Balance Scheduling: Weighting Branch Tradeoffs in Superblocks

Alexandre E. Eichenberger
ECE Department, N.C. State University
alexe@eos.ncsu.edu

Waleed M. Meleis
ECE Department, Northeastern University
meleis@ece.neu.edu

Abstract

Since there is generally insufficient instruction level parallelism within a single basic block, higher performance is achieved by speculatively scheduling operations in superblocks. This is difficult in general because each branch competes for the processor's limited resources. Previous work manages the performance tradeoffs that exist between branches only indirectly. We show here that dependence and resource constraints can be used to gather explicit knowledge about scheduling tradeoffs between branches. The first contribution of this paper is a set of new, tighter lower bounds on the execution times of superblocks that specifically accounts for the dependence and resource conflicts between pairs of branches. The second contribution of this paper is a novel superblock scheduling heuristic that finds high performance schedules by determining the operations that each branch needs to be scheduled early and selecting branches with compatible needs that favor beneficial branch tradeoffs. Performance evaluations for superblocks from SPECint95 indicate that our bounds are very tight and that our scheduling heuristic outperforms well known superblock scheduling algorithms.

Keyword: Superblock, scheduling heuristic, lower bound, ILP compiler technique, weighted completion time.

1 Introduction

Since there is generally insufficient instruction level parallelism within a single basic block, higher performance is achieved by overlapping the operations of consecutive basic blocks. Along with traces [1] and hyperblocks [2], one of the most widely used scheduling units is a *superblock* [3], which forms a sequence of consecutive blocks that are frequently executed and share a unique entry point at the beginning of the first block in the sequence. Superblocks have multiple exits, one at the end of each block in the sequence.

While scheduling algorithms for a single basic block are generally well understood, scheduling superblocks is more difficult because each exit, generally a branch, competes for the limited number of machine resources. In practice, it is often the case that a branch can be scheduled earlier at the expense of some other exits in the superblock. When the taken probability of each branch in a superblock can be estimated, using static predictors [4, 5] or profiling techniques [6, 7], it is often advantageous to schedule heavily taken branches early so as to decrease the expected execution time of the superblock.

Early scheduling algorithms such as Successive Retirement [8] and Critical Path [9, 10, 11, 12] are generally biased toward the first and last exits, respectively, and as a result do not fully exploit the performance improvements possible using superblocks on a wide range of machine widths. A more recent heuristic, Dependence Height and Speculative Yield [1, 13], achieves better performance over a wider range of machines by committing machine resources to operations along the critical path of frequently taken branches. A recent heuristic, G* [8], finds an attractive middle ground between Successive Retirement and Critical Path by repetitively scheduling subsets of the superblock and using useful information from these partial schedules. Another recent heuristic, Speculative Hedge [14], schedules frequently taken branches early without needlessly delaying less frequently taken branches. It proceeds by dynamically computing whether an operation helps each branch because of dependence or resource constraints. Other studies have evaluated the quality of schedules produced by a variety of list-scheduling algorithms for the general weighted completion time problem [15,16].

The major issue not addressed explicitly by previous work concerns performance tradeoffs between branches, i.e. the degree to which scheduling a branch early delays other branches. Most past work deals with branch tradeoffs indirectly, such as assigning higher priorities to operations that help frequently taken branches [1, 8, 13, 14] or operations that help a branch without penalizing more frequently taken branches [14]. We show in this paper that the dependence and resource constraints can be used to gather explicit knowledge about scheduling tradeoffs between branches.

The first contribution of this paper is a set of new, tighter lower bounds on the execution times of superblocks that builds upon some of the best single basic block lower bounds [17, 18]. The proposed bounds, referred to as the Pairwise and Triplewise superblock bounds, are tight because they specifically account for the dependence and resource conflicts between pairs and triples of branches. The bounds are not only used to more accurately evaluate the quality of superblock schedules, but also to extract information that is valuable to a high performance superblock scheduler such as the one described here.

The second contribution of our paper is a novel superblock scheduling heuristic that attempts to find high performance schedules by (1) determining the set of operations that each branch needs to be scheduled early, (2) computing an attractive set of branches with needs that are compatible in the current cycle, and (3) actively favoring some branches over the others. The proposed heuristic, referred to as the Balance superblock heuristic, thus actively weights the branch tradeoffs in a superblock by using the more precise Pairwise bounds.

Performance evaluations for superblocks from SPECint95 indicate that our Pairwise and Triplewise superblock bounds are very tight. For a fully pipelined machine with a mix of 4, 6, and 8 specialized functional units, for example, the best schedules we can find achieve the lower bound for 81.65%, 89.62%, and 96.09% of the superblocks in SPECint95, respectively. The Balance superblock heuristic alone finds such a schedule for 81.35%, 89.58%, and 96.08% of the superblocks of the same benchmark and respective machines.

This paper is organized as follows. Section 2 presents related work in superblock scheduling and Section 3 motivates the proposed bound and scheduling heuristics. Section 4 presents our new Pairwise and Triplewise superblock bounds. Section 5 describes the Balance scheduling heuristic. Measurements and conclusions are presented in Sections 6 and 7, respectively.

2 Background and related work

When scheduling a single basic block, which consists of a sequence of operations with a single entry point and a single exit point, the performance objective is to minimize the number of cycles between the entry and exit points. The most successful heuristic is *Critical Path* [9, 10, 11, 12], which gives highest priority to the operations that are at the beginning of the longest dependence chains in the dependence graph.

One of the most widely used scheduling units to achieve higher degree of instruction level parallelism is a *superblock* [3]. It consists of a sequence of consecutively executed basic blocks with a single entry point (at the entry point of the first block in the sequence) and multiple exit points (at the exit point of each block in the sequence). When scheduling superblocks, the performance objective is to minimize the *average weighted completion time*, i.e. the number of cycles from the entry point to each exit weighted by the exit probability.

Consider the dependence graph shown in Figure 1a, where each node corresponds to an operation and each edge corresponds to a dependence between two operations. We consider in our examples only two types of operations: integer operations (nonbold, plain and shaded) and branch operations (bold, shaded, labeled with exit probability). In our examples each dependence has a unit latency unless otherwise specified, and we schedule for a fully pipelined

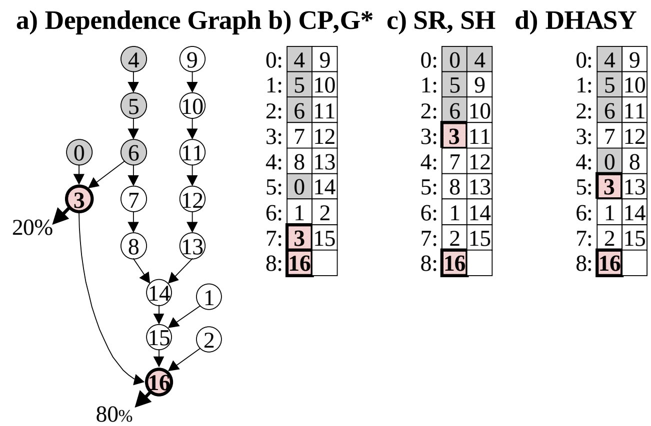

Figure 1. Schedules for Critical Path (CP), Successive Retirement (SR), Dependence Height and Speculative Yield (DHASY), G*, and Speculative Hedge (SH).

machine with two general purpose functional units. The integer operations of the first block (predecessors of branch 3) are shaded and the integer operations of the second block (exclusive predecessors of branch 16) are plain.

We define *EarlyDC* and *LateDC* by ignoring the resource constraints and only considering the dependence constraints. For an operation *v*, we let *EarlyDC[v]* be the earliest cycle that *v* can be issued. For each succeeding branch *b*, we let $LateDC_b[v]$ be the latest cycle that *v* can be issued without forcing *b* to issue in a cycle later than *EarlyDC[b]*. The critical path of the schedule is $CP = \max_v(EarlyDC[v])$.

Applying the Critical Path heuristic to the superblock in Figure 1a gives the schedule shown in Figure 1b. The last exit is scheduled as early as possible on a two issue machine and the side exit is delayed by 4 cycles. In general, Critical Path performs best on wide issue machines where resources are not too constraining.

At the other end of the spectrum, *Successive Retirement* [8] assigns the highest priority to the operations in the first block, the second highest to the operations in the second block, etc. The lowest priority is assigned to operations in the last block of the superblock. Critical Path is then applied to distinguish the priorities of the operations within each block. Successive Retirement gives the schedule shown in Figure 1c, which is optimal since both exits are scheduled as early as possible. In general, Successive Retirement performs best for narrow issue machines where resources are very constraining.

A recently proposed superblock scheduling heuristic [8], referred to as *G** here, attempts to find middle ground between Critical Path and Successive Retirement by selectively applying Successive Retirement to a few critical branches. The first critical branch is the branch with the smallest *rank* among all other branches. For each branch *b*, G* schedules the dependence graph rooted at *b* using a secondary heuristic (Critical Path here). Branch *b*'s rank is the cycle in which b is scheduled divided by the sum of the exit probabilities for b and its preceding branches. G* then removes the critical branch and its predecessors and proceeds recursively with the remaining operations. G* adapts to a wider range of machines than the two previous heuristics but is still sensitive to the secondary heuristic. In Figure 1, only the last branch is critical and G* results in the same schedule as Critical Path.

Another superblock scheduling heuristic, *Dependence Height and Speculative Yield* (DHASY) [1, 13], extends the Critical Path heuristic to superblocks by weighting the critical paths by the exit probabilities. The priority of an operation v is defined as the sum over each successor branch b of $CP{+}1\text{-}LateDC_b[v]$ multiplied by the exit probability of b [13]. This heuristic works quite well in practice; however, it may delay an infrequently taken side exit in some instances where resources are constraining, as shown in Figure 1d.

In Figure 1, Critical Path and G* do not find an optimal schedule because branch 16 cannot be scheduled in fewer than 8 cycles, one cycle more than the longest dependence chains. As noted by Dietrich and Hwu in a similar example [14], resources are the limiting factor for branch 16 since at least $\lceil 16/2 \rceil = 8$ cycles are needed to schedule the 16 predecessors of branch 16 on a two issue machine. This one cycle gap between dependence and resource constraints is just large enough to schedule branch 3 early without delaying branch 16.

Speculative Hedge [14] exploits this observation to schedule frequently taken branches early without needlessly delaying other branches. Before scheduling an operation, Speculative Hedge investigates each ready operation to determine whether it *helps* some of the unscheduled branches. An operation can help a branch for two main reasons. One reason is that the operation is on the critical path to the branch and not scheduling the operation will delay the branch by one cycle. Another reason is that the operation consumes a resource critical to the branch, and scheduling some other operation may delay the branch by one cycle. An operation's priority is the sum of the exit probabilities of all helped branches. Ties are broken by first selecting the operation with the largest number of helped branches and then by the smallest late time. In our first example Speculative Hedge finds the same optimal schedule as Successive Retirement. It does so because every operation helps branch 16 by consuming one of the critical resources constraining branch 16. As a result, the predecessors of branch 3 help both branches and are thus scheduled with a higher priority than the other operations.

3 Motivation for new bounds and heuristic

We now present three observations that motivate our new superblock bound and scheduling heuristic.

Observation 1. We illustrate here the advantage of considering not only if an operation is helping a branch but also if an operation is indirectly delaying a branch by wasting a resource critical to this branch.

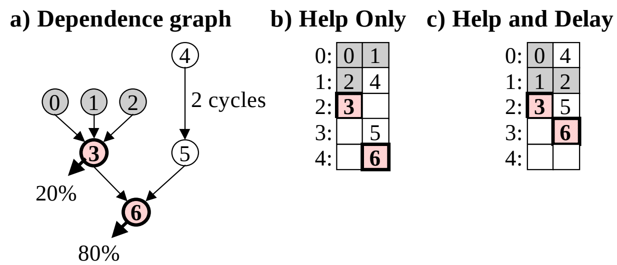

Figure 2. Schedules for Help Only and Help and Delay.

Consider the dependence graph shown in Figure 2a. Branch 3 can be scheduled no earlier than cycle 2 because its three predecessors need at least $\lceil 3/2 \rceil = 2$ cycles on a two issue machine. Similarly, branch 6 can be scheduled no earlier than cycle $\lceil 6/2 \rceil = 3$. Since both branches are resource constrained, operations 0, 1, and 2 help both branches whereas operation 4 helps only one branch. A help-based heuristic would thus assign the highest priority to operations 0, 1, and 2 and find the schedule shown in Figure 2b. Operation 4 is scheduled in cycle 1 and because it initiates a three cycle dependence chain to branch 6, branch 6 is delayed by one cycle.

A more precise analysis in the first cycle reveals that branch 3 needs one out of operations 0, 1, and 2 (because branch 3's predecessors only need three out of the four issue slots in the first two cycles). It also reveals that branch 6 needs operation 4 (because of the dependence chain) and two out of operations 0, 1, 2, and 4 (because branch 6's predecessors need all 6 issue slots in the first three cycles). The needs of these two branches are compatible since scheduling operations 0 and 4, for example, satisfies the specific needs of both branches.

Our new heuristic capitalizes on this observation by scheduling operations that help branches with mutually compatible needs (and large exit probabilities) and that delay branches with unfulfilled needs (and small exit probabilities). It would result in the schedule in Figure 2c.

Observation 2. We show here that determining whether an operation is on a critical path using only the length of dependence chains can be overly optimistic and may lead to suboptimal schedules. This problem is alleviated in part by

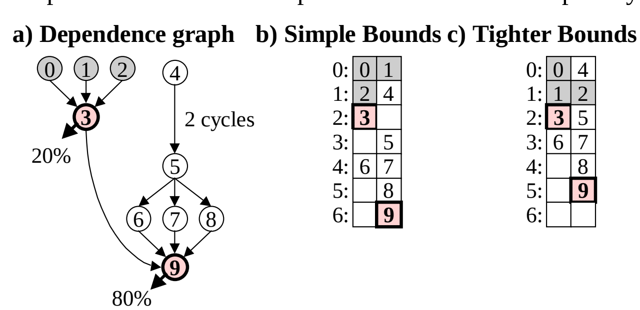

Figure 3. Schedules using simple bounds and tighter bounds.

using a bound that more tightly integrates dependence and resource constraints.

Consider the dependence graph in Figure 3a. As in Figure 2, branch 3 can be scheduled no earlier than cycle 2 and needs at least one out of operations 0, 1, or 2 in the first cycle due to resource constraints. Branch 9 can be scheduled no earlier than cycle $\lceil 9/2 \rceil = 5$ and needs one out of operations 0, 1, 2, and 4 in the first cycle due to resource constraints. The question is: “Does branch 9 need operation 4 due to dependence constraints (as in Figure 2), or not?”

An answer based solely on the length of the dependence chains in the dependence graph is “No” since the longest dependence chain between operations 4 and 9 is only four cycles long. However, not scheduling operation 4 in the first cycle will always delay branch 9 by one cycle, as shown in Figure 3b, because the minimum number of cycles between operations 4 and 9 is four only if operations 6, 7, and 8 can be scheduled in a single cycle. Clearly this is not possible on a two issue machine. The minimum number of cycles between operations 4 and 9 is therefore five and the accurate answer is “Yes, branch 9 needs 4 because of dependences.”

By using the tighter bounds, our heuristic can more accurately compute the minimum number of cycles between operations 4 and 9, giving the schedule in Figure 3c.

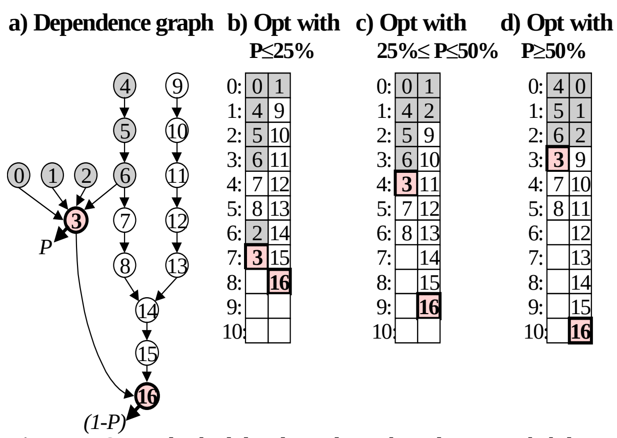

Figure 4. Optimal schedules depends on the side exit probability *P*.

Observation 3. We illustrate here that it is sometimes advantageous to delay a branch to schedule another branch earlier, even when the former has a larger exit probability than the latter.

Consider the dependence graph shown in Figure 4a which is identical to the one in Figure 1a except for operations 1 and 2. Because of this small change, the branches cannot be both scheduled as early as possible. As shown in Figure 4bcd, three different schedules are optimal depending on the side exit probability *P*. With *P*=26%, interestingly, the optimal schedule in Figure 4c delays branch 16 with exit probability 74% by one cycle.

We investigate tradeoffs between branches using a novel Pairwise bound. For the above example the bound indicates that (1) scheduling branch 16 in cycle 8 delays branch 3 by at least four cycles, (2) scheduling branch 16 in cycle 9 delays branch 3 by at least one cycle, and (3) scheduling branch 16 in cycle 10 should not delay branch 3. With this information, our scheduler can better identify the branch tradeoff that minimizes the expected completion time of each branch pair.

4 Lower bounds for superblock scheduling

The improved bounds on the superblock scheduling problem build upon the fast and efficient basic block bounds described by Rim and Jain [18] and Langevin and Cerny [17]. However the results described in the following sections are independent of the algorithm used to compute the lower bound for the basic bock scheduling. We describe an improvement to the Langevin and Cerny algorithm that speeds up the computation of the bound by about 30% in typical dependence graphs. We then describe an improved bound that accounts for the interaction of the resource and dependence conflicts between pairs of branches.

4.1 Lower bounds for basic blocks

In the basic block scheduling problem, the goal is to find a schedule of the operations that satisfies the dependence and resource constraints such that the length of the schedule is minimized.

Rim and Jain. Rim and Jain show that tight lower bounds on the solution to the basic block scheduling problem can be computed by solving a relaxation of this problem [18]. In this case we let *LateDC[i]* be the latest cycle that operation *i* can be issued without causing the last operation to be delayed, ignoring resource constraints. If we let *c* be the schedule length to be minimized, then the *i*th operation issues in cycle t_i such that $EarlyDC[i] \leq t_i \leq LateDC[i] + (c - CP)$. The relaxed scheduling problem is constructed by adding these additional constraints and deleting the dependence constraints. In addition, operations that use functional units that are not fully pipelined are replaced with a series of dependent operations that use fully pipelined units. The solution to the relaxed problem represents a lower bound on the solution to the basic block scheduling problem.

The relaxed scheduling problem can be solved efficiently by the following algorithm. Operations are considered in order of increasing *LateDC* values, and each operation is scheduled in the earliest cycle $t_i \geq EarlyDC[i]$ such that the resource constraints are satisfied. The lower bound is then equal to the maximum of $CP + (t_i - LateDC[i])$ over all operations *i*. The complexity of this algorithm is $O(V+E+c \cdot CP)$, where *V* is the number of operations and *E* is the number of edges in the dependence graph.

Langevin and Cerny. A tighter resource constrained bound, *EarlyRC*, is computed by Langevin and Cerny [17] by recursively using the Rim and Jain algorithm to find lower bounds on the cycles in which every preceding operation can be scheduled. That is, the Early values used by the algorithm are themselves recursively computed using the Rim and Jain algorithm. For an operation i we let $EarlyRC[i]$ be the lower bound returned by the Rim and Jain algorithm when it is applied to the subgraph rooted at operation i with the Early values for each preceding operation j set equal to $EarlyRC[j]$. The complexity of this algorithm is $O(V(V+E+c\ CP))$.

The most computationally expensive part of the Langevin and Cerny algorithm occurs when the late time is computed with a complexity of $O(V)$ and the Rim & Jain bound is computed with a complexity of $O(V+E+c\ CP)$. The following theorem indicates when we may bypass these expensive computations.

Theorem 1 (Trivial Bound Recursion). *Consider the graph G rooted at v where v has a unique direct predecessor p. Assume that the original Langevin and Cerny algorithm is used to compute the bounds for each predecessor of v and yields EarlyRC[p] for p. If the latency* $l_{p,v}$ *from p to v is greater than zero, then* $EarlyRC[v] = EarlyRC[p] + l_{p,v}$.

Proof. Consider the Rim and Jain (RJ) relaxation R (with $EarlyRC$ and $LateDC$) used by Langevin and Cerny (LC) to compute the $EarlyRC[p]$ bound for p and the RJ relaxation R' (with $EarlyRC'$ and $LateDC'$) that is constructed by the original LC algorithm to compute the bound for v. Because v has a unique direct predecessor, $EarlyDC'[x] = EarlyDC[x]$ and $LateDC'[x] = LateDC[x]$ for all predecessors x of v, and $EarlyDC'[v] = EarlyDC[p] + l_{p,v}$ and $LateDC'[v] = LateDC[p] + l_{p,v}$. Since the RJ algorithm considers operations in order of increasing late times, R corresponds to the partial solution of R' where all operations but v have been processed.

We first show that $EarlyRC[v] \geq EarlyRC[p] + l_{p,v}$. By definition of the RJ algorithm, we know that there is at least one operation in R that missed its deadline by $d = EarlyRC[p] - EarlyDC[p]$ cycles. Since R is a partial solution of R', this operation also missed its deadline by d cycles in R'. Thus, $EarlyRC[v]$ is at least $EarlyDC[v] + d$, which may be rewritten as $EarlyRC[p] + l_{p,v}$.

We now show that $EarlyRC[v] \leq EarlyRC[p] + l_{p,v}$. A worst case relaxation R is a relaxation where p misses its deadline by d cycles, allowing v to be placed in cycle $EarlyRC[p] + l_{p,v}$. Operation v satisfies its dependence constraint by being scheduled in this cycle. Furthermore all resources are available in this cycle because $l_{p,v} > 0$ and each operation uses resources at issue time for at most one cycle in the Rim and Jain algorithm. □

The optimized Langevin and Cerny algorithm skips the late time computation and the Rim & Jain bound computation when Theorem 1 holds. This decreases the amount of work by about 30% since approximately 30% of the operations in our benchmark have a unique input register operand and no other dependence constraint.

We apply a variation on this algorithm to compute improved late values that account for resource constraints, $LateRC$. Given a dependence graph G and branch b, we create a new graph G' by deleting all operations that do not precede b and reversing the direction of the remaining dependence edges. Then $LateRC_b[v]$ in G is set equal to $EarlyRC[b]$ in G minus $EarlyRC[v]$ in G', both computed using the Langevin and Cerny algorithm.

4.2 Pairwise bound

The bounds described above, which find a lower bound on the length of a basic block schedule, can be used to find a lower bound on the cost of a superblock schedule. If we let w_i be the exit probability of branch i with latency l_{br}, then $\Sigma_i (EarlyRC[i]+l_{br})*w_i = \Sigma_i EarlyRC[i]*w_i + l_{br}$ over all branches i is one such bound, but it does not take into account the resource conflicts among branches. For example, in Figure 4 branches 3 and 16 have resource constrained early times of 3 and 8, respectively, but if branch 16 issues in cycle 8 then branch 3 cannot issue any earlier than cycle 7. The fact that scheduling one branch as early as possible may delay other branches is not accounted for by this naive bound for superblock scheduling. However a tighter bound can be found by using the Rim and Jain algorithm to quantify this effect. We investigate here this effect for pairs of branches.

We assume that branch i is a predecessor of branch j, and consider the dependence graph G' derived from the original dependence graph G rooted at j with an additional edge from branch i to j with latency $l_{i,j}$. We set the latency $l_{i,j}$ to a given value, and use the Rim and Jain algorithm to find a lower bound on branch j's issue cycle such that i is at least $l_{i,j}$ cycles before j. We define the pair (x_l, y_l) where y_l equals this bound and x_l equals $y_l - l_{i,j}$. We let $l_{i,j}$ vary over all possible latencies and let $(b_{i,j}, b_{j,i})$ equal the (x_l, y_l) pair that minimizes $w_i x_l + w_j y_l$.

Theorem 2 (Pairwise bound). *Consider two branches, i and j, with i predecessor of j in the dependence graph G. Then the pair of values* $b_{i,j}$ *and* $b_{j,i}$ *gives a lower bound on the total weighted completion time of these two branches in any schedule. Furthermore, when computing the lower bound pair, it is sufficient to consider* $l_{i,j}$ *values small enough such that* $y_l = EarlyRC[j]$ *through values large enough such that* $x_l = EarlyRC[i]$*. The* $l_{i,j}$ *values that are considered are never smaller than the latency of branch i and never larger than EarlyRC[j]+1.*

Proof. Assuming that we consider all possible $l_{i,j}$ latencies, we prove this theorem by contradiction. Assume that the $(b_{i,j}, b_{j,i})$ pair is not a lower bound, i.e. there is a solution to the full scheduling problem G with branches i and j in cycle (x', y'), respectively, such that $w_i x' + w_j y' < w_i b_{i,j} + w_j b_{j,i}$. Since we solve the relaxed problem for each possible latency $l_{i,j}$, we must have considered the relaxation G' with $l_{i,j} = y' - x'$ resulting in the pair (x_l, y_l). Because of the validity of the lower bounds produced by the Rim and Jain algorithm, $y_l \leq y'$ and thus $x_l \leq x'$. As a result, $w_i x' + w_j y' \geq$

$w_i\, x_l + w_j\, y_l$, and since by construction $w_i\, x_l + w_j\, y_l \geq w_i\, b_{i,j} + w_j\, b_{j,i}$, the initial assumption leads to a contradiction.

When considering each possible latency $l_{i,j}$, we may start with the largest $l_{i,j}$ value resulting in y_l = *EarlyRC[j]*. Smaller $l_{i,j}$ values cannot decrease y_l below *EarlyRC[j]* but will increase x_l since $x_l = y_l - l_{i,j}$. Thus there is no (x_l, y_l) pair with lower $w_i\, x_l + w_j\, y_l$ below the largest $l_{i,j}$ value resulting in y_l = *EarlyRC[j]*. Similarly, we may stop with the smallest $l_{i,j}$ resulting x_l = *EarlyRC[i]*. Larger $l_{i,j}$ values cannot decrease x_l below *EarlyRC[i]* but will eventually increase y_l when $l_{i,j}$ becomes large enough to contribute to the critical path to j.

Furthermore, we know that the considered $l_{i,j}$ values are never smaller than the latency of branch i because there must be a control flow dependence from branch i to j with branch i's latency since the branches in a superblock are always ordered. Also, the considered $l_{i,j}$ values are never larger than *EarlyRC[j]+1* because this latency is large enough to accommodate all predecessors of j in the last *EarlyRC[j]+1* cycles of the Rim and Jain relaxation. As a result, all the machine resources of the first cycles are exclusively available for the predecessors of i, thus ensuring that x_l = *EarlyRC[i]*. □

An efficient implementation is given in Figure 5. The first $l_{i,j}$ instance investigated (in lines 4-5) attempts to schedule both branches i and j as early as possible. If it is possible, then there is no tradeoff among branches. Otherwise, the latency is decreased until branch j is scheduled as early as possible (lines 7-9). The latency is then increased until branch i is scheduled as early as possible (lines 10-13).

```
(1)  Compute pairwise bound (bij, bji) for i and j
(2)    <consider the subgraph G_l rooted at j with
(3)      edge from i to j with latency l_i,j>
(4)    l_i,j = EarlyRC[j] – EarlyRC[i]
(5)    <Compute and record pair for Gl>
(6)    if (y != EarlyRC[j]) then
(7)      for l_i,j from EarlyRC[j]–EarlyRC[i]-1 to l_br do
(8)        Compute and record pair for Gl
(9)        if (y == EarlyRC[j]) break;
(10)     for l_i,j from EarlyRC[j]–EarlyRC[i]+1 to
(11)       EarlyRC[j]+1 do
(12)       Compute and record pair for Gl
(13)       if (y - l_i,j == EarlyRC[i]) break;
(14)   return <recorded pair <x,y> with min w_i x+w_j y>

(15) Compute and record pair for Gl
(16)   <compute early and late for G_l using EarlyRC
(17)     and LateRC values as bounds>
(18)   y = <compute Rim & Jain bound for G_l using
(19)     early and late>
(20)   <record pair (y - l_i,j, y)>
```

Figure 5. Computing the pairwise lower bound.

4.3 Pairwise superblock bound

While the results in Section 4.2 clearly apply to superblocks with one side exit, we extend here the applicability of pairwise bounds to superblocks with arbitrary many side exits.

Theorem 3 (Average pair bound). *After computing the pairwise bound for each branch pair in a superblock with B branches, a lower bound on the weighted completion time of the superblock is:*

$$w_1\, Avg_{j\neq 1}(b_{1,j}) + w_2\, Avg_{j\neq 2}(b_{2,j}) + \cdots + w_B\, Avg_{j\neq B}(b_{B,j}) + l_{br}$$

where l_{br} is the latency of the branches.

Proof. Recall that the pairwise bound for branches i and j yields the values $b_{i,j}$ and $b_{j,i}$ which satisfy the following inequality,

$$w_i\, b_{i,j} + w_j\, b_{j,i} \leq w_i\, t_i + w_j\, t_j,$$

where t_i and t_j are the issue cycles of i and j in any schedule. For $B=3$ branches, such a constraint can be found for each pair of branches:

$$w_1\, b_{1,2} + w_2\, b_{2,1} \leq w_1\, t_1 + w_2\, t_2$$
$$w_1\, b_{1,3} + w_3\, b_{3,1} \leq w_1\, t_1 + w_3\, t_3$$
$$w_2\, b_{2,3} + w_3\, b_{3,2} \leq w_2\, t_2 + w_3\, t_3.$$

These inequalities can be summed and rearranged, giving

$$w_1\, Avg(b_{1,2}, b_{1,3}) + w_2\, Avg(b_{2,1}, b_{2,3}) + w_3\, Avg(b_{3,1}, b_{3,2}) \leq w_1\, t_1 + w_2\, t_2 + w_3\, t_3\ ,$$

and in general for B branches

$$w_1 Avg_{j\neq 1}(b_{1,j}) + w_2\, Avg_{j\neq 2}(b_{2,j}) + \cdots + w_B\, Avg_{j\neq B}(\, b_{B,j}) \leq \Sigma_i\, w_i t_i.$$

By adding l_{br} to both sides of the above inequality, the right hand side corresponds to the weighted completion time of the superblock and the left hand side is the lower bound stated in Theorem 3. □

4.4 Higher order Superblock bounds

The bound in Sections 4.2 and 4.3 is based on the interaction of branch pairs. Higher order bounds are obtained by investigating similarly the interaction of a larger number of branches, e.g. a triplewise bound based on the interaction of branch triples. Technical details on how to compute the triplewise and higher order bounds can be found in [19].

5 Balance scheduling heuristic

An overview of the balance scheduling heuristic is as follows. The heuristic first computes the (static) EarlyRC/LateRC/Pairwise bounds presented in Section 4 and then proceeds with the main scheduling loop until all operations are scheduled. In this loop, it first updates the (dynamic) bounds and computes the needs of each branch. It then selects an advantageous set of branches with compatible needs. Next, it picks and schedules one of the operations satisfying the needs of the selected branches. When the current cycle is full, it continues with the next (empty) cycle.

We describe the above steps in more detail. For simplicity we restrict our discussion to fully pipelined machines. Non fully pipelined machines are supported by modeling these resources as proposed by Rim and Jain [18].

5.1 Update bounds and compute empty slots

We describe here an efficient technique to compute and dynamically update the *Early* and *Late* values that guide the scheduling decisions. While the techniques in Section 4 could be used here as well, we employ two less compute-intensive bounds. We proceed one branch at a time, in program order. For each branch *b*, only the subgraph *G* of the dependence graph rooted at *b* is considered.

In Step 1, we update *Early* and $Late_b$ using the dependence constraints in *G*, the issue times of scheduled operations, and the initial *EarlyRC* and $LateRC_b$ bounds.

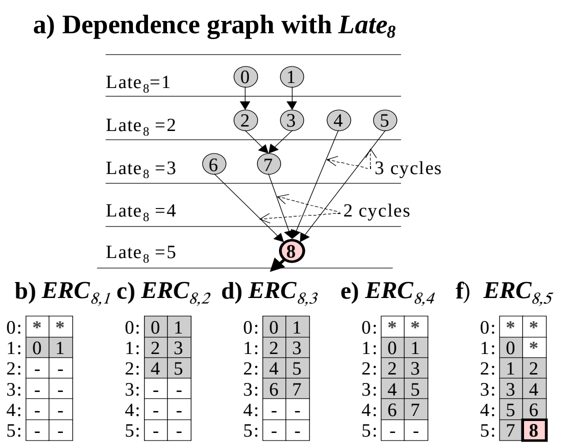

Figure 6. Computing the *ERCs* for 5 subsets of operations.

In Step 2, we evaluate a resource bound. While we could employ the simple resource bound used in Section 3, which simply divides the number of predecessors of *b* by the machine width, this bound is often too weak. For example in Figure 6, this bound indicates that branch 8 can be scheduled at the earliest in cycle $\lceil 8/2 \rceil = 4$, when in fact it can only be scheduled in cycle 5 due to the excessive resource requirements generated by operations 2, 3, 4, and 5.

A good, inexpensive bound is achieved using an observation proposed by Hu [10]. For some cycle *c*, consider the subset of operations S_c that includes each operation *v* such that $Late_b[v] \leq c$. The operations in S_c must be scheduled between the current cycle and cycle *c* because scheduling any operation in S_c later than *c* will by definition delay *b*. If the operations in S_c cannot be scheduled within these cycles, we have found a new, more constraining dynamic resource bound where *b* is delayed by the number of cycles that are needed to accommodate the excess operations. Let *NeedSlot* be the number of operations in S_c and *AvailSlot* be the number of operations that can be accommodated between the current cycle and cycle *c*, inclusive. The number of excess operations is *NeedSlot* – *AvailSlot* and the delay is $\lceil (NeedSlot - AvailSlot) / Width \rceil$. We compute this resource bound for each distinct *c* value in $Late_b$. Each bound is referred to as an *Elementary Resource Constraints with* $Late[b] \leq c$, or $ERC_{b,c}$. The $ERC_{b,c}$ with $c=1...5$ are shown in Figures 6b-f; none of them is more constraining since $Late_8[8]$ already equals 5.

In Step 3, we update *Early* and $Late_b$ accordingly when a more constraining bound is found in Step 2.

In Step 4, we compute the number of empty slots for each $ERC_{b,c}$ (i.e. *AvailSlot* - *NeedSlot*). Empty slots are indicated by starred entries in Figure 6b-f. Empty slot information is important because if there is no empty slot in an $ERC_{b,c}$, as in Figures 6c and 6d, we know that we must now schedule an operation from this $ERC_{b,c}$, or else branch *b* will be delayed.

For machines with multiple resource types, Steps 2 and 4 must be repeated for each resource type *r*, considering only the operations using that resource.

Steps 1, 2, and 4 are relatively time consuming; however, it is not necessary to recompute this data before scheduling each operation. When we determine that the late information of a branch is unchanged (i.e. possibly modified early time does not exceed the late time for *b* computed during the last iteration), we may update the dynamic bounds more inexpensively (*light update*) rather than fully computing them (*full update*).

When the late time of a given branch *b* is unchanged, the late values are unchanged and the empty slot data can be updated as follows. First, when we are scheduling in a new, empty cycle, some of the resources in the previous cycle may not have been fully used. Such resources are lost, and thus the number of empty slots for the given branch *b* and resource type *r* must be decreased accordingly for each *c* value. Second, we must also consider the resources consumed by the last scheduled operation, *lastOp*, if any. If *lastOp* does not uses any resources of type *r*, then the number of empty slots is unchanged. Otherwise, we must consider two cases. In the first case, *lastOp* is not a predecessor of *b* and thus the resources consumed by *lastOp* are wasted from the perspective of *b*. As a result, the number of empty slots for *b* and *r* must be decreased for each *c* value by the number of resources of type *r* consumed by *lastOp*. In the second case, *lastOp* is a predecessor of *b* and thus the resources consumed by *lastOp* are not wasted for each cycle *c* for which *lastOp* is in $ERC_{b,c}$.

5.2 Compute operations needed by each branch

Operations that must be scheduled in the current cycle to allow some branch *b* to be scheduled as early as possible fall into two categories: those that are needed due to dependence constraints and those that are needed due to resource constraints.

Branch *b* needs operation *v* due to dependence constraints if $Late_b[v]$ is smaller than or equal to the current cycle. The set of operations satisfying this condition is referred to as *NeedEach[b]* since *b* needs each operation in *NeedEach[b]* in the current cycle and failing to take each of them will delay the branch by one cycle.

Branch *b* needs *v* because of resource constraints if *v* belongs to an $ERC_{b,c}$ with no empty slot. If there is more than one $ERC_{b,c}$ with no empty slot, we must consider the one with no empty slot and smallest *c*, *cmin*, as it is the most constraining one[1]. The set of operations in $ERC_{b,cmin}$ is referred to as *NeedOne[b]*, since *b* needs one of the operations in *NeedOne[b]* in the current scheduling loop iteration and failing to select one of them will delay the branch by one cycle. For machines with multiple resource types, the above step must be performed for each resource in turn.

In Figure 6c for example, we must select in the current scheduling loop iteration one of the operations in the $ERC_{8,2}$, namely one of operations 0, 1, 2, 3, 4, or 5; this will also satisfy the $ERC_{8,3}$ of Figure 6d.

Note the distinction between *NeedEach[b]* and *NeedOne[b]*: each operation in *NeedEach[b]* is needed in the current cycle; whereas only one operation in *NeedOne[b]* is needed, but it is needed in the current scheduling decision.

Using the above definitions, we say that *b* needs *v* if *v* is in *NeedEach[b]* or *NeedOne[b]*. If either set is empty it is assigned the value nil, meaning that no operation is needed.

5.3 Select compatible branches

Compatible branches are selected one branch at a time by determining if the needs of the current branch can be jointly satisfied with the needs of the previously selected branches. The algorithm keeps track of two sets of operations: (1) the *TakeEach* set which includes the operations that are each needed to satisfy the dependence constraints of the one or more currently selected branches and (2) the *TakeOne* set that includes the operations of which one is needed to satisfy the resource constraints of every currently selected branch.

Consider the conditions that branch *b* must satisfy to be compatible with the currently selected branches. If *b* is selected, the new *TakeEach* set, *TakeEach'*, equals the union of *TakeEach* and *NeedEach[b]*, the operations specifically required by *b*. If there are not enough resources in the current cycle for each operation in *TakeEach'*, then *b* is incompatible with the currently selected branches.

If branch *b* is selected, the new *TakeOne* set, *TakeOne'*, must include the operations in *TakeOne* that also satisfy the resource constraints of *b*, namely *NeedOne[b]*. Thus *TakeOne'* is equal to the operations in the intersection of *TakeOne* and *NeedOne[b]* that are ready and can be accommodated in the current cycle (after accounting for the resources consumed by the operations in *TakeEach'*). If *TakeOne'* is empty, clearly *b* is incompatible.

```
Select compatible branches given BranchOrder:
 TakeEach = TakeOne = nil
 for <each unscheduled branch B with nonnil
      NeedEach and NeedOne, in BranchOrder> do
      <save TakeEach and TakeOne>
      TakeEach |= NeedEach[B]
      if <nonnil TakeEach and TakeOne intersects>
         then TakeOne = nil
      if <every op V in TakeEach is ready and
         fits in CurCycle> then
         TakeOne &= NeedOne[B]
         if <nonnil TakeEach and TakeOne
           intersects> then TakeOne = nil
         TakeOne = <each op V in TakeOne that is
           ready and further fits in CurCycle>
         if <TakeOne is not empty> then
           <branch B is selected; continue>
      <branch B is delayed>
      <restore saved TakeEach and TakeOne>
note: set operations with a nil set are as follows:
 “A & nil = A”, “A | nil = A”, “{each op in nil
 that ...} = nil”, and “<every op in nil> = true”
```

Figure 7. Select compatible branches.

If *b* satisfies the conditions in the previous two paragraphs, *b* is compatible with the currently selected branches, *TakeEach* and *TakeOne* are updated to their new values, and the algorithm proceeds with the next branch.

The branch selection algorithm given in Figure 7 handles the special cases where sets are originally empty (represented by the nil value as opposed to incompatible conditions which are represented by the empty set). It also handles cases where the intersection of TakeEach and TakeOne is not empty.

5.4 Compatible branches using pairwise tradeoff

The algorithm selecting compatible branches in Section 5.3 is sensitive to the order in which the branches are processed. Initially the branches are ordered by decreasing exit probabilities and a branch selection based on this order is performed. If a suitable branch tradeoff is detected, the initial branch order is modified and a new branch selection is performed. After iterating this process a few times, if at all needed, we keep the branch selection with the highest rank. We now describe how to identify a suitable branch tradeoff and how to rank a branch selection.

In a branch selection, the outcome of a branch is either selected, delayed, or ignored. Section 5.3 presented the conditions under which a branch is selected. A nonselected branch is said to be *delayed* if it has some needs and *ignored* if it has none.

---

[1] By definition if $c1 < c2$, the operations in $ERC_{b,c1}$ are always included in the operations of $ERC_{b,c2}$. Thus, selecting some operations of $ERC_{b,cmin}$ satisfies all the other ERCs with no empty slots, whereas the reverse is not necessarily true.

Recall that Observation 3 in Section 3 indicates that in an optimal schedule one branch may be delayed to allow another branch to be scheduled earlier. To take advantage of such branch tradeoffs, we look at each pair of branches *i* and *j* where *i* is delayed and *j* is selected, and the associated pairwise bound $(b_{i,j}\ ,\ b_{j,i})$. If such a pair indicates that best performance is achieved by delaying *i* for the benefit of *j* (i.e. $\text{Late}_i < b_{ij}$), then delaying *i* is a positive outcome and thus the outcome of *i* is revised from delayed to delayedOK. Alternatively, such a pair may indicate that the best performance is achieved by delaying *j* instead of *i* (i.e. $\text{Late}_j < b_{ji}$). If so, it is possible that satisfying the needs of *j* prevented the selection of *i*. Thus, if *j* was indeed processed before *i* in the current branch order, the order of *i* and *j* are interchanged and a new branch selection is evaluated.

The rank of a branch selection is simply the sum of the exit probabilities of the selected and delayedOK branches minus the sum of the exit probabilities of the delayed branches. We keep the branch selection with the largest rank.

5.5 Select best operation

The last scheduling decision is to pick one operation out of the operations in TakeEach or TakeOne. We experimented with several alternatives and the one that performed notably better is the one used by Speculative Hedge [14]. Note that unlike Speculative Hedge which considers each data ready operation, we consider here only the operations that are either in TakeEach or TakeOne (except if they are both nil).

6 Measurements

We present in this section a performance evaluation of our new superblock lower bounds and scheduling heuristics for the SPECint95 benchmark suite. We investigate 6 different VLIW machine configurations. The first three configurations have 1, 2, and 4 *general purpose* functional units and are referred to as *GP1*, *GP2*, and *GP4*, respectively. The next three configurations have 4, 6, and 8 *fully specialized* functional units with a mix defined by the following 4-tuple (#integer units, #memory units, #float units, #branch units) Configuration *FS4* is a (1,1,1,1); *FS6* is a (2,2,1,1); and *FS8* is a (3,2,2,1). All operations are fully pipelined and have a unit latency except for load (2 cycles), float multiply (3 cycles) and float divide (9 cycles). Effects such as cache misses, page faults, and branch mispredictions are factored out to obtain a fair comparison of the scheduling heuristics.

The superblocks were obtained as follows. Classic optimizations were applied to each benchmark program using the IMPACT compiler. The benchmarks were then converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett Packard Laboratories. Superblocks are formed within the LEGO compiler developed at N.C. State University. The resulting 6615 superblocks contain up to 607 operations and 200 branches each.

6.1 Bounds Evaluations

We compare here the quality of the lower bounds computed by the algorithms described in this paper when applied to the SPECint95 benchmark suite for each of the 6 machine configurations. The bounds compared are the naive critical path bound (*CP*), the Hu bound (*Hu*), the Rim and Jain algorithm (*RJ*), the Langevin and Cerny algorithm (*LC*), and our new Pairwise and Triplewise bounds.

	CP			**Hu**			**RJ**		
	Avg	**Max**	**Num**	**Avg**	**Max**	**Num**	**Avg**	**Max**	**Num**
FS4	27.55	93.15	81.88	0.59	27.44	30.87	0.41	24.94	28.93
FS6	4.79	86.49	29.42	0.36	27.53	18.14	0.20	14.76	15.78
FS8	1.55	80.00	15.50	0.64	29.65	12.03	0.18	18.37	7.39
GP1	44.55	93.51	98.38	0.09	17.39	11.04	0.06	17.39	9.90
GP2	13.89	87.18	56.13	0.31	20.00	27.03	0.20	13.85	25.53
GP4	1.24	75.00	11.23	0.08	15.67	6.30	0.03	10.34	5.64

	LC			**Pairwise**			**Triplewise**		
	Avg	**Max**	**Num**	**Avg**	**Max**	**Num**	**Avg**	**Max**	**Num**
FS4	0.17	12.46	26.39	0.05	4.66	22.64	0.00	0.00	0.94
FS6	0.12	13.85	14.95	0.04	5.30	12.32	0.00	0.01	0.59
FS8	0.03	10.42	5.64	0.01	3.02	4.29	0.00	0.00	0.32
GP1	0.04	9.63	9.61	0.01	2.26	8.66	0.00	0.00	0.35
GP2	0.16	13.85	24.99	0.05	5.65	21.69	0.00	0.00	0.95
GP4	0.02	10.34	5.55	0.01	5.17	4.79	0.00	0.00	0.57

Table 1. Performance of bounds relative to the tightest lower bound.

The performance of the bounds is given in Table 1 and is quantified by three numbers: (1) the average percentage difference between the indicated bound and the tightest lower bound, (2) the maximum percentage difference between the indicated bound and the tightest lower bound, and (3) the percentage of the superblocks for which the indicated bound was smaller than the tightest bound

These results demonstrate that the CP bound is much weaker than the other bounds and that the use of resource constraints in Hu, RJ, and LC greatly improves the quality of the lower bounds. The results for RJ and LC are quite similar. This suggests that the extra computational effort invested by LC to compute the recursive bound does not significantly improve the quality of the bound. While the average gaps between both RJ and LC and the Pairwise bound are small, the worst-case gaps can be quite large for RJ and LC, ranging from 9.63% to 24.94% compared to these of Pairwise, ranging from 2.26% to 5.65%. Furthermore, the fraction of the superblocks for which RJ and LC are worse than the Pairwise bound varies from 5.45% to 24.95%. In turn, the Triplewise bound clearly outperforms the Pairwise bound, as the worst gap for Triplewise ranges from 0.00% to 0.01%. Triplewise is better than Pairwise for up to 22.64% of the superblocks and is worst for 0.95% of the superblocks. Therefore the use of Triplewise bound information increases the quality of the bounds computed for a significant fraction of the entire benchmark.

The computational complexity of each bound algorithm is summarized in Table 2 when applying these algorithms to each of the 6615 superblocks in SPECint95 and 6 machine configurations. Complexity expressions are expressed in terms of numbers of operations (V), dependences (E), cycles (C), and types of machine resources (R, which equals 1 and 4 for the GP and FS configurations, respectively). Empirical computational complexity is derived from the worst case complexity by removing factors that are extremely unlikely (e.g. all operations are simultaneously ready) and which do not significantly degrade the quality of the curve-fits. Statistics numbers in Table 2 represent the sum of each loop trip count in the algorithm.

	Computational complexity		**Statistics**	
	Worst-case	**Empirical**	**Average**	**Median**
CP	B(V+E)	B(V+E)	1155.58	152
Hu	B(V+CR)	B(V+CR)	1743.09	314
RJ	B(V+E+C^2)	B(V+C)	1956.21	213
LC	V(V+E+C^2)	V(V+C)	2774.66	275
LC-original	V(V+E+C^2)	V(V+C)	3931.76	361
LC-reverse	BV(V+E+C^2)	BV(V+E+C)	25340.23	631
PW	B^2C(V+E+C^2)	B^2(V+C)	393149.78	221
TW	B^3C^2(V+E+C^2)	B^3(V+C)	1354236.36	156

Table 2. Complexity of the bound algorithms.

These results indicate that the LC bound requires an average of 42% additional computation compared to RC, a surprisingly low overhead given that LC computes a lower bound for each operation whereas RC does so only for each branch. This low LC complexity is mostly the result of Theorem 1 in Section 4.1. Without this optimization (LC-original in Table 2), the bound costs twice as much as RC. Computing Pairwise and Triplewise bounds is significantly more expensive: 2 and 3 orders of magnitude, respectively, compared to RC. Note also the complexity of computing LateRC (LC-reverse in Table 2) which is used by the Balance scheduling algorithm.

6.2 Scheduling Algorithm Evaluation

We investigate here the performance of the superblock scheduling heuristics compared to the tightest lower bound found. Dynamic cycle counts are used through the section. We evaluate here the following primary heuristics: Balance, Successive Retirement (*SR*), Critical Path (*CP*), Dependence Heights and Speculative Yield (*DHASY*), G* (using Critical Path as secondary heuristic), and Help. *Help* is a heuristic based on the main concepts in the Speculative Hedge heuristic [14, 20], namely its resource modeling, its detection of which operations help which branch, and its priority computation. In practice, Help is obtained by omitting the computation of the EarlyRC/LateRC/Pairwise bounds and the selection of compatible branches. We also present results for *Best*, a secondary heuristic which keeps the schedule with the lowest average weighted completion time found by the 6 primary heuristics and a three dimensional cross product of the CP, SR, and DHASY priority functions that invoke a list scheduler 121 times.

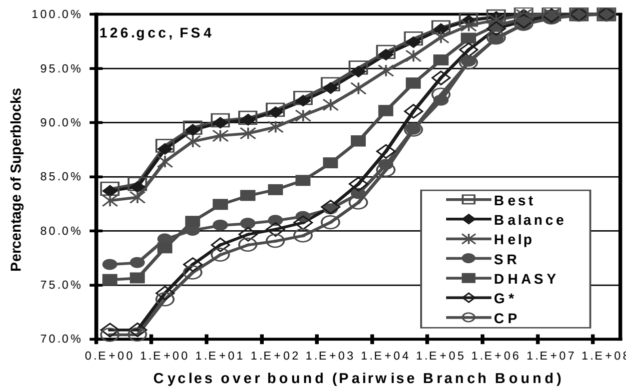

Figure 8. Percentage of superblocks scheduled in no more than a given number of dynamic cycles over bound.

We first evaluate the performance of the above heuristics on the 126.gcc SPECint95 benchmark for a FS4 configuration. The graph in Figure 8 gives the fraction of the superblocks (Y axis) that are scheduled without requiring more than the given number of cycles above the tightest lower bound (X axis). The X axis represents the dynamic number of additional cycles on a logarithmic scale. The graph's Y-intercept gives the fraction of optimally scheduled superblocks, and the heuristics are listed in the legend by decreasing number of optimally scheduled superblocks.

	Bound (cycles)	**Trivial (% of cycles)**	**Slowdown for nontrivial SBs (%)**						
			SR	**CP**	**G***	**DH-ASY**	**Help**	**Bal-ance**	**Best**
FS4	4.90E+12	21.03	2.10	1.25	0.76	0.30	0.37	0.04	0.04
FS6	2.98E+12	41.58	9.26	0.77	0.75	0.71	0.29	0.25	0.14
FS8	2.72E+12	55.89	9.75	0.48	0.54	1.09	0.54	0.18	0.08
GP1	7.36E+12	8.20	0.05	1.93	1.14	0.53	0.02	0.00	0.00
GP2	3.90E+12	20.58	2.75	1.53	0.96	0.32	0.29	0.05	0.05
GP4	2.71E+12	54.02	7.78	0.44	0.37	0.50	0.17	0.10	0.04
Avg	4.09E+12	27.11	5.28	1.07	0.75	0.58	0.28	0.10	0.06

Table 3. Slowdown relative to the tightest lower bound.

In Figure 8, between 70.39% to 83.88% of the superblocks are optimally scheduled. Help performs quite well, with 82.79% optimal superblocks and fewer than 0.53% of superblocks needing more than 10^6 cycles. Balance nearly matches the performance of Best over the entire range of the horizontal axis, with 83.68% of the superblock optimally scheduled and with less than 0.25% of superblocks needing more than 10^6 cycles (vs. 0.22% for Best).

We now evaluate the performance of the superblock scheduling heuristics for the entire SPECint95 benchmark, shown in Table 3. The second column shows the dynamic number of cycles needed to execute each of the 6615 superblocks in SPECint95 as given by our tightest lower bound. The third column shows the fraction of the lower bound cycles spent in trivial superblocks, i.e. superblocks that are optimally scheduled by each of the superblock heuristics. The remaining columns indicate the slowdown in the remaining nontrivial superblocks. Balance is better than all primary heuristics for all configurations and equal to Best for 3 out of the 6 configurations. The last row of Table 3 gives the average slowdown over all machine configurations, with Balance achieving an average slowdown of 0.10%, only 0.04% over that of Best.

	Trivial (% of SB)	Percentage of optimal SB for nontrivial SB						
		G*	CP	SR	DHASY	Help	Balance	Best
Avg	63.38	25.84	25.98	29.15	36.93	64.14	65.70	65.95

Table 4. Average percentage of nontrivial superblocks optimally scheduled.

Table 4 shows the percentage of optimally scheduled nontrivial superblocks. This data suggests that we can reduce compilation time by first comparing the result of DHASY to a lower bound, and then only using a more expensive heuristic when DHASY is not known to be optimal. In this way, the Balance heuristic can be used to schedule 87.44% of the superblocks optimally by applying the heuristic to only about 1/5 of the superblocks.

	Bound cycles	Trivial (% of cycles)	Slowdown for nontrivial SBs					
			SR	CP / G*	DHASY	Help	Balance	Best
Avg	4.09E+12	27.11	5.28	1.07	0.91	0.15	0.34	0.11

Table 5. Slowdown relative to the tightest lower bound. Profiling data unavailable to the schedulers.

All scheduling experiments described so far used perfect profiling information, i.e. the training and actual run are the same. Table 5 indicates the average performance when no profiling data is available. Without profiling data, good results for all scheduling heuristics are obtained when assuming the last branch has a weight of 1000 and all others have unit weight. The performance of SR and CP is unchanged since they do not use profile information, and G* and CP have the same performance since the last branch is always selected as the only critical branch. Compared to the average slowdown in Table 3 over all machine configurations, DHASY increases its slowdown by an additional 0.33%, Help by an additional 0.04%, and Balance by an additional 0.03%. Balance matches the increases of Best (which still uses the actual profiling data to select the best schedule out of the 127 schedules investigated). We can therefore conclude that both Help and Balance are profile insensitive in this benchmark and with these machine configurations.

	Computational complexity		Statistic	
	worst case	empirical	Average	Median
CP	V^2+E	V+E	1483.55	273
DHASY	B(V+E)+V^2	B(V+E)	3694.10	326
G*	B^2(V^2+E)	B(V+E)	67742.83	764
Help	BV(CR+V)	BVR	36565.19	960
Balance	BV(CR+V+B+E)	BVR	38967.35	1151
Balance-full-update	BV(CR+V+B+E)	BV(V+E+R)	674998.84	1692

Table 6. Computational complexity of the heuristics.

The computational complexity of the scheduling heuristics are shown in Table 6 (excluding the computation of bounds from Section 4). While the worst case computational complexity of Help and Balance is high, the *O(BVR)* empirical complexity is similar to the *O(B(V+E))* empirical complexity of DHASY since the number of resource types *R* is typically a small constant. T. Note that when using the light update instead of the full update for the dynamic bounds (in Section 5.1), the cost of the Balance decreases by more than an order of magnitude.

Update once per	Balance heuristic components					
	HlpDel	HlpDel+ Tradeoff	Help	HlpDel+ Bound	Help+ Bound	HlpDel+ Bound+ Tradeoff
Cycle	0.87	0.86	0.82	0.73	0.72	0.71
Op	0.37	0.32	0.28 (1)	0.12	0.12	0.10 (2)

Table 7. Impact of components of Balance heuristic in slowdown for nontrivial superblocks.

Lastly, in Table 7 we investigate which components contribute most to the Balance heuristic. Entries (1) and (2) refer to the previously investigated Help and Balance heuristics, respectively. The first component considers if the heuristic keeps track of whether an operation helps or indirectly delays a branch (*HlpDel*) as proposed in Observation 1, versus only keeping track of whether an operation helps a branch (*Help*). The second components consider if the heuristic uses LC-based bounds as proposed in Observation 2 (*Bound*) as compared to not using these bounds. The third component consider if the heuristic performs branch tradeoffs (*Tradeoff*) as proposed in Observation 3 versus not performing these tradeoffs. Another variable is whether the updating of the bound information is done once per cycle or once per scheduled operation.

By comparing the two rows of Table 7 it can be seen that updating the bound information once per scheduled operation has the largest effect on performance. This allows the scheduler to detect the impact of the first few scheduling decision in a cycle, and use this information to make subsequent scheduling decision. The second most important factor is the use of accurate bounds to better guide the scheduling process, which in effect brings information about yet unscheduled operations earlier in the scheduling process. The third factor is that HlpDelay alone is only beneficial with the bounds, and in fact is best with both bounds and branch tradeoffs. Table 7 also indicates that if the cost of computing the Pairwise bounds needed by the Tradeoff component is too large, the heuristic with Help and Bounds performs nearly as well as the full Balance heuristic.

7 Conclusion

The first contribution of this paper is a set of tighter lower bounds on the execution times of superblocks that specifically accounts for the dependence and resource conflicts between branch pairs and triples. Experiments indicate that the Pairwise and Triplewise superblock bounds are very tight. In the SPECin95 benchmark and for a fully pipelined machine with a mix of 4, 6, and 8 specialized functional units, for example, a schedule achieving the bound is found in 81.65%, 89.62%, and 96.09%, respectively, of the superblocks.

The second contribution of our paper is a novel superblock scheduling heuristic. This Balance heuristic finds a schedule that achieves the bound for 81.35%, 89.59%, and 96.08% of the superblocks for the same respective machine and benchmark. In the nontrivial superblocks of the SPECint95, the average slowdown of Balance compared to the bound is 0.04%, 0.25%, and 0.18%, for the same respective machines, whereas Best (best out of 127 schedules) results in an average slowdown of 0.04%, 0.14%, and 0.08%. The average slowdown for the next best heuristic is double that of the Balance heuristic. When no profiling data is available, Balance still achieves an average slowdown of no more than 0.12%, 0.27%, and 0.27% for the same respective machines. The average slowdown for the next best heuristic is also double that of the Balance heuristic.

Empirical computational complexity indicates that the pairwise superblock bound is $O(N^3)$ whereas the critical path, Hu, Rim and Jain, and Langevin and Cerny bounds are $O(N^2)$, where N approximates the number of operations, branches, and cycles. Similarly the complexity of the Balance scheduling heuristic is $O(N^2)$ whereas only the critical path scheduling heuristic is of a lesser complexity. We furthermore believe that tight bounds and branch tradeoffs may help scheduling beyond superblocks [21, 22].

Acknowledgements

This paper has benefited greatly from discussions with Brian Deitrich regarding the Speculative Hedge heuristic. We are grateful to the Impact group at the University of Illinois, the Elcor group from Hewlett Packard Research Lab, and the Lego group at N.C. State University for their tools in gathering the measurement input.

References

[1] J. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. on Comp*, July 1981.

[2] S. Mahlke et al., Effective compiler support for predicated execution using the hyperblock, *Proc. of the 25th Intl Symp. on Microarchitecture*, pp. 45-54, 1992.

[3] W. Hwu et al., The superblock: an effective technique for VLIW and superscalar compilation, The *Journal of Supercomputing*, 1993, 229-248.

[4] T.Ball and J.Larus, Branch Prediction for free, Proc. of the Conf. on Prog. Language Design and Implementation, 1993.

[5] B. Deitrich, B.-C. Cheng, and W. Hwu, Improving Static Branch Prediction in a Compiler, Proc. of Intl *Parallel Architecture and Compilation Techniques*, 1998.

[6] T.Ball and J.Larus, Optimally profiling and tracing programs", ACM Trans. on Prog. Lang. and Sys.,No. 4, 1994.

[7] A. Eichenberger and S. Lobo, Efficient edge profiling for ILP-processors, Proc. of Intl *Parallel Architecture and Compilation Techniques*, October 1998.

[8] C. Chekuri et al., An analysis of profile-driven instruction level parallel scheduling with application to superblocks, *Proc. of the 29th Intl. Symp. on Microarchitecture*, 1996.

[9] S. Davidson, D. Landskov, B. Shriver, and P. Mallett, Some experiments in local microcode compaction for horizontal machines, *IEEE Trans. on Comp*, C-30, 1981.

[10] T. Hu, Parallel sequencing and assembly line problems, *Operations Research*, Vol. 9, 1961.

[11] W. Kohler, A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems, *IEEE Trans. on Comp*, C-24, Dec 1975.

[12] C. Ramamoorthy and M. Tsuchiya, A high level language for horizontal microprogramming. *IEEE Trans. on Comp*, 1974.

[13] R. Bringmann, Compiler-controlled speculation, PhD thesis, Department of CS, University of Illinois, Urbana, IL, 1995.

[14] B. Deitrich and W. Hwu, Speculative hedge: regulating compile-time speculation against profile variations, *Proc. 29th Intl. Symp. on Microarchitecture*, 1996.

[15] M. Savelsbergh, R. Uma, and J. Wein, An experimental study for LP-based approximation algorithms for scheduling problems, *ACM-SIAM Symp. on Discrete Algorithms*, 1998.

[16] I. Baev, W. Meleis, and A. Eichenberger, Algorithms for total weighted completion time scheduling, *ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[17] M. Langevin and E. Cerny, A recursive technique for computing lower-bound performance of schedules, *ACM Trans. on Design Automation of Elec. Sys.*, 1(4), 1996.

[18] M. Rim and R. Jain, Lower-bound performance estimation for the high-level synthesis scheduling problem, *IEEE Trans. on CAD*, 13(4), 1994.

[19] A. Eichenberger and W. Meleis, Balance Scheduling: Weighting Branch Tradeoffs in Superblocks, TR-99-NCSU.

[20] B. Deitrich, Personal Communication, May 1999.

[21] D. August et al., The Program Decision Logic Approach to Predicated Execution, Proc. of the 26th Intl. Symp. on Computer Architecture, 1999.

[22] M. Schlansker and V. Kathail, Critical Path Reduction for Scalar Programs, Proc. of the 28th *Intl. Symp. on Microarchitecture*, 1995.