented by I have to be scheduled on or before E. When *I"* gets scheduled in C, I has to be scheduled in D. This constraint on the scheduling range for instructions, is used in *closing* a block, i.e. determining that scheduling code into that block is complete. Only when a block is *closed* can the wavefront be advanced across

it, since all blocks above the wavefront represent scheduled blocks into which we cannot schedule any new code. So a block on the wavefront gets *closed* once all instructions that have to be scheduled on or before it have been scheduled. This block on the wavefront may subsequently get *opened* in response to instructions being scheduled in other blocks on the wavefront. Hence a block can flip-flop between *open* and *closed* states. For example in Figure 5, assume D is a *closed* block. When it is decided to schedule I" in C, block D is *opened* since I can no longer be deferred to be scheduled in E. Additionally in a top down scheduling scheme permitting downward code motion as in GCS, constraints on the downward code motion (i.e. how far down an instruction can be moved below its block of origin) need to be taken into account in determining whether a block can be closed. For example as described in Section 5.1, the downward movement of an instruction may be constrained by the availability of a qualifying predicate.

5. Speculation and Predication

GCS selects an instruction to schedule into a block T, from a candidate list of data ready instructions. These instructions may originate from any block in the region connected to T by at least one control flow path. The speculation support in IA-64 allows many dependencies to be ignored when determining data readiness. Control dependencies, and data dependencies on qualifying predicates can be broken using control speculation. Data speculation allows unlikely memory dependencies to be broken. The best candidate is selected from all speculative and non-speculative candidates based on a cost-benefit analysis. This analysis takes into consideration among other things, the instruction's global critical path length, its resource requirements, its speculation and code duplication costs, and the *usefulness* (inverse of *speculativeness*) of the code motion. *Usefulness* is measured in terms of the likelihood that control will flow through the target block T and the block of origin O, given that control reaches T. This is measured as $Prob(BPV(T) \bigcap BPV(O))/Prob(BPV(T))$, where *Prob(pv)* is the function that provides the aggregate probability that control flows along any one of the paths in the path vector pv given that control flows through the region.

Speculation check instructions [1] and recovery code are generated as a byproduct of speculation. Load safety information [13] is used to avoid unnecessary control speculation. GCS also uses predication support in the IA-64 architecture to convert a control speculative instruction to a non-speculative one. If the instruction being speculated across a branch is scheduled after the compare operation that controls the branch, the predicate produced by the compare may be available for use. In such situations it is preferable to predicate the instruction over speculating it. This eliminates the need for a check instruction and recovery code. In addition, when the predicate is false, any adverse effects of a speculative operation on the data cache and TLB are avoided. In some cases when an instruction is moved across multiple branches, the block predicate for the block of origin may not be *available* (i.e. compare not scheduled or result unavailable) at the point where the instruction is being scheduled. However a predicate for an intermediate block in the control dependence chain may be available. Predication with such a predicate will not render the instruction non-speculative, but will reduce the *speculativeness* of the instruction when the instruction executes (i.e. qualifying predicate is true). To do this effectively GCS maintains a predicate promotion list and keeps track of predicates that become available as compare instructions

are scheduled. The predicate promotion list is essentially a form of control dependence information but in the predicate domain.

Instructions may be predicated by another phase of the compiler before *GCS* encounters them. Predicated instructions can also be speculated. *GCS* speculates these instructions by promoting the qualifying predicate to an available predicate in the predicate promotion list. Predicate promotion is thus achieved on the fly while scheduling.

5.1 Downward code motion

Operations such as stores and speculation check instructions cannot be speculated, and therefore tend to stay behind in the block of origin. It is advantageous to move an instruction downward if it does not fit into the schedule for a block. Another motivation to move code down is to empty a block. This can help eliminate an unconditional branch, or expose an opportunity for *multiway* branch generation. Downward code motion can also help reduce the amount of speculation or compensation code needed to expose ILP. Often the guarding predicate for instructions that cannot be speculated may not be available much before the controlling branch, and hence such operations do not move up using predication either. However, these operations can usually be moved down non-speculatively to a *join* block, since the predicate can be used to conditionally execute them. It should be noted that the predicate is only available for use in blocks dominated by the compare instructions that generate the predicate. This places a limit on how far down the code can be moved. In *GCS*, before a block is *closed* and the wavefront advanced across it, the availability of any necessary predicates for the code being moved down is checked.

6. P-ready Code Motion

An instruction I from block O is *M-ready* [10] at a block T, if there is no unscheduled source operand of I on any path that flows through O and T. Consider the example in Figure 6a. In this example I is *M-ready* at B, but instruction J is not due to instruction K. Note that instruction K is on a path from block B to block E that is unlikely to be taken. If the join into block E is eliminated by tail duplicating E, as shown in Figure 6b, then J is now *M-ready* at B, whereas J' is not. The same result can be achieved by only duplicating instruction J using *P-ready* code motion. We define an instruction I from block O to be *P-ready* (partially ready) at a block T, if it is not *M-ready* at T, and if there is no unscheduled operand on at least one path that flows through T and O. When a *P-ready* instruction I is scheduled at a block T, compensation copies of I need to be placed at points where it is *M-ready*. In Figure 6a, J is *P-ready* at B. Figure 6c, shows J scheduled using *P-ready* code motion in block B, with a compensation copy in block D.

P-ready compensation code can increase dynamic instruction count. Hence a decision to select a *P-ready* candidate over an *M-ready* one should be based on probability of executing the *compensation* due to *P-ready* code motion. In *GCS*, both *M-ready* and *P-ready* candidate lists are maintained and the best candidate is chosen based on heuristics. Only those instructions that are important to execute early get moved up to locations where they are *P-ready*. The scheduler has much of the benefits afforded by full tail duplication, and pays the price of tail duplication only on those instructions where it is deemed worthwhile. The current implementation of *P-ready* code motion in *GCS* has one limitation over tail duplication. To preserve the control flow graph within the region during scheduling, *GCS*

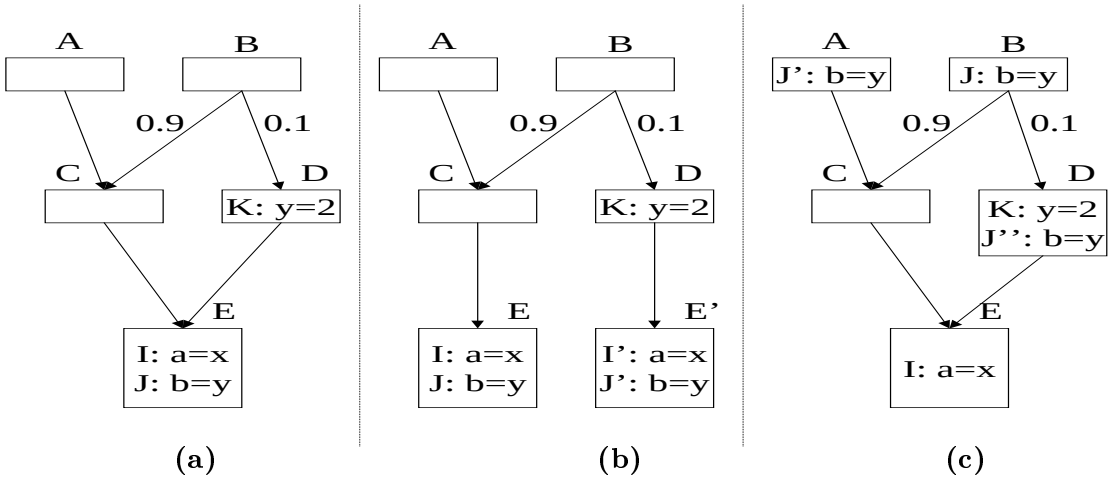


Figure 6: (a): I is M-ready at A, and B. J is M-ready at A but only P-ready at B. (b): Tail duplication of block E makes J M-ready at A and B but duplicates both I and J. (c): P-ready code motion duplicates instruction J but avoids duplication of I.

does not tail duplicate branch instructions through this mechanism. This limitation may be removed in the future.

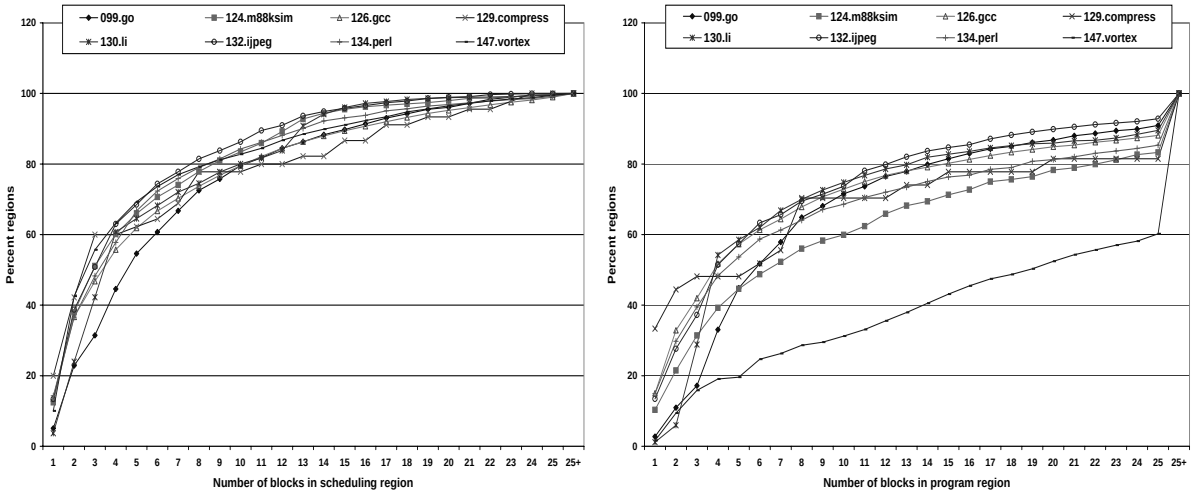
7. Block Reordering And Multiway Branch Generation

In order to schedule branches effectively, *GCS* needs to know the physical block order before scheduling begins. During the scheduling process, some blocks are emptied. Block reordering and branch re-targeting around such blocks often results in the elimination of unconditional branches. In some cases, compensation code may be inserted into a previously emptied block potentially resulting in the addition of a branch. There can be a net performance gain if the number of branches deleted exceeds the number added.

The IA-64 architecture supports the execution of a linear sequence of branch instructions in a single cycle. Such sequences of branch instructions are called *multiway* branches. When a block becomes empty except for a branch instruction, there may be a multiway opportunity if a control flow predecessor that ends with a branch, is also its physical layout predecessor. If such a control flow predecessor is not its physical layout predecessor the block layout may be changed to create the multiway opportunity. Therefore in *GCS* we choose to model all unconditional branches and to update block layout ordering as we schedule [14]. Data on the usefulness of this capability is presented in Section 8.

8. Results

This section presents the results of experiments conducted to measure the effectiveness of various features implemented in the global code scheduler. The SPEC95 integer benchmark suite was used in all experiments. The programs in the suite were run with an input set developed at Intel. This input set attempts to approximate the characteristics of the SPEC95 reference input set, while considerably reducing run time. The compilations used



(a) **(b)**

Figure 7: Cumulative distribution of (a) scheduling region size, (b) program region size, in terms of initial basic blocks.

profile feedback and interprocedural inlining. The same inputs were used for the profiling as well as the measurement runs. Figure 7a shows a cumulative distribution of the size of scheduling regions measured in terms of the number of initial basic blocks they contain. This number does not include any *JS* or *interface blocks*, or nodes representing nested regions. Hence it is a true indication of the size of the scheduling scope. Though *GCS*'s region picking heuristics attempt to build large regions, a large percentage (60%) of regions are 5 blocks or less in size. Figure 7b helps understand this effect. *GCS* picks a *scheduling region* from a *program region*. A single program region is decomposed into one or more scheduling regions. A scheduling region cannot span multiple program regions. A program region is the body of either a loop or a function. Loop and function boundaries therefore represent hard boundaries to the scheduling scope. Figure 7b shows the cumulative distribution of the size of program regions. For all benchmarks except *147.vortex*, the majority (60%) of our program regions are 7 blocks or smaller in size. Improvements in scheduling would be possible by increasing program region size using more aggressive inlining, unrolling, peeling and other methods. A considerable portion of the program code in *147.vortex* is not exercised by the input used. *GCS* region picking heuristics separate and first schedule cold regions (unexercised code) and later nest these cold regions within hot ones. This causes fragmentation of the program region with no detrimental effect. This is the primary reason for smaller scheduling regions in *147.vortex* even though most program regions are large.

The block reordering capabilities of *GCS* were evaluated by measuring the magnitude of the changes to unconditional branches during scheduling. Figure 8 shows the average number of unconditional branches added and the number deleted during scheduling by *GCS* as a percentage of all branches (conditional and unconditional) except calls and returns in the region. As can be seen, on average more than twice as many unconditional branches are deleted as are added. This figure also shows the 2-way and 3-way *multiway* branches

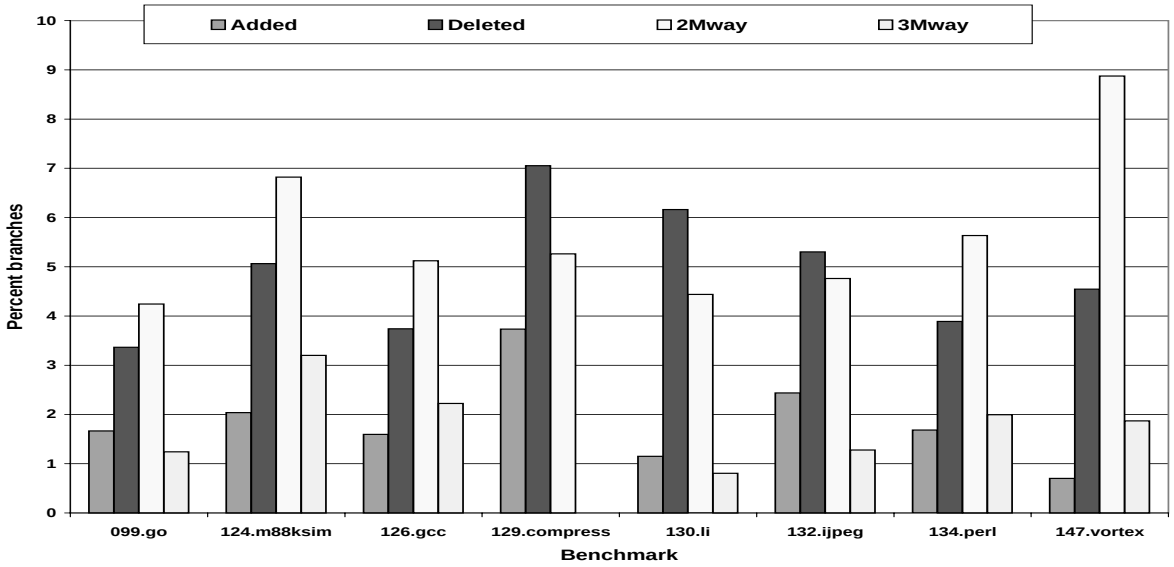


Figure 8: Unconditional branches added/deleted, by reordering blocks during scheduling. Percentage of multiway candidate branches combined into a multiway branch.

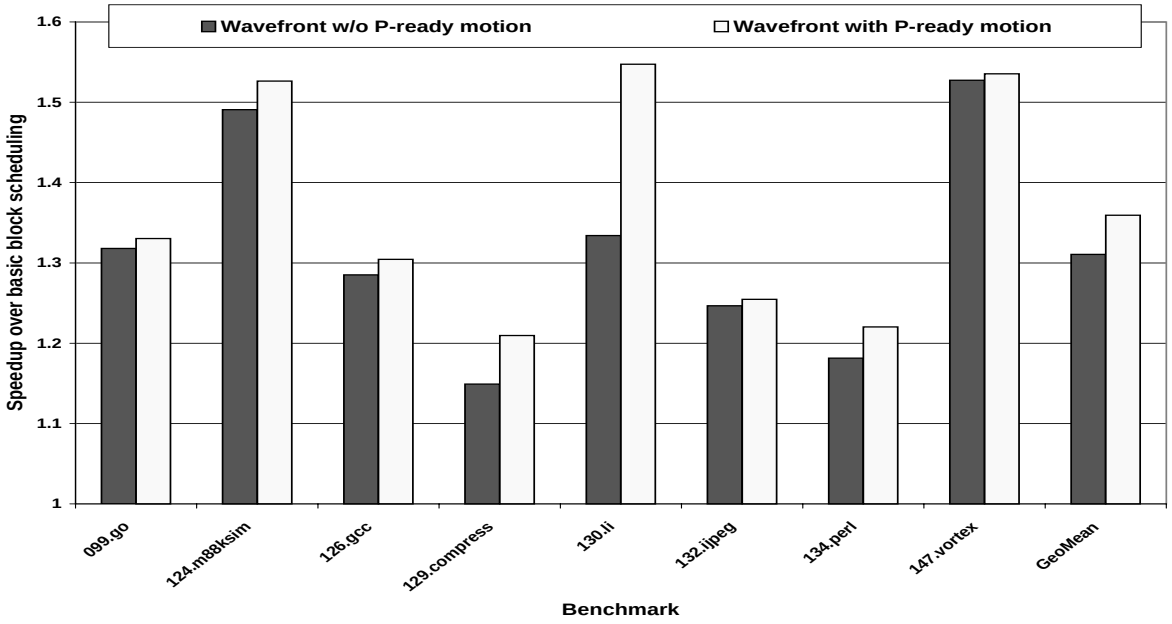


Figure 9: Speedup due to wavefront scheduling over basic block scheduling on the Itanium microarchitecture.

that were formed as a percentage of all branches that can be combined in a *multiway*. This includes calls and returns as well as all unconditional and conditional branches, but does not include speculation check instructions that branch to recovery code. For each 2-way *multiway* generated one branch cycle is saved since the 2-way is executed in one cycle instead of two. For each 3-way *multiway*, at least two cycles are saved.

The performance impact of *wavefront scheduling* in *GCS* is shown in Figure 9. Performance is presented as speedup over *basic block* scheduling. *Basic block* scheduling forms regions out of a contiguous set of blocks and does not include branches except for *call* instructions. The code was scheduled for the Itanium processor from Intel Corporation. The speedup was measured assuming perfect caches. Two data points are presented for each benchmark indicating speedup with and without *P-ready* code motion. No tail duplication of basic blocks is done in either case. The geometric mean for the speedup due to wavefront scheduling is also shown. The mean increase in performance without *P-ready* code motion is about 30%, with *147.vortex* and *124.m88ksim* benefiting from almost a 50% performance increase. Except for *130.li*, the speedup from *P-ready* code motion is modest. *P-ready* code motion itself provides more than a 20% increase in speedup for *130.li*. With respect to scheduling, *P-ready* code motion is more efficient than tail duplication of basic blocks, in that instructions are duplicated only if they yield a performance increase as in *130.li*.

9. Conclusions

An overview of the global code scheduler implemented in Intel's compiler for IA-64 has been provided. *GCS* schedules code in hierarchical acyclic regions, which may include nested regions including nested loops. This paper has described a novel scheduling technique for scheduling subgraphs of the control flow graph, called *wavefront scheduling*. *Wavefront scheduling* and *deferred compensation* help *GCS* to be judicious about the compensation code it generates. This technique is built around a framework of path based analysis. We have described at a high level the use of bit vectors in *GCS* to represent control flow paths to analyze data dependences, control flow relationships, and to do efficient bookkeeping of compensation. A more efficient alternative to tail duplication which we call *P-ready code motion* has been described. This technique can reap most of the scheduling benefit of tail duplication without paying the high code bloat cost. We have also shown the value of modeling and changing the physical block layout during scheduling especially in making effective use of the multiway branch feature in IA-64.

Acknowledgements

The authors would like to acknowledge the contributions of all the Electron code generator team members in the design and implementation of *GCS*. First and foremost, thanks go to Roland Kenner for his wisdom that influenced the design of *GCS*. Kent Fielden and Beatrice Fu provided the right goals and challenges and the time and space that was crucial to develop many ideas implemented in this project. William Chen was intimately involved in the early design and implementation of several aspects of *GCS*, especially all issues dealing with predication, predicate analysis and predicate promotion. Finally, thanks go to all the reviewers of the paper for providing useful feedback.

References

[1] *IA-64 Application Developer's Architecture Guide*. http://developer.intel.com/design/ia64/devinfo.htm: Intel Corporation, 1999.

[2] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transaction on Computers*, vol. C-30, pp. 478–490, July 1981.

[3] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2," *Technical Report HPL-93-43, Hewlett-Packard Laboratories*, June 1993.

[4] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[5] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," *Proceedings of the 25th International Symposium on Microarchitecture (MICRO25)*, pp. 45–54, 1992.

[6] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide-issue processors," *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[7] S. M. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," *Proceedings of the 25th International Symposium on Microarchitecture (MICRO25)*, pp. 55–71, 1992.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM TOPLAS*, vol. 9, pp. 319–349, July 1987.

[9] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," *SIGPLAN'91, PLDI*, 1991.

[10] D. Bernstein, D. Cohen, and H. Krawcyzk, "Code duplication: An assist for global instruction scheduling," *Proceedings of MICRO24, IEEE Computer Society*, pp. 103–113, November 1991.

[11] J. Bharadwaj, "Representation of control flow and data dependence for machine instructions," *Patent No. 5,787,287*, July 28 1998.

[12] J. Bharadwaj, "Method and apparatus for instruction scheduling to reduce negative effects of compensation code," *Patent No. 5,894,576*, April 13 1999.

[13] D. Bernstein, M. Rodeh, and M. Sagiv, "Proving safety of speculative load instructions at compile-time," *4th European Symposium on Programming*, 1992.

[14] C. M. McKinsey and J. Bharadwaj, "Interactive instruction scheduling and block ordering," *Docket No. 43290.P6462. Patent Pending.*, Filed Nov. 3rd 1998.