Scheduling Superblocks with Bound-Based Branch Trade-Offs

Waleed M. Meleis, *Member*, *IEEE*, Alexandre E. Eichenberger, *Member*, *IEEE*, and Ivan D. Baev, *Student Member*, *IEEE*

Abstract—Since instruction level parallelism in basic blocks is often limited, compilers increase performance by creating superblocks that allow operations to be issued speculatively. This is difficult in general because each branch competes for the processor's limited resources. Previous work manages the performance trade-offs that exist between branches only indirectly. We show here that dependence and resource constraints can be used to gather explicit knowledge about scheduling trade-offs between branches. This paper's first contribution is a set of new, tighter lower bounds on the execution times of superblocks that specifically account for the dependence and resource conflicts between pairs of branches. This paper's second contribution 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 trade-offs. Performance evaluations for superblocks from SPECint95 indicate that our bounds are very tight and that our scheduling heuristic outperforms well-known superblock scheduling algorithms.

Index Terms—Superblock, scheduling heuristic, lower bound, ILP compiler technique.

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 [13] and hyperblocks [18], one of the most widely used scheduling units is a superblock [15], 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 [1], [8] or profiling techniques [2], [10], 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 [5] and Critical Path [6], [14], [16], [19], 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. Dependence Height and Speculative Yield [13], [3] 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* [5], 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 [9], 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.

The major issue not addressed explicitly by previous work concerns performance trade-offs between branches, i.e., the degree to which scheduling a branch early delays other branches. Most past work deals with branch trade-offs indirectly, such as assigning higher priorities to operations that help frequently taken branches [13], [5], [3], [9] or operations that help a branch without penalizing more frequently taken branches [9]. We show in this paper that the dependence and resource constraints can be used to gather explicit knowledge about scheduling trade-offs between branches.

In this paper, we use tight bounds on basic block length as the basis for precisely weighting branch trade-offs in superblocks. This paper's first contribution is a set of algorithms to efficiently compute the tightest known resource-aware bounds on superblock execution time. 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

- *W.M. Meleis and I.D. Baev are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115. E-mail: {meleis, baev}@ece.neu.edu.*
- *A.E. Eichenberger is with the Department of Electrical and Computer Engineering, North Carolina State University, Box 7914, Raleigh, NC 27695-7914. E-mail: alexe@eos.ncsu.edu.*

Manuscript received 1 Sept. 2000; accepted 8 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113590.

to a high performance superblock scheduler such as the one described here.

The second contribution of this 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 trade-offs 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 processor with a mix of four, six, and eight specialized functional units, for example, the best schedules we find achieve the lower bound for 81.65 percent, 89.62 percent, and 96.09 percent of the superblocks in SPECint95, respectively. The Balance superblock heuristic alone finds such a schedule for 81.35 percent, 89.58 percent, and 96.08 percent of the superblocks of the same benchmark and respective processors.

This paper is organized as follows: Section 2 motivates the need for precise resource-aware basic block bounds. Section 3 motivates the need for superblock bounds that account for branch trade-offs, describes algorithms for our new Pairwise and Triplewise superblock bounds using the tight basic block bounds, and explains how to extend them to arbitrarily many branches. Section 4 describes the application of these techniques in the Balance scheduling heuristic. Measurements and conclusions are presented in Sections 5 and 6.

2 Lower Bounds for Basic Block Scheduling

Accurate scheduling algorithms rely on tight lower bounds on execution time to predict the trade-offs that exist between scheduling different operations. However, traditional bounds do not precisely account for both resource and dependence constraints. In this section, we define the basic block scheduling problem and motivate the need for more accurate bounds. We show that bounds based on the length of the critical path alone can be overly optimistic and can therefore lead to suboptimal schedules. We then review several tighter lower bounds that integrate dependence and resource constraints.

A basic block consists of a sequence of V operations and E dependence constraints with a single entry point and a single exit point, branch b. The issue time of operation i in a schedule is denoted as t_i and an edge in the dependence graph from operation i to operation j and with latency $l_{i,j}$ implies that $t_i + l_{i,j} \leq t_j$ in any schedule. We let l_{br} equal the latency of a branch operation. The basic block scheduling problem is to schedule the operations such that the dependence and resource constraints are satisfied and such that the schedule's execution time $t_b + l_{br}$ is minimized.

The simplest lower bound on basic block execution time is solely based on the length of the critical path and, thus, implicitly assumes infinite resources. For an operation i, let $Early_i$ be the earliest cycle that i can be issued if we ignore resource constraints, i.e., $Early_i$ is the length of the longest latency path in the DAG to operation i. Therefore, $Early_b$, which we denote CP for Critical Path, is a lower bound on the issue time of the final operation. We let $Late_i$ be the latest cycle that operation i can be issued, ignoring resource constraints, without forcing the final exit b to issue later than $Early_b$.

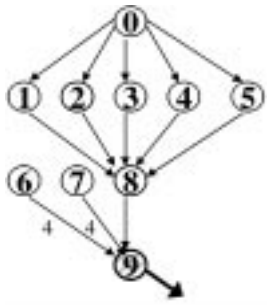

Fig. 1. Dependence graph demonstrating the need for tight, resource-aware lower bounds.

Consider the basic block shown in Fig. 1. In our examples, each dependence has a unit latency unless otherwise specified and we schedule for a fully pipelined processor with two general purpose functional units. In this example, operations 6 and 7 have a latency of 4. Operation 9's Early time is 4, which equals the length of the longest dependence chain passing through the operation. Operation 0 has a Late time of 1, which indicates that operation 0 must be scheduled no later than cycle 1 to avoid delaying operation 9. However, these Early and Late times assume that operations 1-5 can all be scheduled in one cycle. Since no more than two operations can be issued in a cycle, a tighter lower bound on operation 9's issue time is 5 and operation 0 must be issued no later than cycle 0 to avoid delaying operation 9.

The algorithms described in this section compute lower bounds for processors with a set of special-purpose, nonfully pipelined functional units. However, each bound is described assuming all operations in a schedule are of the same type and use fully pipelined units. When multiple types of operations are present, the lower bound is computed with respect to each resource and the tightest lower bound is selected. When an operation uses a nonpipelined unit for L cycles, we replace that operation with L fully pipelined operations connected by a dependence chain.

A better, resource-aware lower bound is achieved using an observation proposed by Hu [14]. We assume that up to m operations can be issued in any cycle. For any integer k, $0 \leq k \leq CP + 1$, we let $n(k)$ be the number of operations i with $Late_i < k$. A lower bound on the issue cycle of the last of these operations is $\lceil n(k)/m \rceil - 1$. An additional time of at least $CP - k + 1$ is needed to satisfy the remaining dependence constraints, therefore, $t_b \geq \max_{k=0...CP}\{\lceil n(k)/m \rceil + CP - k\}$. Note that this expression, for $k = 0$, includes the CP bound. This Hu bound can be computed in $O(V + E + CP)$ time.

Rim and Jain observe that tight lower bounds on the basic block scheduling problem can be computed by finding an optimal solution to a relaxation of the problem [20]. In

any schedule, the following relation must hold for any operation i: $Early_i \leq t_i \leq t_b - (CP - Late_i)$. The relaxed scheduling problem is constructed by adding this constraint for each operation and deleting the dependence constraints, i.e., instead of enforcing the dependence constraints, we enforce a lower bound on the distance between the issue time of each operation and both the beginning of the schedule and the end of the schedule. Minimizing the maximum lateness of any operation in this new problem gives a bound on the original problem.

The relaxed problem can be optimally solved by list scheduling, with the priority of operation i set to $Late_i$, such that $t_i \geq Early_i$ [22]. That is, operations are considered in order of nondecreasing Late times and each operation i is scheduled at the earliest time greater than or equal to its Early time such that the resource constraints are satisfied. The Rim and Jain (RJ) lower bound is then equal to the maximum of $CP + (t_i - Late_i)$ over all operations i. Note that the bound is not simply t_b for the relaxed problem. If t_b is the maximum execution time for the relaxed problem, the RJ bound can be computed in $O(V + E + t_b \cdot CP)$ time.

A tighter resource constrained bound, described by Langevin and Cerny [17], is computed by recursively using the *RJ* bound to find tighter lower bounds on the cycles in which every preceding operation can be scheduled. Instead of using the Early times, which are computed from the length of longest paths in the dependence graph, the Langevin and Cerny (LC) algorithm uses the tighter, resource-aware, RJ bound described above.

The lower bound on the issue time of operation i, $Early_i$, used by the *RJ* bound is tightened by applying the same technique to a subset of the original problem [17]. We let $EarlyRC_i = 0$ for any operation i with no predecessors, as before. For all other operations, we let $EarlyRC_i$ equal the *RJ* lower bound computed for the subproblem consisting of i and all its predecessors using the values of $EarlyRC$ instead of $Early$ for each preceding operation. The value of $EarlyRC_i$ is computed for operation i only after $EarlyRC$ has been computed for all predecessors of i. Since the *LC* relaxation is more constrained than the *RJ* relaxation, *LC* is a tighter bound. The *LC* bound can be computed in $O(V(V + E + t_b \cdot CP))$. We also use a variation on this algorithm to compute Late values, $LateRC$, that account for resource constraints [11].

3 Lower Bounds for Superblock Scheduling

One of the most widely used scheduling units to achieve higher degree of instruction level parallelism is a superblock [15]. 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 execution time, i.e., the number of cycles from the entry point to each exit weighted by the exit probability, while satisfying all resource and dependence constraints. An example is shown in Fig. 2, where branches 3 and 16 are labeled with their exit probablities.

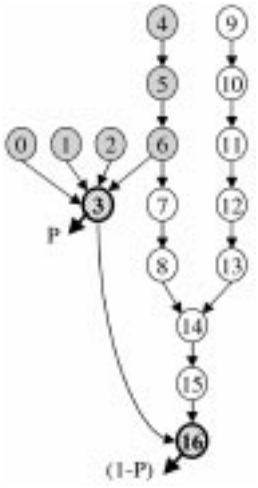

Fig. 2. Dependence graph illustrating the need for pairwise superblock bounds.

In the following sections, we describe new superblock scheduling bounds that account for the interaction of the resource and dependence conflicts between pairs of branches in superblocks. These improved bounds build upon the fast and efficient basic block bounds described in Section 2. However, these results are independent of the algorithm used to compute the lower bound for the basic bock scheduling.

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 execution time of a superblock. If we let w_i be the exit probability of branch i with latency l_{br}, then $\sum_i (EarlyRC_i + l_{br}) \cdot w_i = \sum_i EarlyRC_i \cdot w_i + l_{br}$ over all branches i is one such bound, but it does not take into account the resource conflicts among branches. For example, in Fig. 2, branches 3 and 16 have resource constrained Early times of 3 and 8, respectively, but if branch 16 is issued in cycle 8, then branch 3 cannot be issued 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.

3.1 Pairwise Superblock Bound

We assume that branch i is a predecessor of branch j, and consider the 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 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 3.1 (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 issue time of these two branches in any schedule.*

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

(13)  Compute and record pair for G_l
(14)    <compute Early and Late for G_l using EarlyRC and LateRC values as bounds>
(15)    y = <compute Rim & Jain bound for G_l using Early and Late>
(16)    <record pair (y - l_{i,j}, y)>
```

Fig. 3. Computing the pairwise lower bound.

Proof. 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. □

Lemma 3.2. *Consider two branches, i and j, with i predecessor of j in the dependence graph G. When computing the lower bound pair for i and j as described above, 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. 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 processor 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 Fig. 3. While the results described above apply to superblocks with one side exit, we now extend the pairwise bounds to superblocks with arbitrarily many side exits.

Lemma 3.3 (Average pair bound). *After computing the pairwise bound for each branch pair in a superblock with B branches, a lower bound on the weighted execution time 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. The pairwise bound for branches i and j yields the values $b_{i,j}$ and $b_{j,i}$ which satisfy

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

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:

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

These inequalities can be summed and rearranged, giving

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

and, in general, for B branches,

$$\begin{aligned} &w_1 Avg_{j\neq 1}(b_{1,j}) + w_2 Avg_{j\neq 2}(b_{2,j}) + \cdots + w_B Avg_{j\neq B}(b_{B,j}) \\ &\leq \sum_i w_i t_i. \end{aligned}$$

```
(1)   Compute nth-wise lower bound for branches i_1 ... i_n.
(2)     <consider the subgraph G_l rooted at i_n with edges from i_1 ... i_{n-1} to i_n>
(3)     for l_{i_{n-1},i_n} from <branch latency> to EarlyRC[i_n]+1 do
(4)       for l_{i_{n-2},i_n} from l_{i_{n-1},i_n} + <branch latency> to EarlyRC[i_n]+1 do
(5)         ...
(6)           for l_{i_1,i_n} from l_{i_2} + <branch latency> to EarlyRC[i_n]+1 do
(7)                 <compute Early and Late for G_l using EarlyRC and LateRC values as bounds>
(8)                 y = <compute Rim & Jain bound for G_l using Early and Late>
(9)                 <record nth tuple (y - l_{i_1,i_n}, ... , y - l_{i_{n-1},i_n}, y)>
(10)          end for
(11)        ...
(12)      end for
(13)    end for
(14)    return <recorded nth tuple with minimum average issue weight for i_1 ... i_n>
```

Fig. 4. Computing the *n*th-wise lower bound.

Adding l_{br} to both sides of this inequality gives the weighted execution time of the superblock on the right, and the lower bound stated in Lemma 3.3 on the left. □

3.2 Higher Order Superblock Bounds

The bound in Section 3.1 is based on the interaction of branch pairs. Higher order bounds are obtained by similarly investigating the interaction of a larger number of branches, e.g., a triplewise bound based on the interaction of branch triples.

Consider n branches of a superblock with B branches, $i_1, \ldots, i_n$, in program order. As seen in Section 3.1 for $n = 2$, we let an n-tuple of values $b_{i_1,i_2,\ldots,i_n}, b_{i_2,i_1,i_3\ldots,i_n}, \ldots, b_{i_n,i_1,\ldots,i_{n-1}}$ be the issue cycles for branches $i_1, i_2, \ldots, i_n$, respectively, that give a lower bound on the weighted issue time of these n branches in any schedule. This tuple is computed by varying the latencies between branches $i_1, \ldots, i_{n-1}$ and branch i_n over all possible values, computing the RJ bound for each set of branch latencies, and keeping the solution associated with the smallest weighted issue time. As seen in Lemma 3.2, it is enough to consider latencies in the interval from l_{br} to $EarlyRC_n + 1$, inclusively. The algorithm in Fig. 4 computes the nth-wise bound for the given $i_1, \ldots, i_n$ branches and exploits the fact that branches in a superblock are fully ordered. Note that the innermost loop could be optimized in a fashion similar to the algorithm in Fig. 3. After having computed the nth-wise bound for each nth tuple among the B branches, a lower bound on the weighted execution time of the superblock is:

$$w_1 Avg_{j_2 \neq \ldots \neq j_n \neq 1}(b_{1,j_2,\ldots,j_n}) + w_2 Avg_{j_2 \neq \ldots \neq j_n \neq 2}(b_{2,j_2,\ldots,j_n}) + \cdots + w_B Avg_{j_2 \neq \ldots \neq j_n \neq B}(b_{B,j_2,\ldots,j_n}) + l_{br}.$$

4 Balance Scheduling Heuristic with Branch Trade-Offs

While the goal of basic block schedulers is to minimize overall schedule length, superblock schedulers minimize the number of cycles from the entry point to each exit, weighted by the exit probability. We begin by surveying the most well-known schedulers proposed to solve this problem. For an operation i and a branch b, we let $Late_{b,i}$ be the latest cycle that i can be issued without forcing b to issue later than $Early_b$, while ignoring resource constraints.

The simple Critical Path heuristic sets the priority of each operation equal to its Late time with respect to the last branch and list schedules the operations. Applying the CP heuristic to the superblock in Fig. 5a gives the schedule shown in Fig. 5b (note that CP refers both to a dependence graph's critical path and to the Critical Path heuristic when the distinction between the two uses is clear from the context). The last exit is scheduled as early as possible on a two issue processor and the side exit is delayed by four cycles. In general, CP performs best on wide issue processors where resources are not too constraining.

At the other end of the spectrum, Successive Retirement [5] 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. The Critical Path heuristic is then applied to distinguish the priorities of the operations within each block. Successive Retirement gives the schedule shown in Fig. 5c, which is optimal since both exits are scheduled as early as possible. In general, Successive Retirement performs best for narrow issue processors where resources are very constraining.

A recently proposed superblock scheduling heuristic [5], referred to as $G*$ here, attempts to find middle ground between the Critical Path and Successive Retirement heuristics 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 (CP 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 processors than the two previous heuristics, but is still sensitive to the secondary heuristic. In Fig. 5, 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*) [13], [3], extends the

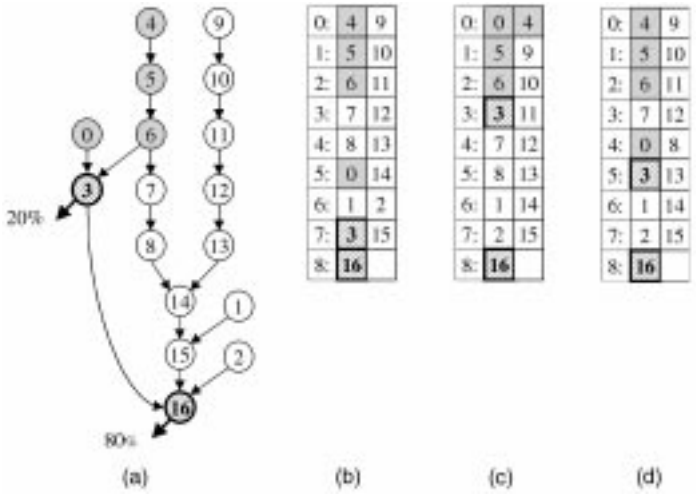

Fig. 5. Schedules for Critical Path (CP), Successive Retirement (SR), Dependence Height and Speculative Yield (*DHASY*), $G*$, and Speculative Hedge (SH). (a) Dependence graph. (b) CP, G*. (c) SR, SH. (d) DHASY.

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 $CP + 1 - Late_{b,v}$ multiplied by the exit probability of b [3]. 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 Fig. 5d.

In Fig. 5, Critical Path and $G*$ do not find an optimal schedule because branch 16 cannot be scheduled in fewer than eight cycles, one cycle more than the longest dependence chains. As noted by Dietrich and Hwu in a similar example [9], 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 processor. This one cycle gap between dependence and resource constraints is just large enough to schedule branch 3 early without delaying branch 16.

Speculative Hedge [9] exploits this observation to schedule frequently taken branches early without needlessly delaying other branches. Before scheduling an operation, Speculative Hedge considers each ready operation to determine whether it helps any unscheduled branches. An operation can help a branch for two main reasons: Because the operation is on the critical path to the branch and not scheduling, it will delay the branch or because the operation consumes a resource critical to the branch and scheduling some other operation may delay the branch. An operation's priority is the sum of the weights 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 Example 5, Speculative Hedge finds the same optimal schedule as Successive Retirement because every operation helps branch 16 by using one of the critical resources constraining branch 16. As a result, the predecessors of branch 3 help both branches and are scheduled with a higher priority than the other operations.

4.1 Overview of Balance Scheduling Heuristic

We now present two observations that motivate our new superblock scheduling heuristic. In the following sections, we describe the details of the algorithm.

4.1.1 Observation 1

It is important that a superblock scheduler consider 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.

Consider the dependence graph shown in Fig. 6a. 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 processor. 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, while operation 4 helps only one branch. A help-based heuristic assigns the highest priority to operations 0, 1, and 2 and finds the schedule in Fig. 6b. Operation 4 is scheduled in cycle 1 and, since 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

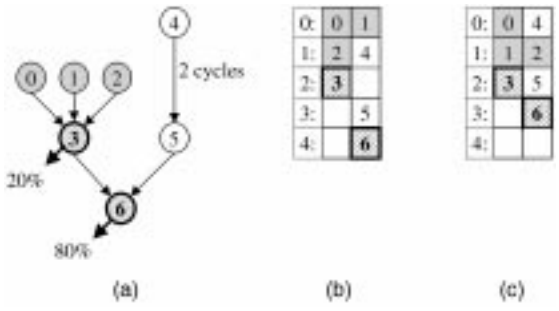

Fig. 6. Schedules for Help Only and Help and Delay. (a) Dependence graph. (b) Help only. (c) Help and Delay.

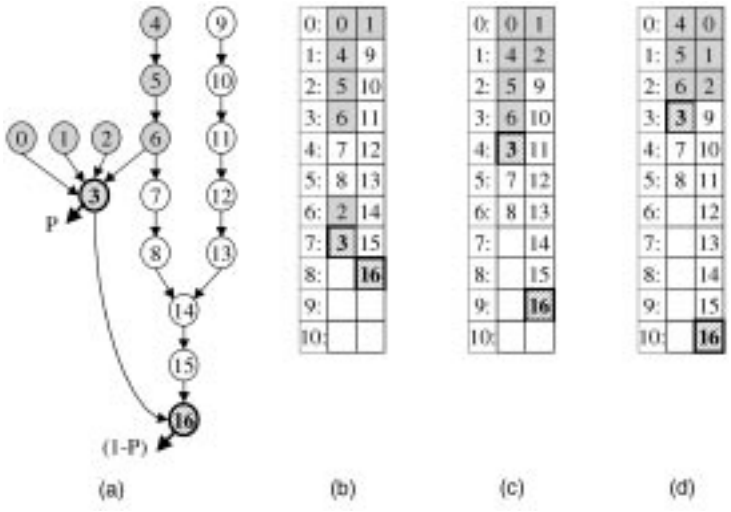

Fig. 7. Optimal schedules depend on the side exit probability P. (a) Dependence graph. (b) Opt with $P \leq 25\%$. (c) Opt with $25\% \leq P \leq 50\%$. (d) Opt with $P \geq 50\%$.

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 six 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 find the schedule shown in Fig. 6c.

4.1.2 Observation 2

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 Fig. 7a which is identical to the one in Fig. 5 except for operations 1 and 2. Because of this small change, the branches cannot both be scheduled as early as possible. As shown in Fig. 7b, Fig. 7c, and Fig. 7d, three different schedules are optimal, depending on the side exit probability P. With $P = 26\%$, interestingly, the optimal schedule in Fig. 7c delays branch 16 with exit probability 74 percent by one cycle.

This trade-off between branches can be precisely understood using the new Pairwise bound described in Section 3.1. 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 trade-off that minimizes the expected execution time of each branch pair.

The Balance scheduling heuristic finds high performance schedules by 1) determining the operations that each branch needs to be scheduled early, 2) computing which branches have compatible needs, and 3) detecting branch trade-offs to actively favor some branches over others. An overview of the balance scheduling heuristic is as follows: The heuristic first computes the (static) $EarlyRC/LateRC/$Pairwise bounds presented in Section 3.1 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. The algorithm is summarized in Fig. 8.

We describe the above steps in more detail in the following sections. For simplicity, we restrict our discussion to fully pipelined processors. Nonfully pipelined processors are supported by modeling these resources as proposed by Rim and Jain [20].

4.2 Update Bounds and Compute Empty Slots

In this section, we describe an efficient technique to compute and dynamically update the Early and Late values that guide the scheduling decisions. While the techniques in Section 3.1 could be used here as well, we employ two bounds that can be computed faster. 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.

In Step 2, we evaluate a resource bound. While we could employ the simplest resource bound which divides the number of predecessors of b by the processor width, this bound is often too weak. For example, in Fig. 9, 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.

Instead, we use the fast Hu bound described in Section 2. For some cycle c, we let $NeedSlot$ be the number of operations that must be scheduled between the current cycle and cycle c and let $AvailSlot$ be the number of operations that can be accommodated between the current

```
(1)   Schedule superblock:
(2)   <compute bounds (EarlyRC/LateRC/branch tradeoffs) as in Section 2>
(3)   <CurCycle = 0 and is empty>
(4)   while <not all ops are scheduled> do
(5)      <update bounds and compute empty slots >
(6)      <compute the operations needed by each branch>
(7)      <select compatible branches with pairwise branch tradeoffs>
(8)      <pick best op among the ops needed by the selected branches>
(9)      <schedule best op>
(10)     if <no further op can be scheduled in CurCycle> then <CurCycle++ and is empty>
(11)  end do
```

Fig. 8. Main scheduling loop of the Balance heuristic.

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 value c in $Late_b$. Each bound is referred to as an *Elementary Resource Constraint with* $Late_b \leq c$ or $ERC_{b,c}$. The $ERC_{b,c}$ with $c = 1 \ldots 5$ are shown in Fig. 9b, Fig. 9c, Fig. 9d, Fig. 9e, Fig. 9f; none of them is more constraining since $Late_{8,8}$ already equals 5.

In Step 3, we update $Early$ and $Late_b$ 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 Fig. 9b, Fig. 9c, Fig. 9d, Fig. 9e, Fig. 9f. Empty slot information is important because if there is no empty slot in an $ERC_{b,c}$, as in Fig. 9c and Fig. 9d, we must now schedule an operation from this $ERC_{b,c}$ or else branch b will be delayed.

For processors 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* [11]) rather than fully computing them (*full update*).

4.3 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 such $ERC_{b,c}$, we consider the one with the smallest c: c_{min} since it is the most constraining one. We let $NeedOne_b$ be the set of operations in $ERC_{b,cmin}$ 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. For processors with multiple resource types, this step is performed for each resource in turn. For example, in Fig. 9c, we must select, in the current scheduling loop iteration, one of the operations in the $ERC_{8,2}$, i.e., one of operations $0 \ldots 5$; this will also satisfy the $ERC_{8,3}$ of Fig. 9d.

While each operation in $NeedEach_b$ is needed in the current cycle, 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

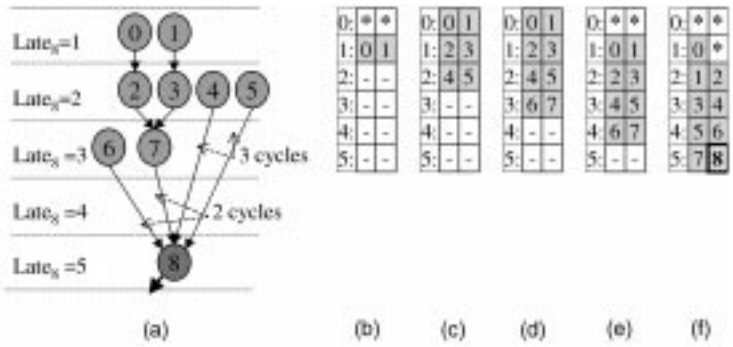

Fig. 9. Computing the *ERC*s for five subseets of operations. (a) Dependence graph with $Late_s$. (b) $\mathrm{ERC}_{8,1}$. (c) $\mathrm{ERC}_{8,2}$. (d) $\mathrm{ERC}_{8,3}$. (e) $\mathrm{ERC}_{8,4}$. (f) $\mathrm{ERC}_{8,5}$.

```
Compute the operations needed by each branch:
    for <each unscheduled branch B> do
    <consider the subgraph G of B and its unscheduled predecessors >
    NeedEach[b] = {each op V in G with Late_{B,V} ≤CurCycle}
    NeedOne[B] = {}
    for <each resource R> do
        <find the smallest C, Cmin, for which EmptySlots[B,R,C] = 0>
        NeedOne[B] |= {each op V in G with Late_{B,V} ≤ Cmin that uses R}
    end do
    if <NeedEach[B] is empty> then NeedEach[B] = nil
    if <NeedOne[B] is empty> then NeedOne[B] = nil
enddo
```

Fig. 10. Compute the operations needed by each branch.

needs v if $v \in NeedEach_b \cup NeedOne_b$. The algorithm that computes $NeedEach$ and $NeedOne$ is given in Fig. 10. If either set is empty, it is assigned the value nil, meaning that no operation is needed.

4.4 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, which 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 $TakeEach \cup NeedEach_b$, the operations specifically required by b. If there are not enough resources 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 satisfy the resource constraints of b, namely $NeedOne_b$. Thus, $TakeOne'$ is equal to the operations in $(TakeOne \cap NeedOne_b)$ 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, b must be incompatible.[1] If b satisfies the conditions in the previous two paragraphs, b is compatible with the currently selected branches, $TakeEach$ and $TakeOne$ are updated, and the algorithm considers the next branch. The branch selection algorithm is shown in Fig. 11.

4.5 Compatible Branches Using Pairwise Trade-Off

The algorithm selecting compatible branches in Section 4.4 is sensitive to the order in which the branches are processed. The branches are ordered by decreasing exit probabilities and a branch selection based on this order is performed. If a suitable branch trade-off is detected, this initial branch order is modified and a new branch selection is performed. After iterating this process, as necessary, we keep the branch selection with the highest rank. We now describe how to identify a suitable branch trade-off and how to rank a branch selection.

In a branch selection, the outcome of a branch is either selected, delayed, or ignored. Section 4.4 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.

Observation 2 indicates that, in an optimal schedule, a branch may be delayed to allow another branch to be scheduled earlier. To exploit such trade-offs, we consider each pair of branches i and j, where i is delayed and j is selected, and the pairwise bound $(b_{i,j}, b_{j,i})$. If the bound indicates that best performance is achieved by delaying i for the benefit of j (i.e., $Late_i < b_{i,j}$), then delaying i is preferred and 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. If so, it is possible that satisfying the needs of j prevented the selection of i. Thus, if j was processed before i in the current branch order, i and j are interchanged and a new branch selection is evaluated. The rank of a branch selection is then the sum of the weights of the selected and $delayedOK$ branches minus the sum of the weights of the delayed branches and we keep the branch selection with the largest rank.

4.6 Select Best Operation

The last scheduling decision is to pick one operation from those in $TakeEach$ or $TakeOne$. From a range of alternatives, we found that the approach used by Speculative Hedge [9] performs the best. Unlike Speculative Hedge, which considers each data ready operation, we consider only operations that are either in $TakeEach$ or $TakeOne$, as shown in Fig. 12.

5 Measurements

We present in this section a performance evaluation of our new superblock lower bounds and scheduling heuristics for

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, while the reverse is not necessarily true.

```
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", "<every op in nil> = true"
```

Fig. 11. Select compatible branches.

the SPECint95 benchmark suite. We investigate six different VLIW processor configurations. The first three configurations have one, two, and four general purpose functional units and are referred to as GP1, GP2, and GP4, respectively. The next three configurations have four, six, and eight 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 (two cycles), float multiply (three cycles), and float divide (nine cycles). Effects of cache misses, page faults, and branch mispredictions are factored out to obtain a fair comparison of the scheduling heuristics.

The superblocks were obtained by applying classic optimizations 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 were formed within the LEGO compiler developed at North Carolina State University. The resulting $6,615$ superblocks contain up to 607 operations and 200 branches each.

5.1 Bounds Measurements

We compare here the quality of the lower bounds on superblock scheduling described in this paper when applied to the SPECint95 benchmark suite for each of the six processor configurations. We compare the new Pairwise and Triplewise superblock bounds to the Langevin and Cerny bound (LC), which is tighter than the naive critical path bound (CP), the Hu bound (Hu), or the Rim and Jain algorithm (RJ).

The performance of the bounds is given in Table 1 and each bound is compared to the tightest bound found for each superblock. This comparison is quantified by: 1) the average percentage difference between the bound and the tightest bound, 2) the maximum percentage difference between the bound and the tightest bound, and 3) the percentage of the superblocks for which the bound was smaller than the tightest bound. While the average gaps between LC and the Pairwise bound is small, the worst-case gap can be large for LC, ranging from 9.63 pecent to 13.85 percent, compared to those of Pairwise, ranging from 2.26 percent to 5.65 percent. In turn, the Triplewise bound outperforms the Pairwise bound, as the worst gap for Triplewise ranges from 0.00 percent to 0.01 percent. Triplewise is better than Pairwise for up to 22.64 percent of the superblocks and is worse for 0.95 percent 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.

```
Pick best op among the ops needed by the selected branches:
    CandidateOps = <ops in TakeEach or TakeOne; all ready ops if both nil>
    for <each op V in CandidateOps> do
        HelpWeight[V] = <sum of the exit probability for each selected unscheduled branch B
            where V is in NeedEach[B] or NeedOne[B]>
        MinLate[V] = <smallest Late_{B,V} among each unscheduled branch B>
    end for
    <keep op V with largest HelpWeight[V], then smallest MinLate[V], then smallest V>
```

Fig. 12. Pick best operation.

TABLE 1
Performance of Bounds Relative to the Tightest Lower Bound (Percents)

	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 2
Complexity of the Bound Algorithms

	Computational complexity		Statistics	
	Worst-case	Empirical	Average	Median
RJ	$B(V+E+C2)$	$B(V+C)$	1,956.21	213
LC	$V(V+E+C^2)$	$V(V+C)$	3,931.76	361
PW	$B^2C(V+E+C^2)$	$B^2(V+C)$	393,149.78	221
TW	$B^3C^2(V+E+C^2)$	$B^3(V+C)$	1,354,236.36	156

The complexity of each superblock bound is given in Table 2 when applying these algorithms to the superblocks in SPECint95 and six processor configurations. Complexity expressions are given in terms of numbers of operations (V), dependences (E), cycles (C), branches (B), and types of processor resources (R, which equals 1 and 4 for the GP and FS configurations, respectively). Empirical complexity is derived from the worst case complexity by removing factors that are unlikely (e.g., all operations are simultaneously ready) and which do not degrade the quality of the curve-fits. Statistics in Table 2 represent the sum of each loop trip count in the algorithm. We see that computing Pairwise and Triplewise bounds is significantly more expensive: 2 and 3 orders of magnitude, respectively, compared with LC.

5.2 Scheduling Measurements

We investigate the performance of the superblock scheduling heuristics compared to the tightest lower bound found. Dynamic cycle counts are used throughout the section. We evaluate the following primary heuristics: Balance, SR, CP, $DHASY$, $G*$ (using CP as secondary heuristic), and $Help$. $Help$ is a heuristic based on the main concepts in Speculative Hedge [9], [7], namely its resource modeling, its detection of which operations help which branch, and its priority computation. We implement $Help$ 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 execution time found by the primary heuristics and a 3D cross product of the CP, SR, and $DHASY$ priority functions that invoke a list scheduler 121 times.

We first evaluate the performance of these heuristics on the 126.gcc SPECint95 benchmark for a FS4 configuration. The graph in Fig. 13 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.

In Fig. 13, between 70.39 percent and 83.88 percent of the superblocks are optimally scheduled. Help performs well, with 82.79 percent optimal superblocks and fewer than 0.53 percent of superblocks needing more than 10^6 cycles. Balance nearly matches Best over the entire range of the horizontal axis, with 83.68 percent of the superblock optimally scheduled and with less than 0.25 percent of superblocks needing more than 10^6 cycles (vs. 0.22 percent 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 $6,615$ 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 heuristics. The remaining columns indicate the slowdown in the remaining nontrivial superblocks. Balance is better

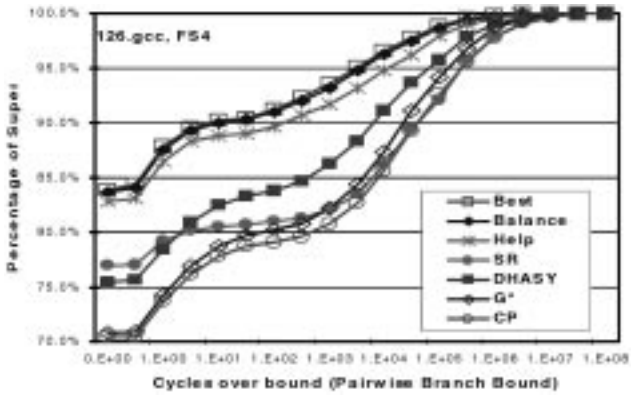

Fig. 13. Percentage of superblocks scheduled in no more than a given number of dynamic cycles over bound.

TABLE 3
Slowdown (Dynamic Cycles) Relative to the Tightest Lower Bound

	Bound (cycles)	Trivial (% of cycles)	Slowdown for nontrivial SBs (%)						
			SR	CP	G*	DHASY	Help	Balance	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 4
Average Percentage of Nontrivial Superblocks Optimally Scheduled

	Trivial (% of SB)	Percentage of optimal SB for nontrivial SB						
		G*	CP	SR	DHASY	Help	Balance	Best
FS4	37.41	13.10	11.18	31.29	25.34	45.98	50.14	50.96
FS6	25.52	36.36	38.40	10.23	41.45	58.25	59.15	59.33
FS8	15.55	49.76	52.90	9.32	51.52	72.72	74.78	74.88
GP1	57.68	3.44	0.21	85.08	20.69	86.83	88.07	88.07
GP2	38.83	12.53	11.04	28.96	33.28	50.65	51.55	51.98
GP4	14.74	39.86	42.13	10.04	49.28	70.39	70.50	70.50
Avg	63.38	25.84	25.98	29.15	36.93	64.14	65.70	65.95

TABLE 5
Slowdown Relative to the Tightest Lower Bound with Profiling Data Unavailable

	Bound (cycles)	Trivial (% of cycles)	Slowdown for nontrivial SBs (%)					
			SR	CP/G*	DHASY	Help	Balance	Best
FS4	4.90E+12	21.03	2.10	1.25	1.09	0.45	0.12	0.05
FS6	2.98E+12	41.58	9.26	0.77	0.69	0.40	0.27	0.26
FS8	2.72E+12	55.89	9.75	0.48	0.74	0.62	0.27	0.17
GP1	7.36E+12	8.20	0.05	1.93	1.47	0.03	0.01	0.00
GP2	3.09E+12	20.58	2.75	1.53	1.09	0.44	0.15	0.07
GP4	2.71E+12	54.02	7.78	0.44	0.35	0.12	0.11	0.10
Avg	4.09E+12	27.11	5.28	1.07	0.91	0.34	0.15	0.11

than all primary heuristics for all configurations and equal to Best for three out of the six configurations. The last row of Table 3 gives the average slowdown over all configurations. Balance achieves an average slowdown of 0.10 percent, only 0.04 percent over that of Best.

Table 4 shows the percentage of optimally scheduled nontrivial superblocks. This data suggests we can reduce compilation time by 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 percent of the superblocks optimally by applying the heuristic to only about $1/5$ of the superblocks.

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. 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 processor configurations, *DHASY* increases its slowdown by an additional 0.33 percent, Help by an additional 0.06 percent, and Balance by an additional 0.05 percent. 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 processor configurations.

The complexity of the scheduling heuristics is shown in Table 6 (excluding the computation of bounds). While the worst case complexity of Help and Balance is high, the $O(B\,V\,R)$ 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. Note that, when using the light update instead of the full update for the dynamic bounds (as described in Section 4.2), the cost of the Balance decreases by more than an order of magnitude.

Last, 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

TABLE 6
Computational Complexity of the Heuristics

	Computational complexity		Statistics	
	Worst-case	Empirical	Average	Median
CP	$V^2 + E$	$V + E$	1,483.55	273
DHASY	$B(V + E) + V^2$	$B(V + E)$	3,694.10	326
G*	$B^2(V^2 + E)$	$B(V + E)$	67,742.83	764
Help	$BV(CR + V)$	BVR	36,565.19	960
Balance	$BV(CR + V + B + E)$	BVR	38,967.35	1151
Balance-full-update	$BV(CR + V + B + E)$	$BV(V + E + R)$	674,998.84	1692

TABLE 7
Impact of Components of Balance Heuristic in Slowdown for Nontrivial Superblocks

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)

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 versus not using these bounds. The third component considers if the heuristic performs branch trade-offs (Trade-off) as proposed in Observation 2 versus not performing these trade-offs. Another variable is whether the updating of the bound information is done once per cycle or once per scheduled operation.

In Table 7, we see 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 decisions in a cycle and to use this information to make subsequent scheduling decisions. The second most important factor is the use of accurate bounds to better guide the scheduling process, which brings information about 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 trade-offs. Table 7 indicates that if the cost of computing the Pairwise bounds needed by the Trade-off component is too large, the heuristic with Help and Bounds performs nearly as well as the full Balance heuristic.

6 Conclusion

The first contribution of this paper is a set of tighter lower bounds on the execution times of superblocks that specifically account for the dependence and resource conflicts between branch pairs and triples. We derive results that significantly decrease the amount of time needed to compute these bounds. Experiments indicate that the Pairwise and Triplewise superblock bounds are very tight. In the SPECint95 benchmark and for a fully pipelined processor with a mix of four, six, and eight specialized functional units, for example, a schedule achieving the bound is found in 81.65 percent, 89.62 percent, and 96.09 percent, respectively, of the superblocks.

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

Empirical complexity indicates that the pairwise superblock bound is $O(N^3)$, whereas the LC bound is $O(N^2)$, where N approximates the number of operations, branches, and cycles. Similarly, the complexity of the Balance heuristic is $O(N^2)$, whereas only the critical path heuristic is of a lesser complexity. We furthermore believe that tight bounds and branch trade-offs may help scheduling beyond superblocks [12], [21].

Acknowledgments

This paper has benefited 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 North Carolina State University for their tools in gathering the measurement input.

REFERENCES

[1] T. Ball and J. Larus, "Branch Prediction for Free," *Proc. SIGPLAN '93 Conf. Programming Language Design and Implementation,* 1993.

[2] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs," *ACM Trans. Programming Languages and Systems,* pp. 1319-1360, 1994.

[3] R. Bringmann, "Compiler-Controlled Speculation," PhD thesis, technical report, Dept. of Computer Science, Univ. of Illinois, 1995.

[4] P. Brucker, M. Garey, and D. Johnson, "Scheduling Equal-Length Tasks under Treelike Precedence Constraints to Minimize Maximum Lateness," *Math. Operations Research,* vol. 2, pp. 275-284, 1977.

[5] C. Chekuri, R. Johnson, B.K. Natarajan, B.R. Rau, and M. Schalnsker, "An Analysis of Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks," *Proc. 29th Ann. Int'l Symp. Microarchitecture (MICRO-29),* 1996.

[6] S. Davidson, D. Landskov, B. Shriver, and P. Mallet, "Some Experiments in Local Code Microcode Compaction for Horizontal Machines," *IEEE Trans. Computers,* vol. 30, 1981.

[7] B.L. Deitrich, personal communication, 1999.

[8] B.L. Deitrich, B.C. Cheng, and W.W. Hwu, "Improving Static Branch Prediction in a Compiler," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 1998.

[9] B.L. Deitrich and W.W. Hwu, "Speculative Hedge: Regulating Compile-Time Speculation against Profile Variations," *Proc. 29th Int'l Symp. Microarchitecture,* pp. 70-79, 1996.

[10] A. Eichenberger and S. Lobo, "Efficient Edge Profiling for ILP Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 1998.

[11] A. Eichenberger and W.M. Meleis, "Balance Scheduling: Weighting Branch Trade-Offs in Superblocks," *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO-32),* 1999.

[12] D. August et al., "The Program Decision Logic Approach to Predicated Execution," *Proc. 26th Int'l Symp. Computer Architecture,* 1999.

[13] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers,* vol. 30, pp. 478-490, 1981.

[14] T. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research,* vol. 9, pp. 841-848, 1961.

[15] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing,* pp. 229-248, 1993.

[16] W. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. Computers,* vol. 24, 1975.

[17] M. Langevin and E. Cerny, "A Recursive Technique for Computing Lower-Bound Performance of Schedules," *ACM Trans. Design Automation,* vol. 1, pp. 443-455, 1996.

[18] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proc. 25th Int'l Symp. Microarchitecture,* pp. 45-54, 1992.

[19] C. Ramamoorthy and M. Tsuchiya, "A High Level Language Horizontal Microprogramming," *IEEE Trans. Computers,* 1974.

[20] M. Rim and R. Jain, "Lower-Bound Performance Estimation for the High-Level Synthesis Scheduling Problem," *IEEE Trans. Computer-Aided Design,* vol. 13, pp. 452-459, 1994.

[21] M. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," *Proc. 28th Int'l Symp. Microarchitecture,* 1995.

[22] B. Simons, "Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines," *SIAM J. Computing,* vol. 12, pp. 294-299, 1983.

Waleed M. Meleis received the BSE degree in electrical engineering from Princeton University in 1990 and the MS and PhD degrees in computer science and engineering from the University of Michigan in 1992 and 1996, respectively. He is an assistant professor in the Department of Electrical and Computer Engineering at Northeastern University, Boston. His research interests are in scheduling algorithms and bounds for modern processors and compilers. He is a member of the IEEE.

Alexandre Eichenberger received the BSE degree in computer science from the Eidgenoessische Technische Hochschule, Zurich, Switzerland, in 1992, and the MS and PhD degrees in computer science and engineering in 1994 and 1996. He is an assistant professor in the Department of Electrical and Computer Engineering at North Carolina State University. His research investigates the interaction of compiler technology and microarchitecture design, including scheduling algorithms for high performance and low register requirements, support for speculative and predicated operations, memory system organization, and exploration of highly concurrent microarchitecture designs. He is a member of the IEEE.

Ivan D. Baev received the Diploma in computer systems engineering from the Technical University, Sofia, Bulgaria, in 1991. He received the MS degree in computer science from Northeastern University and he is currently a doctoral candidate in the Department of Electrical and Computer Engineering. His research interests are in compilation for instruction level parallelism and combinatorial optimization. He is also interested in scheduling algorithms and bounds that apply to detailed ILP processor models. He is a student member of the IEEE.