Profile-Driven Instruction Level Parallel Scheduling

Chandra Chekuri*, Rajeev Motwani*, Richard Johnson,
B. Ramakrishna Rau, Balas Natarajan
Computer Systems Laboratory
HPL-96-16
January, 1996

VLIW scheduling, profile driven scheduling, superblocks, hyperblocks

Code scheduling to exploit instruction level parallelism (ILP) is a central problem in compiler optimization research, in light of the increased use of long-instruction-word computers. 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 are coalesced across branches to form either superblocks or hyperblocks, with the branches labeled with branch probabilities obtained via profiling. Then, the goal of the scheduling heuristic is to minimize the expected completion time of the superblock or the hyperblock.

In this paper, we analyze the general problem of scheduling with profile information and present some experimental results on heuristics that are suggested by our analysis. Specifically, we carry out a theoretical analysis on a simplified abstraction of the problem to obtain insight into its structure, and to develop provably good algorithms. Based on the theoretical analysis of the simplified abstraction, we develop a generic scheme for converting any list scheduling heuristic for basic blocks into a heuristic for the superblocks/hyperblocks with associated profile information. Our techniques are applicable to the general case of scheduling on piplined heterogeneous units with unequal latencies. Experiments show that our scheme offers substantial performance improvement over critical path scheduling on a range of benchmark inputs and machine models.

Internal Accession Date Only
**Stanford University*, Stanford, California
© Copyright Hewlett-Packard Company 1996

1 Introduction

The performance of a VLIW computer 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 of 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 concerning some heuristics that are suggested by our analysis.

In brief, VLIW computers have multiple functional units, all of which are available for simultaneous use. The job of the compiler is to translate the input code into machine instructions, exploiting the parallelism available as best as possible. In carrying out such a translation, a typical compiler has two phases. In the parsing phase, the compiler parses the input code and generates a precedence graph, referred to as the intermediate representation. In the code generation phase, the precedence graph is converted into machine instructions by scheduling each operation on a suitable functional unit.

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. From the viewpoint of developing heuristics, it is simpler to schedule a program one basic block at a time. Unfortunately, scheduling small basic blocks consecutively and separately leads to underutilization of the functional units at the block boundaries. To overcome this limitation, basic blocks that are separated by branch instructions can be combined into a bigger blocks. Two recent successful techniques are *superblock* formation and scheduling [9], and predicated execution using *hyperblocks* [11]. Both superblocks and hyperblocks have a single entry multiple exit property and each branch is a conditional exit. We can allocate probabilities to each exit representing the likelihood of that exit being used in a typical execution. These probabilities can be obtained by profiling — collecting usage statistics when the code is executed. The problem then is to schedule the blocks so as to minimize the expected finish time, where the expectation is taken with respect to the various branch probabilities.

We now make precise our abstraction of the general problem of scheduling with profile information which models both the superblock as well as the hyperblock case. 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 can be seen as the probability that only the portion of the precedence graph rooted at vertex i needs to be computed, i.e., the program represented by the graph exits at vertex i. We are to schedule the vertices of the graph on the m functional units to achieve the shortest weighted execution time. Specifically, we are to find a schedule to minimize $F = \sum_i w_i f_i$, where $f_i = s_i + t_i$ is the finish time of operation τ_i and s_i is its start time. We assume that the target computer has m functional units. In the interest of simplicity, we assume that these units are identical and not pipelined. (The algorithms presented in this paper can be applied in the general case of pipelined and dissimilar units, but our analytic performance guarantees do not carry over.)

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 [5, 10]. It is polynomially solvable when the precedence graph is a forest [7] or a generalized series-parallel graph [1, 10]. 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 [3]. In light of the intractable nature of the problem, we adopt the standard approach for dealing with NP-hard problems [5, 12], i.e., the design of 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 tree case is largely of theoretical interest, but serves well to establish our techniques and provide insight into the design of heuristics for the general case. The S-graph to be defined in the next section is the abstraction of the important practical case of the superblock. We then show that in both these cases, using the optimal sequential schedule as a list to drive a list scheduling algorithm for multiple functional units guarantees a performance ratio of 2. For the S-graph case, the above performance guarantee holds only if all the operations have equal execution time; we also show that if the execution times are arbitrary, no list scheduling algorithm can have good performance guarantees.

Finally, we present a scheme which converts list scheduling heuristics for the single exit case (such as critical path scheduling) into a heuristic for the multiple exit case with associated profile information. This general scheme follows from the structural insight in our basic lemma. In particular, we develop and evaluate a heuristic obtained by augmenting critical path scheduling using our scheme. We cannot show tight performance guarantees on this heuristic, but we do present experimental results on a number of sample superblocks and hyperblocks obtained by applying the IMPACT compiler on SPEC benchmark programs. We report that significant savings are possible as compared to the oblivious algorithm which does not take profile information into account.

2 Preliminaries

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 superblock. 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. In a superblock the conditional exits cannot be executed out of order if the correctness of the execution is to be preserved. This leads to the chaining of the exits in the dag representing the computation and this should make clear the connection between superblocks and their abstraction the S-graphs. Unlike superblocks the hyperblocks are predicated and the branches can be reordered and in some cases the predicates can be fully resolved. In these cases the branches can be independent. From the theoretical point of view this means that in the graphs obtained from hyperblocks there could be weights on arbitrary nodes in the graph and the problem is NP-hard even in the single functional unit case with equal latencies and no good approximation algorithms are known. From a practical point of view since the number of branches might be small one could in some cases opt for an exhaustive search for the optimal for the single unit case.

We say that u immediately precedes v, denoted $u \prec v$, if and only if there is an edge from u to v in the graph. A vertex u precedes a vertex v, denoted $u \overset{*}{\prec} v$, 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 $r_i = t_i/w_i$. 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

$r(A) = t(A)/w(A)$. The notion of the rank of a set of vertices is meant to capture the 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.

We use the superscript m to denote the number of functional units. For example, S^m denotes a schedule for m functional units, and F^m denotes the weighted finish time of S^m, i.e., the cost of S^m. An optimal schedule is denoted S^m_{OPT}, and its cost is denoted by F^m_{OPT}. If $m = 1$, we call a corresponding schedule a sequential schedule, and if $m \geq 2$, we call a corresponding schedule an ILP schedule.

3 Sequential Schedules

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, 4, 7]. 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 in the superblock (or hyperblock), is small. >From a theoretical point of view, the best known approximation algorithm [13] for sequential scheduling of weighted DAGS has a performance guarantee of $O(\log^2 n)$.

3.1 The Basic Lemma

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 $u \stackrel{*}{\prec} v$ or $v \stackrel{*}{\prec} u$.

Definition 1 *Given a weighted precedence graph G, we define G^* to be the smallest precedence-closed* proper *subgraph of G of minimum rank, i.e., among all precedence-closed subgraphs of G, G^* is of minimum rank and has the fewest number of vertices.*

We now prove our main lemma.

Lemma 1 *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. We can assume without loss of generality that C_1 is non empty. 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 value 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

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

For the schedule S,

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

Taking their difference gives

$$\begin{aligned} Cost(S) - Cost(S') &= \sum_{i \leq k} w(B_i)t(C^i) - \sum_{i \leq k} w(C_i)t(B^k - B^{i-1}) \\ &\geq \sum_{i \leq k} w(B_i)\alpha w(C^i) - \sum_{i \leq k} w(C_i)\alpha w(B^k - B^{i-1}) \\ &\geq \alpha \sum_{i \leq k} w(B_i)w(C^i) - \alpha \sum_{i \leq k} w(B_i)w(C^i) \geq 0 . \end{aligned}$$

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. ■

3.2 Constructing the Schedule

Lemma 1 essentially reduces the scheduling problem to the problem of finding G^*. Given G^*, 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. Next, we show that finding G^* and hence finding optimal sequential schedules is relatively straightforward if the precedence graph is a tree or an S-graph.

It is clear that precedence-closed subgraphs of an in-tree are forests of in-trees. Suppose G^* is a forest of in-trees. There must be at least one tree in this forest of rank at most $r(G^*)$. But this tree has fewer vertices than G^*, contradicting the choice of G^*. It follows that G^* is exactly one tree; we refer to this tree as T^*. Determining T^* is straightforward.

To obtain a schedule for the case of an out-tree, we convert it into an in-tree by reversing all the edges and then negating the weights of all the operations. Observe that Lemma 1 is valid even if w_i's are negative. We reverse the optimal schedule of the resulting in-tree to obtain a schedule for the out-tree.

Finding G^* when the precedence graph is an S-graph is also simple. 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. It follows that 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 superblock, this amounts to scheduling the basic blocks comprising the superblock in the order defined by the control flow edges linking the branches guarding each basic block.

4 Lower Bounds on ILP Scheduling

We now show two lower bounds on the cost of the optimal ILP schedule. These lower bounds will be useful in bounding the performance of our list scheduling algorithms for multiple functional units that are to follow.

Lemma 2 $F^m_{\text{OPT}} \geq F^1_{\text{OPT}}/m$.

Proof: Given a schedule S^m on m functional units with a cost F^m, we will construct a sequential schedule S^1 of cost at most mF^m as follows. Order the operations according to their finish times in S^m with the operations finishing early coming earlier in the schedule. This ordering is our schedule S^1. If $u \overset{*}{\prec} v$ then $f^m_u \leq s^m_v \leq f^m_v$ which implies that there will be no precedence violations in S^1. We claim that $f^1_i \leq m f^m_i$ for every operation τ_i. To see this, consider the *total work* done by the m functional units by time f^m_i — it is at most $m f^m_i$. Since any operation which is scheduled earlier than τ_i in S^1 must have finish time less than f^m_i in S^m, the finish time of τ_i is at most the total work before f^m_i. This implies the desired result. ∎

Definition 2 *For any vertex v, define the quantity $p_v = \max_{u \overset{*}{\prec} v} t(P_{uv})$. The path P_{uv}, from the vertex u which maximizes this expression to the vertex v, is referred to as the* critical *path to v, and p_v is the execution time of the critical path to v.*

Lemma 3 $F^m_{\text{OPT}} \geq F^\infty_{\text{OPT}} = \sum_i w_i p_i$

Proof: The first inequality is easy to see since increasing the number of functional units can only improve the optimal solution. The equality can be obtained from the following observation: every operation τ_i finishes by the time p_i and, in any schedule, cannot finish before time p_i. ∎

5 ILP Scheduling of In-Trees

In this section we analyze the standard *list scheduling* algorithm which starts with an ordering on the operations (the list), and greedily schedules each successive operation in the list at the earliest possible time. We use the optimal sequential schedule for trees as the list. The main result is that this algorithm performs within a factor of 2 of the optimal.

Lemma 4 *If S^m is the list schedule using a sequential schedule S^1 as the list, then for any operation τ_i, $f^m_i \leq p_i + f^1_i/m$.*

Proof: The proof depends crucially on the following property of in-trees. Every node (except the root) has exactly one successor. The consequence of this is that the completion of an operation frees at most one operation. Since we use list scheduling, the above observation implies that no operation will be delayed by an operation which comes after it in the list.

Consider the schedule formed by the deleting from S^m all operations that come after operation τ_i in the list. Using similar arguments as above, we can show that the resulting schedule will consist of two segments B_1 and B_2 such that all functional units are in use throughout B_1, and at least one functional unit is idle throughout B_2. Since all the jobs in the resulting schedule are earlier in the list than τ_i, it is clear that the length of B_1 is at most f^1_i/m. Also, at no point in B_2 can all functional units be idle, since this would imply that τ_i was delayed by an operation that came after it in the list. It follows that in B_2, every operation that precedes τ_i is scheduled as early as possible, since there are sufficiently many idle functional units. Hence B_2 is at most p_i long. Since the finish time of τ_i is the sum of the lengths of B_1 and B_2, the claim follows. ∎

Theorem 1 *The list scheduling algorithm, using the optimal sequential schedule as the list, is an approximation algorithm with a performance ratio of 2.*

Proof: Let S^1_{OPT} be an optimal sequential schedule with cost F^1_{OPT}. Then by Lemma 4, in the list schedule S^m we create from S^1_{OPT}, we have $f_i^m \leq p_i + f_i^1/m$ for each operation τ_i. We obtain that

$$\begin{aligned} F^m &= \sum_i w_i f_i^m \\ &\leq \sum_i w_i \left(p_i + \frac{f_i^1}{m} \right) \\ &\leq \sum_i w_i p_i + \frac{1}{m} \sum_i w_i f_i^1 \\ &\leq F^\infty_{\text{OPT}} + \frac{1}{m} F^1_{\text{OPT}} \\ &\leq 2F^m_{\text{OPT}}, \end{aligned}$$

where the last inequality follows Lemmas 2 and 3. ■

6 ILP Scheduling of S-graphs

In this section we analyze a *list scheduling* algorithm for S-graphs that greedily schedules each successive operation in a list at the earliest possible time. As in the tree case, the optimal sequential schedule is used as the list. We show that the algorithm has a performance factor of 2, if all the operations have the same execution time. This should be contrasted with the tree result where we allowed arbitrary execution times. We can also show that if the operations can have arbitrary execution times, then no list scheduling algorithm can offer a performance bound independent of m.

Lemma 5 *If S^m is the list schedule using a sequential schedule S^1 as the list, then for any operation τ_i, $f_i^m \leq p_i + f_i^1/m$.*

Proof: Delete from S^m all operations that come after operation τ_i in the list. This can be done since none of these operations delay τ_i, all operations being of equal execution time. Break the resulting schedule into a sequence of maximal segments $B_1, B_2, \ldots, B_k$, where throughout each segment either all functional units are in use, or throughout each segment at least one functional unit is idle. A segment where all the functional units are in use is called a *full segment*, and other segments are called *partial segments*. Now the sum of the lengths of all full segments is at most f_i^1/m. In each partial segment, every operation was scheduled as early as possible when the schedule was constructed, since idle functional units were available and since no operation which has been removed can delay any of the operations before τ_i (here we use the property that all operations have the same latency). Thus, the sum of the lengths of the partial segments is at most p_i. Since the finish time of τ_i is the sum of the lengths of the full and partial segments, the claim follows. ■

We omit the proof of the following theorem as it is similar to the tree case.

Theorem 2 *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.*

We remark that the preceding theorem is a generalization of Graham's result for the makespan minimization problem [6] and the proof technique is also similar. However, unlike Graham's result, our theorem does not generalize to the case of unequal execution times. Using a more sophisticated scheduling algorithm, we can obtain the following weaker theorem. We omit the proof and the algorithm.

Theorem 3 [Unequal Execution Times] *For an S-graph where operations can have arbitrary execution times, there is a scheduling algorithm with a performance ratio of 4 when provided the optimal sequential schedule.*

We note that Theorem 2 and Theorem 3 hold even if there are weights on arbitrary nodes. However, we do not have an efficient algorithm for finding optimum sequential schedules in the general case. For the makespan problem – scheduling in the single exit case – any list scheduling algorithm performs within a factor of 2 of the optimal [6], even if the latencies were arbitrary. Such is not the case for S-graphs, as evidenced by the following theorem.

Theorem 4 *For S-graphs with unequal execution times, no list scheduling algorithm can offer a performance ratio better than m.*

It should be noted that there is no contradiction between the statements of the preceding theorem and that of Theorem 3 since the algorithm used in Theorem 3 is not a list scheduling based algorithm. The preceding theorem is of theoretical interest only since the ratio of the largest to the smallest execution times in the example used in our proof of the above theorem is very large. But it should be pointed out there are examples which show that the ratio can be proportional to the ratio of the execution times of the longest and the shortest instruction.

7 A Scheduling Heuristic

We now turn our attention to constructing a quality heuristic for profile-driven scheduling, based on our theoretical analysis of the prior sections. Our basic lemma characterized the optimal single unit case in terms of the rank function which used the intuition of work. For the practical pipelined multiple unit case when there is only on exit (single basic block) it is well known that critical path scheduling gives near optimal schedules as opposed to naive list scheduling. The intuition behind our heuristic given below is to marry the optimality of the ranking scheme for weighted graphs on a single functional unit, as shown in the basic lemma, with the effectiveness of critical path scheduling for unweighted graphs on multiple functional units. The heuristic is as follows:

Algorithm `Heuristic-Schedule`

1. Profile-list = empty.
2. Find G^*
3. Append the *critical path* 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.

In words, the algorithm computes the rank of each branch in the precedence graph, where the rank is as defined earlier for the basic lemma. As per the basic lemma, the precedence closed subgraph rooted at the branch vertex with the largest rank should be scheduled first. In keeping with this, the algorithm applies

Model	I Units	FP Units	Mem. Units
m111	1	1	1
m211	2	1	1
m221	2	2	1
m212	2	1	2
m222	2	2	2

Table 1: The five heterogeneous machine models.

Opcode	Time
IALU	1 cycle
FALU	4 cycles
LOAD	2 cycles
STORE	1 cycle

Table 2: Opcodes and execution times.

critical path scheduling to the precedence closed subgraph rooted at the branch vertex with the highest rank. It then deletes this subgraph, and repeats with the rest of precedence graph. To better understand the algorithm it is instructive to look at the following extreme cases. Suppose that a certain exit dominates in that its probability is very close to unity. Then the ranking scheme will ensure that the heuristic carries out a critical path scheduling from that exit. On the other hand, if the probabilities of all exits are equal, the heuristic schedules the branches in order. In both cases, the heuristic behaves in a manner that is intuitively appealing. As remarked earlier, finding G^* for superblocks is simple and takes time proportional to the number of vertices in the DAG. Finding $G*$ for hyperblocks with fully resolved predicates is hard, although if the number of exits is small an exhaustive enumeration is feasible.

The above heuristic can be viewed as an instance of a general paradigm to augment any list scheduling heuristic (for the single exit case) into a heuristic for multiple exits with associated branch information. We chose critical path scheduling in Step 3 since it is known to give near-optimal schedules in practice. Our augmentation scheme relies on the basic lemma, although various other optimizations tailored for the particular application can also be added to the basic scheme.

We now study the performance of the heuristic on a number of superblocks and hyperblocks generated by the IMPACT compiler from the SPEC benchmark programs, with the optimizer turned on. Almost all the programs in the study were integer benchmarks. The branches in the hyperblocks were chained just at they were in the superblocks because IMPACT doesn't yet have the fully resolved predicates which will make the branches independent. We report our results separately for them since the hyperblocks differ from the superblocks in many ways. We report our results on 6859 superblocks and 896 hyperblocks, on the 5 different heterogeneous machine models described in Table 1 and also on 3 homogeneous models where all the units are assumed to be identical with 2,4 and 8 units respectively. The models are non-pipelined machines. For all models, the execution times of the various opcodes are as specified in Table 2. The heterogeneous units are marked mijk, where i, j and k are the number of integer, floating point and memory units respectively. The homogeneous models are denoted ui, where i are the number of units in the model.

We used the critical path algorithm as the baseline in our studies, in that we compare the performance of our algorithm against it. Our algorithm degenerates to critical path scheduling when the last exit in a

Model	% blocks imp	Avg imp %	Max imp %
m111	27.4	11.1	85.7
m211	16.0	11.4	80.0
m212	15.3	11.0	80.0
m221	16.0	11.4	80.0
m222	15.2	11.0	80.0
u2	22.2	15.7	91.7
u4	8.7	11.0	83.3
u8	5.0	8.75	60.0

Table 3: Heuristic's performance on superblocks

block has overwhelmingly high probability. As it happens, this is often the case in the superblocks and hyperblocks in our experiments, since the block formation heuristic in the IMPACT compiler is tuned to favor such. An issue that merits investigation is to measure the performance of the algorithm when the block formation heuristic is more tolerant of side exits.

Tables 3 and 4 show the improvement achieved by our heuristic over critical path scheduling. The columns show the percentage of blocks that enjoy improvement, the average improvement over these blocks, and the maximum improvement over these blocks. More details of the distribution of the improvement can be seen in Figures 2 through 5. The horizontal axes in these figures represent the blocks in the experiment sorted according to the improvement achieved. It is clear that as the number of units increase, the improvement achieved by the algorithm falls. This is to be expected, since as the number of available units goes to infinity, all the schedules approach the optimal schedule. Also the plots for the models with heterogeneous units show that much of the parallelism is in the integer operations and little gains are to be obtained by increasing the number of memory and floating point units.

As we mentioned earlier, our heuristic degenerates to the critical path if the probability of the last exit dominates. On measure of the relative importance of the exits is the critical path ratio, the ratio of the expected critical path length of all the exits over the critical path length of the last exit. If the critical path ratio is smaller than unity, then side exits are important. If the critical path ratio is close to unity, then bottom exits dominate. Studying the improvement achieved by the algorithm for various values of this ratio will give us insight into its behavior. To this end we examine Figure 5, which shows two bar graphs. The boxes with with full lines show the percentage of the total superblocks with the ratio in that range. The boxes with dotted lines show the percentage of blocks within each range that enjoyed improvement with our heuristic, for the homogeneous two unit model. For small values of the critical path ratio, the algorithm shows significant improvement over critical path scheduling. When the ratio is close to unity, little improvement is seen, since the heuristic degenerates to critical path scheduling. Of note is that almost 60% of the superblocks have a critical path ratio greater than 0.9. This could be a result of the block formation heuristic, prompting us to consider block formation heuristics that are more tolerant of side exits.

The above heuristic makes use of the concept of *work* required by a set of instructions, i.e., the sum of the latencies of the instructions. We used the work of the precedence closed subgraph of an instruction as the measure of the finish time of the instruction in the idealized single unit case. In the more general pipelined case a more sophisticated estimated of the above measure can be used.

Model	% blocks imp	Avg imp %	Max imp %
m111	14.0	7.4	54.0
m211	7.6	6.8	45.3
m212	6.6	5.3	45.7
m221	7.6	6.8	45.3
m222	6.6	5.3	45.7
u2	10.7	9.5	56.5
u4	4.2	5.7	35.3
u8	2.7	3.9	22.0

Table 4: Heuristic's performance on hyperblocks.

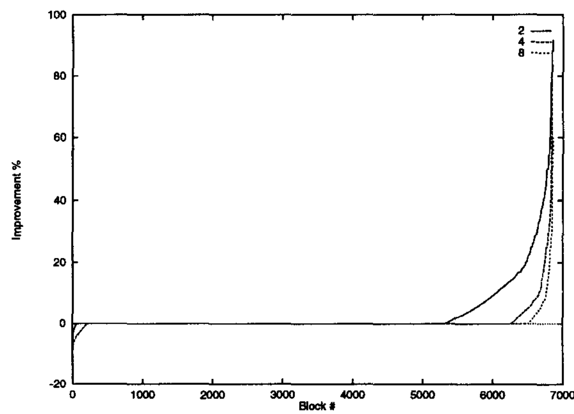

Figure 1: Performance of the heuristic on superblocks: uniform units.

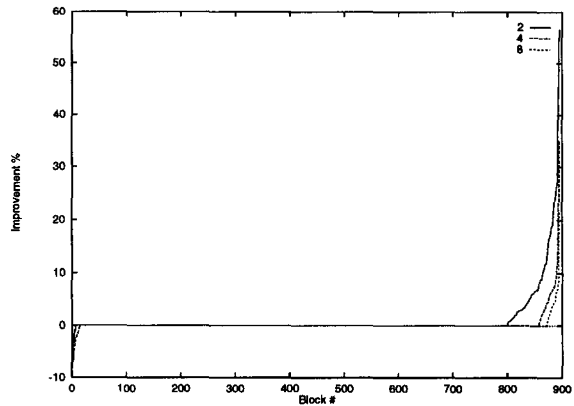

Figure 2: Performance of the heuristic on hyperblocks: uniform units.

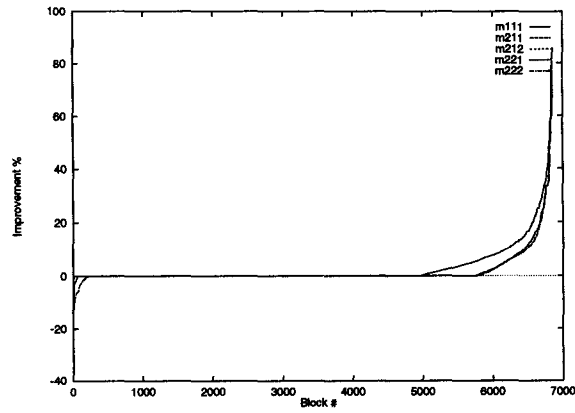

Figure 3: Performance of the heuristic on superblocks: heterogeneous units.

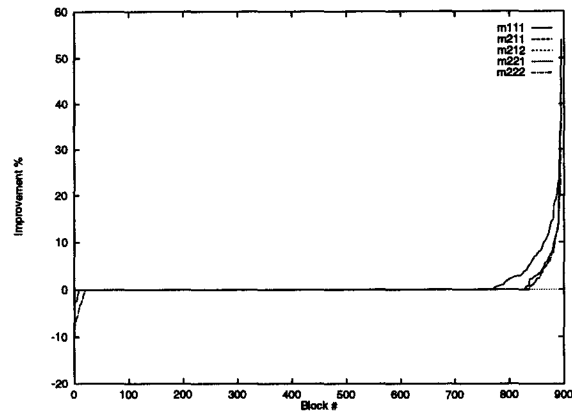

Figure 4: Performance of the heuristic on hyperblocks: heterogeneous units.

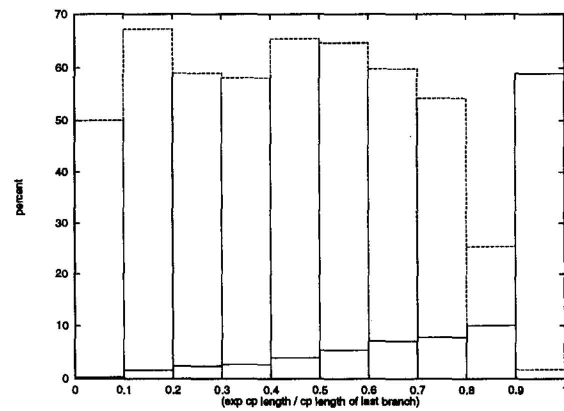

Figure 5: Improvement percent for superblocks plotted against the ratio for the u2 machine.

8 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. As applications of this lemma we obtained optimal sequential schedules for superblocks and trees, and show how these can be used as lists for provably good list scheduling on multiple functional units, provably good for non-pipelined units of equal execution time. In fact, this lemma leads to a generic scheme for converting any list scheduling heuristic for basic blocks into a heuristic for superblocks/hyperblocks with associated profile information. For the general case of pipelined, heterogeneous units with unequal latencies, we presented a heuristic obtained by combining the scheme suggested by our basic lemma with critical path scheduling. Experiments showed that our heuristic offers substantial performance improvement over critical path scheduling on a range of benchmark inputs and machine models. In conclusion, we believe that our results form a good starting point for practical heuristics for the problem of profile-driven scheduling of precedence graphs.

9 Acknowledgements

We thank V. Kathail and M. Schlansker for several helpful discussions.

References

[1] D. Adolphson. Single machine job sequencing with precedence constraints. *SIAM Journal 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] 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).

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

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman, San Francisco (1979).

[6] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416-429 (1969).

[7] W.A. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal of Applied Mathematics*, 23:189–202 (1972).

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

[9] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringamann, R.G. Outllette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7:229–248 (1993).

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

[11] S.A. Mahlke et al. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*, 45–54 (1992)

[12] R. Motwani. *Approximation Algorithms (Volume I).* Technical Report No. STAN-CS-92-1435, Department of Computer Science, Stanford University (1992).

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