SCHEDULING TASKS WITH AND/OR PRECEDENCE CONSTRAINTS[*]

D. W. GILLIES[†] AND J. W.-S. LIU[‡]

Abstract. In traditional precedence-constrained scheduling a task is ready to execute when all its predecessors are complete. We call such a task an AND task. In this paper we allow certain tasks to be ready when just one of their predecessors is complete. These tasks are known as OR tasks. We analyze the complexity of two types of real-time AND/OR task scheduling problems. In the first type of problem, all the predecessors of every OR task must eventually be completed, but in the second type of problem, some OR predecessors may be left unscheduled. We show that most problems involving tasks with individual deadlines are NP-complete, and then present two priority-driven heuristic algorithms to minimize completion time on a multiprocessor. These algorithms provide the same level of worst-case performance as some previous priority-driven algorithms for scheduling AND-only task systems.

Key words. non-preemptive scheduling, list scheduling, minimal length schedules, algorithm analysis, multiprocessor systems, NP-complete problems.

AMS subject classifications. 68M20, 68Q25, 90B35, 90C90.

1. Introduction.

In the traditional model of real-time workloads, dependencies between tasks are represented by partial orders known as precedence constraints. Each task may have several predecessors and may not begin execution until all its predecessors are completed. We call such tasks AND tasks, and the partial order over them is known as AND-only precedence constraints. This traditional model fails to describe many real-time applications encountered in practice. In these applications a task may begin execution when some but not all of its predecessors are completed. We call such a task an OR task. The resulting task system, containing both AND and OR tasks, is said to have AND/OR precedence constraints.

In this paper we are concerned with how to schedule tasks with AND/OR precedence constraints to meet deadlines. We investigate two variants of this problem, called the unskipped and the skipped variants.

In some applications all the predecessors of an OR task must eventually be completed, that is, they cannot be skipped. We call the model for this type of application the *AND/OR/unskipped* model. For example, in robotic assembly [1], one out of four bolts may secure an engine head well enough to allow further work on other parts of the engine head. However, the remaining three bolts must eventually be installed. The unskipped variant also models tasks that share resources. A task may need a resource from one of several predecessors in order to execute and hence is ready to execute when any one predecessor is complete. Such a task can be modeled as an OR task. Again, the other predecessors must eventually be completed. The *AND/OR/unskipped* problem also arises in hard real-time scheduling when the precedence constraints are too strict for tasks to meet their deadlines. By relaxing the precedence constraints of some tasks, and restructuring the application code to accommodate the relaxed constraints, it may be possible for the tasks to meet their deadlines.

[*] This research was supported by the Office of Naval Research (Contracts No. NVY N00014-89-J-1181 and NVY N00014-92-J-1146)

[†] Department of Electrical Engineering, University of British Columbia.

[‡] Department of Computer Science, University of Illinois.

In other applications some predecessors of an OR task may be skipped entirely. We call this the AND/OR/skipped model. One example can be found in the problem of instruction scheduling on superscalar, MIMD, or VLIW processors. On such processors, several different instruction sequences may be used to compute the same arithmetic expression. These different sequences arise from algebraic laws such as associativity and distributivity. Only one sequence needs to be executed, and the other sequences may be skipped. Another application that can be characterized by this model is manufacturing planning [5] because certain manufacturing steps obey associative and distributive algebraic laws. The AND/OR/skipped problem also arises in hard real-time scheduling. When there is insufficient time for a task system to meet its deadlines, we may convert appropriate tasks to imprecise computations [3], which may be modeled as OR tasks whose predecessors may be skipped.

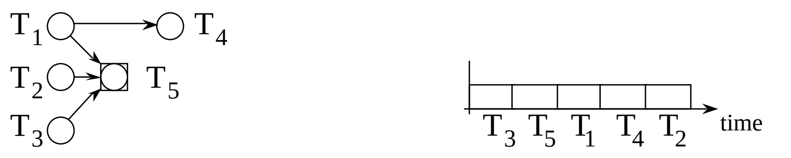
We are concerned with ways to schedule AND/OR precedence-constrained tasks to meet deadlines or to minimize completion time. Most of these problems are generalizations of traditional deterministic scheduling problems that are NP-hard. In this paper we analyze the complexity of the problems that are not known to be NP-hard. For two problems that are known to be NP-hard, we give heuristic algorithms to minimize completion time. The algorithms have small running time and good worst-case performance.

Our work is related to some previous work on deterministic scheduling to meet deadlines [6] [8] and to minimize completion time [9] [10] [13] [14]. We were inspired by an AND/OR model that was proposed as a means of modeling distributed systems for real-time control [18]. Two recent systems incorporated AND/OR precedence constraints of some sort in their implementation [16] [19].

The remainder of this paper is organized as follows. Section 2 describes our assumptions about the AND/OR scheduling problem and introduces the terminology used in later sections. Section 3 investigates the unskipped problem with multiple deadlines and analyzes an algorithm to minimize completion time. In Section 4, we investigate the skipped problem and give a second algorithm to minimize completion time. Section 5 draws conclusions and discusses future work. The appendix contains proofs of the theorems stated in Sections 3 and 4.

2. The AND/OR Model. All the scheduling problems considered here are variants of the following problem. There are m identical processors and a set of tasks $\mathbf{T} = \{T_1, T_2, ..., T_n\}$. Each task T_i must execute on one processor for p_i units of time and is said to have *processing time* p_i. There is a partial order $<$ defined over $\mathbf{T}$. If $T_i < T_j$, then T_i is a *predecessor* of T_j, and T_j is a *successor* of T_i. The task T_i is a *direct predecessor* of T_j if there is no T_k such that $T_i < T_k < T_j$. The task T_j is an *AND* task if its execution may begin only after all its direct predecessors have completed. The task T_j is an *OR* task if its execution may begin after only one of its direct predecessors has completed. The partial order $<$ is an *in-forest* if whenever $T_k < T_i$ and $T_k < T_j$, we have either $T_i < T_j$ or $T_j < T_i$; the partial order $<$ is an *in-tree* if it has a unique element with no successors. A task followed by a series of direct successors $T_{i_1} < T_{i_2} < \cdots$ is called a *task chain.*

The partial order is also represented by a weighted and transitively reduced directed graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$ called the *task graph.* In this graph there is a vertex T_i for every task in the set $\mathbf{T}$. The set $\mathbf{A}$ is known as the *set of arcs.* If T_i is a direct predecessor of T_j in the partial order then $(T_i, T_j) \in \mathbf{A}$. The set $\mathbf{P} = \{p_1, \cdots, p_n\}$ denotes the set of processing times. A task graph together with a set of deadlines $\mathbf{D} = \{d_1, \cdots, d_n\}$ is a 2-tuple $(\mathbf{G}, \mathbf{D})$ that characterizes a scheduling problem; it is



(a) *Task system* (b) *AND/OR/Unskipped schedule*

FIG. 1. *Sample problem and solution.*

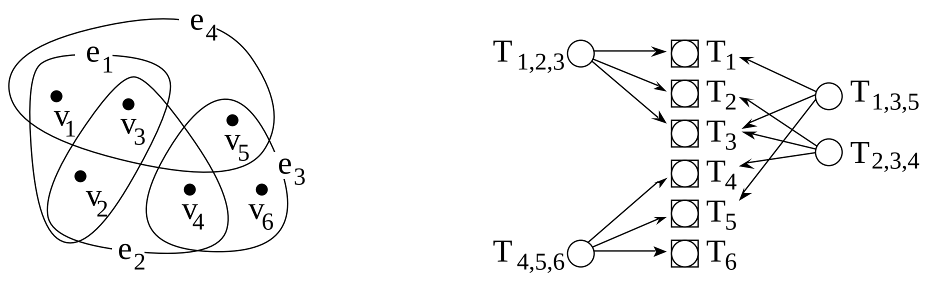
called a *task system*. When several graphs $\mathbf{G_1}, \mathbf{G_2}, \cdots$ are present, the functions $T(\mathbf{G_i})$, $A(\mathbf{G_i})$, and $P(\mathbf{G_i})$ will be used to extract the sets $\mathbf{T}$, $\mathbf{A}$, and $\mathbf{P}$ from the graph $\mathbf{G_i}$.

Let $S(\mathbf{G}, T_i) = \{T_j | (T_i, T_j) \in A(\mathbf{G}), T_i \in T(\mathbf{G})\}$ denote the set of direct successors of T_i, and let $P(\mathbf{G}, T_i) = \{T_j | (T_j, T_i) \in A(\mathbf{G}), T_i \in T(\mathbf{G})\}$ denote the set of direct predecessors of T_i. Let $L(\mathbf{G}, T_j)$ be the length of the longest directed path in $\mathbf{G}$ ending at T_j. More precisely, $L(\mathbf{G}, T_j) = p_j$ if T_j has no predecessors in $\mathbf{G}$, and $L(\mathbf{G}, T_j) = p_j + \max_k\{L(\mathbf{G}, T_k) | (T_k, T_j) \in A(\mathbf{G})\}$ if T_j has predecessors. Let $L^*(\mathbf{G}) = \max\{L(\mathbf{G}, T_j) | T_j \in T(\mathbf{G})\}$ be the length of the longest directed path in a graph $\mathbf{G}$. Let $E(\mathbf{G}, T_i) = \sum_{T_j < T_i \text{ in } \mathbf{G}} p_j$ denote the total processing time of all the predecessors of T_i in $\mathbf{G}$. Let $E^*(\mathbf{G}) = \sum p_i - L^*(\mathbf{G})$ denote the "residual" processing time of an AND-only graph, i.e. the total processing time minus the processing time of the tasks on the longest chain. Later it will be shown that AND-only graphs with minimal $L^*(\mathbf{G})$ and $E^*(\mathbf{G})$ can be used to produce near-optimal priority-driven schedules.

All the tasks with no successors in a task graph are classified as *essential*; this means that they must appear in a valid schedule. If an AND task is essential, then all its direct predecessors are essential. If an OR task T_j is essential, then the scheduling algorithm must choose one direct predecessor T_i to be essential and the precedence constraint $T_i < T_j$ must be obeyed in scheduling the task system. If a task is not classified as essential, then it is *inessential.* We distinguish between two problems referred to as skipped and unskipped problems, respectively. In a skipped scheduling problem, inessential tasks may be left unexecuted. However, in an unskipped problem, inessential tasks must be executed.

Figure 1(a) depicts an AND/OR task system. In the figure AND tasks are depicted by circles and OR tasks are depicted by circles within boxes. Tasks are generally labeled by their *name*, or by their (*name*, *length*), so (T_5, e) would indicate that task T_5 requires e units of processing time. Where necessary, deadlines will be written separately, next to the associated tasks. If the deadlines are omitted from a figure, then the reader should assume that all the deadlines are identical. Every task in this example has a processing time of one and all the tasks have the same deadline, hence, the lengths and deadlines are omitted from this figure. Figure 1(b) depicts a schedule in which T_3 is an essential task, and T_2 is an inessential task. Figure 1(b) shows a schedule of the unskipped task graph from Figure 1(a). If Figure 1(a) were a skipped task graph, then a skipped schedule could be obtained by deleting T_2 from the end of the schedule in Figure 1(b).

The scheduling algorithms in this paper are simple heuristics that never intentionally leave processors idle. These algorithms are known as *priority-driven* or *list-scheduling* algorithms. Whenever a processor is available, a list-scheduling algorithm schedules the ready task with the highest priority according to a priority list. Because



(a) *Exact 3-cover problem* (b) *AND/OR task system*

FIG. 2. *Exact 3-cover transformation.*

they try to make the best local choice at each scheduling decision point, list-scheduling algorithms are also called greedy algorithms. A schedule produced by a list-scheduling algorithm is known as a *list schedule* and the time at which all the tasks in **T** are complete is the *length* of the schedule.

We assume that every task in **T** has ready time equal to zero, thus, an OR task may begin execution as soon as an essential predecessor is completed. In some situations each task T_i has a deadline d_i ; T_i must be completed at or before time d_i. A schedule is called *feasible* if every task completes by its deadline. A task system that has a feasible schedule is called *feasible*. Given a task system our objective is to find a feasible schedule or determine that no feasible schedule exists.

In other situations all the tasks share a common deadline. The problem of finding a feasible schedule in these situations is equivalent to the problem of minimizing the overall completion time, i.e. the time at which the last task completes.

3. Unskipped problems. In this section we discuss the complexity of the AND/OR/unskipped scheduling problem. After showing that most natural problems with deadlines are NP-complete on a single processor, we present a priority-driven heuristic to minimize completion time on m processors. We then explain why no priority-driven heuristic can provide a better worst-case performance bound than the one presented here.

3.1. Scheduling to Meet Deadlines on a Single Processor. There are well-known polynomial-time algorithms [6] [8] for scheduling tasks with AND-only precedence constraints, identical processing times, and arbitrary deadlines on one or two processors. It is natural to ask whether the corresponding AND/OR scheduling problems may be solved in polynomial time. Unfortunately, this extended problem is NP-complete, even when all the deadlines are the same. This fact is expressed in the following theorem.

THEOREM 3.1. *The problem of AND/OR skipped or unskipped scheduling of a task system in which all the OR tasks must meet a common deadline is NP-complete.*

Proof. It suffices to prove that the problem is NP-complete on a single processor. The proof is based on a reduction from exact 3-cover (X3C). Given a hypergraph $H = (V, E)$ of $3n$ vertices and a set of hyper-edges, each of which is incident to three vertices, the problem is to find a set of exactly n edges that covers all the vertices with no overlap. This problem is NP-complete [7].

The exact 3-cover problem can be transformed into an AND/OR scheduling problem as follows. Create a task system $(\mathbf{G}, \mathbf{D})$ composed entirely of unit processing-time tasks. There is an OR task T_i in the task system for each hypergraph vertex v_i in H. In the task system all $3n$ OR tasks have deadline $4n$. Create an AND task $T_{i,j,k}$ for each hyper-edge that connects v_i, v_j, and v_k. The successors of task $T_{i,j,k}$ are the OR tasks T_i, T_j, and T_k. Figure 2 is an example of this transformation. Now we ask if there exists a schedule in which every OR task meets its deadline. Clearly, if the given hypergraph H has an exact 3-cover, n AND tasks corresponding to the cover may execute in the time interval $[0, n]$, thereby allowing all $3n$ OR tasks to complete by time $4n$. If no such cover exists, then at least $n+1$ edges must be used to cover the hypergraph. Hence at least $n+1+3n$ time units must elapse before all the OR tasks are complete regardless of whether this a skipped or an unskipped problem. Thus, if a scheduler produces a feasible schedule, then there is an exact 3-cover, and if the scheduler fails, then no such cover exists. □

The proof of Theorem 3.1 indicates that this scheduling problem is at least as hard as the n-dimensional cover problem, a generalized version of n-dimensional matching. About thirty years ago, T. C. Hu gave a polynomial-time algorithm to schedule an AND-only task system with in-tree precedence constraints on m processors [14]. Thus, there is some hope that if we restrict the AND/OR/unskipped task system to have in-tree precedence constraints, there may exist a polynomial-time algorithm. Unfortunately, the following theorem shows that this AND/OR scheduling problem is NP-complete.

THEOREM 3.2. *The problem of AND/OR/unskipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines, and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

COROLLARY 3.3. *The problem remains NP-complete for task systems in which only the OR tasks have deadlines.*

Proof. The proof is contained in the appendix. □

The proofs of Theorems 3.2 and Corollary 3.1 in the appendix make use of long chains of AND tasks with differing deadlines. We now consider a class of task systems where only two tasks in a chain may have deadlines. In a *simple in-forest*, (1) each in-tree consists of an OR task with a deadline, no successors, and two direct predecessors, and (2) each direct predecessor of an OR task has a deadline and is the root of an in-tree of AND tasks with no deadlines (i.e. the deadlines are infinite). A simple in-forest restricts the allowable precedence constraints and allowable tasks with deadlines in a task system. We have found no simpler non-trivial combination of precedence constraints and deadlines. Surprisingly, even this simplified AND/OR scheduling problem is NP-complete

THEOREM 3.4. *The problem of AND/OR/unskipped scheduling to meet deadlines, where the task system is a simple in-forest with identical processing times, is NP-complete.*

Proof. The proof may be found in [11]. □

Theorems 3.1-3.3 allow us to arrive at the following conclusion. Every AND/OR task graph with k OR tasks, each of which has l direct predecessors, corresponds to a set of l^k different AND-only task graphs. A feasible schedule of such a task system corresponds to an implicit selection of one of these l^k AND-only task graphs. Therefore, when there are $O(\log n)$ OR tasks in the AND/OR task system, it is possible to enumerate in polynomial time the set of all possible AND-only task graphs and apply an optimal AND-only scheduling algorithm such as the one described in [8]. On the other hand, Theorems 3.1-3.3 show that many natural scheduling problems with $O(n)$

TABLE 1
Complexity of AND/OR/unskipped problems.

(a) Scheduling to meet deadlines with identical processing times on 1 processor.

Deadline Location	General Graph 2 Deadlines	In-Tree $O(n)$ Deadlines	Simple In-Forest
On All Tasks	NP-C (Theorem 3.1)	NP-C (Theorem 3.2)	NP-C (Theorem 3.3)
On OR Tasks Only	NP-C (Theorem 3.1)	NP-C (Corollary 3.1)	Trivial

(b) Scheduling to minimize completion time on m processors.

Task Processing Time	General Graph	In-Tree
Identical	NP-C [15] for AND-only	NP-C (Theorem 3.4)
Arbitrary	Minimum-Path Heuristic	Minimum-Path Heuristic

```
Input: Task graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$
Step 1: For each OR task $T_i$ with no OR predecessors:
    (a) Let $T_k$ be a direct predecessor of $T_i$ that minimizes the longest path ending at $T_k$.
        In other words, $T_k \in P(\mathbf{G}, T_i)$ and for all $T_j \in P(\mathbf{G}, T_i)$ with $j \neq k$, $L(\mathbf{G}, T_j) \geq L(\mathbf{G}, T_k)$.
    (b) Convert $T_i$ into an AND task whose only direct predecessor is $T_k$ .
Step 2: The resulting task system has only AND tasks. Schedule this task system using a
    priority-driven algorithm and an arbitrary priority list.
```

FIG. 3. *The minimum path heuristic for general graphs.*

OR tasks are NP-complete. It follows that the complexity of the AND/OR/unskipped problem is determined almost exclusively by the number of OR tasks in the task system and the complexity of the corresponding AND-only scheduling problem. These results are summarized in Table 1(a).

It appears difficult to design a priority-driven scheduling heuristic with good worst-case performance. For the simple problem studied in Theorem 3.3, we have produced examples to show that any algorithm that only considers slacks between deadlines and non-deadline information, one isolated in-tree at a time, may perform $\sqrt{n}$ times worse than an optimal algorithm. Some obvious priority-driven scheduling algorithms such as fewest predecessors first, least slack first, and some generalizations of the algorithms in [4] neglect to compare the deadlines among different in-trees. In the worst case these algorithms may meet only $\sqrt{n}$ deadlines when it is possible to meet n out of $n+1$ deadlines. For more information the reader is referred to [11] [12].

3.2. Scheduling to Minimize Completion Time. We now consider the problem of scheduling AND/OR/unskipped tasks with arbitrary processing times on m processors to meet a common deadline. This problem is equivalent to that of scheduling to minimize the overall completion time. Ullman has shown this problem to be NP-complete [15] for AND-only task systems where all the tasks have identical processing times. However, Hu's algorithm solves this problem in polynomial time for in-tree precedence constraints. Unfortunately, the problem becomes NP-complete when OR tasks are allowed.

THEOREM 3.5. *The problem of scheduling an AND/OR/unskipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

In Figure 3, we present a heuristic that minimizes the completion time of an

AND/OR/unskipped task system with arbitrary processing times. The basic idea is to choose an AND-only graph that minimizes the longest path in $\mathbf{G}$. The heuristic can be implemented to run in time $O(n + |\mathbf{A}|)$ by reversing the direction of the arcs in $\mathbf{G}$ and employing depth-first search. Let $\mathbf{G_o} = (\mathbf{T_o}, \mathbf{A_o}, \mathbf{P_o})$ and W_o denote the implicit AND-only graph and the completion time of the task system according to an optimal schedule. Let $\mathbf{G}' = (\mathbf{T}', \mathbf{A}', \mathbf{P}')$ and W' denote the implicit AND-only graph and the completion time of the task system according to a schedule produced by the Minimum Path Heuristic, respectively. The worst-case performance of the Minimum Path Heuristic depends on the following lemma.

LEMMA 3.6. *$L^*(\mathbf{G}') \leq L^*(\mathbf{G_o})$.*

Proof. Let $H = \{T_i | P(\mathbf{G}', T_i) \neq P(\mathbf{G_o}, T_i)\}$ denote the set of tasks whose predecessors differ between the optimal graph and the graph produced in Step 1 of the Minimum Path Heuristic. If $H = \emptyset$, then the AND-only task graphs are identical and the lemma is established. Otherwise, let $T_i \in H$ be a task for which there exists no $T_j \in H$ with $T_j < T_i$ in $\mathbf{G_o}$. By the construction of $\mathbf{G}'$, $|P(\mathbf{G}', T_i)| = |P(\mathbf{G_o}, T_i)| = 1$. We change $\mathbf{A_o}$, replacing the arc $(P(\mathbf{G_o}, T_i), T_i)$ by $(P(\mathbf{G}', T_i), T_i)$ and obtain no increase in the longest path (by steps 1(a) and 1(b) of the heuristic). This argument is used inductively to transform $\mathbf{G_o}$ into $\mathbf{G}'$ with no increase in the maximum path length. This establishes the lemma. □

The following fact is proved in the well-known paper [13].

LEMMA 3.7. *In any priority-driven schedule, there is a chain of tasks that executes during all the idle periods (when one or more processors are not in use), and this chain is not longer than the completion time of an optimal schedule.*

If W_p denotes the total length of all the idle periods in a schedule produced by the Minimum Path Heuristic, then $W_p \leq L^*(\mathbf{G}') \leq L^*(\mathbf{G_o}) \leq W_o$ by Lemmas 3.1 and 3.2.

THEOREM 3.8. *The worst-case performance of the Minimum Path Heuristic is given by $W'/W_o \leq 2 - 1/m$. Moreover, this bound is tight.*

Proof. Let W_b denote the total length of all the busy periods in a priority-driven schedule. Let W_p denote the total length of all the idle periods in a priority-driven schedule. During the idle periods at least 1 and no more than $m - 1$ tasks execute, and during the busy periods exactly m tasks execute. It should be clear that $W' = W_p + W_b$. Hence, the worst-case completion time of this heuristic may be formulated as a linear program:

$$\begin{aligned} &\text{Maximize} \quad W_p + W_b = W' \\ &\text{subject to} \quad W_p \leq L^*(\mathbf{G}') \;\leq\; L^*(\mathbf{G_o}) \;\leq\; W_o \\ &\qquad mW_b + 1W_p \leq mW_o \end{aligned}$$

Solving the program yields $W_p = W_o$, $W_b = (1 - 1/m)W_o$, i.e. $W'/W_o \leq 2 - 1/m$. □

Examples of AND-only task systems that achieve this bound may be found in [2] and [10]. It is known [10] that no AND-only priority-driven heuristic can avoid $2-1/m$ worst-case performance (because priority-driven heuristics never intentionally idle the processor, and sometimes intentional idling is needed). Our priority-driven heuristic will schedule AND-only task systems as a special case. Hence, it is not possible to get better worst-case performance from an AND/OR scheduling algorithm without a better AND-only scheduling algorithm. In fact, it has been a long-standing open problem to find a better AND-only scheduling algorithm [15].

TABLE 2
Complexity of AND/OR/skipped problems

(a) Scheduling to meet deadlines with identical processing times on 1 processor.

Deadlines Location	General Graph 1 Deadline	In-Tree O(n) Deadlines	Simple In-Forest
On All Tasks	NP-C (Theorem 3.1)	NP-C (Theorem 4.1)	NP-C (Theorem 4.2)
ON OR Tasks Only	NP-C (Theorem 3.1)	NP-C (Theorem 4.1)	[17] Algorithm

(b) Scheduling to minimize completion time on m processors.

Task Processing Time	General Graph	In-Tree
Identical	NP-C [15] ($\geq 3/2 * OPT$)	NP-C (Theorem 4.3)
Arbitrary	No Algorithm	Path-Balancing Heuristic

4. Skipped Problems.

In an AND/OR/skipped scheduling problem, the inessential predecessors of an OR task may be skipped entirely. We first show that when the problems of Section 3 are formulated in the skipped model they remain NP-complete. Then we present a heuristic algorithm for scheduling to minimize completion time on m processors. This heuristic algorithm works for in-tree precedence constraints, but not for arbitrary precedence constraints.

4.1. Scheduling to Meet Deadline.

Theorem 3.1 showed that the problem of AND/OR/skipped scheduling with one deadline and arbitrary precedence constraints is NP-complete on a single processor, therefore, we immediately consider simplifying the precedence constraints.

THEOREM 4.1. *The problem of AND/OR/skipped scheduling to meet deadlines, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

THEOREM 4.2. *The problem of AND/OR/skipped scheduling to meet deadlines, where the task system is a simple in-forest with identical processing times, is NP-complete.*

Proof. The proof may be found in [11]. □

Now we consider the case where the task system is a simple in-forest and only the OR tasks have deadlines. For this type of task system, an algorithm to find a feasible schedule can examine each OR task and choose as its direct predecessor the AND task which has the fewest total predecessors. After these choices are made, the AND-only graph is scheduled using the earliest deadline first rule. This method always produces a feasible schedule if the task system is feasible. If the task system is infeasible it is still possible to maximizes the number of OR tasks that simultaneously meet their deadlines and have essential predecessors. To produce such a schedule, we note that an OR task together with one predecessor subtree consisting of k_i AND tasks may be thought of as one large task with processing time $k_i + 1$. Then the algorithm of [17], which minimizes unit penalty on a single processor, may be used to schedule tasks with processing time $(k_i + 1)$, to maximize the number of OR tasks that meet their deadline.

In summary, we find that the complexity of the skipped problem is always at least as high as the complexity of the unskipped problem. This fact is summarized in Table 2(a).

```
Input: Task graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$
Step 1: Convert the OR tasks in the in-tree $\mathbf{G}$ into AND tasks, to obtain an AND-only graph
    $\mathbf{G}'$ that minimizes $f(\mathbf{G}')$, as follows.
    For each path $C_i = \{T_{x_1} < T_{x_2} < \ldots < T_{x_k}\}$ from the root to a leaf in $\mathbf{G}$ do begin
      (a) [Copy $\mathbf{G}$] $\mathbf{G_C} \leftarrow \mathbf{G}$.
      (b) [Freeze OR tasks along path $C_i$] For each OR task $T_{x_j} \in C_i$ let $\mathbf{A_C} = (\mathbf{A_C} -
          P(\mathbf{G_C}, T_{x_j})) \cup \{(T_{x_{j-1}}, T_{x_j})\}$ (i.e. make $T_{x_j}$ an AND task in $\mathbf{G_C}$).
      (c) [Truncate all paths longer than $C_i$] Let $C_j \neq C_i$ be a longer path in $\mathbf{G_C}$. If no such
          $C_j$ exists, go to Step (d). Otherwise, let $T_k$ be the least OR task on $C_j$. If no such
          $T_k$ exists then go to Step (f). For each $T_l \in P(\mathbf{G_C}, T_k)$ on a path longer than $C_i$, do
          begin remove the arc $(T_l, T_k)$ from $\mathbf{G_C}$ end. If $|P(\mathbf{G_C}, T_k)| = 0$ no AND-only graph
          exists with $C_i$ as the longest path, so go to Step (f). Else Repeat Step (c).
      (d) [Minimize processing time] For each OR task $T_k$ with 2 or more direct predecessors
          and no OR predecessors in the graph $\mathbf{G_C}$, pick as a sole predecessor of $T_k$ the task
          $T_j \in P(\mathbf{G_C}, T_k)$ such that for all $T_i \in P(\mathbf{G_C}, T_k)$ with $i \neq j$, $E(\mathbf{G_C}, T_i) \geq E(\mathbf{G_C}, T_j)$.
      (e) If the resulting AND-only graph yields a lesser value of $f(\mathbf{G_C})$ then let $\mathbf{G}' \leftarrow \mathbf{G_C}$.
      (f) end.
Step 2: The resulting task system $\mathbf{G}'$ contains only AND tasks. Schedule this task system
    using a priority-driven heuristic and an arbitrary priority list.
```

FIG. 4. *The path-balancing heuristic for in-trees.*

4.2. Scheduling to Minimize Completion Time. Table 2(b) gives the complexity of scheduling m processors to minimize completion time. The next theorem concludes our investigation into the complexity of AND/OR scheduling.

THEOREM 4.3. *The problem of scheduling an AND/OR/skipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. ☐

Now we present a heuristic algorithm that minimizes the completion time of an AND/OR/skipped task system with in-tree precedence constraints. Let $f(\mathbf{G}) = E^*(\mathbf{G})/m + L^*(\mathbf{G})$ denote a function of an AND-only precedence graph. This function is an estimate of the worst-case completion time of a priority-driven schedule. Our algorithm converts an AND/OR in-tree into an AND-only in-tree that minimizes this function. In a general graph it is difficult to minimize this function quickly. If $m = 1$, a polynomial-time algorithm to minimize $f(\mathbf{G})$ could be used to solve any exact 3-cover problem (refer to Theorem 3.1), implying P = NP. Because of this the Path Balancing Heuristic described below is restricted to in-tree task graphs. The algorithm appears in Figure 4.

The complexity of the algorithm can be determined as follows. The $O(n)$ possible paths from the root to the leaves can be enumerated in time $O(n)$ using depth-first search. Each iteration of the Steps 1(a) - 1(e) can be carried out together in $O(n)$ time using a recursive depth-first search. Most of the work is done when returning from procedure calls. Hence, the overall complexity of this heuristic is $O(n^2)$.

To derive the worst-case performance of the Path-Balancing Heuristic we begin by showing that Step 1 of this heuristic minimizes $f()$.

LEMMA 4.4. $f(\mathbf{G}') \leq f(\mathbf{G_o})$.

Proof. Consider the longest path of length $L^*(\mathbf{G_o})$ in $\mathbf{G_o}$ This path starts at the tree root and ends at a leaf vertex. Clearly, the Path Balancing Heuristic considers this path in some iteration of Step 1. Step 1(c) of the heuristic ensures that no other paths are longer than this longest path, without increasing $E^*(\mathbf{G}')$ more than is necessary. Step 1(d) of the heuristic chooses the direct predecessors of each OR task to minimize $E^*(\mathbf{G}')$, thus, the heuristic cannot fail to find a graph for which $f(\mathbf{G}')$ is at most $E^*(\mathbf{G_o})/m + L^*(\mathbf{G_o})$. ◻

THEOREM 4.5. *The worst-case performance of the Path Balancing Heuristic is given by:*

$$\frac{W'}{W_o} \leq 2 - \frac{1}{m}. \tag{1}$$

Moreover, this bound is tight.

Proof. Any optimal schedule completes no earlier than the total processing time of the task system divided by m processors, and also no earlier than $L^*(\mathbf{G_o})$. Hence

$$W_o \geq \max\left\{\frac{E^*(\mathbf{G_o}) + L^*(\mathbf{G_o})}{m}, L^*(\mathbf{G_o})\right\}.$$

And by Lemmas 3.2 and 4.1, we have

$$W' \leq E^*(\mathbf{G}')/m + L^*(\mathbf{G}') \leq E^*(\mathbf{G_o})/m + L^*(\mathbf{G_o}).$$

Hence

$$\frac{W'}{W_o} \leq \frac{E^*(\mathbf{G_o})/m + L^*(\mathbf{G_o})}{\max\left\{\frac{E^*(\mathbf{G_o}) + L^*(\mathbf{G_o})}{m}, L^*(\mathbf{G_o})\right\}}. \tag{2}$$

We simplify Equation (2) in two cases.

Case 1. The max{} in (2) evaluates to its first argument. Then we have

$$\frac{W'}{W} \leq \frac{E^*(\mathbf{G_o}) + L^*(\mathbf{G_o})m}{E^*(\mathbf{G_o}) + L^*(\mathbf{G_o})} = B. \tag{3}$$

Note that the max{} in (2) evaluates to its first argument if and only if $L^*(\mathbf{G_o}) \leq E^*(\mathbf{G_o})/(m - 1)$, so we have an upper bound on $L^*(\mathbf{G_o})$. The derivative of the bound in (3) is

$$\frac{dB}{dL^*(\mathbf{G_o})} = \frac{E^*(\mathbf{G_o})(m-1)}{2(E^*(\mathbf{G_o}) + L^*(\mathbf{G_o}))} \geq 0. \tag{4}$$

Because the derivative of (4) is nonnegative for all $m \geq 1$ and $E^*(\mathbf{G_o}) \geq 0$, a maximum of (3) occurs when $L^*(\mathbf{G_o})$ is as great as possible, i.e. $L^*(\mathbf{G_o}) = E^*(\mathbf{G_o})/(m-1)$, thus

$$\frac{W'}{W} \leq \frac{E^*(\mathbf{G_o})(m-1) + E^*(\mathbf{G_o})m}{E^*(\mathbf{G_o})(m-1) + E^*(\mathbf{G_o})} = 2 - \frac{1}{m}.$$

Case 2. The max{} in (2) evaluates to its second argument. This occurs if and only if $E^*(\mathbf{G_o}) \leq L^*(\mathbf{G_o})(m-1)$. We substitute $E^*(\mathbf{G_o}) \leq L^*(\mathbf{G_o})(m-1)$ into the numerator of (2) to obtain (1). ◻

The example in Figure 4 demonstrates that this worst-case bound is tight. Let $\mathbf{T_1} = \{T_1, T_2, T_{4,1}, \ldots, T_{4,m(m-1)/e+1}\}$ and let $\mathbf{T_2} = \{T_2, T_{3,1}, \ldots, T_{3,m}, T_{5,1}, \ldots, T_{5,m}\}$.

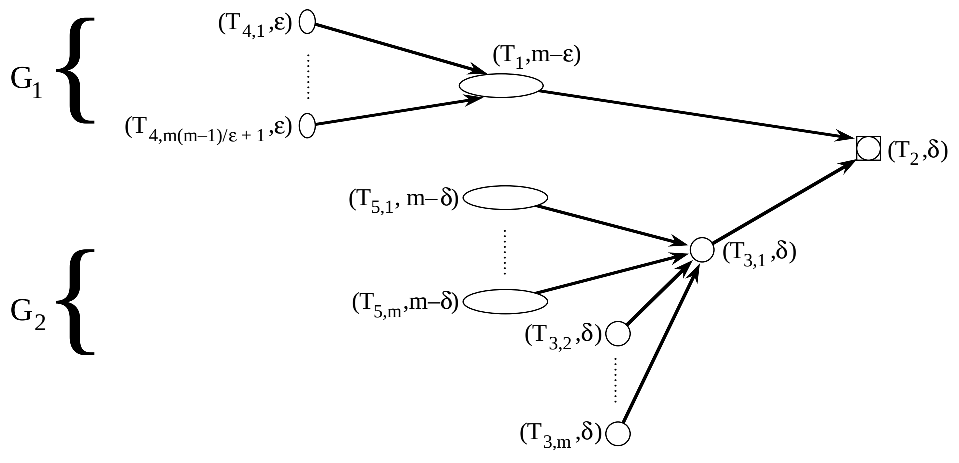


FIG. 5. *A worst-case AND/OR/skipped in-tree.*

The Path Balancing Heuristic chooses between the in-trees $\mathbf{G_1} = (\mathbf{T_1}, \mathbf{A_1}, \mathbf{P_1})$ and $\mathbf{G_2} = (\mathbf{T_2}, \mathbf{A_2}, \mathbf{P_2})$, where $\mathbf{A_1}$ and $\mathbf{A_2}$ denote the associated arc sets. The lengths of the longest paths in these in-trees are $L^*(\mathbf{G_1}) = L^*(\mathbf{G_2}) = m + \delta$, respectively. Furthermore, $E^*(\mathbf{G_1}) = E^*(\mathbf{G_2}) = m^2 - m$. Thus, the Path Balancing Heuristic chooses arbitrarily between these two trees, since either one minimizes $f(\mathbf{G}')$. There is a schedule of length $m + 2\delta$ for $\mathbf{G_2}$, but the shortest possible schedule for $\mathbf{G_1}$ has length $m + m(m-1)/m + \delta$ whenever e divides $(m-1)$ evenly. As $d \rightarrow 0$, the ratio of these schedule lengths approaches $2 - 1/m$.

We now offer additional evidence that the problem of scheduling AND/OR/skipped task systems is much harder than the problem of scheduling AND-only task systems. Consider scheduling an AND/OR/skipped task system derived from an exact 3-cover problem, as described in the proof of Theorem 3.1, on a $(3n + 1)$-processor system. We add to the task system an AND task with $2n + 1$ direct predecessors, and ask if there is a schedule that completes in 2 units of time on $3n + 1$ processors. The task system is feasible if n tasks corresponding to edges in an exact 3-cover together with the additional $2n + 1$ AND tasks begin processing at time 0, and all the tasks corresponding to hypergraph vertices together with the other added AND task begin their processing at time 1. Hence, there is a schedule with a completion time of 2 if and only if there is an exact 3-cover. It follows that unless P = NP no polynomial-time AND/OR/skipped scheduling heuristic can guarantee a worst-case completion time of less than 3/2 times the length of an optimal schedule. In contrast to this, if the task system is AND-only, it is known [15] that no polynomial-time heuristic can guarantee a worst-case completion time of less than of 4/3 times the length of an optimal schedule.

5. Conclusion.

We have analyzed the skipped and unskipped variants of the AND/OR scheduling problem with deadlines. In the skipped variant, some tasks may be left unscheduled, but in the unskipped variant all tasks must be scheduled. When tasks have identical processing times, deadlines, and there is a single processor, the problem was shown to be NP-complete, even for drastically simplified precedence constraints. We presented an efficient priority-driven heuristic to minimize completion time on m processors, and showed that its worst-case performance bound cannot be improved by using a different priority-driven heuristic. We also presented a heuristic to minimize the completion time of an AND/OR/skipped task system with in-tree precedence constraints. We derived the worst-case performance for this algorithm and explained why the algorithm cannot be extended to handle general task graphs with the same performance unless P = NP.

Throughout this paper we assumed that only one direct predecessor task had to be completed before an OR task could begin. Under a more general assumption, OR task T_i can begin once k_i predecessor tasks are complete. The algorithms and theorems in this paper require minor modifications to handle this more general case. There is also a similar AND/OR model where individual arcs (and not tasks) can be AND arcs or OR arcs. By using tasks with a processing time of zero, our model can simulate this other model. There are also situations where both OR/skipped and OR/unskipped tasks are present in a single in-tree. With slight modifications our AND/OR/skipped heuristic can be used to handle such mixed task systems. Details of these transformations and algorithms appear in [12].

During this investigation we reached several conclusions about the complexity of AND/OR scheduling. Contrary to our intuition, the skipped problems we considered were generally of higher complexity than the corresponding unskipped problems. This can be seen by comparing Table 1 and Table 2, and the proofs in the appendix. In the problem of scheduling to meet deadlines, we have several observations. It was generally not helpful to restrict the in-degree of OR tasks in the task graph. It was also not helpful to restrict deadlines to only the OR tasks, or to restrict the task graph to be an in-tree or an in-forest or even a simple in-tree, the simplest relation possible for this type of problem.

A. Appendix.

This appendix presents the proofs of Theorems 3.2, 3.4, 4.1, 4.3, and Corollary 3.1. Proofs of Theorems 3.3 and 4.2 may be found in both [11] and [12]. Except where noted, all proofs refer to the scheduling of a single processor.

THEOREM 3.2. *The problem of AND/OR/unskipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines, and in-tree precedence constraints, is NP-complete.*

Proof. Our proof is based on a reduction from 3SAT. Given an instance of a 3SAT problem, with k boolean variables and n clauses, we will create k OR tasks. For each variable x_i which occurs in l_i clauses we create an in-tree containing one OR task and two chains of length l_i. One chain corresponds to truth for the associated variable, and the other corresponds to falsity. Therefore, there are $3n$ tasks in all chains corresponding to truth, and $3n$ tasks in all chains corresponding to falsity. The OR tasks are given deadlines of $e = 3n + k$. An example is shown in Figure 6. This example is an in-tree for a variable x that appears in 4 clauses. Deadlines are depicted above or below the tasks. Because of the deadlines of the OR tasks, in any feasible schedule k OR tasks and k chains execute throughout the time interval $[0, e]$, and no other tasks may execute in this interval. This leaves k task chains to execute in the time period $[e, e + 3n]$ in a feasible schedule.

For each 3SAT clause we assign an interval of three time units starting at time e. Hence the time intervals $[e, e+3], [e+3, e+6], \dots, [e+3n-3, e+3n]$ correspond to clause 1, clause 2, ... , clause n, respectively. Each interval of time is divided into two parts. In the first two time units, tasks in leftover chains corresponding to truth or falsity in a clause may execute. In the third time unit, only a task corresponding to truth may execute. To enforce this rule, we give later deadlines to the tasks/terms that would make each clause true. In Figure 6, variable x occurs in

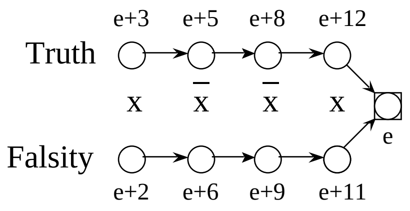


FIG. 6. *An in-tree for a variable x appearing in the first 4 clauses.*

the first 4 clauses of the 3SAT expression. It appears uncomplemented in clauses 1 and 4, and complemented in clauses 2 and 3. If x appears in the 3SAT expression for the i'th time as an uncomplemented variable in clause j, the deadline for the i'th task in the truth predecessor chain is $e+3j$, and is $e+3j-1$ for the i'th task in the falsity predecessor chain. These deadlines are exchanged if the i'th appearance of x is as a complemented variable in clause j. We give all the OR tasks a common AND successor with a deadline of infinity, to form a single tree.

If a scheduling algorithm finds a feasible schedule, then each task that executes in the interval $[e+3j-1, e+3j]$ corresponds to a variable (or a complemented variable) that is true in clause j. If the variable were not true, then the deadline of the task would have expired one time unit earlier. Furthermore, the task chains guarantee that the truth or falsity of a variable is consistent among different 3SAT clauses. Thus, a schedule is feasible if and only if there is a satisfying truth assignment. □

All the other proofs in this appendix and in [11] and [12] are modifications of the proof of Theorem 3.2. In particular, Theorems 3.3 and 4.2 require a large simple in-tree for each term in a 3SAT expression, and have been omitted for brevity.

COROLLARY 3.1. *The problem remains NP-complete if only the OR tasks have deadlines.*

Proof. We make the following changes to the proof of Theorem 3.2: replace the in-trees of the type depicted by Figure 6 by new in-trees such as the one in Figure 7. This is done by adding an AND task with a deadline of e to the beginning of each truth and falsity chain, converting each AND task with a deadline into an OR task with one or two extra AND predecessor tasks, and setting $e = 3n + 5k$. As in the previous proof, the last deadline associated with a variable in a clause is $e+3n$. Note that there are exactly $e+3n$ tasks of the type that are shaded in Figure 7, in the entire task system. Becuase of their deadlines, the shaded tasks must execute in the time interval $[0, e+3n]$, and the unshaded tasks must execute after time $e+3n$ in any feasible schedule.

It is not difficult to verify that in a feasible schedule the tasks that execute in the time interval $[e, e+3n]$ correspond to a satisfying 3SAT truth assignment. □

THEOREM 4.1. *The problem of AND/OR/skipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines on the OR tasks only, and in-tree precedence constraints, is NP-complete.*

Proof. This theorem extends the previous corollary to skipped tasks. We use nearly the same in-trees as in Corollary 3.1. However, we set $e = 2k$, we give the OR tasks at the root of each in-tree a deadline of $e+3n+k$ rather than e, and we replace each unshaded task by a chain of e AND tasks. It is not difficult to check that the task chains that execute in the time intervals $[e+3j-1, e+3j]$, $1 \le j \le n$, correspond to a truth assignment satisfying the 3SAT clauses. □

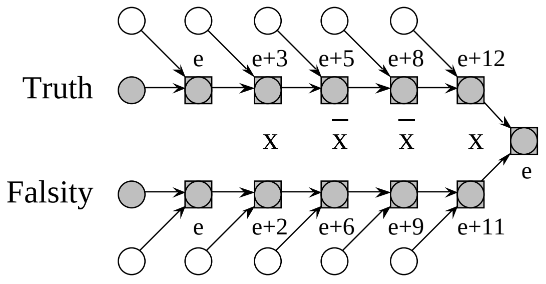


FIG. 7. *An in-tree for scheduling with deadlines on OR tasks only.*

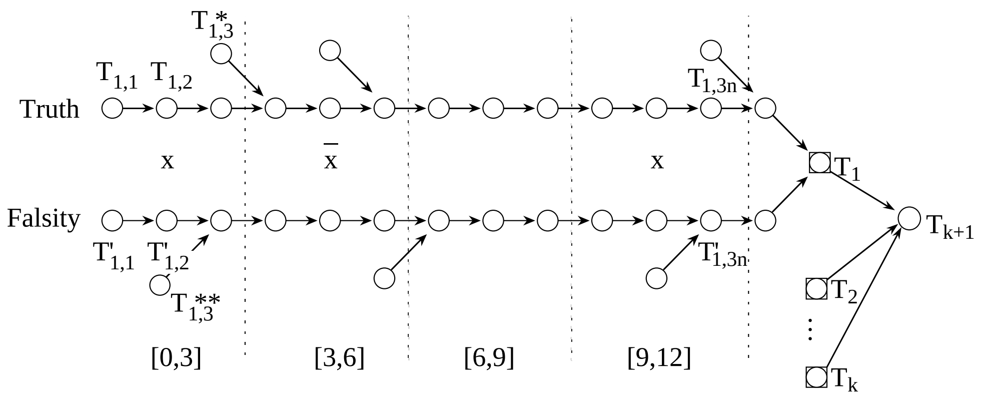


FIG. 8. *In-tree task system for AND/OR/skipped/unskipped tasks on m processors.*

THEOREM 4.3. *The problem of scheduling an AND/OR/skipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. Given a 3SAT problem with k boolean variables and n clauses, we specify a system with $m = k + 1$ processors. For each variable x_i we create an in-tree with one OR task T_i at the root and two predecessor chains of length $3n + 1$. One chain corresponds to truth, and the other corresponds to falsity. All the tasks $T_1 \ldots T_k$ have a common AND direct successor T_{k+1}. For each 3SAT clause we assign an interval of 3 units of time starting at time zero. Hence the intervals $[0, 3], [3, 6], \ldots, [3n - 3, 3n]$ correspond to clause 1, clause 2, . . ., clause n. If a variable x_i appears uncomplemented (complemented) in clause j, we create two AND tasks $T^*_{i,j}$ and $T^{**}_{i,j}$ and make their successors $T_{i,3j+1}$ and $T'_{i,3j}$ ($T_{i,3j}$ and $T'_{i,3j+1}$) respectively. Figure 8 illustrates the transformation for a 3SAT problem with $n = 4$ clauses. The variable x appears in the first, second, and fourth clauses of the 3SAT problem instance, and x is complemented in the second clause. The predecessor chains of length $3n + 1$ are used to simulate multiple deadlines, which are not allowed by the problem statement.

If a scheduling algorithm finds a feasible schedule with an overall completion time of $3n + 3$, then by interchanging tasks among different processors, we can transform the schedule so that processors one through k execute a truth or falsity chain of length $3n+1$ in the time interval $[0, 3n+1]$, and processor $k+1$ executes only tasks of type $T^*_{i,j}$ or $T^{**}_{i,j}$ in the same time interval. Then each task that executes in the time interval

$[3j-1, 3j], 1 \leq j \leq n$, on processor $k+1$ corresponds to a variable or complemented variable that is true in clause j of the 3SAT problem instance. Because only one truth or falsity chain for each OR task executes in the time interval $[0, 3n+1]$, the truth or falsity of a variable is consistent among different 3SAT clauses. Thus, a feasible schedule can be found if and only if there is a satisfying truth assignment. ☐

THEOREM 3.4. *The problem of scheduling an AND/OR/unskipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is nearly identical to the proof of Theorem 4.3. Given a 3SAT problem, we generate the same in-tree as in the proof of Theorem 3.4, except we add a chain of $6n+6$ AND successors to task T_{k+1}. Then we ask if there is a schedule with an overall completion time of $9n+9$. In such a schedule k task chains without essential tasks have plenty of time to complete in the time interval $[3n+3, 9n+9]$. It is not difficult to see that there are tasks that execute in the time intervals $[3j-1, 3j]$, $1 \leq j \leq n$, that correspond to a satisfying truth assignment. ☐

Acknowledgement. We wish to acknowledge Mohlafi Sefika, who implemented the path balancing heuristic and improved its description, and also Sandra Broadrick-Allen, who helped to improve later versions of this paper.

REFERENCES

[1] P.-R. CHANG, *Parallel algorithms and VLSI architectures for robotics and assembly scheduling*, Ph.D. thesis, Purdue University, West Lafayette, IN, 1988.

[2] E. G. COFFMAN, JR., ED., *Computer and Job Shop Scheduling Theory*, John Wiley, New York, NY, 1976.

[3] J. Y. CHUNG, J. W.-S. LIU, AND K. J. LIN, *Scheduling Periodic Jobs That Allow Imprecise Results*, IEEE Trans. Computers, 39 (1990), pp. 1156-1174.

[4] E. G. COFFMAN, JR., J. Y. LEUNG, AND D. W. TING, *Bin packing: maximizing the number of pieces packed*, Acta Informatica, 9 (1978), pp. 263-271.

[5] L. S. HOMEM DE MELLO AND A. C. SANDERSON, *AND/OR graph representation of assembly plans*, Proc. AAAI (1986) pp. 1113-1119.

[6] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, SIAM J. Comput., 6 (1977), pp. 416-428.

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: a Guide to the Theory of NP-completeness*, W. H. Freeman and Co., San Francisco, CA, 1979.

[8] M. R. GAREY, D. S. JOHNSON, B. B. SIMONS, AND R. E. TARJAN, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput. 10 (1981), pp. 256-269.

[9] D. W. GILLIES AND J. W.-S. LIU, *Greed in resource scheduling*, Proc. IEEE Real-Time Systems Symposium, 10 (1989), pp. 285-294.

[10] D. W. GILLIES AND J. W.-S. LIU, *Greed in resource scheduling*, Acta Informatica, 28 (1991), pp. 755-775.

[11] D. W GILLIES, AND J. W.-S. LIU, *Scheduling Tasks with AND/OR Precedence Constraints*, Rep. No. UIUCDCS-R-90-1627 (UIUC-ENG-1766), Department of Computer Science, Univ. of Illinois, Urbana, 1991.

[12] D. W GILLIES, *Algorithms to schedule tasks with AND/OR precedence constraints*, Ph.D. Thesis, Department of Computer Science, Univ. of Illinois, Urbana, 1993.

[13] R. L GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416-429.

[14] T. C HU, *Parallel sequencing and assembly line problems*, Operations Res., 9 (1961), pp. 841-848.

[15] E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS, *Sequencing and Scheduling: Algorithms and Complexity*, Rep. No. BS-R8908, Centre for Mathematics and Computer Science, Amsterdam, Holland, 1989.

[16] M. C. MCELVANY, *Guaranteeing deadlines in MAFT*, Proc. IEEE Real-Time Systems Symposium, 9 (1988), pp. 130-139.

[17] **J. M. Moore**, *An n job, one machine sequencing algorithm for minimizing the number of late jobs*, Management Sci., 15 (1968), pp. 102-109.

[18] **D. Peng and K. G. Shin**, *Modeling of concurrent task execution in a distributed system for real-time control*, IEEE Trans. Computers, 36 (1987), pp. 500-516.

[19] **V. Saletore and L. V. Kale**, *Obtaining first solution faster in AND and OR parallel execution of logic programs*, North American Conference on Logic Programming, 1 (1989), pp. 390-406.