Scheduling Time-Critical Instructions on RISC Machines.

Conference Paper *in* ACM Transactions on Programming Languages and Systems · January 1990

DOI: 10.1145/155183.155190 · Source: DBLP

2 authors:

Krishna Palem
Rice University
147 PUBLICATIONS 3,324 CITATIONS
SEE PROFILE

Barbara Simons
IBM
80 PUBLICATIONS 1,902 CITATIONS
SEE PROFILE

Scheduling Time-Critical Instructions on RISC Machines

KRISHNA V. PALEM
IBM T. J. Watson Research Center
and
BARBARA B. SIMONS
IBM Santa Teresa Laboratory

We present a polynomial time algorithm for constructing a minimum completion time schedule of instructions from a basic block on RISC machines such as the Sun SPARC, the IBM 801, the Berkeley RISC machine, and the HP Precision Architecture. Our algorithm can be used as a heuristic for RISC processors with longer pipelines, for which there is no known optimal algorithm. Our algorithm can also handle time-critical instructions, which are instructions that have to be completed by a specific time. Time-critical instructions occur in some real-time computations, and can also be used to make shared resources such as registers quickly available for reuse. We also prove that in the absence of time-critical constraints, a greedy scheduling algorithm always produces a schedule for a target machine with multiple identical pipelines that has a length less than twice that of an optimal schedule. The behavior of the heuristic is of interest because, as we show, the instruction scheduling problem becomes NP-hard for arbitrary length pipelines, even when the basic block of code being input consists of only several independent streams of straightline code, and there are no time-critical constraints. Finally, we prove that the problem becomes NP-hard even for small pipelines, no time-critical constraints, and input of several independent streams of straightline code if either there is only a single register or if no two instructions are allowed to complete simultaneously because of some shared resource such as a bus

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

General Terms· Algorithms

Additional Key Words and Phrases: Compiler optimization, deadline, greedy algorithm, instruction scheduling, latency, NP-complete, pipeline processor, register allocation, RISC machine scheduling

1. INTRODUCTION

Many code optimization problems for parallel and pipelined machines can be modeled as deterministic scheduling problems. Typically, these scheduling problems involve rearranging generated object code that is derived from a

Authors' addresses: K. V. Palem, IBM Research Division, T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. e-mail: rpalem @watson.ibm.com. B. B. Simons, Compiler Technology Institute, IBM Santa Teresa Laboratory, 555 Bailey Road, San Jose, CA 95141.

single basic block of source code. The object code instructions have deterministic behavior and often require a single unit of execution time on the CPU [19, 25, 26]. A fast algorithm for rearranging the object code in a basic block to minimize execution time can improve the quality and efficiency of the code generated by the compiler. In particular, a minimum execution time schedule contains the smallest possible number of no-ops or idle cycles, thereby utilizing all of the processor cycles effectively. In addition, it is frequently necessary to guarantee that time-critical instructions are completed early in the schedule. For example, the early execution of certain instructions might help to minimize spillage induced by register allocation.

To illustrate the notion of time critical, assume that instruction s_i initializes a value which is subsequently referenced by instructions $s'_1, s'_2, \ldots, s'_k$. Suppose that s_i is given an early deadline and that $s'_1, s'_2, \ldots, s'_k$ are given somewhat later, but still early, deadlines. Then, if a schedule satisfies the deadline constraints, the register that is used to store the value initialized by s_i will be available for other use no later than the latest deadline that is assigned to $s'_1, s'_2, \ldots, s'_k$.

We present a fast algorithm that takes a basic block of code as its input and constructs a minimum completion time schedule for a generic model of RISC machines. This model approximates several early RISC processors such as the Sun SPARC [28], the IBM 801 [26, 27], the Berkeley RISC [19], and the HP Precision Architecture [13]. It also constructs an optimal schedule satisfying time-critical constraints for such machines. Our algorithm can be used as a heuristic for constructing "fast" schedules for RISC processors with long or multiple pipelines for which it is not optimal such as the Intel 80860 [7].

Any scheduling algorithm that never inserts a no-op if some instruction is available for scheduling is called a *greedy* scheduling algorithm. We show that in the absence of time-critical constraints, a greedy scheduling algorithm produces a schedule for target machines with multiple identical pipelines that has a completion time less than twice that of an optimal schedule. The factor of two is a worst-case guarantee, and in practice most greedy algorithms tend to perform better.

This paper contains three NP-completeness results. One proves that the problem of producing schedules with minimum overall completion time is NP-hard if the depth of the pipeline grows as part of the input, even when the basic block of code being input consists of only several independent streams of straightline code, and these instructions have no time-critical constraints. The other two results demonstrate how the introduction of resource constraints can make a problem NP-complete. In particular, the instruction scheduling problem is NP-hard for a single small-depth pipeline, even if the inputs are only independent streams of straightline code without time-critical constraints, if there is only a single register, or if two instructions are not allowed to complete simultaneously.[1] A weaker NP-completeness result for

[1]The completion time-constraint problem was brought to our attention as an open problem at the Workshop on Programming Languages and Compilers for Parallel Computing, Cornell University, Aug., 1988.

the instruction scheduling problem with register constraints is claimed in [17], but the proof is flawed [15].

2. DESCRIPTION OF THE MODEL

We consider target machines in which every machine instruction requires one cycle of CPU time. If the operands of the instruction are derived from on-chip registers, then such instructions are fetched, decoded, and executed in one cycle. In contrast, some instructions, such as LOADs, require additional cycles due to latencies introduced by memory access.

We use the standard directed acyclic graph (DAG) representation of basic blocks [17]. Each node in the DAG corresponds to an instruction, and each edge corresponds to a dependence. An instruction cannot be executed until all of its predecessors are completed. Furthermore, if an instruction, say a LOAD, requires additional time to complete, instructions that depend on that load must be delayed until the entire LOAD has been completed. The additional delay, which is represented as a weight on the appropriate out-edges from the LOAD to its immediate descendents, is called an *interinstructional latency*, or *latency*, for short. The value of the latency is the additional delay beyond the unit of time required by the CPU.

Figure 1 shows a simple DAG, all the edges of which have latency 1, and two possible schedules for that DAG. Schedule S_1 illustrates how unnecessary idle time can be introduced if the nodes are scheduled in a suboptimal order. Clearly, schedule S_2, which completes execution earlier than S_1, is preferable. The idle time in schedule S_1 could have been introduced either at compile time using no-ops or at runtime, if the target machine has hardware interlocks. Therefore, depending on the machine, the problem is either to minimize the number of idle cycles caused by no-ops produced by the compiler or to minimize the number of cycles during which the interlocks are activated.

In addition, the input might contain time-critical instructions. Such an instruction has associated with it a nonnegative integer called a *deadline*. The deadline could be either a real-time constraint or a value chosen by the programmer or compiler to try to improve performance. In this case, the problem is to construct a schedule in which all instructions are completed by their deadlines. We do not address the question of how to assign deadlines. A schedule in which all the nodes are completed by their deadlines is called a *feasible* schedule; otherwise it is *infeasible*. A problem instance is feasible or infeasible according to whether or not a feasible schedule exists for that instance. If the input has no deadlines, then by default it is feasible. An instruction completed after its deadline is *tardy*. If instructions are allowed to be tardy, a *minimum tardiness* schedule is a schedule in which the maximum tardiness of any node is minimized.

2.1 Some Definitions

Assume we have a set of instructions that form a basic block. Techniques such as trace scheduling [8, 9] or global compaction [1, 22] can be used to

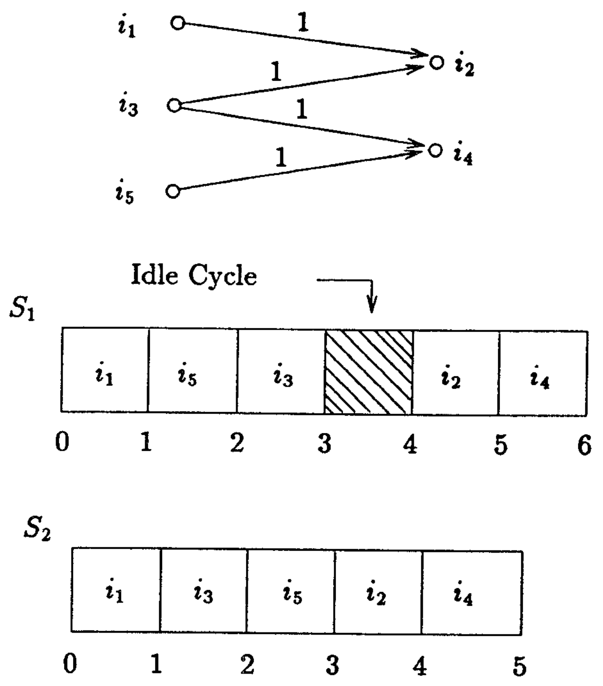


Fig. 1. A DAG with two possible schedules.

increase the size of this basic block. The problem input is a DAG $G = (N, E)$ that represents the basic block, where each node $i \in N$ corresponds to one of the instructions, and each edge $(i, j) \in E$ corresponds to a dependence. In addition, each edge has a nonnegative integer weight $w(i, j)$, which is the latency of edge (i, j). If a node i must be completed by a certain time t, then i has a *deadline* $d(i) = t$. If the target machine has multiple (identical) processors, then the delays involved in transferring data items between these processors through on-chip register banks is encoded by appropriately incrementing the edge latencies.

Formally, a schedule S specifies for each instruction or node a start time $S(i)$ and a processor $M(i)$ from the set $1, 2, \dots, m$ of identical processors in the target machine such that:

(1) For $i, j \in N$, $i \neq j$, and $M(i) = M(j)$, $|S(i) - S(j)| \geq 1$. (No two nodes are executed simultaneously on the same processor.)

(2) $S(j) \geq S(i) + w(i, j) + 1$ for $(i, j) \in E$. (The earliest start time of a node depends on the start time, latency, and processing time of its predecessors.)

If there are no deadlines, then the goal of the algorithm is to construct a schedule with *minimum completion time*, that is, $\max_i\{S(i) + 1\}$ is minimized; if there are deadlines, then the goal is to construct a feasible schedule that starts at time 0. If for some assignment of deadlines there is no feasible schedule, then the algorithm should return that information and also construct a minimum tardiness schedule. We show in Section 4.1.2 that if the rank algorithm, defined below, constructs a minimum tardiness schedule for a problem instance with deadlines, it also constructs a minimum completion time schedule for an instance of the same problem without deadlines.

2.2 Relationship to Pipeline Scheduling

The pipeline model studied in this paper is more general than the classical or standard notion of a pipeline machine. A *standard pipeline machine of length* k is a machine with k unit length stages for which an instruction that enters the first stage of the pipeline at time t exists from the last stage at time $t + k$. A new instruction can enter the pipeline at times $t + 1, t + 2, \dots$. In this model the start time of an instruction is at least k units greater than that of any instruction on which it depends, and as many as k instructions may be in the pipeline simultaneously. The model of a standard pipeline is a special case of the latency model in which all the latencies are $k - 1$.

A generalization of the standard pipeline model is obtained by allowing an instruction to exit the pipeline at some stage prior to the last stage. An instance of this problem can be represented by having identical latencies on all the out-edges of a node, but allowing different nodes to have different values on their out-edges. For the algorithms in this paper, we consider only the most general version of the model, namely one in which different out-edges of the same node can have different latencies.

2.3 Compiler Construction Issues

We briefly discuss the interaction between the scheduler and other stages of optimization in the compiler, particularly register allocation. If the register allocation phase precedes the scheduling phase, then, by forcing instructions to use the same register, the register allocator can create dependences between instructions that are not otherwise dependent. Consequently, unnecessary hazards might be introduced. This problem can be caused by any shared resource that is allocated at compile time. Another example in which unnecessary hazards could be introduced is a target machine in which two instructions that complete simultaneously access a single (limited) resource, such as a bus. Deadlines might provide a technique for handling this problem.

There are different approaches for handling the interaction between the scheduler and the register allocator. In the approach used by Hennessy and Gross [16, 17], the instruction scheduler is explicitly constrained by hazards that are introduced by the register allocator and by memory access. Gibbons and Muchnick [13] deal with register allocation by introducing edges in the DAG to prevent instructions that share registers from overlapping. Register allocation is handled in the PL.8 compiler [3] by having the instruction scheduler preceded by a first register allocation phase and succeeded by a second register allocation phase. In the first phase the allocation is done for a target machine with an unbounded number of registers. A register is reused in the first phase only when the reuse is guaranteed not to add any additional constraints. After instruction scheduling, register allocation for the actual target machine is performed, and hazards are eliminated by appropriately introduced *spill code*. The latter two approaches obviate the need for the scheduler to explicitly deal with constraints introduced by register allocation, other than those encoded into the input DAG.[2] A mixed strategy that switches back and forth between instruction scheduling and register allocation is presented by Goodman and Hsu [12].

We assume that the compiler designer has employed a technique such as that in [13] or [3]. Consequently, the problem of instruction scheduling is separated from that of register allocation. Similar approaches can be used to separate the scheduler from constraints introduced by other shared resources, such as a bus. For a more detailed discussion, see our chapter on instruction scheduling in [2].

3. PREVIOUS WORK

In [3–6, 10, 13, 16–18, 20, 21, 23, 30], aspects of instruction scheduling for pipeline and related machines are studied. A survey of deterministic scheduling results for pipelined machines is contained in [20]; some of these results can be found in more detail in [6, 10, 21]. In [23], Palem characterizes a general sufficient condition for pipelined scheduling problems to be solvable in polynomial time, and shows that some new problems, as well as several previously known polynomially solvable scheduling problems, satisfy this condition.

We have already discussed the approach taken for the PL.8 compiler [3]. There are a number of other results that are based on greedy scheduling. Hennessy and Gross [16, 17] present a heuristic that runs in time $O(n^4)$, where n is the number of nodes in the DAG, and report performance results for their heuristic on the MIPS machine [18]. There is no analysis of the worst-case performance of the heuristic. Gibbons and Muchnick [13] describe a heuristic for the case in which the latencies are 0 or 1, with the substantially improved running time of $O(n^2)$. Although they report good performance for the heuristic, they too do not do a worst-case analysis of the quality of the schedules produced by their heuristic. Bernstein and Gertner [4] give an algorithm for optimally scheduling an arbitrary graph with latencies of 0 or 1 on a single processor. Since their algorithm uses transitive reduction as a preprocessing step, the running time of their algorithm is either that of transitive reduction[3] or $O(n^2)$, if preprocessing costs are ignored. Their algorithm does not handle time-critical instructions. In [5], Bernstein, Rodeh, and Gertner analyze the worst-case behavior of the greedy scheduling algorithm for a target machine with a single pipeline. In Section 5 we generalize that result to the multiple pipeline or processor case.

[2] If code scheduling is done prior to register allocation and no consideration is given to minimizing register lifetimes, additional register spill may be introduced. The use of deadlines and forms of backward scheduling may help eliminate this problem. A detailed discussion is beyond the scope of this paper.

[3] The running time is the minimum of $O(ne)$, where e is the number of edges in the original DAG, and the running time of matrix multiplication.

4. THE RANK ALGORITHM

A standard technique for instruction scheduling is to use a greedy scheduling algorithm that always schedules a node whenever there is at least one available node. The input to the greedy algorithm is an ordered list of nodes, a DAG G, which represents the dependences between the nodes, latencies, which can be any nonnegative integer, and m, the number of processors in the target architecture.

We refer to each possible integer time at which a node can start as a *time step*.[4] Time step t *finishes* at time $t + 1$. At each time step the greedy algorithm scans the list, choosing up to m eligible nodes on each scan to be scheduled, giving priority to the nodes earliest on the list. A node is eligible if all of its predecessors in G have been scheduled on an earlier scan and the relevant latency constraints have been satisfied. If no node is eligible, the value of the current time step is increased to the earliest time for which some node is eligible. At the end of the scan, the chosen nodes are deleted from the list, and new nodes may become eligible. The process is repeated until the list is empty.

The input to the greedy algorithm could be any list, including an arbitrarily ordered one, that does not use information about the graph structure to prioritize nodes.[5] The rank algorithm, defined below, uses information about the latencies between a node i and each of i's successors, as well as the deadline of node i, to compute the *rank* of i, written *rank* (i). *The rank of a node is an upper bound on the finish time of that node in any feasible schedule.* Once the ranks are all computed, the algorithm constructs a list, based on the ranks, which it then schedules greedily.

The rank algorithm constructs a feasible single processor schedule or a minimum tardiness schedule for an arbitrary input DAG if all the latencies are either 0 or 1, and some or all of the nodes have preassigned deadlines. Although the rank algorithm is not guaranteed to find a feasible schedule for an arbitrary DAG if the latencies are greater than 1, or if there is more than one pipeline in the target machine, we conjecture that its behavior as a heuristic is quite good in the general case. There are some preliminary test results in [7] showing the rank algorithm performing better on the Intel 80860 than the Warren algorithm, which is used for instruction scheduling in the IBM RS/6000 [30]. In at least one special case, namely the monotone interval ordered graph (see Section 4.1.5), the rank algorithm constructs an optimal schedule for arbitrary latencies, deadlines, and processors. The approximation bound of Section 5 applies to the rank algorithm, as well as to any other greedy scheduling algorithm, if there are no preassigned deadlines.

[4] In a (forward) schedule we assume that 0 is the first time step.

[5] The *highest level first* algorithm, a more sophisticated version of the greedy algorithm, uses some information about the graph structure to construct the ordered list. However, the highest level first algorithm is not guaranteed to construct an optimal schedule even for some simple cases of the instruction scheduling problem, as illustrated in schedule S_1 of Figure 1.

4.1 The Algorithm

(1) Compute the *ranks* of all the nodes. If some node is assigned a rank less than or equal to 0, return the information that the problem instance is infeasible.[6]

(2) Construct *list*, which is an ordered list of nodes in *nondecreasing* order of their ranks.

(3) Apply the greedy scheduling algorithm to *list*.

4.1.1 *Computing the Ranks.* The *weighted length* of a path p is the sum of its constituent edge latencies and the number of nodes in p, excluding the endpoints of p. Let $\mathbf{w}^+(i,j)$ denote the weighted length of the longest path from node i to a successor j. In Figure 2, $\mathbf{w}^+(i_1, i_2) = \mathbf{w}^+(i_1,i_3) = \mathbf{w}^+(i_3,i_4) = 1$, $\mathbf{w}^+(i_2, i_4) = 0$, and $\mathbf{w}^+(i_1,i_4) = 3$.

The rank of node i is computed after $\mathbf{w}^+(i,j)$ and $rank(j)$ have been computed for all nodes j that are successors of i. If j is a node that does not have preassigned deadlines, then $d(i) \leftarrow D$, where D is some integer that is sufficiently large that all nodes are guaranteed to be completed by time D. An example of such a value is $(k+1)n$, where k is the maximum latency. A node with no successors is called a *sink*. If i is a sink, then $rank(i) \leftarrow d(i)$.

For Figure 2, suppose $d(i_1) = d(i_2) = d(i_3) = d(i_4) = 6$. Then $rank(i_4) = 6$. If i has only a single successor j, then $rank(i) = min\{rank(j) - 1 - \mathbf{w}^+(i,j), d(i)\}$. In Figure 2 $rank(i_2) = 5$ and $rank(i_3) = 4$.

Let i be a node with more than one successor, whose successors' ranks have all been computed. We construct a sorted list $sw(i)$ of the successor set of node i. The nodes in $sw(i)$ are sorted in nonincreasing order by the $\mathbf{w}^+$ values relative to i, that is, if $\mathbf{w}^+(i,j) > \mathbf{w}^+(i, p)$, then node j occurs before node p in $sw(i)$. For node i_1 of Figure 2, the possible values for $sw(i_1)$ are $sw(i_1) = i_4 i_3 i_2$ and $sw(i_1) = i_4 i_2 i_3$. Let $sw(i)_q$ be all the nodes j in $sw(i)$ for which $\mathbf{w}^+(i,j) = q$. Because $sw(i)$ is sorted, the nodes in $sw(i)_q$ are contiguous in $sw(i)$.

We next sort each segment $sw(i)_q$ by nonincreasing order of ranks. Let $swr(i)$ be the resulting list. The nodes in $swr(i)$ are all the successors of node i stored in the nonincreasing order based on $\mathbf{w}^+$; all the nodes with the same $\mathbf{w}^+$ value are further sorted by their ranks. For Figure 2, the only possible choice for $swr(i_1)$ is $swr(i_1) = i_4 i_2 i_3$.

A schedule for a target machine with m processors can be represented by a matrix in which each row represents one of the processors of the target machine, and each column represents a time step. A *slot* is a single entry in the matrix, and represents a specific time step on a specific processor. A slot is *available* if no node has been assigned to the specific start time and processor represented by the slot.

To compute $rank(i)$, we select nodes in the order in which they appear in $swr(i)$, starting at the beginning of $swr(i)$. Each time we select a node j, we *backward schedule* j by greedily scheduling j in an available slot with the latest possible start time less than $rank(j)$. In particular, we schedule j in

[6] If the problem instance is infeasible, a minimum tardiness schedule is constructed.

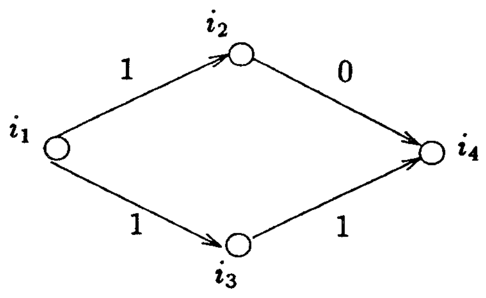


Fig. 2. An example DAG.

the time step, finishing at the largest D', such that:

(1) $D' \leq rank(j)$, and

(2) the number of nodes that occur before j in $swr(i)$ and have been assigned this time step is strictly less than m.

The rank of i with respect to j equals $min\{D' - 1 - \mathbf{w}^+(i,j),\ d(i)\}$. The $D' - 1$ is the start time of node j. The rank of i with respect to j gives the latest time that node i can *finish* if node j is to be completed by its completion time in the backward schedule. We compute the rank of i with respect to each of its successors; $rank(i)$ is the *smallest* of these values.

4.1.2 *Correctness of the Ranks.* Below we present the key theorem and proof for the rank algorithm. It shows that if no nodes are completed later than their deadlines, they must also be completed no later than their ranks. We assume that nodes with no preassigned deadlines are given the default deadline D. Note that the proof is entirely general, and holds for any number of processors and any latencies.

THEOREM 4.1. *Let G be a DAG and S a schedule for G in which every node is completed by its deadline. Then every node i in S is completed no later than $rank(i)$.*

PROOF. If i is a sink node, then the theorem follows trivially from the definition of rank.

Suppose that i is not a sink node and assume inductively that the theorem holds for all successors of i. If $rank(i) = d(i)$, then the theorem obviously holds. So assume that $rank(i) = D' - 1 - \mathbf{w}^+(i, j') < d(i)$ for some j' and D'. By the manner in which rank is computed, j' is scheduled in the backwards schedule in time step $D' - 1$. If $D' = rank(j')$, then the result follows immediately from the assumption that the ranks of all the successors of i satisfy the theorem, together with the definition of $\mathbf{w}^+(i, j')$.

Now assume that $D' < rank(j')$ and let S_{backward} with completion time T_{backward} be the backwards schedule as it exists immediately after the insertion of j'.

Case 1. There are no idle slots in the time steps finishing at $D' + 1$, $D' + 2, \ldots, T_{\text{backward}}$. Since nodes are scheduled as late as possible in S_{backward}, $rank(j) \leq T_{\text{backward}}$, for $j \in S_{\text{backward}}$. From the order in which nodes are

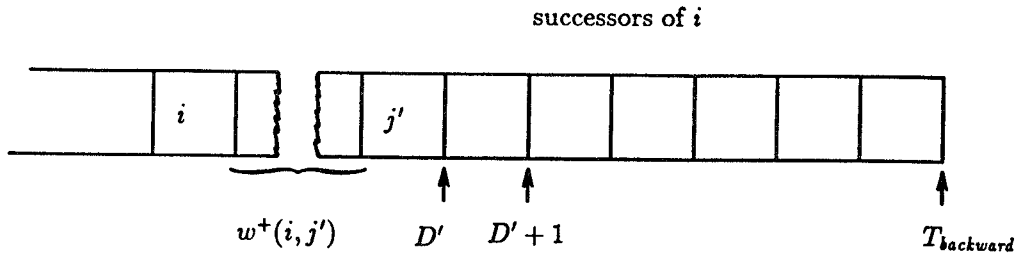


Fig. 3. An illustration of Case 1 of the proof for $m = 1$.

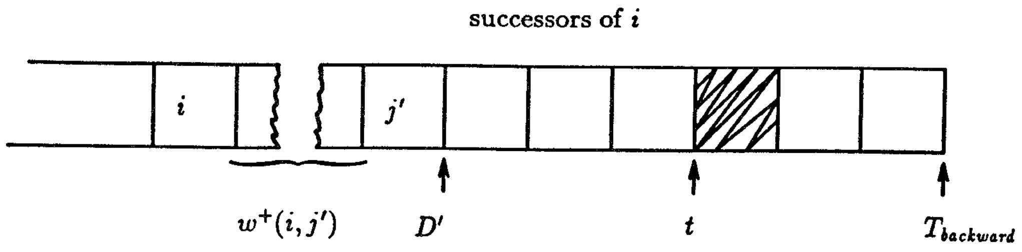


Fig. 4. An illustration of Case 2 of the proof for $m = 1$.

placed in S_{backward}, we get $\mathbf{w}^+(i,j) \geq \mathbf{w}^+(i,j')$, $j \in S_{\text{backward}}$ (see Figure 3). A simple pigeon-hole argument suffices to prove that if i is completed later than *rank*(i) in the forward schedule, some successor of i will complete later than time T_{backward}. This contradicts the assumption that all of i's successors are completed by their ranks.

Case 2. There is some idle slot in the time steps finishing at $D' + 1$, $D' + 2, \ldots, T_{\text{backward}}$. Let t be the start time of the smallest such time step containing an idle slot. Since S_{backward} is constructed greedily, it follows that all nodes in time steps with start time less than t have rank no greater than t. Also, by assumption, all these time steps have no idle slots. Therefore, all nodes scheduled in times steps prior to t must be completed by time t and have a $\mathbf{w}^+$ value at least as great as $\mathbf{w}^+(i,j')$ (see Figure 4). The theorem again follows from a pigeon-hole argument. □

If I is a problem instance, then I_δ is defined to be the problem instance that is obtained by adding δ to every preassigned deadline of I. If a node has no preassigned deadline, then it is given the preassigned deadline of $D + \delta$ (rather than D). We also define *rank*(i) to be the ranks computed for instance I, and *rank*$_\delta(i)$ to be the rank of node i in I_δ.

LEMMA 4.2. *Let I and I_δ be as defined above. Then rank$_\delta(i)$ = rank$(i) + \delta$.*

PROOF. The proof follows directly from the definition of rank. □

Corollaries 4.3 and 4.4 show how the rank algorithm can be used to solve the minimum tardiness problem and the minimum completion time problem in the absence of deadlines.

COROLLARY 4.3. *Assume that the rank algorithm constructs a feasible schedule for a class of problems if one exists. Then it also constructs a minimum tardiness schedule for infeasible instances from that class.*

PROOF. If a problem instance I is infeasible, then there exists a sufficiently large δ such that when δ is added to all the deadlines, the corresponding problem instance I_δ is feasible. Since, by assumption, the rank algorithm constructs a feasible schedule if one exists, the smallest δ for which a feasible schedule can be constructed is the value of the minimum tardiness. It follows from Lemma 4.2 that *list* is the same for both I and I_δ. Therefore, the same schedule is constructed for both problem instances. □

COROLLARY 4.4. *Suppose the rank algorithm constructs a minimum tardiness schedule for a class of problems. Then the rank algorithm also constructs a minimum completion time schedule for inputs in which there are no preassigned deadlines.*

PROOF. The proof follows from the technique of giving each node the same number D as a deadline. If D is precisely the minimum completion time for the problem instance, then the rank algorithm will construct a schedule with 0 tardiness. If D is not the minimum completion time, by Lemma 4.2, the rank algorithm will construct the identical schedule as for the case in which D is the minimum completion time. □

4.1.3 *The Running Time of the Rank Algorithm.* For the analysis of the worst-case running time, we assume that the input DAG is a connected graph. We also assume that the input DAG is a transitively closed graph, where the *transitive closure* of a graph $G = (N, E)$ is a graph $G' = (N, E')$, which consists of all the nodes from G together with an edge from i to j if there is a path in G from i to j. Otherwise, the transitive closure is automatically computed during the computation of the $\mathbf{w}^+$ values.

Computing the $\mathbf{w}^+$ *values.* The computation of all the $\mathbf{w}^+$ values takes time $O(en)$. Given the $\mathbf{w}^+$ values, constructing the lists $sw(i)$ and $swr(i)$ involves sorting the various successor sets. We can use any sorting algorithm that has a worst-case running time of $O(n \log n)$. Since each edge in the transitively closed graph is used for processing the rank of only one node, the total time required for sorting all the sets $sw(i)$ and $swr(i)$ is $O(e' \log n)$, where e' is the number of edges in the transitive closure of G.

Backscheduling using UNION-FIND. Once the list $swr(i)$ has been constructed, the backward scheduling step of the rank computation for node i is performed. If the backward scheduling is done in a straightforward fashion, it will increase the running time of the algorithm. Therefore, we implement this step using the UNION-FIND algorithm [29] on the values of the ranks of the successors of i.

Suppose there are n_i distinct ranks values among the successors of i. We create n_i single node trees, $tree[1], tree[2], \ldots, tree[n_i]$, with a single rank associated with each tree. We order the trees by their associated ranks, such that $rank\ (tree[p - 1]) < rank(tree[p])$. So $rank(tree[1])$ is the smallest rank in $swr(i)$, and $rank(tree[n_i])$ is the largest rank in $swr(i)$.

Each *tree*$[p]$ has a field called *capacity*$[p]$ associated with it. We set *capacity*$[1] \leftarrow m \times \mathit{rank}(\mathit{tree}[1])$, and *capacity*$[p] \leftarrow m \times (\mathit{rank}(\mathit{tree}[p]) - \mathit{rank}(\mathit{tree}[p-1]))$, where m is the number of processors. *capacity*$[p]$ is the number of nodes that *can* be inserted into the backward schedule in the slots greater than $\mathit{rank}(\mathit{tree}[p-1])$ and less than or equal to $\mathit{rank}(\mathit{tree}[p])$. Each *tree*$[p]$ also has a field called *content*$[p]$; initially *content*$[p] \leftarrow 0$ for $1 \leq p \leq n_i$.

Suppose node $j \in \mathit{tree}[p]$ is the next node in $\mathit{swr}(i)$ to be processed. We process j by setting *content*$[p] \leftarrow$ *content*$[p] + 1$. If as a result *content*$[p] =$ *capacity*$[p]$, and $p > 1$, then *tree*$[p]$ is made a subtree of *tree*$[q]$, where $\mathit{rank}(\mathit{tree}[q])$ is the largest rank of any tree with rank less than $\mathit{rank}(\mathit{tree}[p])$. (Initially, $q = p - 1$, but as trees are made subtrees of other trees we can have $q < p - 1$.)

If *content*$[1]$ ever becomes greater than *capacity*$[1]$, then there is no feasible schedule and $\mathit{rank}(i) \leq 0$.[7]

We use subroutines FIND and UNION to make one tree a subtree of another. FIND is used to determine the name of the tree with the largest rank less than the rank associated with the tree that has just been filled to its capacity. This is done by calling FIND$(\mathit{rank}(\mathit{tree}[p-1]))$. The value q that is returned is the name of the tree containing $\mathit{rank}(\mathit{tree}[p-1])$. UNION$(p, q)$ is then called to make *tree*$[p]$ a subtree of *tree*$[q]$.

The UNION-FIND algorithm processes each of the e' edges exactly once during the course of the entire computation. There is an implementation of the UNION-FIND algorithm [29] that has a running time of $O(e'\alpha(e'))$, where $\alpha(\cdot)$ is the inverse of Ackermann's function, and e' is the number of edges in the transitive closure of G. (For all practical purposes, the value of the inverse Ackermann function can be assumed to be a very small integer.)

Greedy scheduling. The greedy algorithm portion of the rank algorithm can be implemented in $O(e' + n \log n)$. Two priority queues are used to implement the greedy algorithm. The first is used to record when nodes whose predecessors have all been scheduled become eligible. This is done by inserting a node into the priority queue as soon as all of its predecessors have been scheduled, using as the key the minimum start time that would not violate any of the latencies. A node is removed from the first priority queue when the time step being scheduled is equal to its key. The node is then inserted into the second queue, using the location of the node in the original list as the key.

THEOREM 4.5. *The running time for the rank algorithm is $O(en + e' \log n)$ if the $\mathbf{w}^+$ values are not part of the input, or $O(e' \log n)$ otherwise, where n is the number of nodes, e is the number of edges in the original DAG, and e' is the number of edges in the transitively closed DAG.*

[7] If a minimum tardiness schedule is being constructed, then assigning a node a rank less than or equal to zero is not a contradiction.

4.1.4 *Arbitrary DAG,* 0 *or* 1 *Latencies, and a Single Processor.* The more important class of problems for which the rank algorithm constructs an optimal schedule is the class of arbitrary DAGs in which the latencies are either 0 or 1, the deadlines are arbitrary, and there is a single processor ($m = 1$). As noted earlier, this class models versions of several different RISC machines.[8]

THEOREM 4.6. *Let $G = (N, E)$ be a DAG, the edges of which have latencies of either* 0 *or* 1 *and the nodes of which have arbitrary deadlines. Then the rank algorithm constructs a feasible single processor schedule for G, whenever one exists, and constructs a minimum tardiness schedule otherwise.*

PROOF. Assume for contradiction that the problem instance is feasible, but the rank algorithm fails to construct a feasible schedule.

Let S_{rank} be the partial schedule that had been constructed just before the first node is scheduled to be complete after its rank, and let i be that node. By assumption, all the nodes in S_{rank} are scheduled to complete by their ranks. Backtrack over S_{rank} examining the contents of each time step until the first of the following three events occurs:

(1) a node j is encountered such that $rank(j) > rank(i)$ (greedy scheduling implies that some node is scheduled in S_{rank} prior to j),
(2) an idle slot is encountered,
(3) all of S_{rank} has been examined.

Suppose the first condition holds, namely node j is encountered with $rank(j) > rank(i)$. Let t be the time step at which j is scheduled to start and let Σ be the set of nodes scheduled to start at time steps $t + 1, t + 2, \ldots, rank(i) - 1$, together with node i itself. (Node j is not included in Σ.) By the definition of node j, all nodes in Σ have rank less than $rank(j)$. Also, since $i \in \Sigma$, $|\Sigma| = rank(i) - t$.

Let j' be the node scheduled to start at time step $t - 1$; j' must be a predecessor of all the nodes in Σ, since otherwise the greedy algorithm would have scheduled one of those nodes at time step t. If $k \in \Sigma$ is an immediate successor of j' and there are no other paths from j' to k, then $w(j', k) = 1$, since otherwise the greedy algorithm would schedule node k at time t. Therefore, for all successors k' of j' we have $\mathbf{w}^{+}(j', k') \geq 1$. Consequently, the backward schedule will cause some successor k' of j' to have a finish time (D') no greater than $t + 1$. By the definition of rank, this gives $rank(j') \leq t - 1$, contradicting the assumption that all the nodes in S_{rank} are completed by their ranks. Therefore, this condition cannot occur.

If the second condition holds, namely an idle slot is encountered, then again let node j' be the node immediately preceding the idle slot; the argument is the same as above. Again, this condition cannot occur.

[8]As computation speeds increase, latencies from memory accesses are increasing relative to computation time. The rank algorithm can be used as a heuristic for those cases in which the memory latencies exceed 1.

If the last condition holds, then S_{rank} has no idle time, and all the nodes in S_{rank} have rank no greater than $rank(i)$. Consequently, from the pigeon-hole principle, there is a node whose rank is no greater than zero. From Theorem 4.1, we conclude that this contradicts the existence of a feasible schedule. (Intuitively, at least two nodes must be scheduled in the same slot, contradicting the fact we are constructing a schedule for a single processor.)

If the problem instance is infeasible, it follows from Corollary 4.3 that the rank algorithm constructs a minimum tardiness schedule. □

4.1.5 *Monotone Interval-Order, Arbitrary Latencies, and Multiple Processors.* Even though the general instruction scheduling problem is NP-complete for arbitrarily large latencies, it is still possible that fast (polynomial time) algorithms exist for interesting classes of graphs. One such class, for which the rank algorithm constructs a feasible schedule whenever one exists, is called *monotone interval-orders*. For this problem class the number of processors in the target machine can be arbitrary, and the latencies and deadlines can assume arbitrary integer values.

An *interval-order graph* is a DAG $G = (N, E)$, where N is a set of closed intervals in the real line. The edges of G are derived from the order between these intervals, as follows. For $i, j \in N$, $(i, j) \in E$ if and only if for any pair of numbers x and y with $x \in i$ and $y \in j$, $x < y$. Each edge has a latency, and each node either has a preassigned deadline or is assigned one by the algorithm. The only constraints on their values are that the latencies are nonnegative and all nodes that do not have preassigned deadlines are assigned the same large deadline by the algorithm. The following lemma is from [24].

LEMMA 4.7. *Let $G = (N, E)$ be an interval-order graph. Then for $i, j \in N$, either all the predecessors of i are also predecessors of j or all predecessors of j are also predecessors of i.*

A *monotone* interval-order graph is one in which, given any pair of edges (i, j) and (i, j'), $w(i, j) \geq w(i, j')$ whenever the predecessors of j' are also predecessors of j.

THEOREM 4.8. *Let $G = (N, E)$ be a monotone interval-order graph with arbitrary latencies and deadlines, and assume that there are $m \geq 1$ processors. Then the rank algorithm constructs a feasible schedule for G whenever one exists, and constructs a minimum tardiness schedule otherwise.*

PROOF. As in Theorem 4.6, we initially assume for contradiction that the rank algorithm fails to construct a feasible schedule for G, but there is a feasible schedule for G. Let S_{rank} be the partial schedule that had been constructed by the greedy scheduling algorithm when it first determines that the problem instance is infeasible, and let i be the first node that is not completed by its rank. For $j \in S_{rank}$, let $S_{rank}(j)$ be the start time of j. Clearly, $rank(j) > S_{rank}(j)$ for all nodes in S_{rank} except i. We consider the following three cases.

Case 1. There are precisely m nodes with ranks bounded above by $rank(i)$ scheduled at each of the time steps $0, 1, \ldots, rank(i) - 1$. Then by a simple pigeon-hole argument, together with Theorem 4.1, there does not exist any feasible schedule.

Case 2. There is either an idle slot or some node with rank greater than $rank(i)$ scheduled at time step $rank(i) - 1$. Then i must have a predecessor, say j, such that $S_{rank}(j) + \mathbf{w}^+(j, i) + 1 \geq rank(i)$. Otherwise, i would have been scheduled to start at time step $rank(i) - 1$. From the definition of rank, we get that $rank(j) \leq S_{rank}(j)$, contradicting the assumption that any node in a time step smaller than $rank(i)$ has a start time less than its rank.

Case 3. There is some time step $0 \leq t' < rank(i) - 1$ such that there is either an idle slot or some node with rank greater than $rank(i)$ scheduled at time step t'. Let t be the largest such time step, and let Σ be the set of nodes with start times of $\{t + 1, t + 2, \ldots, rank(i) - 1\}$ together with node i. Clearly, $|\Sigma| = (rank(i) - t - 1) \times m + 1$. Any node $i' \in \Sigma$ must have a predecessor j such that $S_{rank}(j) + \mathbf{w}^+(j, i') + 1 > t$. Otherwise, i' would have been assigned start time t. We say that j is a *constraining node of* i'.

Let $k \in \Sigma$ be the node in Σ with the smallest sized predecessor set, i.e. $|pred(k)| \leq |pred(k')|$, for $k, k' \in \Sigma$. By Lemma 4.7 every node in $pred(k)$ is a predecessor of all the nodes in Σ. Let $j \in pred(k)$ be a constraining node for k. Then, because G is a monotone interval order, $\mathbf{w}^+(j, k') \geq \mathbf{w}^+(j, k)$ for $k' \in \Sigma$. Consequently, j is a constraining node for all $k' \in \Sigma$. But now the rank computation for j results in a rank for j which is less than the finish time for j in S_{rank}, contradicting the assumption that i is the first node in S_{rank} with this property.

If the problem instance is infeasible, it follows from Corollary 4.3 that the rank algorithm constructs a minimum tardiness schedule. □

5. THE GREEDY SCHEDULING HEURISTIC

Most scheduling algorithms are greedy in that they do not introduce idle time if some instruction is available for scheduling. The result in this section holds for any greedy scheduling algorithm applied to an instruction scheduling problem containing an arbitrary DAG, arbitrary latencies, an arbitrary number of processors, and no preassigned deadlines. The analysis is worst case, and the algorithms will tend to perform better in practice.

THEOREM 5.1. *Let $G = (N, E)$ be an arbitrary DAG with arbitrary latencies between* 0 *and* k. *Then the greedy scheduling algorithm constructs a schedule for G on a target machine with m processors that is guaranteed to be no more than a factor of* $2 - 1/m(k + 1)$ *worse than optimal.*

PROOF. Consider the greedy schedule constructed for the given DAG G with the assumption that we have as many processors as we can use at our disposal, as opposed to only m. We use S_∞ to denote this schedule, with T_∞ being the completion time of S_∞. Let S_{greedy} be a schedule constructed by the greedy algorithm for the given DAG with a target machine of m processors, and let T_{greedy} be the completion time of S_{greedy}.

We say that a time step in a schedule is *active* provided it has at least one node scheduled in it. Otherwise, it is *idle*. If P is a path in G, the number of *idle slots in P* is defined to be the sum of the latencies of the edges of P. We define $idle_\infty$ to be the maximum number of idle slots of any path in G. The *length* of a path P is the sum of the number of nodes and the number of idle slots in P. By construction, T_∞ is the length of the longest path in G.

LEMMA 5.2. *The maximum number of time steps in S_{greedy} containing at least a single idle slot is T_∞; the maximum number of idle time steps in S_{greedy} (that is, time steps containing only idle slots) is $idle_\infty$.*

PROOF. Any time step in S_{greedy} that has an idle slot must contain either a node or an idle slot from every path in G containing a node that remains unscheduled at that time step. Therefore, by the pigeon-hole principle, the maximum number of time steps in S_{greedy} containing at least a single idle slot is T_∞. Similarly, any idle time step in S_{greedy} must contain an idle slot from every path in G containing a still unscheduled node, and again the result follows from the pigeon-hole principle. □

Let $active_\infty = T_\infty - idle_\infty$, and let P be a path in G with $idle_\infty$ idle slots. If $|P| < T_\infty$, then add a chain of $T_\infty - |P|$ nodes with latency 0 between them to P. Let P' be P together with the additional chain of nodes, and let G' be G with P' substituted for P. Because $|P'| = T_\infty$, the length of S_∞ with G' substituted for G remains T_∞. Since $active_\infty$ is the number of nodes in P', and since there can be at most k idle slots following any node in a path, we have:

$$T_\infty \le (k + 1) active_\infty. \tag{1}$$

Since the number of idle time steps in S_{greedy} is no more than $idle_\infty = T_\infty - active_\infty$ and since S_{greedy} is a schedule for m processors, there can be no more than $m(T_\infty - active_\infty)$ idle slots in the idle time steps of S_{greedy}. Since each active step of S_{greedy} contains at least one node, there can be at most $(m - 1)(active_\infty)$ idle slots in the active time steps of S_{greedy}. Therefore, if ν_{greedy} is the number of slots in S_{greedy},

$$\begin{aligned} \nu_{greedy} &\le n + m(T_\infty - active_\infty) + (m - 1) active_\infty \\ &= n + mT_\infty - active_\infty \end{aligned} \tag{2}$$

Substituting $\nu_{greedy} = mT_{greedy}$ in (2), we get

$$T_{greedy} \le (1/m)(n - active_\infty) + T_\infty. \tag{3}$$

Substituting (1) in (3) gives

$$\begin{aligned} T_{greedy} &\le (1/m)(n - T_\infty/(k + 1)) + T_\infty \\ &= n/m + T_\infty(1 - 1/(m(k + 1))). \end{aligned} \tag{4}$$

Substituting $T_{opt} \ge n/m$ and $T_{opt} \ge T_\infty$ in (4), we get

$$T_{greedy} \le T_{opt} + T_{opt}(1 - 1/(m(k + 1))) \tag{5}$$

or

$$T_{greedy}/T_{opt} = 2 - 1/(m(k + 1)). \quad \square \tag{6}$$

The completion time of a greedy schedule is within a factor of 2 of the completion time of an optimal schedule. However, the quality of the greedy schedule can degrade as the number of processors in the target machine, as well the latencies, increase. There are examples in [20] in which a schedule constructed by the greedy algorithm can be arbitrarily close to the bound given by (6), thereby showing that the bound is tight.

6. NP-COMPLETENESS RESULTS

All of the NP-completeness reductions use a DAG that is a set of chains. A *chain* is a subgraph that consists of only a simple path, in which every node has at most one in-edge and one out-edge, and the graph is connected. Since a chain corresponds to a basic block of code with *straightline* dependences (threads) and since a chain is a very simple graph (for example, a chain is also a tree), these are very strong negative results. We remind the reader that an NP-completeness proof for a simple case automatically implies that the more complex cases are NP-hard. In particular, since a chain is a very simple DAG, our NP-completeness results also apply to more complex DAGs. Similarly, since our NP-completeness result in Section 6.1 holds for machines with a single register, it also holds for machines with multiple registers.

The NP-completeness reductions are all from the *3-partition problem* [11], which is defined as follows. Given a multiset A containing $3n$ integers and a positive integer bound B, where $B/4 < a_i < B/2$ for all $a_i \in A$ and $\sum_{i=1}^{3n} a_i = Bn$, is there a partition of A into n triples of three elements each such that the sum of the integers in each triple equals B?[9]

6.1 Registers

If there is only a single processor and a single register on the target machine and some but not necessarily all of the nodes are preassigned to the register, then constructing a minimum completion time schedule is NP-complete. To the best of our knowledge, ours is the first correct proof that the addition of register constraints can transform a version of the instruction scheduling problem, for which a polynomial time algorithm exists (from Theorem 4.6), into an NP-complete problem.

When a value is stored in a register by instruction i, a new value cannot be inserted into the register until after all the instructions which access the current value have been executed. We define a *register constraint* as follows. If $w_{max}(i)$ is the maximum latency for all edges (i, j), then a new node cannot be inserted into the register until at least w_{max} time units after the completion of instruction i [17]. Because we are presenting a negative result, this very weak definition of a register constraint only strengthens the result.

THEOREM 6.1. (The register allocation problem). *Let $G = (N, E)$ be a DAG that is a set of chains for which the latencies are all equal to 1 and there is*

[9]Because the 3-partition problem is strongly NP-complete [11], a reduction that is polynomial in the *value* of the numbers in the 3-partition problem instance is sufficient for a proof of NP-completeness.

only one register. The problem of determining if there is a single processor schedule for G having a completion time no greater than D, for some given D, is NP-complete.

PROOF. Membership in NP is obvious. We show that the problem is NP-hard by reducing the 3-partition problem to the register allocation problem.

Given an instance of the 3-partition problem, we construct an instance of the register allocation problem in which all latencies are one. For each a_i there is a corresponding chain called a *number chain*, as shown in Figure 5. Each number chain $C(a_i)$ consists of two subchains called the *first subchain* and the *second subchain*, each containing a_i nodes. Nodes that are assigned to the register are called *register nodes* and nodes that are not assigned to the register are *nonregister nodes*. All of the nodes in the first subchain are nonregister nodes and all of the nodes in the second subchain are register nodes.

There is also a *place-holding* chain, C_{ph}, that consists of n subchains $C^1_{ph}, \dots, C^n_{ph}$ (see Figure 5). C^1_{ph} contains $2B + 1$ nodes, with the first B nodes being register nodes and the remaining $B + 1$ nodes being nonregister nodes. C^i_{ph}, $2 \le i \le n$, contains $2B$ nodes, with the first $B - 1$ nodes being register nodes and the remaining $B + 1$ nodes being nonregister nodes. C_{ph} is constructed by linking subchains $C^1_{ph}, \dots, C^n_{ph}$ in order with latency 1 edges. If the node in C_{ph} is started at time 0, since all edges have latency 1 and the last node of C^n_{ph} has no out-going edge, the earliest completion time for C_{ph} is $4Bn + 1$. We set $D = 4Bn + 1$; consequently, any schedule with a completion time of $4Bn + 1$ must both begin and end with a node from C_{ph}.

In any feasible schedule for an instance of the register constraint problem created by the above transformation, every node of the place holding chain C_{ph} must be scheduled as soon as it is available. Suppose that there is a solution to the 3-partition problem and suppose that a_i, a_j, and a_k comprise one of the triples in the solution. We construct a schedule S with completion time $4Bn + 1$ for the corresponding instance of the register constraint problem by inserting a number chain node between each pair of nodes from C_{ph}. The nodes in S alternate between register and nonregister nodes, except for the last node from subchain C^i_{ph}, which is a nonregister node and is followed by a nonregister node from a number chain (Figure 6). An equivalent description is that all register nodes are both preceded and succeeded by a nonregister node, with the exception of the first register node of C^1_{ph}, which is the first node in S.

More specifically, schedule each of the B nodes in the first subchains of $C(a_i)$, $C(a_j)$, and $C(a_k)$—all of which are nonregister nodes—after each of the first B register nodes from C^1_{ph}. In a similar manner we schedule each of the B register nodes from the second subchains of $C(a_i)$, $C(a_j)$, and $C(a_k)$ after the first B nonregister nodes of C^1_{ph}. This does not increase the completion time of C_{ph}, since each edge in C_{ph} has latency 1. The process is repeated for each of the triples in the solution of the 3-partition problem, each

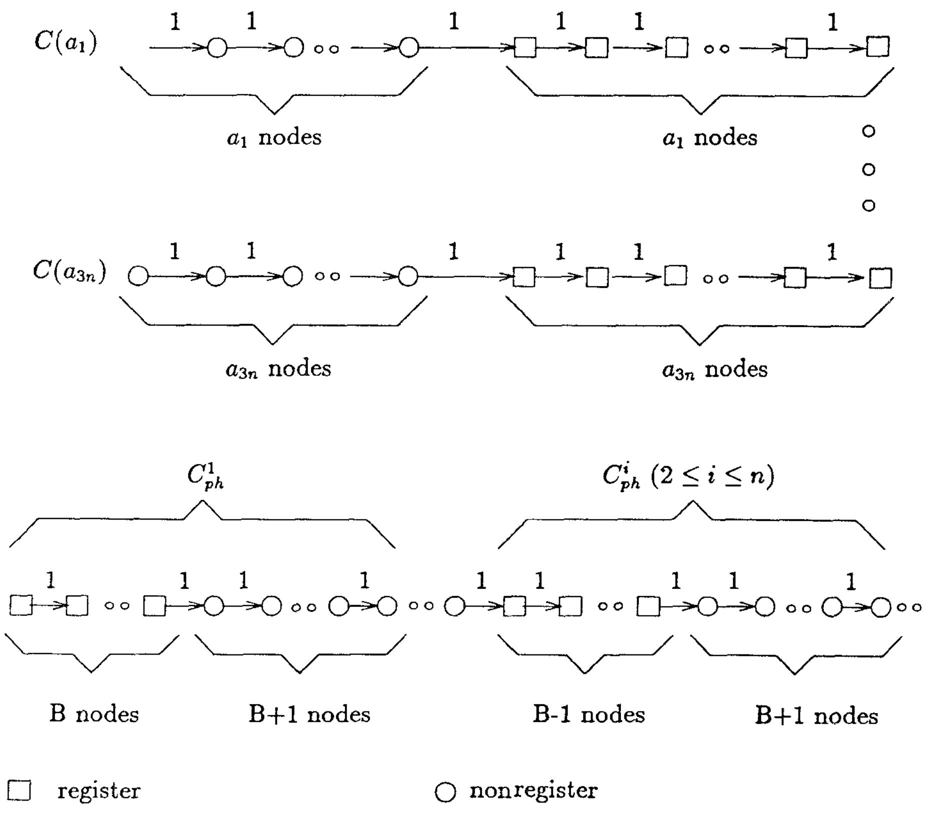


Fig. 5. The number and place-holding chains for the register-constraint problem.

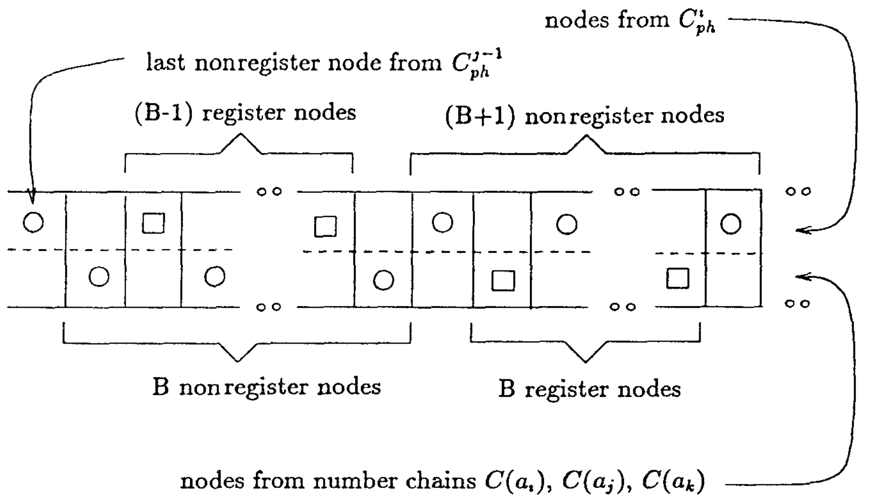


Fig. 6. Structure of a feasible schedule for the register-allocation problem.

time interleaving register nodes with nonregister nodes. There is no idle time in the schedule, and the completion time is $4Bn + 1$.

Conversely, suppose that we are given a schedule S for the multiple chain problem that completes by time $4Bn + 1$. This implies that S has no idle time and that all the nodes from the number chains are interleaved with nodes from C_{ph}. It also implies that the interleaving strictly alternates between register nodes and nonregister nodes, except at the boundaries, as described above.

Given schedule S, let $\overline{R}_i$ be the nodes that are either predecessors or successors of the register nodes of C^i_{ph} in S, and let R_i be the nodes that have nonregister nodes of C^i_{ph} as both their predecessor and successor nodes in S, $1 \leq i \leq n$.

Since, by assumption, S has no idle time, all nodes in $\overline{R}_i$ must be nonregister number chain nodes. By the construction of C_{ph}, $|\overline{R}_i| = B$. Consequently:

(*) All nodes in R_i must be register number chain nodes. Also, $|\overline{R}_i| = |R_i| = B$.

Lemma 6.2. *Each* $\overline{R}_i$, $1 \leq i \leq n$, *consists of all the nodes from the first subchains of precisely three number chains.*

Proof. First consider $\overline{R}_1$ and suppose that $\overline{R}_1$ contains nodes from only $C(a_i)$, $C(a_j)$, and $C(a_k)$. If $a_i + a_j + a_k > B$, then, without loss of generality, we can assume that $\overline{R}_1$ does not contain all the nonregister nodes from a_k. Since a register number chain node cannot be scheduled until all of the nonregister nodes from that number chain are scheduled, R_1 must contain some nonregister node. But this contradicts (*). Therefore, $a_i + a_j + a_k \leq B$. Using a similar argument, if $a_i + a_j + a_k < B$, then once again R_1 must contain some nonregister node. Therefore, $a_i + a_j + a_k = B$. A simple induction argument gives the lemma for $\overline{R}_i$. □

The theorem follows immediately from Lemma 6.2.

6.2 Completion Time Constraint

We now consider the case in which the target machine has a single processor and there is a shared resource required by each instruction upon its completion. (We ignore any other constraints such as register constraints). If two instructions complete simultaneously, we say that a *completion time hazard* occurs. The scheduling problem is to construct a minimum completion time schedule in which no two instructions complete simultaneously. For example, a completion time constraint exists if it might be necessary to access a bus upon the completion of the execution of an instruction. Since we are proving a negative result, we have assumed a particularly simple model for the target machine, so that the only form of pipelining that occurs is when the second cycle of a 2-cycle instruction overlaps with the first cycle of another 2-cycle instruction.

THEOREM 6.3. (The completion time constraint problem). *Let $G = (N, E)$ be a DAG that is a set of chains for which the latencies are only* 0 *or* 1. *The problem of determining if there is a single processor schedule for G having a completion time no greater than D, for some given D, in which no two instructions complete simultaneously is NP-complete.*

PROOF. Membership in NP is obvious. We show that the problem is NP-hard by reducing the 3-partition problem to the completion time constraint problem.

Given an instance of the 3-partition problem, we construct an instance of the completion time constraint problem. As in the previous reduction, for each a_i there is a corresponding chain, called a *number chain.* Each number chain $C(a_i)$ consists of $n + 1$ subchains called *subchain* $(i, 1)$, *subchain* $(i, 2), \dots,$ *subchain* $(i, n + 1)$, as shown in Figure 7.

Each subchain except for subchain $(i, n + 1)$ has a_i nodes, which are called *number nodes*. Subchain $(i, n + 1)$ has K nodes, where $K = Bn$; these nodes of subchain $(i, n + 1)$ are called *enforcer nodes*. All number nodes and enforcer nodes, except for the last node of subchain $(i, n + 1)$, have an outgoing edge with latency 1. Subchains (i, j) and $(i, j + 1)$, $1 \leq j \leq n$, are connected by a *connecting* node $u_{i,j}$ (see Figure 7). The outgoing edge connecting $u_{i,j}$ to subchain $(i, j + 1)$ has latency 0. There are Bn^2 number nodes, $3nK = 3Bn^2$ enforcer nodes, and $3n^2$ connecting nodes, giving a total of $4Bn^2 + 3n^2$ number chain nodes.

There is also a *place-holding* chain, C_{ph}, (see Figure 8) that consists of $2n$ subchains: $C_{ph}^1, C_{ph}^2, \dots, C_{ph}^{2n}$. Subchain C_{ph}^j contains $jB + 1$ *place-holder* nodes for $1 \leq j \leq n$ and $B(2n - j) + 3K + 1$ place-holder nodes for $n + 1 \leq j \leq 2n$. All the nodes in subchain C_{ph}^j, except for the last node of subchain C_{ph}^{2n}, have an outgoing edge with latency 1. Subchains C_{ph}^j and C_{ph}^{j+1}, $1 \leq j < 2n$, are connected by a *connecting node* v_j, the out-edge of which has latency 0, as shown in Figure 8. □

We now count the total number of nodes in C_{ph} and the minimum completion time for C_{ph} alone.

LEMMA 6.4. *There are* $4Bn^2 + 4n - 1$ *nodes in* C_{ph}. *Moreover, it takes at least* $8Bn^2 + 6n - 2$ *time units to schedule* C_{ph}.

PROOF. There are $(B + 1) + (2B + 1) + \cdots + (Bn + 1) = Bn(n + 1)/2 + n$ place-holder nodes in subchains $C_{ph}^1, C_{ph}^2, \dots, C_{ph}^n$. Also, there are $((n - 1)B + 3K + 1) + ((n - 2)B + 3K + 1) + \cdots + (B + 3K + 1) + (3K + 1) = n(n - 1)/2 + 3nK + n$ place-holder nodes in subchains $C_{ph}^{n+1}, C_{ph}^{n+2}, \dots, C_{ph}^{2n}$. This gives a total of $4Bn^2 + 2n$ (since $K = Bn$) place-holder nodes. Furthermore, there are $2n - 1$ connecting nodes in C_{ph}, giving a total of $4Bn^2 + 4n - 1$ nodes in C_{ph}. When C_{ph} is scheduled alone, each place-holder node, except for the last, one has a unit of idle time following it, resulting in $4Bn^2 + 2n - 1$ units of idle time. Summing nodes and idle time for C_{ph}, we get $8Bn^2 + 6n - 2$ as the minimum completion time for C_{ph}. □

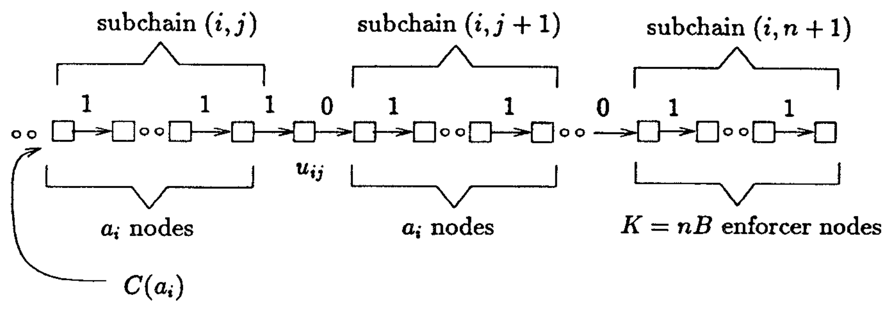


Fig. 7. The number chains for the completion time constraint problem.

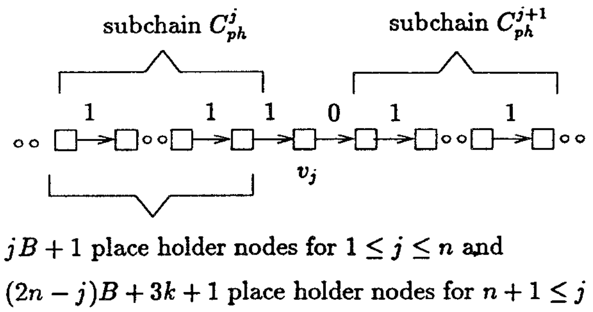

subchain C^j_{ph}
subchain C^{j+1}_{ph}
1
1
1
0
1
1
v_j
$jB + 1$ place holder nodes for $1 \le j \le n$ and
$(2n - j)B + 3k + 1$ place holder nodes for $n + 1 \le j \le 2n$

Fig. 8. The place-holding chains for the completion time constraint problem.

The total number of nodes in the input is $4Bn^2 + 3n^2 + 4Bn^2 + 4n - 1 = 8Bn^2 + 3n^2 + 4n - 1$. The intuition behind the proof is that a solution to the 3-partition problem allows an interleaving of nodes such that the only idle time in the optimal schedule is one unit for each connecting node of C_{ph}, giving a total of $2n - 1$ units of idle time and an overall completion time of $8Bn^2 + 3n^2 + 6n - 2$. Consequently, the 3-partition problem has a solution if and only if there is a schedule for the above multiple chain-scheduling problem with a completion time of $8Bn^2 + 3n^2 + 6n - 2$.

First, suppose that there is a solution to the 3-partition problem. We construct a schedule with a completion time of $8Bn^2 + 3n^2 + 6n - 2$, as follows. Given a schedule S, define *interval* i in S to be the set of nodes that are scheduled before v_i and are not in interval j for any $j < i$. In each interval, schedule the place-holder nodes from C_{ph} as soon as they become available.

Suppose that a_i, a_j, and a_k comprise one of the triples in the solution. Begin S with a node from C_{ph}, and schedule all the nodes in the subchains $(i, 1)$, $(j, 1)$, and $(k, 1)$ in interval 1 during the B units of idle time forced by the place-holder nodes of C^1_{ph} without introducing any additional idle time. After the last node from C^1_{ph} is scheduled, there is forced idle time of one unit, since the next node in C_{ph}, v_1, is a connecting node with a latency 0 out-edge. Schedule all the connecting nodes from $C(a_i)$, $C(a_j)$, and $C(a_k)$, respectively $u_{i,1}$, $u_{j,1}$, and $u_{k,1}$, immediately prior to the latency 0 connecting node v_1 of C_{ph}. In this fashion none of the number chain-connecting nodes introduces any additional idle time into the schedule, and all of the number chain-connecting nodes are in interval 1.

Next, schedule the nodes from subchains $(i,2)$, $(j,2)$, and $(k,2)$ in B of the $2B$ units of idle time of interval 2 provided by the place-holder nodes in C_{ph}^2. Also schedule the first subchains from another triple in the solution to the 3-partition problem in the remaining B idle slots of C_{ph}^2 (see Figure 9). These nodes are followed by the corresponding connecting nodes from the number chains, which are scheduled immediately before v_2, and consequently are in interval 2.

Repeat the above process for each of the triples in the solution of the 3-partition problem, each time interleaving number nodes with place-holder nodes from C_{ph}^i, $1 \leq i \leq n$. Since the number nodes are all scheduled during what would be idle time if only nodes from C_{ph} were scheduled, they do not increase the completion time of the schedule. (The connecting nodes increase the completion time, but that has been accounted for in the computation of the bound on the completion time.)

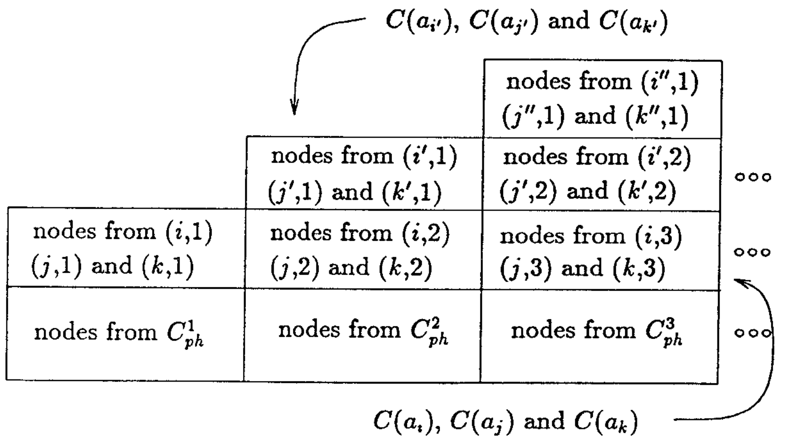
Subchain C_{ph}^{n+l}, $1 \leq l \leq n$, has $(n-l)B + 3k + 1$ units of idle time (slots) interleaved with its place-holder nodes. Schedule $3K$ slots of interval $n - l$ with the $3K$ enforcer nodes from those number chains $C(a_{i'})$, $C(a_{j'})$, and $C(a_{k'})$ whose first subchains $(i', 1)$, $(j', 1)$, and $(k', 1)$ are scheduled in the slots associated with C_{ph}^l. Schedule number nodes from appropriate subchains of the remaining $3(n-l)$ number chains in the $(n-l)B$ idle slots. (We assume that the chains can be partitioned into triples such that each triple corresponds to a solution in the 3-partition problem.) The connecting nodes are scheduled as before. The number of units of idle time produced by this strategy is exactly $2(n-1)$, one unit of idle time for each connecting node from C_{ph}. Therefore, the total length of the schedule is $8Bn^2 + 3n^2 + 6n - 2$.

Conversely, suppose that we are given a schedule S for the problem that completes by time $8Bn^2 + 3n^2 + 6n - 2$. It follows from previous arguments that S has precisely $2n - 1$ units of idle time. Therefore, each v_i has a set of number chain-connecting nodes that are scheduled together with v_i in consecutive time slots. Furthermore, there must be $3j$, $1 \leq j \leq n$, and $3(2n-j)$, $n < j < 2n$, number chain-connecting nodes scheduled in consecutive time slots together with v_j. Otherwise, there will be fewer than $3K$ enforcer nodes available during interval $n + q$ for some $q \leq n$. The details, which are left to the reader, are similar to those of Lemma 6.8 below.

We say that a unit of idle time on time step t_i is *associated* with interval i if the last node that precedes the idle time is from interval i. Since no two nodes can complete at the same time, we have the following:

LEMMA 6.5. *There is precisely one unit of idle time associated with each interval i, $1 \leq i < 2n$.*

The following lemma shows how S encodes the solution of the 3-partition problem.



$C(a_{i'})$, $C(a_{j'})$ and $C(a_{k'})$
nodes from $(i'',1)$ $(j'',1)$ and $(k'',1)$
nodes from $(i',1)$ $(j',1)$ and $(k',1)$
nodes from $(i',2)$ $(j',2)$ and $(k',2)$
nodes from $(i,1)$ $(j,1)$ and $(k,1)$
nodes from $(i,2)$ $(j,2)$ and $(k,2)$
nodes from $(i,3)$ $(j,3)$ and $(k,3)$
nodes from C_{ph}^1
nodes from C_{ph}^2
nodes from C_{ph}^3
$C(a_i)$, $C(a_j)$ and $C(a_k)$

Fig. 9. Structure of a feasible schedule for the completion time constraint problem.

LEMMA 6.6. *Interval q, $1 \le q \le n$, contains all the nodes from the following subchains of the number chains: (i_1, q), (j_1, q), (k_1, q), $(i_2, q-1)$, $(j_2, q-1)$, $(k_2, q-1), \ldots, (i_q, 1)$, $(j_q, 1)$, $(k_q, 1)$. Moreover, $a_{i_1} + a_{j_1} + a_{k_1} = a_{i_2} + a_{j_2} + a_{k_2} = \cdots = a_{i_q} + a_{j_q} + a_{k_q} = B$.*

PROOF. The proof follows from Lemma 6.5 and the assumption that S completes by time $8Bn^2 + 3n^2 + 6n - 2$. The theorem now follows. □

6.3 Arbitrary Latencies

Hennessy and Gross [17] have shown that the instruction-scheduling problem is NP-complete for arbitrary DAGs with latency constraints of 0 and two other values. We now prove that even if the DAG is restricted to a set of chains, the problem remains NP-complete.

THEOREM 6.7. *Let $G = (N, E)$ be a DAG that is a set of chains for which the latencies can be 0 or two other values. The problem of determining if there is a single processor schedule for G having a completion time no greater than D, for some given D, is NP-complete.*

PROOF. Membership in NP is obvious. We show that the problem is NP-hard by reducing the 3-partition problem to an instance of the arbitrary latencies problem.

Given an instance of the 3-partition problem, we construct an instance of the arbitrary latencies problem. This construction is illustrated in Figure 10. (Note that 0, B, and $(2n - 1)B$ are the latencies in this instance.)

For each a_i there is a corresponding chain, called a *number chain*. Each number chain $C(a_i)$ consists of two subchains called the *first subchain* and the *second subchain*, each containing a_i nodes linked by edges with latency 0. The first and the second subchains are in turn linked together by a latency $(2n - 1)B$ edge that is called a *separator edge*. There is also a *place-holding chain*, C_{ph}, that consists of $2n$ subchains $C_{ph}^1, \ldots, C_{ph}^{2n}$, with each containing

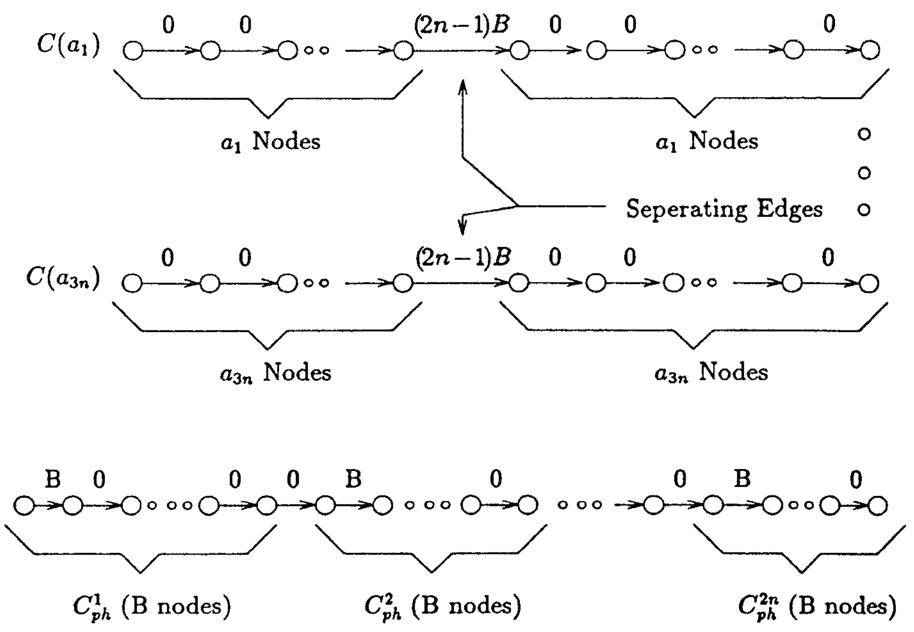


Fig. 10. Structure of a feasible schedule for the arbitrary latencies problem.

B nodes. The first node in each subchain C^j_{ph} is called a *partition* node and is followed by a latency B edge; all the remaining $B - 2$ edges in the subchain C^j_{ph} are latency 0 edges. These $2n$ subchains are connected in series by latency 0 edges to form C_{ph} (see Figure 10). If the first node of C_{ph} is started at time 0, the earliest completion time for the entire chain is $4Bn$; we use this value for the completion time, that is, $D = 4Bn$. To prove Theorem 6.7, we show that the 3-partition problem has a solution if and only if there is a schedule for the above multiple chain-scheduling problem with a completion time of $4Bn$.

First suppose that there is a solution to the 3-partition problem and suppose that a_i, a_j, and a_k comprise one of the triples in the solution. Clearly, we can schedule all the nodes in the first subchains of $C(a_i)$, $C(a_j)$, and $C(a_k)$ immediately after the partition node of C^1_{ph}. The latency (of $2Bn - B$ units) on the separator edges allows us to schedule the nodes in the second subchains of $C(a_i)$, $C(a_j)$, and $C(a_k)$ in the interval immediately following the partition node of C^{n+1}_{ph}. We repeat the above process for each of the triples in the solution of the 3-partition problem, each time scheduling the beginning of the second subchain $2Bn$ time units later than the beginning of the corresponding first subchain. Because all of the nodes are scheduled and there is no idle time in the schedule, the completion time of the schedule is $4Bn$.

Conversely, suppose that we are given an instance of the arbitrary latencies problem that has been constructed from an instance of 3-partition using the above transformation. Let S be a schedule for that instance of the arbitrary latencies problem, and suppose that S completes by time $4Bn$. This implies that all the nodes from the number chains are scheduled during the latency or idle time of B units after C^j_{ph}, $1 \le j \le 2n$. The following lemma completes the proof.

LEMMA 6.8. *There are nodes from precisely three of the number chains scheduled in S between the partition nodes of C^i_{ph} and C^{i+1}_{ph}, for $1 \le i \le n$. Furthermore, these nodes consist of all the nodes in the corresponding first subchains.*

PROOF. Suppose that the lemma does not hold. The schedule must begin with the first node of C_{ph}, since otherwise it would complete at some time later than $4Bn$. Furthermore, each node from C_{ph} must be scheduled in S as soon as it becomes available. Because of the latency of $(2n - 1)B$ units on the separator edges (together with the unit running time of all nodes), all the first subchain nodes must be scheduled before any of the second subchain nodes in any schedule with completion time of $4Bn$.

Let p_i be the partition node in subchain C^i_{ph}, $1 \le i \le 2n$. Without loss of generality, we consider the B units of idle time in C_{ph} between nodes p_1 and p_2. Clearly, nodes from at least three number chains must be scheduled between p_1 and p_2. Otherwise, since $a_i < B/2$ for all i, there must be at least one unit of idle time between p_1 and p_2 in S. Suppose that nodes from three number chains $C(a_i)$, $C(a_j)$, and $C(a_k)$ are scheduled after p_1, but that not all the nodes of the first subchain of $C(a_k)$ are scheduled before p_2. Because of the latency of the separator edge, this implies that nodes from the second subchain of $C(a_k)$ cannot be scheduled until after p_{n+2} has been scheduled. Since $a_i + a_j < B$, this implies that there will be idle time in S between p_{n+1} and p_{n+2}, which contradicts the assumption that S has a completion time of $4Bn$.

Now suppose that nodes from more than three subchains of the number chains are scheduled between nodes p_1 and p_2 in S. Then, no more than three of these subchains, say a_i, a_j, and a_k, are completely scheduled prior to the scheduling of p_2. Moreover, it must be the case that $a_i + a_j + a_k < B$. Therefore, there will be idle time between p_{n+1} and p_{n+2}.

Consequently, the lemma holds and the theorem follows. □

ACKNOWLEDGMENT

We thank Dave Alpern, Paul Hudak, Mary Van Deusen, Mark Wegman, Kenny Zadeck, and Kevin McAuliffe for helpful discussions and suggestions. We are also grateful to David Bernstein for pointing out an error in an earlier draft and for several useful comments. We thank the anonymous referees for constructive comments and observations.

REFERENCES

1. AIKEN, A., AND NICOLAU, A. Loop quantization: An analysis and algorithm. Tech. Rep. 87-821, Cornell Univ., March, 1987.
2. ALLEN, F., ROSEN, B, AND ZADECK, K., ED. Optimization in compilers. To appear.
3. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction* (1982), 22–31.

4. Bernstein, D., and Gertner, I. Schedulıng expressions on a pıpelined processor wıth a maxımal delay of one cycle. *ACM Trans. Programm. Lang. Syst. 11*, 1 (Jan 1989), 57–66
5. Bernstein, D., Rodeh, M., and Gertner, I. Approximation algorithms for scheduling arithmetic expressions on pıpelined machines. *J. Algorithms 10* (March 1989), 120–139.
6. Bruno, J., Jones, J. W. III, and So, K. Determınistic scheduling wıth pipelınes processors. *IEEE Trans. Comput. C-29* (1980), 308–316.
7. Dave, A. Code generation for the Intel 80860. M.S. thesis, Indian Institute of Science, Bangalore, India, 1991.
8. Ellis, J. *Bulldog: A Compiler for VLIW Archıtectures*. The MIT Press, Cambridge, Mass., 1986
9. Fisher, J. A. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30*, 7 (July 1981), 478–490.
10. Gabow, H. Scheduling UET systems on two uniform processors and length-two pipelines. *SIAM J. Comput. 17* (1988), 810–829.
11. Garey, M. R., and Johnson, D. S. *Computers and Intractabılıty* W. H. Freeman, San Francisco, 1979.
12. Goodman, J., and Hsu, W-C. Code scheduling and register allocation in large basıc blocks. In *Proceedıngs of the Internatıonal Conference on Supercomputing* (July 1988), ACM Press, New York, 442–452.
13. Gibbons, P., and Muchnick, S Efficient instruction scheduling for pipelined architecture. In *Proceedıngs of the ACM Symposium on Compıler Constructıon* (Palo Alto, Calıf., June 1986), 11-16.
14. Graham, R. L. Bounds for certain multiprocessor anomalıes. *Bell Syst. Tech. J. 45* (1966), 1563–1581.
15. Gross, T. Personal communıcation.
16. Hennessy, J., and Gross, T. Code generation and reorganization in the presence of pipeline constraints. In *Proceedings of the 9th Annual ACM Symposıum on Prıncıples of Programming Languages* (Albuquerque, N.M., Jan. 1982), 120–127.
17. Hennessy, J. and Gross, T. Postpass code optımization of pipeline constraints *ACM Trans. Program. Lang. Syst.* 5 (1983), 422–448.
18. Hennessy, J. Jouppi, N. Gill, J., Baskett, F., Strong, A., Gross, T., Rowen, C., and Leonard, J. The MIPS machine. In *Proceedıngs IEEE Coupcon* (San Francisco Feb. 1982), 2–7.
19. Katevenis, M. *Reduced Instructıon Set Computer Archıtecture for VLSI.* MIT Press, Cambridge, Mass., 1984.
20 Lawler, E. Lenstra, J. K., Martel, C., Simons, B., and Stockmeyer, L. Pipeline scheduling: A survey. Tech. Rep. RJ 5738, IBM Research Div., San Jose, Calif., 1987.
21. Leung, J. Y.-T., Vornberger, O., and Witthoff, J. On some variants of the bandwidth minimızation problem. *SIAM J. Comput. 13* (1984), 650–667
22. Nicolau, A. Loop quantization or unwinding done right. In *Proceedings of the 1st International Conference on Supercomputıng Lecture Notes in Computer Science*, June, 1987, Springer Verlag, New York, 294–308.
23. Palem, K. On the complexity of precedence constrained scheduling. CS-TR-86-10, Dept. of Computer Sciences, Univ. of Texas, Austin, 1986.
24. Papadimitriou, C., and Yannakakis, M. Scheduling interval-ordered tasks. *SIAM J. Comput. 8* (1979), 405–409.
25. Patterson, D. Reduced instruction set computers. *Commun. ACM 28* (1985), 8–21
26. Radin, G. The 801 minıcomputer. *IBM J. Res. Dev. 27* (1983), 237–246.
27. Simpson, R. The IBM RT personal computer. *Byte*, Extra Edition (1986), 43–78
28. *The SPARC(tm) Archıtecture Manual.* Ver. 7, SUN Micro Systems, 1987
29. Tarjan, R. E. Efficiency of a good but not linear set unıon algorithm *J ACM 22* (April, 1975), 215–225.
30. Warren, H. Instruction scheduling for the IBM RISC System/6K processor. *IBM J. Res. Dev.* (1990), 85–92.

Received February 1991; revised August 1992; accepted October 1992