Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks

C. Chekuri
Dept. of Comp. Sci.
Stanford Univ.
Stanford, CA 94305

R. Johnson
Hewlett Packard Labs
1501 Page Mill Rd
Palo Alto, CA 94304

R. Motwani
Dept. of Comp. Sci.
Stanford Univ.
Stanford, CA 94305

B. Natarajan[1], B.R. Rau, M.Schlansker
Hewlett Packard Labs
1501 Page Mill Rd
Palo Alto, CA 94304

Abstract

Code scheduling to exploit instruction level parallelism (ILP) is a critical problem in compiler optimization research, in light of the increased use of long-instruction-word machines. Unfortunately, optimum scheduling is computationally intractable, and one must resort to carefully crafted heuristics in practice. If the scope of application of a scheduling heuristic is limited to basic blocks, considerable performance loss may be incurred at block boundaries. To overcome this obstacle, basic blocks can be coalesced across branches to form larger regions such as super blocks. In the literature, these regions are typically scheduled using algorithms that are either oblivious to profile information (under the assumption that the process of forming the region has fully utilized the profile information), or use the profile information as an addendum to classical scheduling techniques. We believe that even for the simple case of linear code regions such as super blocks, additional performance improvement can be gained by utilizing the profile information in scheduling as well. We propose a general paradigm for converting any profile-insensitive list scheduler to a profile-sensitive scheduler. Our technique is developed via a theoretical analysis of a simplified abstract model of the general problem of profile-driven scheduling over any acyclic code region, yielding a scoring measure for ranking branch instructions. The ranking digests the profile information and has the useful property that scheduling with respect to rank is provably good for minimizing the expected completion time of the region, within the limits of the abstraction. While the ranking scheme is computationally intractable in the most general case, it is practicable for super blocks and suggests the heuristic that we present in this paper for profile-driven scheduling of super blocks. Experiments show that our heuristic offers substantial performance improvement over prior methods on a range of integer benchmarks and several machine models.

[1]Address all correspondence to this author

1. Introduction

The performance of a VLIW machine depends strongly on the ability of the compiler to exploit instruction level parallelism (ILP) in programs. Unfortunately, the task of the compiler is made difficult by the presence of a number of intractable optimization problems such as instruction scheduling and register allocation. In this paper, we study one such problem — scheduling with profile information. We believe that our results will serve as a good starting point for developing practical heuristics that are to be included in compilers for VLIW machines. As evidence, we present experimental results validating heuristics suggested by our analysis.

A basic block is a program fragment that may only be entered at the top and exited at the bottom. The precedence graph of a single basic block will be a directed acyclic graph (DAG) [2], and in practice, typically consists of fewer than 10 vertices. Scheduling small basic blocks consecutively and separately leads to underutilization of the functional units due to sequentialization effects at the block boundaries. To overcome this limitation two broad approaches have been proposed. One called if-conversion, eliminates the branches via hardware support for predicated execution, allowing instructions to be moved outside of their basic blocks, see [4] for instance. The other approach does not require hardware support, and involves the formation of larger code regions such as traces, [7], and super blocks, [14]. A super block consists of a sequence of basic blocks strung together, with conditional exits at the branch points that separate the basic blocks. Super blocks are typically formed as follows. Given is a code region with branch probabilities available at each branch in the region. These probabilities can be obtained by profiling — collecting usage statistics when the code is executed. Starting at the entry point to the code region, follow the dominant fork at each branch. If at a certain branch, the probability of reaching the branch from the start falls below a certain threshold level, terminate the path. Alternatively, if neither fork in the branch is predominant, terminate the path. The chain of basic blocks along the path traced in this manner is a super block. Delete the chain of basic blocks defining the super block from the code region and repeat the process on the modified portion. Since some of the basic blocks that are deleted might have entries into them from other blocks in the code region, these blocks must be duplicated in the modified region. This process is called "tail duplication." It is clear that the amount of tail duplication will affect the size of the overall code, thereby affecting performance in the face of fixed instruction cache sizes. In light of this, the threshold and the parameters defining the notion of predominance are empirically determined and are outside the scope of this paper.

Much of the literature on scheduling ignores profile information, under the assumption that the region formation algorithm has fully digested the profile information. For instance, super blocks are typically scheduled with heuristics that are oblivious to the profile information, although there are techniques that use profile information as an addendum to classical scheduling techniques, e.g., the speculative yield technique of [3], following [7]. (Example 1 examines several of these techniques.) As region formation algorithms become more sophisticated and produce non-linear code regions encompassing balanced branches, they will be less effective in digesting the profile information. As a result, it will be increasingly important that the code-region be annotated with profile information, and that the scheduling technique effectively utilize this profile information. We believe that our paper offers a first step in this direction.

One might argue that it would be better to tackle the problem of profile driven scheduling from first principles, by treating general code regions, rather than processed code regions such as super blocks. However, there are two good reasons for restricting our study to processed regions. **(1)** as we will see shortly, profile-driven scheduling of generic code regions is computationally intractable and is impractical unless the number of branches in the region is small, say fewer than 16; and, **(2)** tail duplication during the formation of simpler regions such as super blocks often exposes more instruction level parallelism than is extant in the generic code region. Thus, it is a legitimate goal to study good scheduling heuristics even for the limited case of super blocks.

We now make precise our abstraction of the general problem of scheduling with profile information. We are given a directed acyclic precedence graph derived from the source program as described above. Each vertex in the graph represents an operation τ_i with specified execution time t_i, which is the time required to execute τ_i. Each vertex also carries a weight w_i. The weight w_i is the probability that the program exits at vertex i. In other words, w_i is the probability that only the portion of the precedence graph rooted at vertex i needs to be computed, i.e., only those vertices upon which vertex i depends need to be computed. We assume that the target machine has m functional units. We are to schedule the vertices of the graph on the m functional units to achieve the lowest cost, i.e., shortest weighted execution time. Specifically, we are to find a schedule to minimize $\sum_i w_i f_i$, where $f_i = s_i + t_i$ is the finish time of operation τ_i and s_i is its start time.

The general problem in which every node has a weight has been shown to be NP-hard even for $m = 1$ provided we permit arbitrary precedence constraints on the operations [9, 15]. The problem is polynomially solvable when the precedence graph is a forest [12] or a generalized series-parallel graph [1, 15]. For $m \geq 1$, the problem is NP-hard even without precedence constraints, unless the weights are all identical in which case it is polynomially solvable; on the other hand, the problem is strongly NP-hard even when all weights are identical and the precedence graph is a collection of chains [5]. In light of the intractable nature of the problem, we adopt the standard approach of designing approximation algorithms with a bounded performance ratio. The performance ratio of an approximation algorithm is defined as the worst-case ratio of the cost of the approximate solution and the optimal solution.

We begin with a general lemma that shows how to construct an optimal sequential schedule for a general precedence graph with weights. The construction of the lemma can be efficiently exploited only for two restricted versions of the problem, the case where the precedence graph is a tree, i.e., each vertex has exactly one outgoing edge, and the S-graph case where the weights are non-zero only on a single path. The S-graph, to be defined formally in the next section, is the abstraction of the super block. We then show that using the optimal sequential schedule as a list to drive a list scheduling algorithm for multiple functional units guarantees a performance ratio of 2. Finally, in a heuristic extension to our basic lemma, we present a generic scheme for converting any list scheduling algorithm that is insensitive to profile information into a scheduling algorithm for super blocks that is sensitive to profile information. We cannot show tight performance guarantees on this heuristic, but we do present experimental results on a number of sample super blocks obtained by applying the Impact compiler on SPEC benchmark programs. We report that significant savings are possible as compared to prior methods.

Example 1 *Consider the precedence graph of Figure 1, which is the precedence graph of a super block with three branch vertices, vertices 3, 7, and 28 as marked in the figure. The probability that the program will exit via vertex 3, is 0.1. Similarly the probability that it will exit via vertices 7 and 28 are 0.3 and 0.6 respectively. We assume we are given a non-pipelined processor with two identical functional units with all operations having a latency of one cycle. We now schedule this graph on the processor in three ways. First, we carry out critical path scheduling. This is equivalent to*

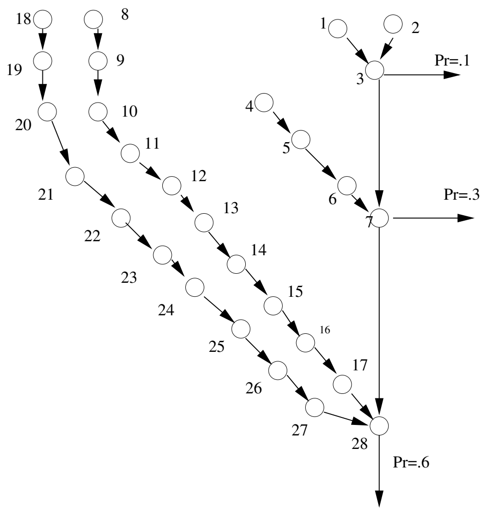

Figure 1. Precedence graph of a super block. Each vertex corresponds to unit latency operation. The probability labels on the branch exits of the graph are the probabilities that the exits will be taken.

list scheduling with the distance from the last exit (vertex 28) as the list priority. Table 1 shows the schedule we would obtain in this case, along with the expected finish time of the schedule.

Secondly, we schedule using speculative yield priorities, as in [3]. Here, the priority of a vertex is the weighted sum of its longest path length from each exit, where the weight is the probability of the exit being taken. The list of vertices sorted in descending order of priority is the list for scheduling. Table 2 shows the schedule we would obtain in this case, and it has a better expected finish time than scheduling by critical path from the last exit.

Lastly, we construct a list schedule that seeks to retire each exit in order at the earliest possible time, as described below. We will call this successive retirement *scheduling. Construct a priority list, based on the critical path priority, of all vertices preceding and inclusive of the earliest exit, (vertex 3 in the figure). Delete all vertices on this list from the graph. Iterate this procedure until the graph is consumed, appending in order the lists created at each iteration. List-schedule, using the list so created. Table 3 shows the schedule we would obtain in this case, along*

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
FU1	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*4*
FU2	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*
Cycle	*9*	*10*	*11*	*12*	*13*	*14*	*15*	
FU1	*1*	*15*	*3**	*16*	*17*	*27*	*28**	
FU2	*5*	*2*	*6*	*26*	*7**			

Table 1. List schedule for the super block of Figure 1 for a two functional unit machine, using the critical path from the last exit as the list. Exits are marked with a *. Expected completion time = 14.0 cycles.

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
FU1	*8*	*9*	*10*	*11*	*12*	*4*	*13*	*2*
FU2	*18*	*19*	*20*	*21*	*22*	*23*	*1*	*5*
Cycle	*9*	*10*	*11*	*12*	*13*	*14*	*15*	
FU1	*14*	*3**	*15*	*16*	*7**	*27*	*28**	
FU2	*24*	*6*	*25*	*26*	*17*			

Table 2. Speculative yield schedule for the super block of Figure 1 for a two functional unit machine. Exits are marked with a *. Expected completion time = 13.9 cycles.

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
FU1	*1*	*3**	*5*	*6*	*7**	*19*	*20*	*21*
FU2	*2*	*4*	*8*	*18*	*9*	*10*	*11*	*12*
Cycle	*9*	*10*	*11*	*12*	*13*	*14*	*15*	
FU1	*22*	*23*	*24*	*25*	*26*	*27*	*28**	
FU2	*13*	*14*	*15*	*16*	*17*			

Table 3. Successive retirement list schedule for the super block of Figure 1 for a two functional unit machine. Exits are marked with a *. Expected completion time = 10.7 cycles.

with the expected finish time of the schedule. Although the successive retirement schedule ignored the profile information completely, it has the lowest finish time of the three schedules.

2. Theoretical Results

Let $G = (V, E)$ denote the precedence graph. A sink in the graph is a vertex with no outgoing edges. We assume, without loss of generality, that the graph has exactly one sink, since we can easily ensure this by the addition of a dummy vertex with in-edges from the sinks of the given graph. Each vertex i is assigned a weight w_i. Let P be a path from a source to a sink in a precedence graph G. We now define the notion of an S-graph, the graph-theoretic abstraction of a super block. Recall that a super block consists of a chain of basic blocks with conditional exits at the branch points separating the basic blocks. The graph G is said to be an S-graph with respect to P if the weights w_i are zero everywhere except on the path P. Without loss of generality, we assume that the weight on the sink is non-zero. If not, we can delete the sink and break the graph into a number of components, retaining only the component containing P. The precedence graph of a super block will contain precedence edges between the branch vertices, since these cannot be executed out of order. Since the branch vertices are the only vertices with non-zero exit probabilities, the precedence graph of a super block is an S-graph.

We say that u immediately precedes v if and only if there is an edge from u to v in the graph. A vertex u precedes a vertex v if and only if there is a path from u to v. For any vertex $u \in V$, let G_u denote the subgraph of G induced by the set of vertices preceding u. A subgraph is said to be closed under precedence if for every vertex u in the subgraph, all vertices preceding u are also in the subgraph.

We define the rank of a vertex i to be the ratio $r_i = t_i/w_i$. Although a vertex can have infinite rank, as will become evident shortly, we will only be interested in those of finite rank. For any set of vertices $A \subseteq V$, we define its weight as $w(A) = \sum_{v_i \in A} w_i$, and its execution time as $t(A) = \sum_{v_i \in A} t_i$; based on this, the rank is $r(A) = t(A)/w(A)$. For instance, the rank of the set of vertices preceding vertex 7 in Figure 1 is 7/.4 = 17.5. The notion of the rank of a set of vertices is meant to capture their relative importance, comparing the sum of their weights to the cost of executing them. Intuitively, the sum of the weights is the contribution made by the set of vertices to the weighted finish time, while the sum of their execution times is the delay suffered by the rest of the graph as a result of scheduling the set of vertices first. As will become evident in our basic lemma, the notion of rank plays a key role in characterizing the optimal sequential schedule.

2.1. The Basic Lemma

In this section we develop a basic lemma characterizing optimal sequential schedules of weighted precedence graphs, i.e., schedules on a single functional unit. Efficient algorithms for the two cases where the precedence graph is a tree, and the precedence graph is an S-graph, can be obtained as special applications of the basic lemma. Previously, optimal sequential algorithms for weighted trees have been described in the literature [1, 8, 12]. Applying the basic lemma to general weighted graphs would cost time exponential in the number of vertices with non-zero weights, a cost that can be practical, if the number of such vertices, i.e., the number of branches is small. From a theoretical point of view, the best known approximation algorithm [11] for sequential scheduling of weighted DAGS has a performance guarantee of 2; however, this algorithm is based on rounding solutions of linear programs and is impractical for the compiler setting.

The following terms are defined with respect to a specific schedule S. We use the term *segment* to refer to a set of consecutive operations in a schedule. Two segments B_1 and B_2 in the schedule are *independent* if there are no operations $u \in B_1$ and $v \in B_2$ such that u precedes v or v precedes u.

Given a weighted precedence graph G, we define G^* to be the smallest precedence-closed *proper* subgraph of G of minimum rank.

We now prove our main lemma.

Lemma: For any graph G, there exists an optimal sequential schedule where the optimal schedule for G^* occurs as a segment which starts at time zero.

Proof: Let S be an optimal schedule for G in which G^* is decomposed into a minimum number of maximal segments. Suppose that G^* is decomposed into two or more segments in S. For $k > 1$, let $B_1, B_2, \ldots, B_k$ be the segments of G^* in S, in increasing order of starting times. Let the segment between B_{i-1} and B_i be denoted by C_i. Let α denote $r(G^*)$ and C^j denote the union of the blocks $C_1, C_2, \ldots, C_j$. From the definition of G^* it follows that $r(C^j) \geq \alpha$ since otherwise we could have included C^j in G^*. Let B^j similarly denote the union of the blocks $B_1, B_2, \ldots, B_j$. It follows that $r(B^k - B^j) < \alpha$ for otherwise $r(B^j) \leq \alpha$ and B^j is smaller than G^*.

Let S' be the schedule formed from S by moving all the B_i's ahead of C_i's while preserving their order within themselves. The schedule S' is legal since G^* is precedence closed. We will show that the cost of S' is no more than that of S which will finish the proof. While comparing the costs of the two schedules, we can ignore the contribution of the vertices that come after B_k since their status remains the same in S'. For the schedule S' we have

$$Cost(S') = \sum_{i \le k} w(C_i)t(B^k) + \sum_{i \le k} w(C_i)t(C^i) + \sum_{i \le k} w(B_i)t(B^i) \ .$$

For the schedule S,

$$Cost(S) = \sum_{i \le k} w(B_i)t(C^i) + \sum_{i \le k} w(C_i)t(B^{i-1}) + \sum_{i \le k} w(C_i)t(C^i) + \sum_{i \le k} w(B_i)t(B^i) \ .$$

Taking their difference gives,

$$Cost(S) - Cost(S')$$

$$\begin{aligned} &= \sum_{i \le k} w(B_i)t(C^i) - \sum_{i \le k} w(C_i)t(B^k - B^{i-1}) \ . \\ &\ge \sum_{i \le k} w(B_i)\alpha w(C^i) - \sum_{i \le k} w(C_i)\alpha w(B^k - B^{i-1}) \\ &\ge \alpha \sum_{i \le k} w(B_i)w(C^i) - \alpha \sum_{i \le k} w(B_i)w(C^i) \ge 0 \ . \end{aligned}$$

The second inequality above follows from our earlier observations about $r(C^i)$ and $r(B - B^j)$. The third step follows from a simple reordering of the order of summation.

2.2. Schedules

The main lemma essentially reduces the scheduling problem to the problem of finding G^*. Then, we can recursively schedule G^* and the graph formed by removing G^* and put their schedules together to obtain an optimal schedule for the entire graph. Unfortunately, the problem of finding G^* for an arbitrary precedence graph is NP-hard. However, if the number of vertices in the graph that have non-zero weight are small, then G^* can be feasibly determined by exhaustive enumeration. Next, we show that finding G^* and hence finding optimal sequential schedules is relatively straightforward if the precedence graph is an S-graph. Let G be an S-graph with respect to a path P. If G^* has a sink not on path P or a sink of zero weight, such a sink can be deleted and both the rank and the number of vertices in G^* reduced appropriately. Thus G^* must be a subgraph with a single sink, and the sink must be a vertex on P with non-zero weight–determining G^* is straightforward. The schedule so obtained is essentially the one obtained by greedily scheduling successive vertices on the path defining the S-graph as early as possible. In terms of the corresponding super block, this amounts to scheduling the basic blocks comprising the super block in control-flow order, exactly the successive retirement schedule given in Example 1.

We can also obtain good ILP schedules for super blocks from the sequential schedule. Specifically, we can show that list scheduling using the optimal sequential schedule as the list, gives good approximate solutions for S-graphs. We defer the proof to the full paper.

Theorem: For an S-graph where all operations have equal execution time, the list scheduling algorithm, using the optimal sequential schedule as the list, is an approximation algorithm with a performance ratio 2.

3. The Practical Heuristic

Notice that our theorem for profile-driven ILP scheduling above is quite limited since it requires that all operations have equal execution time. For the practical situation, we offer a quality heuristic, based on our theoretical analysis from the earlier sections.

3.1. The Modified Rank Function

In our basic lemma, we computed the rank of a set of vertices to be the sum of the latencies in the set divided by the sum of the exit probabilities. For the single unit case, the numerator is a good measure of the length of the time required to compute the set of vertices. Extending our notion of rank from the sequential setting to the ILP setting, we will replace the numerator by the length of the schedule to compute the set of vertices. We call this the modified rank or the *mrank* of a set of vertices, where

$$mrank(A) = \frac{\text{length of schedule for A}}{\text{sum of exit probabilities in A}}.$$

As in the basic lemma, G^* is defined to be the smallest precedence closed subgraph of G of minimum *modified* rank. The intuition behind the modified rank is that the numerator is the time required to retire A, while the denominator is the benefit in retiring A. Thus, the ratio reflects the amount of computatational time required per unit of exit probability. Minimizing this ratio in selecting G^* has the effect of maximizing the "return on investment" in the schedule. Given an S-graph, G^* can found by the following simple procedure.

Algorithm Finding G^* under modified rank

For each branch b of the S-graph,
  On the given processor,
  construct a list schedule for the
  subgraph G_b rooted at b,
  ignoring profile information
  Let T be the length of the schedule
  and let W be the sum of the exit
  probabilities of all exits in G_b.
  rank(G_b) = T/W
G^* is G_b for the earliest b in control
order that has the minimum modified rank.

In the algorithm for computing G^* under modified rank, there is considerable flexibility in selecting the list scheduler, including those that are oblivious to profile information.

3.2. The Heuristic

Now that we know how to compute G^* under modified rank, we can proceed to the scheduling heuristic, given below. In words, the heuristic converts any list scheduler for precedence graphs, to one that is sensitive to profile information for S-graphs. In this sense, the heuristic takes a profile-insensive list scheduling algorithm, and bootstraps it to be profile-sensitive. To start, the heuristic finds G^* under modified rank, using the insensitive list scheduler. It then makes the list for G^* the initial portion of the list for G. The heuristic deletes G^* from G, and iterates, appending the lists each time till all of G is consumed.

Algorithm Scheduler

1. Profile-list = empty.
2. Find G^* under modified rank
   using the insensitive scheduler
3. Append the schedule list of G^* to Profile-list.
4. Remove G^* from the DAG
5. If there are branches remaining, then goto Step 2.
6. List schedule using Profile-list.

Example 2 *We return to the graph of Figure 1, and apply our scheduling heuristic to it. We will use critical path scheduling as the insensitive list scheduler that is oblivious to profile information. Once again, we assume a processor with two identical functional units and equal latencies for all vertices. We have 3 candidates for G^* initially consisting of the subgraph rooted at vertex 3, denoted by G_3, the subgraph rooted at vertex 7, denoted by G_7 and the entire graph G. Computing their modified ranks, we get $rank(G_3) = 2/0.1 = 20, rank(G_7) = 4/0.4 = 10$ and $rank(G_{28}) = 15/1 = 15$, where the numerators are the length of the critical path schedules on the given processor. Since G_7 has the lowest rank, it is G^*. We therefore set Profile-list to be the critical path list for G_7, and remove G_7 from G. Since vertex 28 is the only exit remaining, we append its critical path list to Profile-list. List scheduling using Profile-list on the two functional unit processor yields the schedule of Table 4. Notice that the expected finish time of this schedule is lower than that of the critical path schedule of Table 1 the speculative yield schedule of Table 2 and the successive retirement schedule of Table 3.*

Cycle	1	2	3	4	5	6	7	8
FU1	4	5	6	7*	8	9	10	11
FU2	1	2	3*	18	19	20	21	22
Cycle	9	10	11	12	13	14	15	
FU1	12	13	14	15	16	17	28*	
FU2	23	24	25	26	27			

Table 4. The list schedule constructed by our heuristic for the super block of Figure 1 for a two functional unit machine, using a critical path scheduler as the insensitive scheduler. Exits are marked with a *. Expected completion time = 10.5 cycles.

4. Experimental Results

We now study the performance of the heuristic on a number of optimized super blocks generated by the Impact compiler from the SPEC benchmark programs. We restrict our attention to integer benchmarks, since broadly speaking the floating-point benchmarks yield super blocks with near-zero side-exit probabilities, [16]. We used the Impact compiler to compile these benchmarks, decomposing each program into super blocks and basic blocks only. We report our results on scheduling these blocks over two different classes of machine models, processors with uniform functional units, and processors with heterogenous functional units. All the models are non-pipelined with opcode execution times of as specified in Table 6. The assumption that the machines are not pipelined is only in the interest of simplicity, and is not an inherent limitation of our technique. The uniform processor models have 2, 4, and 8 identical functional units respectively, and are denoted u_2, u_4 and u_8. While uniform machine models are unrealistic in practice, they serves well to study the effect of scaling the number of functional units in a processor. The heterogenous models h_3, h_5 and h_8 are as shown in 5. Model h_3 has one IALU, one FALU and one

Model	#IALU	#FALU	#MEM
h_3	1	1	1
h_5	2	1	2
h_8	3	2	3

Table 5. Functional units of the three heterogenous processors.

Opcode	Time
IALU	1 cycle
FALU	4 cycles
FDIV	8 cycles
LOAD	2 cycles
STORE	1 cycle
BRANCH	1 cycle

Table 6. Opcodes and execution times.

load/store unit, model h_5 has two IALUs, one FALU and two load/store units, while model h_8 has three IALUs, two FALU and three MEM (load/store) units. We assume that BRANCH operations can be performed on the FALU.

First, we use the critical path scheduler as the profile-insensitive scheduling algorithm to drive our heuristic, and compare its performance against three algorithms: **(1)** critical path scheduling from the last exit; **(2)** speculative yield as in Example 1 and [3]; **(3)** successive retirement as in Example 1.

Table 7 shows the improvements achieved by our heuristic over critical-path scheduling for the benchmarks studied over the various machine models. For each benchmark and machine model, we show the improvement in the total schedule length of the benchmark. Formally, we define the *total schedule length* of a benchmark to be the weighted sum of the schedule lengths for all the basic blocks and super blocks for that benchmark, where the weights are the execution frequencies obtained via profiling. In our experience, this is a good measure of the run-time of the benchmark on a typical machine with a sufficient number of registers. Table 8 shows the improvements achieved by our heuristic over speculative yield scheduling, and Table 9 shows the improvements achieved by our heuristic over successive retirement scheduling. Referring to Table 9, notice that on narrow machines such as u_2 and h_3, little performance gain is evidenced. This is because successive retirement is optimal on the sequential processor as shown in our theoretical analysis, and is likely a good schedule on narrow machines.

To substantiate our claim that our heuristic is a general paradigm for converting a profile-insensitive scheduler to

Benchmark	Improvement %					
	u_2	u_4	u_8	h_3	h_5	h_8
espresso	12.8	9.2	5.4	11.3	10.5	8.9
li	6.0	1.4	1.2	3.6	2.1	0.9
compress	6.1	5.9	5.2	5.8	5.3	3.8
alvinn	0.6	0.8	0.9	0.6	0.6	0.8
ear	2.0	1.9	1.5	0.5	1.6	1.0
sc	4.7	4.8	5.1	5.4	6.2	5.1
cccp	4.3	4.7	4.1	3.1	4.6	3.8
cmp	1.2	2.2	2.5	1.1	1.8	2.0
eqn	2.4	2.3	2.1	2.0	1.9	2.0
grep	0.7	1.0	0.9	0.6	0.8	0.8
lex	1.9	0.8	0.9	1.6	1.3	1.0
qsort	10.9	11.3	5.7	6.2	6.2	9.9
tbl	1.4	1.3	1.4	0.9	1.1	1.2
wc	3.8	3.8	4.0	3.0	3.7	4.5
yacc	6.9	4.1	4.3	4.7	4.5	4.4
Average	4.1	3.5	2.8	3.1	3.3	3.1

Table 7. Comparison of heuristic against critical path scheduling. Critical path scheduling is the profile-insensitive scheduler. Shown is improvement in total schedule length of benchmarks.

Benchmark	Improvement %					
	u_2	u_4	u_8	h_3	h_5	h_8
espresso	5.4	3.7	4.7	3.5	3.8	4.8
li	1.3	0.4	0.5	0.3	0.5	0.4
compress	3.6	4.4	5.1	3.6	3.1	3.0
alvinn	0.8	0.4	0.9	0.6	0.6	0.9
ear	1.9	0.8	1.5	0.4	1.6	1.6
sc	2.8	4.2	5.5	3.0	3.7	4.6
cccp	2.8	3.8	4.0	2.3	3.4	4.0
cmp	1.2	2.2	2.5	1.1	1.8	2.0
eqn	1.4	2.0	2.1	1.5	2.0	2.1
grep	0.5	0.9	1.0	0.5	0.8	0.9
lex	1.1	0.7	0.8	1.0	0.9	0.9
qsort	8.9	10.1	6.4	2.9	3.8	7.6
tbl	0.6	1.5	1.3	0.7	1.0	0.7
wc	2.9	3.8	4.0	2.1	2.2	5.1
yacc	3.6	3.4	4.2	2.8	3.9	3.7
Average	2.4	2.7	2.8	1.6	2.1	2.6

Table 8. Comparison of heuristic against speculative yield scheduling. Critical path scheduling is the profile-insensitive scheduler. Shown is improvement in total schedule length of benchmarks.

Benchmark	Improvement %					
	u_2	u_4	u_8	h_3	h_5	h_8
espresso	-0.4	4.7	4.9	-0.4	5.2	6.7
li	-2.1	1.2	0.3	-0.1	3.0	3.8
compress	0.7	4.9	5.3	0.5	4.3	6.1
alvinn	1.1	1.5	1.2	0.5	-0.1	1.7
ear	0.8	-0.5	1.7	0.3	0.6	1.2
sc	0.4	4.9	6.2	2.3	4.0	4.3
cccp	1.8	6.9	3.7	2.5	7.7	5.2
cmp	0.1	3.5	2.5	1.1	3.4	0.2
eqn	2.8	-0.2	2.1	0.9	6.3	6.4
grep	-0.3	0.1	1.0	0.2	2.2	-0.3
lex	1.6	1.7	1.1	2.0	1.5	1.5
qsort	2.1	6.4	9.5	2.7	4.9	8.4
tbl	-3.5	-0.4	1.1	0.1	3.1	1.8
wc	2.9	5.8	4.5	2.1	7.1	3.2
yacc	2.6	5.1	5.4	3.1	5.0	5.3
Average	0.7	2.8	3.2	1.1	3.6	3.5

Table 9. Comparison of heuristic against successive retirement scheduling. Critical path scheduling is the profile-insensitive scheduler. Shown is improvement in total schedule length of benchmarks.

Benchmark	Improvement %					
	u_2	u_4	u_8	h_3	h_5	h_8
espresso	-0.4	4.7	4.9	-0.4	5.1	6.7
li	-2.1	1.5	0.3	0.1	2.9	4.1
compress	0.7	4.1	4.8	0.5	4.7	5.6
alvinn	1.1	1.5	1.2	0.5	-0.1	1.7
ear	0.7	-0.5	1.7	0.3	0.6	1.2
sc	0.4	4.9	6.0	2.3	4.0	4.2
cccp	1.7	6.9	3.7	2.5	7.8	5.1
cmp	0.1	3.5	2.5	1.1	3.4	0.2
eqn	2.8	-0.3	2.1	0.9	6.2	6.4
grep	-0.3	0.1	1.0	0.2	2.2	-0.3
lex	1.6	1.7	1.1	2.0	1.4	1.5
qsort	1.5	6.7	9.7	2.4	4.8	4.9
tbl	-3.5	-0.4	1.1	0.1	3.1	1.8
wc	2.6	5.8	4.5	2.1	7.1	3.2
yacc	2.6	5.3	5.4	3.1	5.0	5.0
Average	0.6	2.8	3.1	1.1	3.6	3.2

Table 10. Comparison of heuristic against successive retirement scheduling. Successive retirement scheduling is the profile-insensitive scheduler. Shown is improvement in total schedule length of benchmarks.

profile-sensitive one, we apply our heuristic using successive retirement as the profile insensitive scheduler, over the same set of benchmarks and machine models. The results are shown in Table 10.

4.1. Discussion

In our performance studies above, the total schedule length of a benchmark depends on the nature and mix of basic blocks and the super blocks produced during compilation. Our heuristic is designed to improve the performance of super blocks that have side exits with substantial exit frequency. If the compiler is not aggressive in creating such super blocks, or if side exits occur very infrequently, the opportunities for performance gains are limited. To examine this in detail, we introduce the notion of the critical path ratio of a super block, which aims to measure the relative importance of the side exits of a super block. To this end, we define the expected critical path length as the weighted sum of the lengths of the critical paths of all the exits, weighted by their exit probabilities. The critical path ratio is the ratio of the expected critical path length to the length of the critical path of the last exit. If the critical path ratio is small compared to unity, then the side exits are significant, and if the critical path ratio is close to unity, then the last exit is predominant. It is clear that every basic block will have a critical path ratio of unity.

Figure 2 shows the average improvement achieved by our heuristic over critical path scheduling, as a function of the critical path ratio. The plots represent averages over all the basic blocks and super blocks obtained from compiling the benchmarks studied. The plots marked u_2, u_4 and u_8 in the figure refer to the respective uniform processor models. As an example of how to read these plots, observe that blocks with critical path ratio of 0.2 enjoy a 30% improvement on average when scheduled by our heuristic, as compared to scheduling by critical path from the last exit, with respect to the two-functional unit machine u_2. As the critical path ratio nears unity, the achieved improvement falls off, as is to be expected since in this case the last exit is predominant, and our heuristic converges to critical path scheduling. Notice also that as the number of available functional units increases from u_2 to u_4 and u_8, the achieved improvement falls off. This is because critical path scheduling is increasingly good for wider processors, (optimal in the limiting case of infinitely wide processors), and there is reduced opportunity for performance gains by rearranging the schedule. Also shown in the figure is the distribution of the blocks, depicted as cumulative percentage against critical path ratio. As an example of how to read this plot, observe that roughly 30% of the blocks in the sample have a critical path ratio of 0.8 or less. At this value of critical path ratio, the performance improvement is down to a few percent. Hence, the remaining 70% of the blocks are not good candidates for improvement via our scheduling heuristic. This suggests that if super block formation heuristics could form super blocks with lower critical path ratios, our scheduling algorithm would have increased opportunity for performance gains.

Another factor that affects the performance gains realized by our scheduling heuristic is the amount of parallelism present in a super block. If a super block has little parallelism, then the critical path schedule will not saturate the processor, and little performance gain can be obtained since the schedule is not constrained by resources. On the other hand, if the super block has a lot of parallism in it, the critical path schedule will saturate the processor, and much performance gain can be had by rearranging the schedule in favor of high probability exits. A good measure of the parallelism available in a block is the processor utilization factor of the schedule for the block. This is essentially the the average load on the processor during the schedule expressed as a percentage. Formally, the *procesor utilization factor* is the number of cycles for which each functinal unit is busy, summed over all functional units, and expressed as a percentage of the product of the length of the schedule and the number of functional units.

Thus we have two independent factors that can affect the gains realized by our scheduling heuristic **(1)** The importance of the side exits as reflected in the critical path ratio and **(2)** the amount of parallelism available as reflected in the processor utilization. We will now examine the results of Table 7 in light of these two factors. To do so, let us extend the notion of the critical path ratio to benchmarks—the critical path ratio of a benchmark is the weighted sum of the critical path ratios of the blocks composing it, where the weights are the execution probabilities of each block. Similarly, the utilization factor of a benchmark is the weighted sum of the utilization factors of the blocks composing it, where the weights are the execution probabilities of each block. Our heuristic should perform well when the critical path ratio is small and the utilization factor is large. We test this hypothesis in Figure 3. The horizontal axis in the plot is the critical path ratio and the vertical axis is the processor utilization for critical path schedule on processor model u_4. Each box in the figure represents a benchmark, with the center of the box corresponding to its critical path ratio and processor utilization on the horizontal and vertical axes respectively.

The length of the side of each box is directly proportional to the improvement achieved by our heuristic on the benchmark, corresponding to the entry in column u_4 of Table 7.

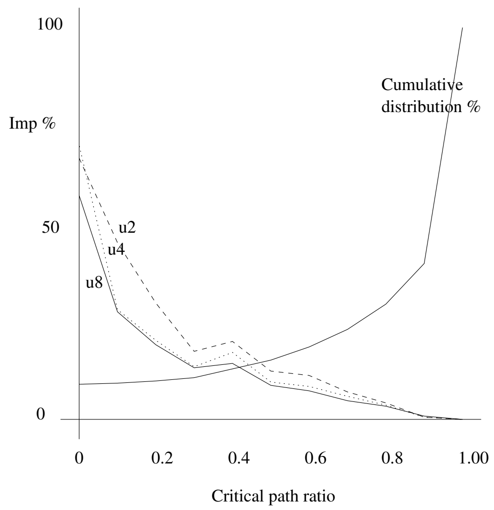

Figure 2. Performance gains of our heuristic over critical path scheduling, critical path scheduling is the profile-insensitive scheduler. Shown is improvement in run time of the blocks as a function of critical path ratio, and the distribution of the blocks as cumulative percentage against critical path ratio.

The benchmarks that have a high critical path ratio enjoy very little performance gain independent of their processor utilization. These benchmarks are shown as small box. Also, benchmarks that have little parallelism, manifested as low processor utilization, enjoy very little performance gain even if they have low critical path ratio. Thus, the performance gains of Table 7 are well explained by our intuition, and lends support to the conclusion that the heuristic exhibits gains where gains are possible.

5. Acknowledgements

We thank the Impact group at the University of Illinois for permission to use the Impact compiler in this study. C. Chekuri was supported by NSF Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corp. R. Motwani was supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corp.

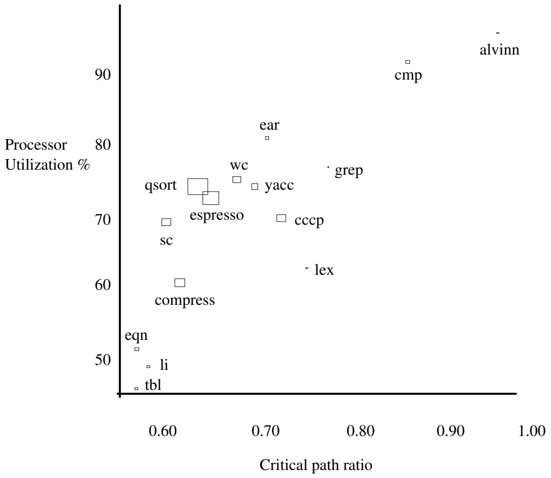

Figure 3. Scatter plot of performance gains of our heuristic over critical path scheduling on the u_4 processor model, critical path scheduling is the profile-insensitive scheduler. Each box corresponds to a benchmark, and the length of the side of the box is proportional to the percentage improvement in performance.

6. Conclusion

We presented a theoretical analysis of the general problem of scheduling a precedence graph with profile information. Our main theoretical result is a general lemma characterizing optimal sequential schedules for a weighted precedence graph. In a heuristic extension to this lemma, we presented a generic scheme for converting profile-insensitive list scheduling algorithms into a profile-sensitive scheduling algorithm for super blocks. Experiments show that in some settings, our heuristic can offer substantial performance improvement over prior methods on a range of benchmarks.

References

[1] D. Adolphson. Single machine job sequencing with precedence constraints. *SIAM J. on Computing*, 6:40–54 (1977).

[2] A. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools.* Addison Wesley, Reading, MA (1988).

[3] R. A. Bringmann, Enhancing instruction level parallelism through compiler controlled optimization. M.S Thesis, University of Illinois, 1992.

[4] J.C. Dehnert and R.A. Towle, Compiling for the Cydra-5 *J. of Supercomputing*, 7:181-228, (1993).

[5] J. Du, J.Y.T. Leung, and G.H. Young. Scheduling chain structured operations to minimize makespan and mean flow time. *Information and Computation*, 92:219–236 (1991).

[6] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. IEEE Trans. on Computers, Vol C-30:478-490.

[7] J. A. Fisher. Global code generation for instruction level parallelism. Tech. Rep. HPL-93-43, Hewlett Packard Labs, June 1993.

[8] M.R. Garey. Optimal task sequencing with precedence constraints. *Discrete Math.*, 4:37–56 (1973).

[9] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, San Francisco (1979).

[10] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM J. on App. Math.*, 17:416-429 (1969).

[11] L.A. Hall, A.S. Schulz, D.B. Shmoys and J. Wein. Scheduling to minimize average completion time: offline and on-line algorithms. In *Proc. of the 7th ACM-SIAM Symp. on Discrete Algorithms*, 1996, 142–151.

[12] W.A. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM J. of App. Math.*, 23:189–202 (1972).

[13] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848 (1961).

[14] W.W. Hwu et al. The super block: An effective technique for VLIW and superscalar compilation. *J. of Supercomputing*, 7:229–248 (1993).

[15] E.L. Lawler. Sequencing jobs to minimize total weighted completion time. *Annals of Discrete Math.*, 2:75–90 (1978).

[16] S. A. Mahlke. Exploiting Instruction Level Parallelism in the Presence of Conditional Branches. Ph.D Thesis, U. of Illinois, Urbana, IL, (1996).

[17] S.A. Mahlke et al. Effective compiler support for predicated execution using the hyperblock. In *Proc. 25th Int. Symp. Microarchitecture (MICRO 25)*, 45–54 (1992)

[18] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Proc. of ICALP* (Springer-Verlag), 751–762 (1991).