Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle

DAVID BERNSTEIN
IBM T. J. Watson Research Center
and
IZIDOR GERTNER
Technion-Israel Institute of Technology

Consider a pipelined machine that can issue instructions every machine cycle. Sometimes, an instruction that uses the result of the instruction preceding it in a pipe must be delayed to ensure that a program computes a right value. We assume that issuing of such instructions is delayed by at most *one* machine cycle. For such a machine model, given an unbounded number of machine registers and memory locations, an algorithm to find a shortest schedule of the given expression is presented and analyzed. The proposed algorithm is a modification of Coffman-Graham's algorithm [7], which provides an optimal solution to the problem of scheduling tasks on two parallel processors.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Pipelined processors

1. INTRODUCTION

In the past few years more and more attention has been paid to pipelined computer architectures, many of which were designed and implemented [13, 16]. We consider the constraints imposed by a pipelined processor on the computation of a straight-line program.

Computing two successive instructions on a pipelined processor may result in a situation such that the second instruction must be delayed until the value of its operand computed by the first instruction is ready. Some of the existing pipelined processors identify this situation in hardware; on the others a NOP

This work was done while D. Bernstein was a graduate student at the Technion-Israel Institute of Technology.

Authors' current addresses: D. Bernstein, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; I. Gertner, Center for Large-Scale Computation, The Graduate School, CUNY, 25 W. 43 St., Suite 400, NYC, NY 10036

(NOP stands for No OPeration) must be explicitly coded between two such instructions. In both cases the execution time of a program may increase. The problem is: Find an order of evaluation of the expression that results in a minimal completion time program under pipelined constraints.

A set of expressions included in a straight-line program can be conveniently represented by a directed acyclic graph (dag) [1, 2]. We assume that such dags are built in the compiler after the instruction selection phase was completed. The problem of scheduling dags with pipelined constraints was previously considered in several papers [4, 11, 12]. Let us discuss the differences among the approaches undertaken in these works and our paper.

Our main assumption on the nature of a pipelined architecture at hand is, as it was in [4] and [12], that interlocks (delays) exist only among instructions that are connected with an edge in a dag. In [12] it was shown that, if the maximal delay *d* of the instructions due to pipelined constraints is unbounded, the problem of finding an optimal order of evaluation is NP-complete. In [11] an even harder problem was considered, namely, the case where interlocks may exist among data-unrelated instructions. The algorithm proposed in [11] takes care of such constraints in a heuristic way.

The second issue of interest is whether there exist additional constraints on the machine resources. Assuming a finite number of machine registers in a machine makes the problem of finding an optimal order of evaluation intractable even on sequential (nonpipelined) machines ([2, 5]). Due to recent advances in microelectronics technology, our assumption that there is a large number of machine registers is quite reasonable. Our view of the problem is that scheduling algorithms must be incorporated in compilers for pipelined machines before register allocation is done. Subsequently, global register allocation can be performed using graph-coloring algorithms [17]. We understand that in this scheme the scheduler will likely lengthen the lifetimes of variables, producing a sequence of machine instructions that requires more registers to be allocated. This, in turn, might require more spill code to be produced during the register allocation stage. However, since the future computers will have many machine registers, we believe that these limitations are not serious.

As opposed to this view, the scheduling problem was treated in [11] and [12] after the register allocation was done, in a post-pass code optimization phase. (We are aware that in [12] it was necessary since their machine had no hardware interlocks.) In this case, new constraints which arise as a result of the unrelated usage of the same register in different portions of the code limit the possible reordering of instructions. To take care of these additional register constraints, a specialized dag representation was used in a heuristic algorithm of [11], while in [12] a standard dag representation was used, resulting in an algorithm of higher time complexity compared to that of [11].

Finally, our last assumption is that instructions are delayed by at most *one* machine cycle. This assumption makes our model suitable to the recently developed RISC-architecture processors [19]. As appears in [14], this assumption holds for RISC I, RISC II ([15]), and for 801 ([20]). Also, it is true for the HP Precision Architecture [11]. However, in the MIPS machine, as reported in [12], sometimes instructions are delayed by two machine cycles.

For a machine model in hand, we present an algorithm to find an optimal order of evaluation of the given expression. The proposed algorithm is a modification of Coffman-Graham's algorithm, which finds a shortest schedule for a set of equal execution time tasks on two-processor systems ([7]). In [4], the original algorithm of Coffman and Graham was used to schedule optimally expressions under the assumption that all the delays are exactly one time cycle (i.e., an interlock exists between any two instructions adjacent in the dag). In a complementary effort [3], we analyze the behavior of Coffman-Graham's algorithm in the case of delays greater than one. We show that the ratio of the length of the schedule produced by the algorithm over the length of an optimal schedule is at most $2 - 2/(d + 1)$, where d is the maximal allowed delay.

The complexity of our algorithm is identical to that of Coffman-Graham's algorithm, which was initially proved to be $O(n^2)$ [7]. Then, Sethi [21] showed that this algorithm (and therefore our algorithm) can be implemented in $O(n\alpha(n) + e)$ where e is the number of edges in a dag, and $\alpha(n)$ is a slowly growing function (a functional inverse of the Ackermann function). Recently, the complexity of Coffman-Graham's algorithm was reduced by Gabow to $O(n + e)$ using the algorithm from [8] and the data structure from [9]. The advantage of Gabow's implementation is that it does not require the dags to be free of transitive edges, as previous works (and our algorithm) assume. Unfortunately, Gabow's algorithm is not suitable for pipelined machines since it produces directly a schedule for a two-processor system rather than a priority list of tasks that can be converted into a schedule for different types of machines.

To this end, our result supports the observation in [12] that the problem of finding an optimal order of evaluation for a pipelined processor is similar to that of finding an optimal schedule for a number of parallel processors. As a consequence of this, allowing $d \geq 2$ is a hard problem since the problem of optimal scheduling for three (or more) parallel processors has been open for quite a long time [10]. As mentioned above, approximation algorithms for this case can be found in [3].

The rest of the paper is organized as follows. In the next section we start with some preliminary definitions. Then in Section 3 the algorithm is presented, followed by the optimality proof (Section 4). We conclude with a discussion of directions for further research.

2. PRELIMINARIES

2.1 The Machine Model

Our machine has an unbounded number of general purpose registers $r_1, r_2, \ldots$ and memory cells $mem_1, mem_2, \ldots$. It supports the following operations:

(1) Load operation: $(mem_i) \rightarrow (r_j)$

(2) Store operation: $(r_i) \rightarrow (mem_j)$

(3) Arithmetic operations: $A((r_{i_1}), \ldots, (r_{i_k})) \rightarrow (r_j)$, where $k \geq 1$.

The execution times of all three types of operations are equal to one *time unit*. The major feature of the machine is its pipelined structure. Usually, the machine

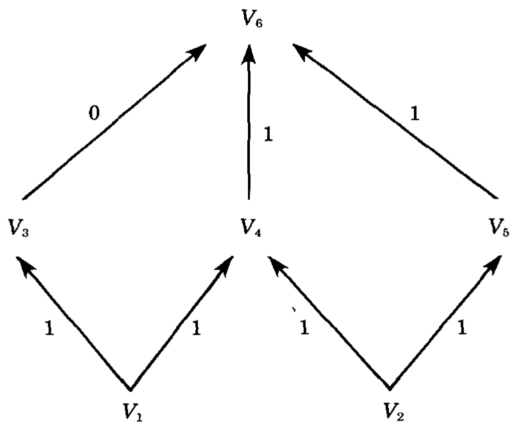

Fig. 1. An example of a dag.

issues instructions every time unit. Due to data dependencies among the instructions in a program, the issue of some of the instructions may need to be delayed by an integral number of time units. Given two instructions I_1 and I_2, where I_2 uses the result of I_1, we denote by $D(I_1, I_2)$ the number of time units by which I_2 must be delayed when it follows after I_1. Also, let $d = \max(D(I_1, I_2))$ for all I_1 and I_2 over the instruction set of the machine. In the sequel, we assume that $d = 1$ (for a discussion on the case $d \geq 2$, see [3]).

2.2 Expression Dags and Their Evaluation

Following [1] and [2], computations are represented by directed acyclic graphs (dags). Each vertex of a dag corresponds to a machine instruction whose operands are computed at the children of this vertex in a dag. An example of a dag is given in Figure 1.

On sequential machines, a dag $G = (V, E)$ defines a partial order of the computation of the vertices of V. For every two vertices, v and u, of G such that $(v, u) \in E$, u can be initiated only after v has been completed. However, for pipelined machines, the schedules of G must obey additional restrictions. In the sequel, for every $(v, u) \in E$, we denote the delay times by $D((v, u))$, which is equal to $D(I_1, I_2)$, where I_1 and I_2 are the machine instructions that compute v and u, respectively. Notice that the appropriate values of $D((v, u))$ may vary from architecture to architecture. In the dag of Figure 1, the integers near the edges denote the corresponding delay times.

A *legal* schedule S of a dag G is a one-to-one mapping of V into the set N of the positive integers, such that for every two vertices, v and u, $(v, u) \in E$, $S(u) - S(v) > D((v, u))$. The range of S is interpreted as *time slots* (of unit length each) at which instructions are performed. A time slot of S, at which no instruction can be executed due to pipe limitations, is called a NOP. Since $d = 1$, in the rest of the paper we can assume that G has no transitive (redundant) edges.

Two legal schedules for the dag of Figure 1 appear in Figure 2, where i in column j means that v_i is executed at time slot j. Notice that time slot 6 of S^1 is

N	1	2	3	4	5	6	7
S^1	1	2	3	4	5		6
S^2	2	1	5	4	3	6	

Fig. 2. Two legal schedules for the dag of Figure 1.

a NOP since $D(v_5, v_6) = 1$. The *completion time* $c(S)$ of a schedule S is defined by $\max_{v \in V} S(v)$. For example, in Figure 2, $c(S^1) = 7$ and $c(S^2) = 6$. Throughout this work we will be interested in minimizing the completion time, which is equivalent to minimizing the number of NOPs. An *optimal* schedule S of G is a legal schedule for which $c(S)$ is the smallest. S^2 of Figure 2 is an optimal schedule of the dag of Figure 1 since it has no NOPs.

2.3 List Schedules

Let $G = (V, E)$ be a computation dag. If $(v, u) \in E$ we say that u is an *immediate successor* of v, and v is an *immediate predecessor* of u. Also, if there exists a directed path in G from v to u, we say that u is a *successor* of v, and v is a *predecessor* of u. In the sequel, we denote by $index(S, v)$ the time slot of a schedule S at which a vertex v is executed, and by S_i a vertex which is scheduled in S at time slot i. Given a schedule S of G, u is *ready at time slot k*, if for each of its immediate predecessors v, $index(S, v) \leq k - 1 - D(v, u)$.

Now we consider an important class of schedules, called *list schedules* [6]. Informally, given a *priority list* L of the vertices of G, the list schedule S which *corresponds* to L can be constructed by the following procedure:

(1) Iteratively schedule the elements of S starting at time slot 1 such that during the ith step, L is scanned from left to right, and the first ready vertex not yet scheduled is chosen to be executed at time slot i.

(2) If no such job is found, a NOP is inserted into S at time slot i.

Consider a class of optimal schedules for G. Since all the machine instructions have unitary execution time, there is no reason in optimal schedules to leave the machine idle if a ready vertex exists. Therefore, for our problem, an optimal schedule can always be found among the class of list schedules. In the sequel we consider list schedules only. The obvious question is how to obtain the right priority list L.

3. THE LABELING ALGORITHM

Let $G = (V, E)$ be a dag and let v be a vertex of G. We denote by $P(v)$ the set of immediate successors of v in G, that is, $u \in P(v)$ if and only if $(v, u) \in E$. Also, let $R(v)$ be the subset of $P(v)$ such that $u \in R(v)$ if and only if $D((v, u)) = 1$.

First, we present the original algorithm of Coffman and Graham [7], then we show how to modify it to produce an optimal priority list for the pipelined machine at hand. The algorithm uses the lexicographic order among decreasing sequences of positive integers. For example, by the lexicographic order, $\{7, 3, 2\} < \{7, 4\}$ and $\{7, 3, 2\} < \{7, 3, 2, 1\}$. Conveniently, we denote an empty sequence by $\{\ \}$, and every nonempty sequence is greater than an empty sequence.

Let $|V| = n$. The algorithm assigns to each vertex v of G a *label* $l(v) \in \{1, 2, \ldots, n\}$. The label $l(v)$ is defined as follows:

(a) An arbitrary vertex v with $P(v)$ empty is chosen, and $l(v) = 1$.

(b) Suppose the labels $1, 2, \ldots, i - 1$ have already been assigned by the algorithm. For each vertex v, for which the map l has been computed for all elements of $P(v)$, let $M(v)$ denote the decreasing sequence of positive integers formed by ordering $\{l(u) \mid u \in P(v)\}$. Choose a vertex v' for which $M(v')$ is minimal (break tie at will) and set $l(v') = i$.

(c) Repeat the assignment in (b) until all the vertices are labeled.

The modified labeling algorithm proceeds in the same way as above except that in Step (b), $M(v)$ is formed by ordering the set $\{l(u) \mid u \in R(v)\}$. Finally the optimal priority list L is determined by ordering vertices with the higher labels first. An optimal schedule S, which corresponds to L, can be determined by applying the list-scheduling process described in Section 2 to L. Notice that L only defines the relative priority of the vertices of G, and there might be a situation in which $l(v) > l(u)$ and $index(S, u) < index(S, v)$.

Consider how the algorithm labels the vertices of the dag of Figure 1. First, since $P(v_6)$ is empty, it assigns $l(v_6) = 1$. Then, it computes $M(v_3) = \{\,\}$ and $M(v_4) = M(v_5) = \{1\}$. Since $M(v_3)$ is minimal, $l(v_3) = 2$. Then, the algorithm assigns label 3 to either v_4 or v_5. Assume that $l(v_5) = 3$, then $l(v_4) = 4$. Now we can construct $M(v_1) = \{4, 2\}$ and $M(v_2) = \{4, 3\}$. Since $M(v_1) < M(v_2)$, $l(v_1) = 5$ and $l(v_2) = 6$. Thus, $L = \{2, 1, 4, 5, 3, 6\}$. The optimal schedule S, which corresponds to L, is S^2 of Figure 2. Notice that in S^2, v_5 was scheduled before v_4 even though $l(v_4) > l(v_5)$. This happened since, when v_5 was scheduled, v_4 was not ready yet. This demonstrates further, as was mentioned before, that L defines only the relative priority of the vertices in a dag, and the final order of the vertices is determined only in the list-scheduling process.

Notice that if we had applied the original algorithm of Coffman and Graham to the dag of Figure 1, we could have $l(v_3) = 4$, $l(v_4) = 3$ and $l(v_5) = 2$. In this case, $M(v_1) = \{4, 3\}$, while $M(v_2) = \{3, 2\}$. Thus, $l(v_2) = 5$ and $l(v_1) = 6$, which results in $L' = \{1, 2, 3, 4, 5, 6\}$. The schedule S, which corresponds to L', is S^1 of Figure 2, which is suboptimal.

4. THE OPTIMALITY PROOF

In this section we prove that the labeling algorithm of Section 3 is optimal. Before we proceed with the main theorem we need the following result.

LEMMA 1. *Let v and u be vertices of G such that neither of them is a successor of the other and $l(v) > l(u)$. If there exists a vertex x in $R(u) - R(v)$, then there exists a vertex y in $P(v) - R(u)$ such that $l(y) > l(x)$.*

PROOF. Consider the moment t at which u is labeled by the algorithm.

Case 1. At time t there exists $y \in P(v)$ which has not yet been labeled by the algorithm.

Since u is labeled at time t, $y \notin P(u)$, and therefore, $y \notin R(u)$. Thus, $y \in P(v) - R(u)$. Since y is labeled after u, $l(y) > l(u)$. Notice that for every successor w of u, $l(u) > l(w)$. Therefore, since $x \in R(u)$, $l(y) > l(u) > l(x)$, and we are done.

Case 2. At time t all $y \in P(v)$ have already been labeled by the algorithm.

Since the algorithm decided to label u before v, $M(u) \leq M(v)$. Notice that since $x \in R(u) - R(v)$, $l(x)$ appears in $M(u)$ and does not appear in $M(v)$. In this case, $M(v) \neq M(u)$, and therefore, $M(v) > M(u)$. Therefore, there must exist y such that $l(y)$ appears in $M(v)$ and does not appear in $M(u)$ with $l(y) > l(x)$. Since $l(y)$ appears in $M(v)$, $y \in R(v)$, and therefore $y \in P(v)$. Also, since $l(y)$ does not appear in $M(u)$, $y \notin R(u)$. Therefore, $y \in P(v) - R(u)$, and the lemma is proved. $\square$

To demonstrate how Lemma 1 can be applied, consider the dag of Figure 1. Recall that $l(v_1) = 5$ and $l(v_2) = 6$. Also, $R(v_1) = \{v_3, v_4\}$ and $P(v_2) = R(v_2) = \{v_4, v_5\}$. Notice that $v_3 \in R(v_1) - R(v_2)$. Now apply Lemma 1 substituting v_2, v_1, and v_3 instead of v, u, and x, respectively. Thus, by Lemma 1, there exists a vertex y in $P(v_2) - R(v_1)$ whose label is greater than $l(v_3)$. Indeed, $v_5 \in P(v_2) - R(v_1)$ and $l(v_5) > l(v_3)$.

In the sequel, we are going to prove the optimality of the algorithm of Section 3 by induction on the size of G. Let L be the priority list computed by the labeling algorithm for G. To set up an induction proof, we will show a somewhat stronger statement: given G and a set NR (for Not Ready) of the leaves (vertices with no immediate predecessors) of G which are not ready at time slot 1 and become ready only at time slot 2, the schedule S which corresponds to L is optimal. We denote this *modified scheduling problem* by a pair (G, NR). Notice that, since the list-scheduling process is applied on L, the vertex with a highest label among all the vertices which do not belong to NR will be scheduled first at S. By convention, if NR includes all the leaves of G, we assume that the first time slot of S is a NOP.

The relationship between the original and the modified scheduling problems is straightforward. Initially, we are given a dag G and NR is empty. After vertices $v_1, \ldots, v_s$ have been scheduled, we are left with $G' = G - \{v_1, \ldots, v_s\}$ and $NR' = \{x \mid x \in R(v_s)\}$. Notice that the addition of the set NR to the scheduling problem did not change its nature. Indeed, after v_s has been scheduled in S, none of the vertices which belong to $R(v_s)$ can be scheduled immediately after v_s, and they are released only one time unit later. Thus, the schedules that result when the algorithm of Section 3 is applied to the original and to the modified scheduling problems are identical (assuming that initially NR is empty).

Let us demonstrate the concept of the modified scheduling problem on the dag of Figure 1. Initially, NR is empty, and v_2 is scheduled first since it has the highest label. Then, we are left with $G' = G - \{v_2\}$, and $NR' = \{v_4, v_5\}$. Since v_1 has the highest label in G' and is ready, it is scheduled second. Now, $G'' = G - \{v_1, v_2\}$, and $NR'' = \{v_3, v_4\}$. At this moment, v_4 has the highest label in G'', but since it is not ready (it appears in NR''), v_5 is scheduled next. This process is continued until we get the same schedule as before.

THEOREM 2. *Let a modified scheduling problem be defined by (G, NR) and L be the priority list produced by the labeling algorithm for G. The schedule S which corresponds to L is optimal.*

PROOF. By induction on the number of vertices n of G.

Basis. If $n = 1$ or $n = 2$, the theorem trivially holds.

Induction assumption. Assume that for all dags with at most $n - 1$ vertices the theorem holds.

Induction step. Let $| V | = n$. Assume, by contradiction, that there exists a schedule $SOPT$ of G such that $c(SOPT) < c(S)$. Let $S_1 = v$.

Claim 1. $SOPT_1 \neq v$.

Proof of Claim 1. Assume, by contradiction, that $SOPT_1 = v$. Let $G' = G - \{v\}$, and $NR' = R(v)$. Since G' has only $n - 1$ vertices, we can apply the induction hypothesis to (G', NR') to get a schedule S' such that $c(S') \leq c(SOPT) - 1$. Notice that $S'_1 \notin R(v)$. Therefore, $c(S) = c(S') + 1 \leq c(SOPT)$, a contradiction. $\square$

By Claim 1, we assume that $SOPT_1 \neq v$, and let $SOPT_1 = u$. Notice that both v and u are leaves of G. Also, since v has been chosen by the algorithm to be scheduled while u was ready, we conclude that $l(v) > l(u)$. Let $index(SOPT, v) = k$ ($k \geq 2$), and assume that $SOPT$ is an optimal schedule such that k is minimal.

Claim 2. If $k > 2$ then there exists an optimal schedule $SOPT$ of G such that for all $2 \leq i \leq k - 1$, $l(SOPT_i) > l(v)$.

Proof of Claim 2. Assume, by contradiction, that there exists $2 \leq i \leq k - 1$ such that $l(SOPT_i) < l(v)$. Let $\overline{G} = G - \{SOPT_1, \ldots, SOPT_{i-1}\}$ and $\overline{NR} = R(SOPT_{i-1})$. Let w be the vertex of $\overline{G}$ which is ready at time slot 1 and has a maximal label. Since v is ready at this slot, $l(w) \geq l(v)$ (notice that w may be v itself). Since $\overline{G}$ has at most $n - 1$ vertices, we can apply the induction hypothesis to $\overline{G}$ to get that w can be scheduled at time slot i of $SOPT$. If $w = v$, it contradicts the minimality of k. Thus, $w \neq v$ and $l(w) > l(v)$. $\square$

Thus, by Claim 2, if $k > 2$ we can assume that for all $2 \leq i \leq k - 1$, $l(SOPT_i) > l(v) > l(u)$. Therefore, by way of computing the labels in the algorithm, none of $SOPT_i$, $2 \leq i \leq k - 1$ is a successor of u. Also, since v is a leaf of G, either in case of $k = 2$ or in case of $k > 2$ nothing prevents the algorithm from setting $SOPT_1 = v$ and $SOPT_k = u$, except that $SOPT_{k+1}$ would probably have to be delayed. (We address this issue subsequently.) Let the resultant schedule be denoted by $SOPT'$. In the sequel, we will prove that there exists a reordering on the part of $SOPT'$ that appears after time slot k such that the resultant schedule $SOPT'$ is legal and $c(SOPT') \leq c(SOPT)$, contradicting Claim 1.

Case 1. Either $SOPT_{k+1} = \text{NOP}$ or $SOPT_{k+1} \notin R(u)$.
Since there is no delay between u and $SOPT_{k+1}$, $SOPT'$ is a legal schedule, $c(SOPT') = c(SOPT)$, and we are done.

Case 2. $SOPT_{k+1} \in R(u)$.
Let $\hat{G} = G - \{SOPT_1, \ldots, SOPT_k\}$ and $\widehat{NR} = R(v) \cap R(u)$. Notice that $\hat{G}$ has less than n vertices. Also notice that $SOPT_{k+1}$ is a leaf of $\hat{G}$, and since

$v = SOPT_k$, $SOPT_{k+1} \notin R(v)$, which leads to $SOPT_{k+1} \notin \widehat{NR}$. Let x be a vertex in $\hat{G}$ with a maximal label that is ready at time slot 1 of $(\hat{G}, \widehat{NR})$. Notice that there is at least one such vertex since $SOPT_{k+1}$ is ready at slot 1 of $(\hat{G}, \widehat{NR})$.

Claim 3. $x \notin R(u)$.

Proof of Claim 3. Assume, by contradiction, that $x \in R(u)$. Since x is ready at time slot 1 of $(\hat{G}, \widehat{NR})$, $x \notin \widehat{NR}$. Thus, $x \notin R(v) \cap R(u)$. Since $x \in R(u)$, this means that $x \notin R(v)$, that is, $x \in R(u) - R(v)$. Notice that neither v nor u is a successor of the other (since $v = S_1$ and $u = SOPT_1$). Thus, since $l(v) > l(u)$, we can apply Lemma 1 to get that there exists $y \in P(v) - R(u)$ such that $l(y) > l(x)$. Notice that, since $y \notin R(u)$, $y \notin \widehat{NR}$. Thus, if y is ready at time slot 1 of $(\hat{G}, \widehat{NR})$, this will be the contradiction to the assumption that x is such a vertex with a maximal label. The only reason that y is not ready at time slot 1 of $(\hat{G}, \widehat{NR})$ is that y is not a leaf of $\hat{G}$. Let z be a predecessor of y in $\hat{G}$ with a maximal label. Thus, $l(z) > l(y) > l(x)$. Notice that z is a leaf of $\hat{G}$. Since $y \in P(v)$, z is not a successor of v; otherwise an edge (v, y) will be transitive in $\hat{G}$, and recall that we have assumed that dags have no transitive edges (see Section 2). Thus, $z \notin R(v)$, which leads to $z \notin \widehat{NR}$. Therefore z is ready at time slot 1 of $(\hat{G}, \widehat{NR})$, and this is the contradiction to the assumption that x is such a vertex with a maximal label. □

Now let us apply the induction hypothesis to $(\hat{G}, \widehat{NR})$. We are going to substitute in $SOPT'$ the part of the schedule that computes $\hat{G}$ by the schedule produced by the algorithm for $(\hat{G}, \widehat{NR})$. Notice that in $SOPT$, none of the vertices which belong to $R(v)$ can be scheduled at time slot $k + 1$ since $SOPT_k = v$. Thus, by setting $\widehat{NR} = R(v) \cap R(u)$, we do not restrict the class of optimal schedules for $\hat{G}$ as compared to those that can come out in $SOPT$. By the induction hypothesis, there exists a schedule $\hat{S}$ of $(\hat{G}, \widehat{NR})$ such that $\hat{S}_1 = x$ and $c(\hat{S}) \le c(SOPT) - k$. Consider again the schedule $SOPT'$, which is defined as follows:

(1) $SOPT'_1 = v$.

(2) For $2 \le i \le k - 1$, $SOPT'_i = SOPT_i$.

(3) $SOPT'_k = u$.

(4) For $k + 1 \le i \le k + |\hat{S}|$, $SOPT'_i = \hat{S}_{i-k}$.

Notice that $SOPT'_{k+1} = x$. Since by Claim 3, $x \notin R(u)$, $SOPT'$ is a legal schedule. Thus $c(SOPT') = k + c(\hat{S}) \le c(SOPT)$. By Claim 1, this contradicts the assumption that $c(SOPT) < c(S)$. □

5. CONCLUSIONS

In this paper we presented an optimal scheduling algorithm for the case in which the maximal delay d of instructions due to pipelined constraints is one time unit. As discussed in the introduction, the existence of a polynomial time optimal algorithm for $d \ge 2$ is an open question, but the problem seems to be intractable. One of the possible directions for future research is to develop efficient approximation algorithms (for example, see [3]). The other is to develop algorithms for restricted cases of the general problem. One of the classical restrictions in the area of code generation is the assumption that the given expression has no common subexpressions, i.e. it can be represented by a tree rather than by a dag (consider, for example [1]). For this case, only when all the delays are identical, a simple optimal algorithm was proposed in [18]. Another assumption, that may be suitable to certain RISC architectures, is that only instructions that use the result of a preceding load instruction must be delayed. In both cases the existence of polynomial time optimal algorithms is an open problem.

ACKNOWLEDGMENT

We would like to thank the referees whose remarks improved the presentation of the paper.

REFERENCES

1. AHO, A. V., AND JOHNSON, S. C. Optimal code generation for expression trees. *J. ACM 23*, 3 (July 1976), 488–501.
2. AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. Code generation for expressions with common subexpressions. *J. ACM 24*, 1 (Jan. 1977), 146–160.
3. BERNSTEIN, D., RODEH, M., AND GERTNER, I. Approximation algorithms for scheduling arithmetic expressions on pipelined machines. To be published in the *J. Alg.* (Mar. 1989).
4. BRUNO, J., JONES, J. W., AND SO, K. Deterministic scheduling with pipelined processors. *IEEE Trans. Comput. C-29*, 4 (Apr. 1980), 308–316.
5. BRUNO, J. L., AND SETHI, R. Code generation for a one-register machine. *J. ACM 23*, 3 (July 1976), 502–510.
6. COFFMAN, E. G. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.
7. COFFMAN, E. G., JR., AND GRAHAM, R. L. Optimal scheduling for two-processor systems. *Acta Inf. 1* (1972), 200–213.
8. GABOW, H. N. An almost-linear algorithm for two-processor scheduling. *J. ACM 29*, 3 (July 1982), 766–780.
9. GABOW, H. N., AND TARJAN, R. E. A linear time algorithm for a special case of disjoint set union. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Apr. 1983), ACM, New York, 1983, 246–251.
10. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability, A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, 1979.
11. GIBBONS, P. B., AND MUCHNICK, S. S. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Symposium on Compiler Construction* (Palo Alto, Calif., June 1986). ACM, New York, 1986, 11–16.
12. HENNESSY, J., AND GROSS, T. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 422–448.
13. HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.
14. KATEVENIS, M. G. H. *Reduced Instruction Set Computer Architectures for VLSI.* MIT Press, Cambridge, 1985.
15. KATEVENIS, M. G. H., et al. The RISC II Micro-architecture. *J. VLSI Comput. Syst. 1*, 2 (Jan. 1985), 138–152.
16. KOGGE, P. M. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.
17. LARUS, J. R., AND HILFINGER, P. N. Register allocation in the SPUR Lisp Compiler. In *Proceedings of the ACM Symposium on Compiler Construction* (Palo Alto, Calif., June 1986), ACM, New York, 1986, 255–263.
18. LI, H. F. Scheduling trees in parallel/pipelined processing environments. *IEEE Trans. Comput. C-26*, 11 (Nov. 1977), 1101–1112.
19. PATTERSON, D. A. Reduced instruction set computers. *Commun. ACM 28*, 1 (Jan. 1985), 8–21.
20. RADIN, G. The 801 minicomputer. *IBM J. Res. Dev.* 27, 3 (May 1983), 237–246.
21. SETHI, R. Scheduling graphs on two processors. *SIAM J. Comput. 5*, 1 (Mar. 1976), 73–82.

Received June 1986; revised June 1987 and September 1988; accepted September 1988