Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations

Jia Xu

Abstract— **We present a scheduling algorithm that solves the problem of finding a feasible nonpreemptive schedule whenever one exists on M identical processors for a given set of processes such that each process starts executing after its release time and completes its computation before its deadline, and a given set of precedence relations and a given set of exclusion relations defined on ordered pairs of process segments are satisfied. This algorithm can be applied to the important problem of automated pre-run-time scheduling of processes with arbitrary precedence and exclusion relations on multiprocessors in hard-real-time systems.**

Index Terms— **Automated pre-run-time scheduler, deadlines, exclusion, hard-real-time systems, multiprocessor, optimal scheduling algorithm, precedence.**

I. Introduction

In [17], Xu and Parnas described a scheduling algorithm that solves the problem of finding a feasible schedule whenever one exists for a given set of processes such that each process starts after its release time and completes its computation before its deadline, and a given set of precedence relations and a given set of exclusion relations defined on ordered pairs of process segments are satisfied on a single processor. That algorithm would not necessarily solve the problem if there were multiple processors. In this paper, we provide a scheduling algorithm that can deal with multiprocessors.

We present a scheduling algorithm for solving the following problem: we are given a set of processes, where each process consists of a sequence of segments. Each segment is required to precede a given set of other segments. Each segment also excludes a given set of other segments, i.e., each segment cannot be executed simultaneously with any segment in the set that it excludes. For each process, we are given a release time, a computation time, and a deadline. It is also assumed that we know the computation time and start time of each segment relative to the beginning of the process containing that segment.

Our problem is to find a nonpreemptive schedule on M identical processors for the given set of processes such that each process starts executing after its release time (the release time is the earliest time a process is allowed to start) and completes its computation before its deadline, and all the precedence and exclusion relations on segments are satisfied. Note that if we can solve this problem, then we can also solve the special case where the release times and deadlines of each process are periodic by solving the problem for the set of processes occurring within a time period that is equal to the least common multiple of the periods of the given set of processes.

The multiprocessor algorithm presented here, as well as the single-processor algorithm in [17], were designed to be used by a pre-run-time scheduler for scheduling processes with arbitrary precedence and exclusion relations in hard-real-time systems [5]. In such systems, precedence relations may exist between process segments when some process segments require information produced by other process segments. Exclusion relations may exist between process segments when some process segments must exclude other process segments to prevent errors caused by simultaneous access to shared resources, such as data, I/O devices, etc.

In many hard-real-time applications, the bulk of the computation can be confined to periodic processes where the sequencing and timing constraints are known in advance. That is, the release times and deadlines of processes besides the precedence and exclusion relations defined on them are known in advance. General techniques also exist for transforming a set of asynchronous processes into an equivalent set of periodic processes [14], [15]. Thus it is possible to use a pre-run-time scheduler to make scheduling decisions before run time. Pre-run-time scheduling has many advantages compared to run time scheduling: precious run time resources required for run time scheduling and context switching can be greatly reduced, and, more importantly, it is easier to guarantee in advance that real-time deadlines will be met.

However, up to now, the automated pre-run-time scheduler for processes with arbitrary precedence and exclusion relations on multiprocessors has remained "an unsolved problem" [5]. In the past, practitioners working on safety-critical hard-real-time systems, such as in the area of submarine data processing and aircraft weapon delivery programs, have been observed doing such problems by hand. The result of this "hand scheduling" is "spaghetti code" that is very hard to verify and maintain.

The algorithm presented here makes it possible to automate the task of pre-run-time scheduling processes with arbitrary precedence and exclusion relations on M identical processors. We are currently working on producing a practical system that

Manuscript received October 31, 1990; revised August 12, 1992. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under an operating grant and an equipment grant. Recomended by Peter Ng.

The author is with the Department of Computer Science, York University, North York, Ontario M3J 1P3, Canada.

IEEE Log Number 9206642.

uses both this algorithm and the single-processor algorithm presented in [17] to systematically search for a feasible schedule when given a set of release time, deadline, precedence, and exclusion relation parameters. Such a system would greatly facilitate the task of pre-run-time scheduling because it would virtually eliminate any possibility of errors in the computation of schedules. Not only would it be capable of finding a feasible schedule whenever one exists, it would also be capable of informing the user whenever no feasible schedule exists for a given set of parameters much faster and reliably than any ad hoc or manual method. In the latter case, it could also provide the user with useful information on which parameters should be modified in order to obtain a feasible schedule. Such a system would be particularly useful for applications in which changes in the system often occur and schedules have to be frequently recomputed.

II. Related Work

In [14], Mok treats in detail techniques that allow one to use a pre-run-time scheduler to make scheduling decisions before run time for both periodic and asynchronous processes by replacing asynchronous processes with an equivalent set of periodic processes. Here, we overview some of the previous work that are related to optimal multiprocessor scheduling algorithms for a hard-real-time environment.

For solving the problem of finding a feasible schedule on multiprocessors for a set of processes where each process must execute between a given release time and deadline, optimal scheduling algorithms exist that solve the special case where all processes are completely preemptable, i.e., each process consists of a single segment that can be preempted by any other process [12]. In the case where a feasible nonpreemptive schedule is required, optimal scheduling algorithms exist for the special case where all processes have unit computation time [16]. Both of these two special cases can be solved in polynomial time. If processes have arbitrary computation times, then the problem of finding a feasible nonpreemptive schedule becomes NP complete in the strong sense, even if only one processor is used [6], which effectively excludes the possibility of the existence of a polynomial time algorithm for solving the problem. Surveys of deterministic scheduling problems and algorithms can be found in [4], [7], and [9].

Other algorithms and synchronization methods focus on on-line or dynamic scheduling, where the primary objective is to try to dynamically construct a feasible schedule at run time. Some heuristics have been proposed or studied [11], [8]. A related problem is scheduling tasks with resource requirements [2]. In [1], an optimal algorithm is given for the case of unit time tasks with zero or one resource. In [20] and [21] heuristics are studied for the case of arbitrary length tasks with multiple resources. In [10] an algorithm is given for determining an upper bound on the response time of each task under resource constraints. Most algorithms or synchronization methods for on-line scheduling employ heuristics to achieve short run times.

Since our algorithm was designed primarily for pre-run-time scheduling, where it is assumed that the major characteristics of processes are known in advance, and schedules are computed off-line, the ability to find a feasible schedule whenever one exists, rather than the algorithm's run-time, was our foremost concern. For this reason, we concentrated our effort on finding optimal algorithms that find a feasible schedule whenever one exists, rather than heuristics.

We do not know of any published optimal scheduling algorithm that solves the problem of finding a feasible nonpreemptive schedule whenever one exists on M identical processors for a given set of processes with arbitrary computation times such that each process starts executing after its release time and completes its computation before its deadline, and a given set of precedence relations and a given set of exclusion relations defined on ordered pairs of process segments are satisfied. Such problems occur frequently in many real-world situations. Multiprocessor algorithms that assume that all processes are completely preemptable were not applicable to our problem, since assuming all processes are completely preemptable would allow simultaneous access to shared resources, which could have disastrous consequences. Multiprocessor algorithms that assume that all processes have unit computation time were also not applicable to our problem, since, in reality, processes rarely have the same computation time.

Although the problem that our algorithm solves is NP complete and it is possible to construct pathological problem instances where the algorithm would require an amount of computation time that is exponentially related to the problem size, we believe that such pathological problem instances are unlikely to occur in practical hard-real-time system applications. Our experience has shown that even with difficult problems of very large size, the algorithm is still capable of finding a feasible schedule whenever one exists within reasonable time.

A very useful property of this algorithm is that at each intermediate stage of the algorithm a complete schedule is constructed. At the beginning, the algorithm starts with a schedule that is obtained by using an earliest-deadline-first strategy. Then it systematically improves on that initial schedule until a feasible schedule is found, or until it determines that no feasible schedule exists. Thus, even if we have to terminate the algorithm prematurely, it would still provide a complete schedule that is at least as good as any schedule obtained by using an earliest-deadline-first heuristic.

In Section III, we provide an overview of the algorithm. Basic notation and definitions are introduced in Section IV. In Section V we show how to improve on a valid initial solution. In Section VI we describe the strategy used to search for an optimal or feasible solution. The empirical behaviour of the algorithm is described in Section VII. Finally, conclusions are presented in Section VIII.

III. Overview of the Algorithm

From the computation time and start time of each segment relative to the beginning of the process containing that segment, and the release time, computation time, and deadline of

each process, one should be able to compute the release time, computation time, and the deadline for each segment.

Our algorithm finds a valid feasible schedule in which all segments in the schedule complete their computation before their deadlines while satisfying a given set of "$EXCLUDE$" relations and a given set of "$PRECEDE$" relations defined on ordered pairs of segments, whenever such a valid feasible schedule exists. The set of $EXCLUDE$ relations and the set of $PRECEDE$ relations are initialized to be identical with those exclusion and precedence relations required in our original problem.

The algorithm will terminate as soon as it finds a feasible schedule that meets all deadline constraints. If the original problem is nonfeasible, i.e., no feasible schedule exists that can satisfy all deadline constraints, the algorithm will also terminate as soon as it determines that this is the case.

Our algorithm uses a branch-and-bound technique. It has a search tree where at its root node we use an earliest-deadline-first strategy to compute a schedule called a "valid initial solution" that satisfies the release time constraints and all the initial $EXCLUDE$ and $PRECEDE$ relations.

At each node in the search tree, we find the latest segment l in the valid initial solution computed at that node. We identify two "expand" sets of pairs of segments $G_1[l] = \{(i, j)\}$ and $G_2[l] = \{(i_1, j_1)\}$ such that a feasible solution can be found only if segment j must be completed before segment i is started, or if segment j_1 is scheduled to start not later than segment i_1.

For each pair of segments (i, j) in the expand set $G_1[l]$, we create a successor node in which we add the relation "j $PRECEDES$ i." For each pair of segments (i_1, j_1) in the expand set $G_2[l]$, we create a successor node in which we add the relation "j_1 $BEFORE$ i_1." Then if a valid initial solution for the successor node is computed using the new additional relations, j would be completed before i is started, or j_1 would be scheduled to start not later than i_1.

For each node in the search tree, we also compute a lower bound on the lateness of any schedule leading from that node. The node that has the least lower bound among all unexpanded nodes is considered to be the node that is most likely to lead to a feasible solution—we always branch from the node that has the least lower bound among all unexpanded nodes. In case of ties, we choose the node with least lateness among the nodes with least lower bound.

We continue to create new nodes in the search tree until we either find a feasible solution, or until there exists no unexpanded node that has a lower bound less than the least lateness of all valid initial solutions found so far. In the latter case, no feasible valid initial solution exists in which all the deadline constraints can be satisfied.

The ways in which we use $PRECEDES$ and $BEFORE$ relations to schedule a segment j such that j is completed before a segment i for each pair (i, j) in the expand set $G_1[l]$, or let segment j_1 be started no later than another segment i_1 for each pair (i_1, j_1) in the expand set $G_2[l]$ cover *all* possible ways of finding a feasible valid initial solution.

In the following section, we formally define all the terms mentioned above.

IV. Valid Initial Solution

A. Notation and Definitions

In order to solve the foregoing problem, we first introduce the following definitions and notations:

Let the *set of processes* be denoted by P.

Each process $p \in P$ consists of a finite sequence of *segments* $p[0], p[1], \ldots, p[n[p]]$, where $p[0]$ is the first segment and $p[n[p]]$ is the last segment in process p.

For each segment i, we define a *release time* $r[i]$, a *deadline* $d[i]$, and a *computation time* $c[i]$.

It is assumed that $r[i]$, $d[i]$, and $c[i]$ have integer values.

Let the *set of all segments* belonging to processes in P be denoted by $S(P)$.

Each segment i consists of a sequence of *segment units* $(i, 0), (i, 1), \ldots, (i, c[i]-1)$, where $(i, 0)$ is the first segment unit and $(i, c[i]-1)$ is the last segment unit in segment i.

We define the *set of segment units of* $S(P)$:

$$U = \{(i, k) | i \in S(P) \wedge 0 \leq k \leq c[i] - 1\}.$$

Assume that the *total number of processors* available is M.

We define the *set of processor time units*:

$$R = \{(t, m) | t \in [0, \infty) \wedge 1 \leq m \leq M\}.$$

Intuitively, a segment unit is the smallest indivisible granule of a process. The total number of segment units in each segment is equal to the computation time required by that segment. Processor resources are measured in terms of processor time units. Each segment unit requires one processor time unit (one unit of time on one processor) to execute.

A *schedule* of a set of processes P is a total function π: $U \rightarrow R$ satisfy the following properties:

i) $\forall (i_1, k_1), (i_2, k_2) \in U, \forall t, m, t \in [0, \infty), 1 \leq m \leq M$: $(\pi(i_1, k_1) = (t, m) \wedge ((i_1 \neq i_2) \vee (k_1 \neq k_2))) \Rightarrow (\pi(i_2, k_2) \neq (t, m))$.

ii) $\forall (i, k_1), (i, k_2) \in U$: $(k_1 < k_2 \wedge \pi(i, k_1) = (t_1, m_1) \wedge \pi(i, k_2) = (t_2, m_2)) \Rightarrow (t_1 < t_2)$.

iii) $\forall p, i, j, p \in P, 0 \leq i, j \leq n[p]$: $(i < j \wedge \pi(p[i], c[p[i]] - 1) = (t_1, m) \wedge \pi(p[j], 0) = (t_2, m_2)) \Rightarrow (t_1 < t_2)$.

Condition i) above states that on each processor no more than one segment can be executing at any time. Condition ii) states that a schedule must preserve the ordering of the segment units in each segment and that no more than one segment unit belonging to the same segment can execute at the same time. Condition iii) states that a schedule must preserve the ordering of the segments in each process.

We say that *segment i executes at time t on processor m* iff $\exists k, 0 \leq k \leq c[i] - 1 : \pi(i, k) = (t, m)$.

We say *segment i executes from* t_1 to t_2 *on processor m* iff $\exists k, m, \forall t, 0 \leq k \leq c[i] - 1, 1 \leq m \leq M, 0 \leq t \leq t_2 - t_1 - 1 : \pi(i, k + t) = (t_1 + t, m)$.

We say that *processor m is idle at time t* iff $\nexists j, k : \pi(j, k) = (t, m)$.

We define the *start time* of segment i to be $s[i]$ such that $\pi(i, 0) = (s[i], m)$.

We define the *completion time* of segment i to be $e[i]$ such that $\pi(i, c[i] - 1) = (e[i] - 1, m)$.

The *lateness of a segment* i in schedule P is defined by $e[i] - d[i]$.

The *lateness of a schedule of* P is defined by $\max\{e[i] - d[i] | i \in S(P)\}$.

We define a *latest segment* to be a segment that realizes the value of the lateness of the schedule.

A *nonpreemptive schedule* of a set of processes P is a schedule of P satisfying the following properties:

$$\forall i, m, t, k, i \in S(P), 1 \le m \le M, t \in [0, \infty), 0 \le k \le c[i] - 1 : \pi(i, 0) = (t, m) \Rightarrow \pi(i, k) = (t + k, m).$$

We introduce the $PRECEDE$ relation and $EXCLUDE$ relation on ordered pairs of segments together with the notion of a "valid schedule."

We impose the restriction that if segment i $EXCLUDES$ segment j, then segment j must also $EXCLUDE$ segment i, i.e.

$$(i\ EXCLUDES\ j) \Leftrightarrow (j\ EXCLUDES\ i)$$

A *valid schedule* of a set of processes P is a nonpreemptive schedule of P that (in addition to satisfying all the properties of a nonpreemptive schedule) satisfies the following properties:

$\forall i, j \in S(P)$:

i) $s[i] \ge r[i]$

ii) $(i\ PRECEDES\ j) \Rightarrow (e[i] \le s[j])$

iii) $(i\ EXCLUDES\ j) \Rightarrow (e[i] \le s[j]) \vee (e[j] \le s[i])$.

Condition i) above states that each process can only start execution after its release time. Condition ii) states that in a valid schedule, if segment i $PRECEDES$ segment j, then under all circumstances, segment j cannot start execution before segment i has completed its computation. Condition iii) states that in a valid schedule, if segment i $EXCLUDES$ segment j, then either segment j must complete its computation before segment i can start execution, or, segment i must complete its computation before segment j can start execution.

We initialize the set of $PRECEDE$ relations and the set of $EXCLUDE$ relations to be identical with the precedence and exclusion relations that must be satisfied in the original problem. In addition, in order to enforce the proper ordering of segments within each process, we let $p[k]$ $PRECEDE$ $p[k+1]$ for all $p \in P$, and for all k, $0 \le k \le n[p] - 2$. Thus, a valid schedule would satisfy all the release time, exclusion, and precedence constraints in the original problem.

A *feasible schedule* of a set of processes P is a valid schedule of P such that its lateness is less than or equal to zero.

An *optimal scheduling algorithm* is a scheduling algorithm that, given any set of processes P, finds a feasible schedule for P whenever one exists.

We introduce a third relation that will be used in our algorithm—the $BEFORE$ relation on ordered pairs of segments. In any schedule, a set of $BEFORE$ relations is satisfied if and only if the following properties are satisfied:

$$\forall i, j \in S(P): i\ BEFORE\ j \Rightarrow s[i] \le s[j].$$

We adjust release times and deadlines according to $PRECEDES$, $EXCLUDES$, and $BEFORE$ relations.

The *adjusted release time* $r'[i]$ of segment i is defined by

i) $r'[i] = r[i]$, if $\nexists j$: $(j\ PRECEDES\ i \vee j\ BEFORE\ i)$

or

ii) $r'[i] = \max\{r[i], r'[j] + c[j]\}$, if $\exists j : j\ PRECEDES\ i \vee (j\ BEFORE\ i \wedge j\ EXCLUDES\ i)$

or

iii) $r'[i] = \max\{r[i], r'[j]\}$, if $\exists j : \neg(j\ PRECEDES\ i) \wedge (j\ BEFORE\ i \wedge \neg(j\ EXCLUDES\ i))$.

In ii) above, if there exists some segment j such that j $PRECEDES$ i; or j is $BEFORE$ i and j $EXCLUDES$ i, then i cannot start until j has completed. Since the earliest time that j can be completed is $r'[j] + c[j]$, i's adjusted release time, that is, its real release time, should be $r'[j] + c[j]$ if it is greater than $r[i]$. In iii) above, if there exists some segment j such that j does NOT $PRECEDE$ i and j is $BEFORE$ i and j does NOT $EXCLUDE$ i, then i cannot start until j has started. Since the earliest time that j can be started is $r'[j]$, i's adjusted release time should be $r'[j]$ if it is greater than $r[i]$.

The *adjusted deadline* $d'[i]$ of segment i is defined by

i) $d'[i] = d[i]$, if $\nexists j : (i\ PRECEDES\ j \vee i\ BEFORE\ j)$

or

ii) $d'[i] = \min\{d[i], d'[j] - c[j]\}$, if $\exists j : i\ PRECEDES\ j \vee (i\ BEFORE\ j \wedge i\ EXCLUDES\ j)$

or

iii) $d'[i] = \min\{d[i], d'[j] - c[j] + c[i]\}$, if $\exists j : \neg(i\ PRECEDES\ j) \wedge (i\ BEFORE\ j \wedge \neg(i\ EXCLUDES\ j))$

In ii) above, if there exists some segment j such that i $PRECEDES$ j; or i is $BEFORE$ j and i $EXCLUDES$ j, then j cannot start until i has completed. In order to complete j's computation before j's deadline, the latest time by which j must be started is $d'[j] - c[j]$. Then i's adjusted deadline, that is, its real deadline, should be $d'[j] - c[j]$ if it is smaller than $d[i]$. In iii) above, if there exists some segment j such that i does NOT $PRECEDE$ j and i is $BEFORE$ j and i does NOT $EXCLUDE$ j, then j cannot start until i has started. In order to complete j's computation before j's deadline, the latest time by which j must be started is $d'[j] - c[j]$, which is also the latest time by which i must be started. This implies that the latest time by which i must be completed is $d'[j] - c[j] + c[i]$. So i's adjusted deadline should be $d'[j] - c[j] + c[i]$ if it is smaller than $d[i]$.

As examples, in the leaf node of Example 1 (Fig. 1), there exists a new relation A $PRECEDES$ D. According to ii) in the definition of adjusted release times above, D's adjusted release time is $r'[D] = \max\{r[D], r'[A] + c[A]\} = \max\{10, 0 + 18\} = 18$. According to ii) in the definition of adjusted deadlines, A's adjusted deadline is $d'[A] = \min\{d[A], d'[D] - c[D]\} = \min\{36, 35 - 17\} = 18$. In the leaf node of Example 3 (Fig. 3), there exists the new relation A $BEFORE$ C, and A does NOT $EXCLUDE$ C and A does NOT $PRECEDE$ C. According to iii), A's adjusted deadline is $d'[A] = \min\{d[A], d'[C] - c[C] + d[A]\} = \min\{55, 25 - 15 + 18\} = 28$.

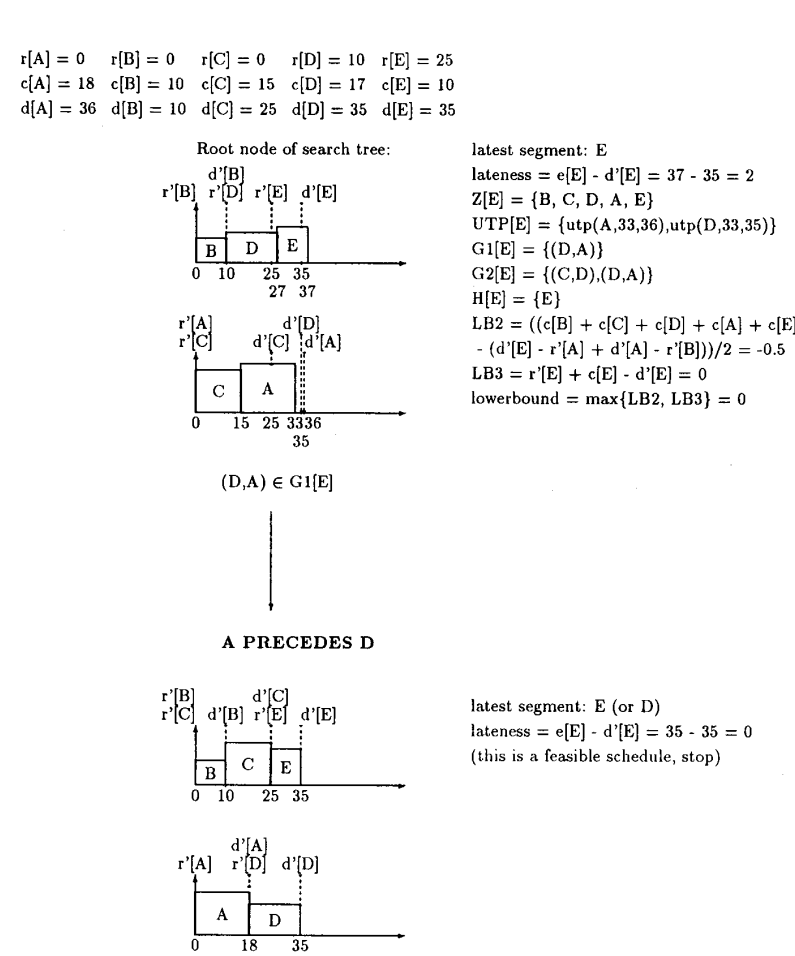


Fig. 1. Example 1.

At any time $t, t \in [0, \infty)$, we say segment "j is $ELIGIBLE$ at t on m" iff:

i) $\nexists i, i \neq j : i$ executes at t on m

ii) $t \geq r'[j] \wedge \neg(e[j] \leq t) \wedge \neg(s[j] < t) \wedge \neg(\exists m_1, m_1 \neq m : j$ executes at t on $m_1)$

iii) $\nexists i : i\ PRECEDES\ j \wedge \neg(e[i] \leq t)$

iv) $\nexists i : i\ EXCLUDES\ j \wedge s[i] < t \wedge \neg(e[i] \leq t)$

v) $\nexists i : i\ BEFORE\ j \wedge \neg(s[i] \leq t)$

The above definition guarantees that at any time t, if segment j is $ELIGIBLE$ at t on m, then j can be put into execution at t on processor m while satisfying the $BEFORE$ relations and all the properties of a valid schedule.

A *valid initial solution* for a set of processes P is a valid schedule of P that (in addition to satisfying all the properties of a valid schedule) satisfies the following properties:

$\forall t \in [0, \infty)$:

i) $\forall j, j \in S(P) : (s[j] = t \wedge j$ executes at t on $m)$
$\Rightarrow j$ is $ELIGIBLE$ at t on m

ii) $\forall j$:
$\exists i : i$ is $ELIGIBLE$ at t on m
$\wedge((d'[i] < d'[j])$
$\vee(d'[i] = d'[j] \wedge (\exists k : i\ BEFORE\ k$
$\vee\ i\ PRECEDES\ k) \wedge (\nexists k : j\ BEFORE\ k$
$\vee j\ PRECEDES\ k))$
$\vee(d'[i] = d'[j] \wedge (\exists k : i\ BEFORE\ k$
$\vee\ i\ PRECEDES\ k) \wedge (\exists k : j\ BEFORE\ k$
$\vee j\ PRECEDES\ k) \wedge c[i] > c[j]))$
$\vee(d'[i] = d'[j] \wedge (\nexists k : i\ BEFORE\ k$
$\vee i\ PRECEDES\ k) \wedge (\nexists k : j\ BEFORE\ k$
$\vee j\ PRECEDES\ k) \wedge c[i] > c[j]))$
$\Rightarrow \neg(j$ executes at t on $m)$

iii) $\exists m$: processor m is idle at t
$\Rightarrow \nexists i : i$ is $ELIGIBLE$ at t on m

Condition i) above states that a segment j can start execution at time t on processor m only if j is $ELIGIBLE$ at t on m. Condition ii) states that in a valid initial solution, if there exists at least one segment i such that i is $ELIGIBLE$ at time t on m, and either i has a shorter deadline than segment j; or, if i and j have the same deadline and i is $BEFORE$ or $PRECEDES$ some segment k but j is not $BEFORE$ and does not $PRECEDES$ any segment k; or, if i and j have the same deadline and i is $BEFORE$ or $PRECEDES$ some segment k and j is $BEFORE$ or $PRECEDES$ some

TABLE I

	i PC j	j PC i	i EX j	j EX i	i BF j	j BF i	$\neg(j$ EX $i)$
i PC j:		x				x	
i EX j:							x
i BF j:		x				x	

i PC j : i *PRECEDES* j.
i EX j : i *EXCLUDES* j.
i BF j : i *BEFORE* j
x: inconsistent

segment k, and i has a longer computation time than j; or, if i and j have the same deadline and i is not *BEFORE* and does not *PRECEDES* any segment k and j is not *BEFORE* and does not *PRECEDES* any segment k, and i has a longer computation time than j; then segment j cannot start executing at time t on processor m. Condition iii) states that in a valid initial solution, a processor can be idle at all times only if there is no *ELIGIBLE* segment that can be executed on that processor at that time. Condition iii) effectively guarantees that all segments will eventually be completed in a valid initial solution, provided that all relations on segments are "consistent" defined as follows.

We define each pair of relations on segments indicated by an "x" in Table I to be *inconsistent*. All other pairs of relations on segments are *consistent*.

In addition to satisfying release time, exclusion, and precedence constraints, a valid initial solution also satisfies execution priority constraints defined by the set of *BEFORE* relations and deadlines.

Initially, we set the set of *BEFORE* relations to be empty. New *BEFORE* relations as well as new *PRECEDE* relations are defined and used by the algorithm to reschedule the latest segment earlier in order to improve on existing valid initial solutions.

B. A Procedure for Computing a Valid Initial Solution

The procedure given at the bottom of this page uses an earliest-deadline-first strategy to compute a valid initial solution in which release time constraints and a given set of *EXCLUDE*, *PRECEDE*, and *BEFORE* relations are enforced.

At the release time or completion time of each segment, the procedure checks to see whether each processor is idle or not. If some processor m is idle, then the procedure first selects among all *ELIGIBLE* segments the segment that has the shortest deadline for execution. If more than one segment have the shortest deadline then the procedure selects among them the segment that *PRECEDES* or is *BEFORE* some other segment. If more than one segment not only has the shortest deadline, but also *PRECEDES* or is *BEFORE* some other segment, then the procedure selects among them the segment that has the greatest computation time. If there are still ties, the procedure selects an arbitrary one among them. Note that this selection process corresponds exactly to the conditions specified in ii) in the definition of a valid initial solution. Because the procedure only selects segments that are *ELIGIBLE* for execution, condition i) in that definition is also satisfied.

At any time t, a necessary condition for a segment j to be *ELIGIBLE* at t on some processor m is $t \geq r'[j]$. First, the procedure checks at $t = r'[j]$ for every segment j, to see whether there exists an idle processor m such that j can be put into execution at t on m. If at $t = r'[j]$ there exists an idle processor m, the procedure will put j into execution at $t = r'[j]$ on m unless there exists some other segment i such that i has not completed its computation and either 1) i contradicts the conditions iii)–v) for j to be *ELIGIBLE* at $t = r'[j]$ on m; or 2) i must be selected for execution at t on m before j according to the conditions specified in ii) in the definition of a valid initial solution. In the case of either 1) or 2), segment j cannot be put into execution until some other segment i completes its computation. In either case, whenever some other segment i completes its computation, the procedure will check again to see whether j can be put into execution at that time according to the same conditions. Thus, the procedure also satisfies condition iii) in the definition of a valid initial solution. See Examples 1–4

```
t ← 0
  while ¬(∀i: e[i] ≤ t) do
    begin
      if (∃i: t = r'[i] ∨ t = e[i]) then
        begin
          for m:= 1 to M do
            begin
              if (processor m is idle at t) then
                begin
                  Among the set {j|j is ELIGIBLE at t on m} select the segment j that has min d'[j]. In case of ties,
                  first select the segment j such that ∃k: j BEFORE k∨j PRECEDES k. In case of further ties,
                  select the segment j that has max c[j]. Let j execute from t to t + c[j] on processor m, i.e.,
                  ∀k, 0 ≤ k ≤ c[j] − 1: π(j, k) ← (t + k, m) (s[j] ← t; e[j] ← t + c[j]) end
                end
            end
        t ← t + 1
    end
```

(Figs. 1–4) for examples of schedules corresponding to valid initial solutions.

V. How to Improve on a Valid Initial Solution

A. The Set $Z[l]$ and Usable Time Periods

Suppose that the valid initial solution is not feasible. Let l be the latest segment in that valid initial solution. (If there exists more than one segment that have maximum lateness, then let l be the segment that completed last among those segments.)

A feasible schedule may be found only if l can be rescheduled earlier.

We define the set of segments $Z[l]$ recursively as follows:

1) $l \in Z[l]$;
2) $\forall i,\ s[i] < s[l]$,
   if $\exists j,\ j \in Z[l]$;
   $r'[j] < s[j] \wedge r'[j] < e[i]$
   then $i \in Z[l]$

As an example, in the valid initial solution of the root node of the search tree of Example 3 (Fig. 3): E is the latest segment. $E \in Z[E]$ from 1); From 2), $F \in Z[E]$ because $s[F] < s[E] \wedge r'[E] < e[F] \wedge r'[E] < s[E]$; $D \in Z[E]$ because $s[D] < s[E] \wedge r'[E] < e[D] \wedge r'[E] < s[E]$; $A \in Z[E]$ because $s[A] < s[E] \wedge r'[F] < e[A] \wedge r'[F] < s[F]$; $B, C \in Z[E]$ because $s[B] = s[C] < s[E] \wedge r'[A] < e[B] < e[C] \wedge r'[A] < s[A]$; Thus $Z[E] = \{B, C, A, D, F, E\}$.

We introduce the notion of the *set of usable time periods of segment l*: $UTP[l] = \{utp(i, t_s, t_e)\}$. Each *usable time period*, $utp(i, t_s, t_e)$, is defined as a continuous period of time $[t_s, t_e]$ that lies between the adjusted release time $r'[i]$ and the adjusted deadline $d'[i]$ of some segment i in the set $Z[l]$ such that some processor m is not occupied by any segment in the set $Z[l]$, and i does not execute at any time between t_s and t_e.

$$\begin{aligned}
&\forall utp(i, t_s, t_e) \in UTP[l]: \\
&\quad \exists i,\ i \in Z[l]:\ r'[i] \le t_s < t_e \le d'[i] \\
&\quad \wedge (e[i] \le t_s \vee t_e \le s[i]) \\
&\quad \wedge (\exists m,\ 1 \le m \le M,\ \forall t \in [t_s, t_e - 1],\ \nexists j \in Z[l]: \\
&\qquad j \text{ executes at } t \text{ on processor } m).
\end{aligned}$$

In any schedule that corresponds to a valid initial solution:

1) $Z[l]$ is a set of segments whose start time precede (and include) l's start time in a period of continuous utilization of processors such that within that time period there does not exist any time where *all* M processors are idle.
2) If the schedule is not feasible, then a feasible schedule can be found only if it is possible to reschedule some segment $i \in Z[l]$ such that some portion of a usable time period $utp(i, t_s, t_e) \in UTP[l]$ is occupied by i.
3) If the schedule is not feasible and either no usable time period exists for the latest segment l, or, no segment $i \in Z[l]$ can occupy any portion of any usable time period of l without causing itself or some other segment to be late, then no feasible schedule exists for the given set of segments.

(A proof of these properties can be found in Appendix II.)

B. The Expand Sets $G_1[l]$ and $G_2[l]$

We define two *expand sets of* l, $G_1[l]$ and $G_2[l]$ as follows:

$$\begin{aligned}
G_1[l] = \{&(i, j) | i, j \in Z[l] \\
&\wedge i \ne j \\
&\wedge r'[j] + c[j] + c[i] \le d'[i] \\
&\wedge \exists utp(i, t_s, t_e) \in UTP[l]: e[j] = t_s \\
&\quad \wedge \neg(i\ PRECEDES\ j) \\
&\quad \wedge \neg(i\ BEFORE\ j)\} \\
G_2[l] = \{&(i, j) | i, j \in Z[l] \\
&\wedge i \ne j \\
&\wedge M \ge 2 \\
&\wedge UTP[l] \ne \varnothing \\
&\wedge r'[i] + c[i] < d'[i] \\
&\wedge r'[j] + c[i] \le d'[i] \\
&\wedge \neg(i\ PRECEDES\ j) \\
&\wedge \neg(i\ BEFORE\ j) \\
&\wedge \exists k:\ k \in Z[l] \wedge s[i] < s[k] \wedge r'[k] < e[i] \wedge r'[k] < s[k] \\
&\quad \wedge r'[k] + c[i] \le d'[i] \\
&\quad \wedge \neg(i\ PRECEDES\ k) \\
&\quad \wedge \neg(i\ BEFORE\ k) \\
&\wedge s[j] = \min\{s[k_1] | (s[i] < s[k_1] \wedge k_1 \in Z[l]) \\
&\quad \wedge r'[k_1] + c[i] \le d'[i] \\
&\quad \wedge \neg(i\ PRECEDES\ k_1) \\
&\quad \wedge \neg(i\ BEFORE\ k_1)\}\}
\end{aligned}$$

The expand subset $G_1[l]$ contains all pairs of segments i and j in $Z[l]$ such that if j is completed before i is started by adding the relation $j\ PRECEDES\ i$ and recomputing a new valid initial solution, then i may occupy some portion of a usable time period and allow the latest segment l to be scheduled earlier, thus reducing the maximum lateness.

The expand subset $G_2[l]$ contains all pairs of segments i and j in $Z[l]$ such that if j is scheduled not later than i by adding the relation $j\ BEFORE\ i$ and recomputing a new valid initial solution, then some segment may occupy some portion of a usable time period and allow the latest segment l to be scheduled earlier, thus reducing the maximum lateness.

Through the expand set $G_2[l]$, $BEFORE$ relations are used to reverse the ordering of each segment i and the "next" segment j in $Z[l]$ in the valid initial solution ($s[j] = \min\{s[k_i] | s[i] < s[k_1]\}$). Since a feasible schedule can be found only if it is possible to re-schedule some segment $i \in Z[l]$ such that some portion of a usable time period $utp[i, t_s, t_e] \in UTP[l]$ is occupied by i, it is possible to enumerate all feasible solutions through the expand set $G_2[l]$ alone.

For each pair of segments i and j such that i occurs before j in the valid initial solution: if $r'[j] + c[i] > d'[i]$ then reversing the order of i and j by adding the relation j

BEFORE i will result in i being late; if i *PRECEDES* j or i *BEFORE* j then one cannot add the relation j *BEFORE* i; if there does not exist any segment k that can be moved earlier by occupying a portion of the space previously occupied by segment i, i.e., $\nexists k: k \in Z[l] \wedge s[i] < s[k] \wedge r'[k] < e[i] \wedge r'[k] < s[k]$, then moving segment i later by adding the relation j *BEFORE* i will not have any beneficial effect on the lateness of the latest segment l. In these cases, in the definition of $G_2[l]$, the algorithm excludes further consideration of any ordering in which j *BEFORE* i.

Through the expand set $G_1[l]$ one can accelerate considerably the search for a feasible solution. For each segment i such that a usable time period $utp[i, t_s, t_e]$ exists, by adding the relation j *PRECEDES* i where segment j is the segment immediately preceding the usable time period $utp[i, t_s, t_e]$, i.e., $e[j] = t_s$, $G_1[l]$ positions segment i such that either i immediately occupies a portion of $utp[i, t_s, t_e]$, or, later on i could be pushed forward to occupy a portion of $utp[i, t_s, t_e]$ when pairs of segments occurring before i are reordered by *BEFORE* relations through subsequent expand sets $G_2[l]$.

For each pair of segments i and j such that i occurs before j in the valid initial solution: if $r'[j] + c[j] + c[i] > d'[i]$ then adding the relation j *PRECEDES* i will result in i being late; if i *PRECEDES* j or i *BEFORE* j then one cannot add the relation j *PRECEDES* i. In these cases, the algorithm excludes from further consideration any ordering in which j *PRECEDES* i in the definition of $G_1[l]$.

We note that in the single processor case, i.e., when $M = 1$, it is possible to enumerate all feasible solutions by using *PRECEDES* relations defined by the expand set $G_1[l]$ alone.

In Example 1 (Fig. 1), the valid initial solution of the root node of the search tree has two usable time periods of E: $utp(D, 33, 35)$ and $utp(A, 33, 36)$. The expand subset $G_1[E]$ contains one pair of segments: (D, A). The expand subset $G_2[E]$ contains two pairs of segments: (D, A) and (C, D). Corresponding to the pair $(D, A) \in G_1[E]$, we add the relation A *PRECEDES* D. Then when we recompute a new valid initial solution that satisfies the relation A *PRECEDES* D, segment D occupies the usable time period $utp(D, 33, 35)$, allowing E to be completed before its deadline, which results in a feasible schedule. Note that the relation A *PRECEDES* D results in $r'[D]$ and $d'[A]$ having a new adjusted value 18.

In Example 2 (Fig. 2), the valid initial solution of the root node of the search tree has one usable time period of G: $utp(E, 53, 55)$. The expand subset $G_1[G]$ is empty because the segment immediately preceding the usable time period $utp(E, 53, 55)$ is E itself. The expand subset $G_2[G]$ contains two pairs of segments: (D, A) and (C, A). Corresponding to the pair $(D, A) \in G_2[G]$, we add the relation A *BEFORE* D. Then when we recompute a new valid initial solution that satisfies the relation A *BEFORE* D, segment E occupies the usable time period $utp(E, 53, 55)$, allowing G to be completed before its deadline, which results in a feasible schedule.

In Example 3 (Fig. 3), the valid initial solution of the root node of the search tree has three usable time periods of E: $utp(C, 15, 20)$, $utp(F, 40, 49)$ and $utp(A, 40, 55)$. The expand subset $G_1[E]$ is empty. The expand subset $G_2[E]$ contains three pairs of segments: (C, A), (A, D), and (F, E). Corresponding to the pair $(C, A) \in G_2[E]$, we add the relation A *BEFORE* C. Then when we recompute a new valid initial solution that satisfies the relation A *BEFORE* C, segment C occupies a portion of the usable time period $utp(C, 15, 20)$, allowing E to be completed before its deadline, which results in a feasible schedule.

In Example 4 (Fig. 4), the algorithm finds a feasible schedule in which a set of *EXCLUDE* relations as well as an initial set of *PRECEDES* relations are satisfied.

C. A Lateness Lower Bound

We now show how to compute a lower bound on the lateness of any valid initial solution satisfying a given set of *EXCLUDE*, *PRECEDE*, and *BEFORE* relations.

let $H[l] = \{i | i \in Z[l] \wedge d'[i] \leq d'[l]$
$\wedge \nexists k \in Z[l]: d'[k] > d'[l] \wedge s[i] \leq s[k]$
$\wedge \nexists t, m: s[i] \leq t < s[l] \wedge$ processor m is idle at $t\}$

if $|H[l]| \geq M$ then:

let $r'_{e_1}, r'_{e_2}, \ldots, r'_{e_M}$ be the M earliest adjusted release times in $H[l]$, i.e.,

$$r'_{e_1} = \min\{r'[i] | i \in H[l]\},$$
$$\exists i_1, r'[i_1] = r'_{e_1}:$$
$$r'_{e_2} = \min\{r[i] | i \in (H[l] - \{i_1\})\},$$
$$\exists i_2, r'[i_2] = r'_{e_2}:$$
$$r'_{e_3} = \min\{r'[i] | i \in (H[l] - \{i_1, i_2\})\},$$
$$\ldots$$
$$\exists i_{M-1}, r'[i_{M-1}] = r'_{e_{M-1}}:$$
$$r'_{e_M} = \min\{r'[i] | i \in (H[l] - \{i_1, i_2, \ldots, i_{M-1}\})\}.$$

let $d'_{l_1}, d'_{l_2}, \ldots, d'_{e_M}$ be the M latest adjusted deadlines in $H[l]$, i.e.,

$$d'_{l_1} = \max\{d'[i] | i \in H[l]\},$$
$$\exists j_1, d'[j_1] = d'_{l_1}:$$
$$d'_{l_2} = \max\{d'[i] | i \in (H[l] - \{j_1\})\},$$
$$\exists j_2, d'[j_2] = d'_{l_2}:$$
$$d'_{l_3} = \max\{d'[i] | i \in (H[i] - \{j_1, j_2\})\},$$
$$\ldots$$
$$\exists j_{M-1}, d'[j_{M-1}] = d'_{l_{M-1}}:$$
$$d'_{l_M} = \max\{d'[i] | i \in (H[l] - \{j_1, j_2, \ldots, j_{M-1}\})\}.$$

$$LB_1 = \frac{\sum_{i \in H[l]} c[i] - \sum_{k=1}^{M} (d'_{l_k} - r'_{e_k})}{M}$$

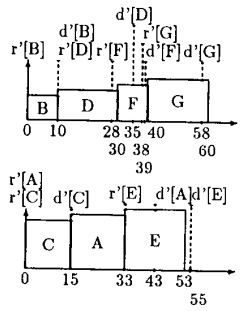
Similarly, if $|Z[l]| \geq M$ then:

let $r''_{e_1}, r''_{e_2}, \ldots, r''_{e_M}$ be the M earliest adjusted release times in $Z[l]$.

let $d''_{l_1}, d''_{l_2}, \ldots, d''_{e_M}$ be the M latest adjusted deadlines in $Z[l]$.

r[A] = 0	r[B] = 0	r[C] = 0	r[D] = 10	r[E] = 33	r[F] = 28	r[G] = 38
c[A] = 18	c[B] = 10	c[C] = 15	c[D] = 20	c[E] = 20	c[F] = 10	c[G] = 20
d[A] = 43	d[B] = 10	d[C] = 15	d[D] = 35	d[E] = 55	d[F] = 39	d[G] = 58

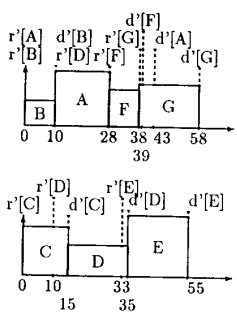
Root node of search tree:



latest segment: G
lateness = e[G] - d'[G] = 60 - 58 = 2
Z[G] = {B, C, D, A, F, E, G}
UTP[G] = {utp(E,53,55)}
G1[G] = ∅
G2[G] = {(D,A),(C,A)}
H[G] = {B, C, D, A, F, E, G}
LB1 = LB2 = ((c[B] + c[C] + c[D] + c[A] + c[F] + c[E] + c[G]) - (d'[G] - r'[A] + d'[E] - r'[B]))/2 = 0
LB3 = r'[G] + c[G] - d'[G] = 0
lowerbound = max{LB1, LB2, LB3} = 0

(D,A) ∈ G2[G]

A BEFORE D



latest segment: G
lateness = e[G] - d'[G] = 58 - 58 = 0
(this is a feasible schedule, stop)

Fig. 2. Example 2.

$$LB_2 = \frac{\sum_{i \in Z[l]} c[i] - \sum_{k=1}^{M} (d''_{l_k} - r''_{e_k})}{M}$$

$LB_3 = \max\{r'[i] + c[i] - d'[i] | i \in S(P)\}$
if $|H[l]| \geq M$ then lowerbound $= \max\{LB_1, LB_2, LB_3\}$
else if $|H[l]| < M \wedge |Z[l]| \geq M$ then lowerbound $=$ $\max\{LB_2, LB_3\}$
else lowerbound $= LB_3$

In the computation of the lower bound function, LB_1 considers the subset $H[l]$ of segments in $Z[l]$ that do not result in idle intervals in the schedule and that do not have deadlines greater than the deadline of the latest segment l.

A necessary condition for obtaining a feasible schedule is that the sum of the computation times of all the segments in the set $H[l]$, i.e., $\sum_{i \in H[l]} c[i]$ must at least fit into the sum of the M processor time intervals that are bounded by the M earliest adjusted release times and M latest deadlines in $H[l]$, i.e.,

$$\sum_{k=1}^{M} (d'_{l_k} - r'_{e_k})$$

Suppose that it is possible to schedule each of the M segments in $H[l]$ that have the M least release times first on each of the M processors; and, it is possible to schedule the M segments in $H[l]$ that have the M greatest deadlines last on each of the M processors; and, it is possible to schedule the remaining segments in $H[l]$ between the first and last segments on each processor such that there is no gap in the schedule and the differences between the completion times and deadlines of the last segment on every processor, that is, their lateness, are exactly the same. This amounts to trying to evenly fit the computation times of all the segments in $H[l]$ into the M largest processor time intervals. Furthermore, suppose that in such a schedule we have an ideal situation where each subset of segments in $H[l]$ that execute on a same processor have the same release times as the first segment in the subset on that

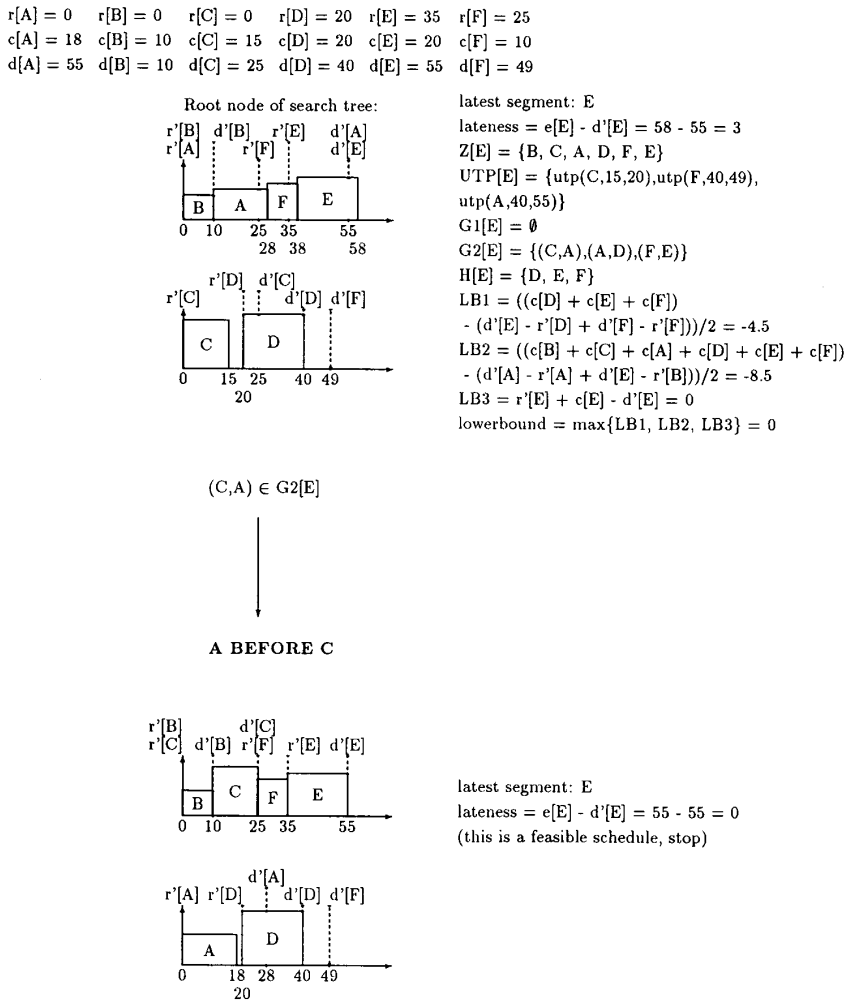


Fig. 3. Example 3.

processor, and have the same deadlines as the last segment in the subset on that processor. Then no matter how we rearrange the order of the segments in such a hypothetical schedule, the lateness of the schedule cannot be reduced, because if we try to reduce the lateness of the last segment on one processor by decreasing the total of the computation times of segments on that processor, we would have to increase the total of the computation times of segments on some other processor, which would increase the lateness of the last segment on the other processor. Any attempt to replace the last segment on a processor with a segment that has a greater deadline would not help either, because the segments that have the greatest deadlines are already scheduled at the end on each processor. The lateness of such a hypothetical schedule would be equal to the lateness of the last segment on each processor, which is

$$\frac{\sum_{i \in H[l]} c[i] - \sum_{k=1}^{M} (d'_{l_k} - r'_{e_k})}{M}.$$

When scheduling the actual set of segments in $H[l]$, one cannot achieve a smaller lateness than the lateness of this hypothetical schedule. This is because the actual set of segments $H[l]$ would always have release times that are greater or equal to that already assumed and always have deadlines that are shorter or equal to that assumed. Furthermore, it may not be possible to schedule them such that there is no gap in the schedule and the lateness of the last segment on each processor are exactly the same. This proves that LB_1 is a lower bound on the lateness of the set of segments in $H[l]$.

LB_2 considers all the segments in $Z[l]$, and can be proved similarly. LB_3 is a trivial lower bound that considers each segment individually in the whole set of $S(P)$.

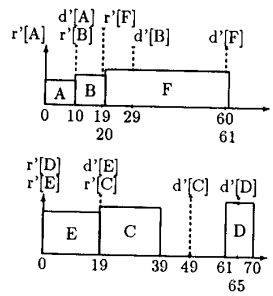
The lower bounds LB_1 and LB_2 are valid only when there are at least M segments in the sets $H[l]$ and $Z[l]$ respectively, whereas LB_3 is valid regardless of the number of segments in the sets $H[l]$ and $Z[l]$.

VI. Searching for an Optimal or Feasible Solution

We now define a search tree that has as its root node the valid initial solution that satisfies all the $EXCLUDE$ and *PRECEDE* relations in the original problem specification.

r[A] = 0	r[B] = 10	r[C] = 19	r[D] = 0	r[E] = 0	r[F] = 19	**F EXCLUDES D**
c[A] = 10	c[B] = 10	c[C] = 20	c[D] = 9	c[E] = 19	c[F] = 41	**A PRECEDES B**
d[A] = 10	d[B] = 29	d[C] = 49	d[D] = 65	d[E] = 19	d[F] = 60	

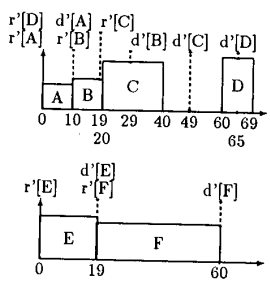
Root node of search tree:



latest segment: D
lateness = e[D] - d'[D] = 70 - 65 = 5
Z[D] = {A, E, B, C, F, D}
UTP[D] = {utp(C,39,49),utp(D,39,61)}
G1[D] = ∅
G2[D] = {(E,D),(B,C),(C,F),(F,D)}
H[D] = { D}
LB2 = ((c[A] + c[E] + c[B] + c[F] + c[C] + c[D])
- (d'[D] - r'[A] + d'[F] - r'[D]))/2 = -8
LB3 = r'[E] + c[E] - d'[E] = 0
lowerbound = max{ LB2, LB3} = 0

(C,F) ∈ G2[D]

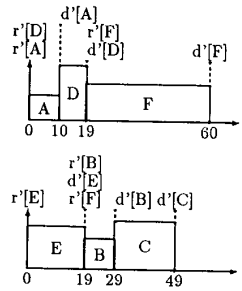
F BEFORE C



latest segment: D
lateness = e[D] - d'[D] = 69 - 65 = 4
Z[D] = {A, E, B, F, C, D}
UTP[D] = {utp(C,40,49)}
G1[D] = ∅
G2[D] = {(E,D),(B,F),(F,D),(C,D)}
H[D] = {D}
LB2 = ((c[A] + c[E] + c[B] + c[F] + c[C] + c[D])
- (d'[D] - r'[A] + d'[F] - r'[D]))/2 = -8
LB3 = r'[E] + c[E] - d'[E] = 0
lowerbound = max{ LB2, LB3} = 0

(B,F) ∈ G2[D]

F BEFORE B



latest segment: F
lateness = e[F] - d'[F] = 60 - 60 = 0
(this is a feasible schedule, stop)

Fig. 4. Example 4.

At each node in the search tree, we compute the lower bound and the expand sets. Let segment l be the latest segment in the valid initial solution computed at that node.

For each pair of segments $(i, j) \in G_1[l]$, we create a successor node that corresponds to a new problem in which we assign a new relation j *PRECEDES* i. If we apply the aforementioned procedure and compute a new valid initial solution in which the new relations are enforced, then segment j will be completed before segment i is started in the new schedule. Then segment i will occupy a portion of a usable time period of l, allowing l to be scheduled earlier if possible.

For each pair of segments $(i, j) \in G_2[l]$, we create a successor node that corresponds to a new problem in which we assign a new relation j *BEFORE* i. If we apply the aforementioned procedure and compute a new valid initial solution in which the new relations are enforced, then segment i will be started at a time later than or equal to the start time of segment j in the new schedule. Then some segment may occupy a portion of a usable time period of l, allowing l to be scheduled earlier if possible.

After generating the valid initial solution for each new successor node, we test it for feasibility. If a feasible solution is not discovered among any of the resulting problems, then we proceed to create new successor nodes in a similar manner. We use a strategy of branching from the node with the least lower bound. In case of ties, we choose the node with least lateness among the nodes with least lower bound.

The steps of the algorithm are listed as follows. (For a more detailed implementation of the algorithm see Appendix I.)

Step 0: Compute an initial valid solution and the corresponding lowerbound. Find the latest segment l and its lateness. If its lateness is less than or equal to zero then stop, a feasible schedule has been found. If its lowerbound is greater than zero then stop, no feasible schedule can ever be found. Otherwise, call this node the parent node.

Step 1: Find the expand sets $G_1[l]$ and $G_2[l]$ and create $|G_1[l]| + |G_2[l]|$ new child nodes.

For each node corresponding to a pair of segments $(i, j) \in G_1[l]$, assign a new relation j *PRECEDES* i. For each node corresponding to a pair of segments $(i, j) \in G_2[l]$, assign a new relation j *BEFORE* i. Let each child node inherit all relations assigned to any of its predecessor nodes.

Recompute a valid initial solution, lowerbound, and find the latest segment and its lateness for each child node.

Step 2: If steps 3 and 4 have been performed for all child nodes then chose the parent node and go to step 5, otherwise, select the child node with the least lateness.

Step 3: Set minlateness $\leftarrow$ min {minlateness, lateness(childnode)}.

If minlateness is less than or equal to zero then stop, a feasible solution has been found.

Step 4: If lowerbound(childnode) > 0 then close this child node and return to step 2, this node will never lead to a feasible solution.

Return to step 2.

Step 5: Select among all open nodes a node with the least lower bound, in case of ties, select the node with least lateness. Call this node the parent node and go to step 1. ■

(See examples 1–4 in Figs. 1–4.)

One could also adopt a strategy of terminating the search whenever a schedule has been found such that its lateness is within a prespecified ratio of optimal. An upperbound on that ratio can be computed with the formula (lateness$-L$)/L where L is the least lower bound of all nodes belonging to the open node set.

VII. Empirical Behavior of the Algorithm

We have written a program in Pascal that implements the multiprocessor algorithm previously described. We have tested our algorithm on a variety of sample problems. To be concise, here we only describe the set of characteristics of one typical sample problem.

Each instance of our typical sample problem consists of 420 segments, which belong to 180 processes. Among them are 100 "interdependent" processes and 80 "independent" processes. Segments that belong to interdependent processes exclude and/or precede segments belonging to one to four other interdependent processes, whereas segments belonging to independent processes do not have exclusion relations or precedence relations with segments belonging to other processes.

Each interdependent process consists of three to five segments. Precedence relations are defined on consecutive pairs of segments belonging to each interdependent process to ensure the proper ordering of segments within the process.

Computation times of interdependent processes are randomly drawn to lie between 1 and a maximum computation time $c_{\max}$.

Interdependent processes are divided into "groups," where segments of processes in each group exclude and/or precede segments of processes in the same group.

Interdependent processes belonging to the same group S_k have the same release time r_{S_k} for the first segment in each process. r_{S_k} is randomly drawn to lie between 0 and R_{S_k}, where

$$R_{S_k} = Lengthfactor * \left(\frac{1}{M}\right) * \left(\sum_{i \in S(P)} c[i] - \sum_{j \in S_k} c[j]\right).$$

Interdependent processes belonging to the same group S_k have the same deadline d_{S_k} for the last segment in each process. d_{S_k} is randomly drawn to lie between $D1_{S_k}$ and $D2_{S_k}$, where

$$D1_{S_k} = R_{S_k} + Lengthfactor * \left(\frac{1}{M}\right) * \left(\sum_{j \in S_k} c[j]\right)$$

and

$$D2_{S_k} = Lengthfactor * \left(\frac{1}{M}\right) * \left(\sum_{i \in S(P)} c[i]\right).$$

The release time of each segment after the first is given a value that is equal to the release time of the first segment plus the sum of the computation times of all segments that precede it in the same interdependent process.

TABLE II
LENGTH FACTOR BRANCHES NUMBER OF NODES

No. of Segments	M	$\lvert P_{\text{inter}}\rvert$	$\lvert P_{\text{indep}}\rvert$	$c_{\max}$	Length Factor Min	Length Factor Max	No. of Trials	Branches > 2	Branches > 5	No. of Nodes Min	No. of Nodes Max	No. of Nodes Avg
50	2	10	16	20	1.0	2.0	40	1	0	1	176	6.5
125	2	25	40	20	1.0	2.0	40	1	0	1	538	23.7
250	2	50	80	20	1.0	2.0	40	2	1	1	845	27.3
330	2	75	75	20	1.0	2.0	40	2	1	1	1154	36.3
420	2	100	80	20	1.0	2.0	40	2	1	1	1367	39.5
420	3	100	80	20	1.0	2.0	40	1	0	1	775	32.6

M is the number of processors; $|P_{\text{inter}}|$ is the number of interdependent processes; $|P_{\text{indep}}|$ is the number of independent processes; $c_{\max}$ is the maximum computation time of a segment; *min* and *max Length Factor* are, respectively, the minimum and maximum values of *Length Factor; No. of Trials* is the number of cases generated and tested. > 2 and > 5 *Branches* are, respectively, the number of trials in which the algorithm branched more than two and five times; *Min, Max,* and *Avg No. of Nodes* are, respectively, the minimum, maximum, and average number of nodes generated in the search tree in trials in which the algorithm did not branch more than five times.

The deadline of each segment before the last is given a value that is equal to the deadline of the last segment minus the sum of the computation times of all segments that it precedes in the same interdependent process.

Each independent process consists of one single segment. The computation times of independent processes are also randomly drawn to lie between a minimum and maximum computation time. The release time $r[i]$ of each independent process consisting of a single segment i is randomly drawn to lie between 0 and

$$Lengthfactor * \left(\frac{1}{M} * \left(\sum_{j \in S(P)} c[j]\right) - c[i]\right).$$

The deadline $d[i]$ of each independent process consisting of a single segment i is randomly drawn to lie between $r[i] + c[i]$ and

$$Lengthfactor * \left(\frac{1}{M}\right) * \left(\sum_{j \in S(P)} c[j]\right).$$

We have tried to structure our sample problem instances in a way that we think will resemble real application problems. For example, the release times and deadlines of segments were chosen in a way that would greatly reduce the probability of having trivially inconsistent release times and deadlines. We gave identical release times for the first segment and identical deadlines for the last segment for all interdependent processes that belong to the same group based on the fact that interdependent processes that have exclusion and/or precedence relations on their segments are most likely to be constrained to occur in the same time period.

In general, a smaller value of *Lengthfactor* results in smaller amounts of time between the release times and deadlines of segments, which makes it more difficult to find a feasible solution. However, when the value of *Lengthfactor* decreases to a certain point then it might be easy to determine that no feasible solution exists.

We used the number of nodes generated in the search tree as the primary performance measure because it is machine independent and because it basically determines the size of the problem that can be computed effectively. We also observed the number of branches required by the algorithm, because this has a major effect on the number of nodes generated.

Table II contains results of testing the algorithm. In each row, 40 test cases were generated as described before. The length factor in the test cases were randomly drawn to lie between 1.0 and 2.0.

Our tests indicated that the algorithm is quite effective. In only a few cases did our algorithm have to branch more than twice to find a feasible schedule or determine that no feasible schedule exists. When the problem size in terms of the number of segments increased, we found that the increase in the average number of nodes generated in the search tree was quite moderate. Thus we believe that our algorithm should be capable of handling relatively large problem sizes.

We also performed tests on many variations of the typical sample problem described earlier. For example, we varied the ratio of the number of independent processes to the number of interdependent processes, the number of segments in each process, the number of precedence relations and exclusion relations defined on segments that belong to different interdependent processes, the number of interdependent processes within each group, and the minimal and maximum computation times between which the computation times of segments belonging to independent and interdependent processes were randomly drawn. However, on these variations, we did not find any significant difference in the effectiveness of our algorithm.

VIII. CONCLUDING REMARKS

In this paper, we presented an algorithm that solves the problem of finding a nonpreemptive feasible schedule whenever one exists for a set of processes on M processors that satisfies a given set of release time, deadline, precedence, and exclusion constraints defined on process segments. We do not know of any other published algorithm that is capable of solving this problem.

The algorithm can be used to automate pre-run-time scheduling of processes with arbitrary precedence and exclusion relations on multiprocessors in hard-real-time systems.

A previously published algorithm [17] allows selective preemptions but is restricted to single-processor systems. The algorithm presented in this paper does not allow preemptions, but it can be applied when there are multiple processors. Both algorithms use a branch-and-bound strategy, but the strategies for generating the "expand sets" and the lower-bound function are completely different in the two algorithms. The concepts of "*BEFORE*" *relations*, "*adjusted deadlines*," and "*usable time periods*" are fundamental to the algorithm in this paper and did not exist in [17].

Although the algorithm in this paper does not explicitly allow preemptions, the formulation of the problem in terms of precedence relations defined on ordered pairs of process segments makes it possible to allow preemptions at predefined points in a process by dividing processes into separate segments at those points and defining appropriate precedence relations on pairs of consecutive segments to guarantee the proper ordering of segments within each process.

In [20] a heuristic for scheduling nonpreemptive tasks with resource constraints was presented. In that heuristic, CPU's are modeled as resources. For every resource R_j and every process p_i, whether p_i must use R_j is prespecified rather than determined by the scheduler. The heuristic in [20] is a good technique when each process must be scheduled on a particular fixed processor and no precedence relations are needed. However, the algorithm presented in this paper can be applied when precedence relations must be satisfied; and the processors are interchangeable.

Since the original problem is NP complete, finding a feasible solution whenever one exists requires exponential time in the worse case. Thus one may consider applying other heuristic techniques such as local neighborhood search and simulated annealing as alternative techniques to attack this problem.

Local neighborhood search attempts to improve on an initial solution by a series of local incremental changes. Simulated annealing attempts to avoid getting stuck in a poor but locally optimal solution by randomizing local alterations. In some cases, such techniques would take less time than a branch-and-bound algorithm to compute a schedule. However, these techniques are purely heuristic approaches that do not necessarily find a feasible solution even if one exists, nor are they capable of determining that no feasible schedule exists.

Because the algorithm presented in this paper is used for pre-run-time (off-line) scheduling, not for run-time (on-line) scheduling, we consider the algorithm's ability to find a feasible schedule whenever one exists, and, its ability to determine that no feasible schedule exists when that is the case, to be more important than the absolute time it takes to compute a schedule. This led us to choose "optimal techniques" over heuristic approaches.

When implementing this algorithm, it may be advantageous to make space-time trade-offs to match available resources. If our major constraint is space instead of time, we might consider only storing at each node partial information that is different from the information stored at its ancestor nodes, then whenever we need complete information to proceed at a certain node, we use the information stored at its ancestor nodes to reconstruct the complete information required at that node.

One may also include an initial problem parameter verification stage that performs a preliminary analysis of all the initial problem parameters and modifies or rejects if necessary any problem parameters that are either redundant or inconsistent with other parameters prior to using this algorithm.

For future work, we will explore ways of generalizing this algorithm to solve the problem of scheduling processes with release times, deadlines, precedence, and exclusion relations on M processors for the case where certain segments of a process are preemptable at any point in time by certain segments of other processes, and for the case where exclusion regions within each process overlap or are embedded within each other.

Appendix I
An Implementation of the Main Algorithm

```
begin {main}
  nodeindex: = 0;
  initialize(PC(nodeindex), EX);
  BF(nodeindex) ← ∅;
  nonfeasible: =false;
  feasible: =false;
  opennodeset ← ∅;
  if consistent(PC(nodeindex), EX, BF(nodeindex)) then
  begin
    schedule(nodeindex) ← validinitialsolution(PC
    (nodeindex), EX, BF(nodeindex));
    leastlowerbound: =lowerbound(nodeindex);
    if lowerbound(nodeindex)> 0 then
      nonfeasible: =true;
    if lateness(nodeindex) ≤ 0 then
      feasible: = true;
    if not (nonfeasible or feasible) then
    begin
      opennodeset ← {nodeindex};
      minlateness: =lateness(nodeindex);
      minlatenode: =nodeindex;
      while not (nonfeasible or feasible or
        spacetimelimitsexceeded) do
      begin
        lowestboundset ← {nd | lowerbound(nd)
          =leastlowerbound ∧ nd ∈ opennodeset };
        select parentnode such that
          lateness(parentnode) = min{lateness(node) | node
            ∈ lowestboundset}};
        l : =latestsegment(schedule(parentnode));
        firstchildnode: =nodeindex + 1;
        for each pair of segments (i, j) ∈ G1 (parentnode)
          begin
            nodeindex: =nodeindex + 1;
            PC(nodeindex) ← PC(parentnode) ∪ {(i, j)};
          end;
        for each pair of segments (i, j) ∈ G2(parentnode)
          begin
            nodeindex: =nodeindex + 1;
            BF(nodeindex) ← BF(parentnode) ∪ {(i, j)};
          end;
```

```
        opennodeset ← opennodeset - {parentnode};
        for childnode: =firstchildnode to nodeindex do
        begin
          if not feasible and consistent(PC(childnode), EX,
            BF(childnode)) then
          begin
            schedule(childnode) ←
              validinitialsolution(PC(childnode), EX,
                BF(childnode));
            if lateness(childnode) < minlateness then
            begin
              minlateness: =lateness(childnode);
              minlatenode: =childnode;
            end;
            if lateness(childnode) ≤ 0 then
              feasible: =true
            else
              if lowerbound(childnode) ≤ 0 then
              opennodeset ← opennodeset ∪ {childnode}
          end;
        end;
        leastlowerbound ← min{lowerbound(nd) | nd
          ∈ opennodeset};
        if opennodeset = ∅ and not feasible then
          nonfeasible := true;
      end;
      minlateschedule := schedule(minlatenode);
    end;
  end;
end.
```

(end of algorithm)

In the algorithm above, a node in the "opennodeset" is a node that does not have successors but may be selected as the node to be branched from next. "PC(nodeindex)" and "BF(nodeindex)" are, respectively, the set of *PRECEDE* relations and the set of $BEFORE$ relations associated with the node identified by "nodeindex." "EX" is the (constant) set of $EXCLUDE$ relations. "schedule(nodeindex)" is the valid initial solution computed using PC(nodeindex), EX and BF(nodeindex). "lateness(nodeindex)" is the lateness of schedule(nodeindex). "lowerbound(nodeindex), G_1(nodeindex), and G_2(nodeindex) are, respectively, the lowerbound and the two expand sets computed from schedule(nodeindex).

The algorithm terminates as soon as a feasible schedule in which all deadlines are met is found or when a predefined space/time limit is exceeded or when it is determined that no feasible schedule exists.

Appendix II
Proof of Some Properties of a Valid Initial Solution

Property 1: $Z[l]$ is a set of segments whose start time precede (and include) l's start time in a period of continuous utilization of processors such that within that time period there does not exist any time where all M processors are idle.

Proof: First we prove by contradiction that for any segment j, if $r'[j] < s[j]$, then there does not exist any time t between $r'[j]$ and $s[j]$ such that all M processors are idle.

Suppose this is not true, i.e., there exists some time t, $r'[j] \leq t < s[j]$, such that all M processors are idle. According to condition iii) in the definition of a valid initial solution, this implies that no segment is $ELIGIBLE$ at t on any processor m. Then at least one of the conditions in the definition of "j is $ELIGIBLE$ at t on m" must be false. But among those five conditions, segment j satisfies the first two, i.e., i) and ii), since $r'[j] \leq t < s[j]$. Condition iv) must also be true, otherwise if there exists some i such that: $i\ EXCLUDES\ j \wedge s[i] < t \wedge \neg(e[i] \leq t)$, this would imply that i is executing at time t on some processor m. So one of the two remaining conditions iii) or v) must be false, i.e., there must exist some segment i_1, such that either: $i_1\ PRECEDES\ j \wedge \neg(e[i_1] \leq t)$; or $i_1\ BEFORE\ j \wedge \neg(e[i_1] \leq t)$. According to the definition of adjusted release times, for any such segment i_1, $r'[i_1] \leq r'[j]$ must hold, which in turn implies that $r'[i_1] \leq t < e[i_1]$ must hold. From our assumption that all M processors are idle at t, this implies that $r'[i_1] \leq t < s[i_1]$ must hold, which is exactly the same condition that segment j satisfies. Then the same reasoning that we applied to segment j would also be applicable to segment i_1. If we continue on, there must exist an infinite set of segments $i_1, i_2, \ldots, i_k, \ldots$, such that for each segment i_k in that set: $r'[i_k] \leq t < s[i_k]$ must hold, which is clearly impossible, since there only exists a finite number of segments. This proves that there does not exist any time t between $r'[j]$ and $s[j]$ such that all M processors are idle.

For any pair of segments i and j that satisfies the conditions specified in the definition of $Z[l]$, i.e., $j \in Z[l] \wedge r'[j] < s[j]$ and i started before j started, i.e., $s[i] < s[j]$, and i completed its computation after j's release time, i.e., $r'[j] < e[i]$; i must belong to the set of segments that were executing at a time between $r'[j]$ and $s[j]$. Then, from what was proved earlier, there does not exist any time between $s[i]$ and $e[j]$ such that all M processors are idle. If we recursively apply such reasoning starting from $j = l$, it follows that for any segment $i \in Z[l]$, there does not exist any time between $s[i]$ and $e[l]$ such that all M processors are idle.

Property 2: If the schedule is not feasible, then a feasible schedule can be found only if it is possible to re-schedule some segment $i \in Z[l]$ such that some portion of a usable time period utp$(i, t_s, t_e) \in UTP[l]$ is occupied by i.

If the schedule is not feasible and either no usable time period exists for the latest segment l, or, no segment $i \in Z[l]$ can occupy any portion of any usable time period of l without causing itself or some other segment to be late, then no feasible schedule exists for the given set of segments.

Proof: If the schedule is not feasible, then in order to reduce the lateness of the schedule the latest segment l in the schedule must be scheduled earlier. But the only way to schedule l earlier, is to reschedule some segment i that was previously scheduled earlier than l, so that l can execute during a time when i was executing.

Rescheduling any segment i that is not in $Z[l]$ will not help to schedule l earlier. If $i \notin Z[l]$, then according to the definition of the set $Z[l]$ there are only two possible cases: either $s[i] \geq s[l]$, or $\nexists j \in Z[l] : r'[j] < s[j] \wedge r'[j] < e[i]$. In the first case, rescheduling i will not help, since executing l during the time that i was previously executing will only result

in l being scheduled to start at the same time or later. In the second case, for all $j \in Z[l]$: either $r'[j] = s[j]$ and j cannot be scheduled earlier, or $r'[j] \geq e[i]$ and j cannot be executed during any time that i was previously executing, because no segment can be put into execution before its release time.

Suppose we try to reschedule some segment i in $Z[l]$ that was previously scheduled earlier than l, so that l can execute during a time when i was executing. Let us assume that by doing so l is scheduled earlier by x time units. Then since those x processor execution time units were taken away from i, i would have to be given back the same number of processor execution time units. Because there does not exist any usable time period of l in the valid initial solution, that is, there does not exist any time period that lies between the adjusted release time $r'[i]$ and the adjusted deadline $d'[i]$ of some segment i in the set $Z[l]$ such that some processor m is not occupied by any segment in the set $Z[l]$, those x time units cannot be given back to i by letting either i or some other segment in $Z[l]$ execute at any time during which some processor was previously not occupied by any segment in $Z[l]$. So the only remaining possibility would be to let i execute in the last x time units of l's previous execution, which means that the completion time $e[i]$ of i will be equal to or greater than l's completion time $e[l]$. Because the original schedule was obtained by using an earliest-deadline-first strategy in the definition of a valid initial solution, and i was scheduled before l, this implies that either $d'[i] \leq d'[l]$ or $s[i] < r'[l]$ must be true in the original schedule. In the former case, rescheduling i to execute during the last x time units of l's previous execution, would result in i being at least as late as l was previously. In the latter case, scheduling i after l will result in an additional gap in the schedule with a length greater than or equal to $s[i] - r[l]$. But, since there did not exist any usable time period that lies between the adjusted release time $r'[i]$ and the deadline $d'[i]$ of any segment i in $Z[l]$ such that some processor was not occupied by any segment in $Z[l]$, the additional gap would push at least one segment in $Z[l]$ to complete its computation beyond its original completion time, causing it to miss its deadline.

Acknowledgment

The author is greatly indebted to D. L. Parnas for motivating this work and for providing much needed support, help, and encouragement during the course of this work. The author is also grateful to the editor, P. Ng, and to the anonymous referees for the many helpful suggestions and comments that have led to important improvements in the presentation of this paper.

References

[1] J. Blazewicz, "Deadline scheduling of tasks with ready times and resource constraints," *Inform. Proc. Lett.*, vol. 8, pp. 60–63, Feb. 1979.

[2] J. Blazewicz, J. K. Lenstra, and A. H. G. R. Kan, "Scheduling subject to resource constraints: Classification and complexity," *Discrete Appl. Math.*, vol. 5, pp. 11–24, 1983.

[3] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints," *Nav. Res. Log. Quart.*, vol. 18, pp. 511–519, Dec. 1971.

[4] E. G. Coffman, Jr., *Computer and jobshop scheduling theory.* New York: Wiley-Interscience, 1976.

[5] S. R. Faulk and D. L. Parnas, "On synchronization in hard-real-time systems," *Commun. ACM*, vol. 31, pp. 274–287, Mar. 1988.

[6] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP completeness.* San Francisco: Freeman, 1979.

[7] M. J. Gonzalez, Jr., "Deterministic processor scheduling," *Computing Surveys*, vol. 9, pp. 173–204, Sept. 1977.

[8] D. Gunsfield, "Bounds for naive multiple machine scheduling with release times and deadlines," *J. Algorithms*, vol. 5, pp. 1–6, 1984.

[9] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent developments in deterministic sequencing and scheduling: a survey," in *Proc. NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, Durham, England in *Deterministic and stochastic scheduling.* M. A. H. Dempster *et al.*, Eds., Dordrecht: D. Reidal Publ. Co., July 1981.

[10] D. W. Leinbaugh, "Guaranteed response time in a hard real-time environment," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 85–91, Jan. 1980.

[11] G. K. Manacher, "Production and stabilization of real-time task schedules," *J. ACM*, vol. 14, pp. 439–465, July 1967.

[12] C. Martel, "Preemptive scheduling with release times, deadlines, and due dates," *J. ACM*, vol. 29, pp. 812–829, July 1982.

[13] A. K. Mok and M. L. Detouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proc. 7th IEEE Texas Conf. on Computing Systems*, Nov. 1978.

[14] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1983.

[15] ——, "The design of real-time programming systems based on process models," in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1984.

[16] B. Simons, "Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines," *SIAM J. Comput.*, vol. 12, pp. 294–299, May 1983.

[17] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Eng.*, vol. 16, pp. 360–369, Mar. 1990.

[18] ——, "On satisfying timing constraints in hard-real-time systems," *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 70–84, Jan. 1993.

[19] ——, "Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections." In *Proc. 11th Annual IEEE Int. Phoenix Conf. on Computers and Communications (IPCCC-92)*, Scottsdale, AZ, April 1–3, 1992.

[20] W. Zhao, K. Ramamrithan, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 564–577, May 1987.

[21] W. Zhao, K. Ramamrithan, and J. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Comput.*, vol. C-36, pp. 949–960, Aug. 1987.

Jia Xu received the Docteur en Sciences Appliquées degree in computer science from the Université Catholique de Louvain, Belgium in 1984.

He is currently an Associate Professor of Computer Science at York University, North York, Ontario, Canada. From 1984 to 1985 he was a postdoctoral fellow at the University of Victoria, Victoria, B.C., Canada. From 1985 to 1986 he was a postdoctoral fellow at the University of Toronto, Toronto, Ontario, Canada. He is a Fellow of the Canadian Institute for Telecommunications Research, Ontario, Canada. His research interests include real-time systems, scheduling, software engineering, and database management systems.