Minimizing Total Completion Time on Uniform Machines with Deadline Constraints

TEOFILO F. GONZALEZ

University of California, Santa Barbara, Santa Barbara, CA

JOSEPH Y.-T. LEUNG

New Jersey Institute of Technology, Newark, NJ

AND

MICHAEL PINEDO

Stern School of Business, New York University, New York, NY

Abstract. Consider n independent jobs and m uniform machines in parallel. Each job has a processing requirement and a deadline. All jobs are available for processing at time $t = 0$. Job j must complete its processing before or at its deadline and preemptions are allowed. A set of jobs is said to be *feasible* if there exists a schedule that meets all the deadlines. We present a polynomial-time algorithm that given a feasible set of jobs, constructs a schedule that minimizes the total completion time $\sum C_j$. In the classical $\alpha \mid \beta \mid \gamma$ scheduling notation, this problem is referred to as $Qm \mid prmt, \bar{d}_j \mid \sum C_j$. It is well known that a generalization of this problem with regard to its machine environment results in an NP-hard problem.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Mean flow time, uniform machines, deadline constraints, polynomial-time algorithms

---

The work of J. Y.-T. Leung and M. Pinedo was supported in part by the National Science Foundation (NSF) Grants DMI-0300156 and DMI-0245603, respectively.

Authors' addresses: T. F. Gonzalez, Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106, e-mail: teo@cs.ucsb.edu; J. Y.-T. Leung, Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, e-mail: leung@oak.njit.edu; M. Pinedo, Stern School of Business, New York University, New York, NY 10012, e-mail: mpinedo@stern.nyu.edu.

1. *Introduction*

Consider m uniform machines in parallel and n jobs. Machine i has speed v_i, and $v_1 \leq v_2 \leq \cdots \leq v_m$. Job j has a processing requirement p_j and deadline $\bar{d}_j$. Preemptions are allowed, i.e., the processing of any job may be interrupted at any time and resumed immediately on another machine or at a later time on the same machine or on another machine. Jobs may be preempted any number of times. However, a job cannot be processed simultaneously on two or more machines. If job j is processed only on machine i, then the time it spends on machine i is p_j/v_i. All jobs are available for processing at time $t = 0$ and job j must complete its processing before or at its deadline $\bar{d}_j$. A set of jobs is said to be *feasible* if there exists a schedule that meets all its deadlines; such a schedule is called a feasible schedule. Given a feasible set of jobs, our objective is to find a schedule that minimizes the total completion time $\sum C_j$. In the 3-field notation $\alpha \mid \beta \mid \gamma$ introduced by Graham et al. [1979], this problem is referred to as $Qm \mid prmt, \bar{d}_j \mid \sum C_j$.

The special case with m machines and n jobs without deadlines, that is, $\bar{d}_j = \infty$ for all j, can be solved via the preemptive rule which at any point in time assigns the job with the smallest remaining processing requirement to the fastest machine. This rule, which in the literature has been referred to as the SRPT-FM rule, will always generate a schedule with minimum $\sum C_j$ (see Pinedo [2002]).

Leung and Pinedo [2003] developed for the special case with m identical machines in parallel, that is, $Pm \mid prmt, \bar{d}_j \mid \sum C_j$, a polynomial-time algorithm that works in $O(mn^3 \log(mn))$ time. Gonzalez [1978] and McCormick and Pinedo [1995] developed a polynomial-time algorithm for the special case with machines that have different speeds but where all jobs have a common deadline, that is, $Qm \mid prmt, \bar{d}_j = \bar{d} \mid \sum C_j$. As we will see later, the scheduling problem becomes more complex when the machines have different speeds and the jobs have different deadlines. Sitters [2001] showed that the more general problem with unrelated machines, that is, $Rm \mid prmt, \bar{d}_j \mid \sum C_j$ is strongly NP-hard, even when the jobs have no deadline; that is, $\bar{d}_j = \infty$ for all j. One problem still remains open, namely $Qm \mid prmt, \bar{d}_j \mid \sum C_j$. Cho and Sahni [1980] developed an efficient feasibility procedure for this problem that can be used to discard infeasible problem instances. Therefore, one may assume without loss of generality that the input to any procedure is a problem instance with at least one feasible schedule.

In this article, we provide a polynomial-time algorithm for $Qm \mid prmt, \bar{d}_j \mid \sum C_j$. The algorithmic framework we present is somewhat similar to the framework adopted by Leung and Pinedo [2003] for the case of identical machines. Their framework is based on two procedures, namely the *Scheduling Procedure (SP)* and the *Job-Ordering Procedure (JOP)*. The framework in this article with uniform machines also consists of two procedures. However, the SP procedure in this article is significantly more complex than the corresponding procedure for identical machines. The main reason is that in the case of identical machines scheduling decisions can be made by considering only a subset of the jobs, whereas in the case of uniform machines one needs to consider all jobs. The proof of correctness for the identical machine problem involves a triple interchange argument, whereas the one in this article is more complex and requires what we in what follows refer to as a *massive interchange* argument. Our job-ordering procedure for uniform machines is similar to the one developed by Leung and Pinedo for identical machines.

This article is organized as follows: In the next section, we outline the framework for the polynomial-time algorithm for $Qm \mid prmt, \bar{d}_j \mid \sum C_j$. Then, we explain its main procedures, namely the *Scheduling Procedure (SP)* and the *Job-Ordering Procedure (JOP)*. In the subsequent two sections, we prove the various properties of these two procedures. In the fifth section, we discuss the time complexity of our algorithm and in the last section we present extensions and conclusions.

Throughout this article, we use the following notation and terminology. We consider several different types of schedules, namely, complete schedules, partial schedules, and temporary schedules. A complete schedule specifies the processing times of all n jobs; such a schedule is denoted by a σ with an appropriate subscript or superscript. A partial schedule specifies for each one of the n jobs some part of its processing time, which may be anywhere from zero to its total processing time; a partial schedule is typically denoted by S with an appropriate subscript or superscript. Note that not every partial schedule is a complete one, but complete schedules may also be referred as partial schedules. A temporary schedule is a partial schedule that is generated in a step of a procedure in order to verify a feasibility condition and obtain some information; after the verification has been completed and the necessary information is obtained, the temporary schedule is discarded. A temporary schedule usually carries a subscript T.

2. *The Algorithmic Framework*

We assume that we are given a feasible set of jobs. All schedules are assumed to be feasible unless stated otherwise. Let

$$D_1 < D_2 < \cdots < D_z$$

denote the distinct deadlines of the n jobs and let $D_0 = 0$. Every schedule σ induces a deadline $d_j(\sigma)$ for each job j, where $d_j(\sigma)$ is defined as the smallest D_k with $C_j \leq D_k$ in σ. If the context is clear, we drop the σ and simply denote the induced deadline by d_j. Note that the induced deadline d_j of job j may be smaller than its original deadline $\bar{d}_j$. However, every feasible schedule has the property that $d_j \leq \bar{d}_j$ for each job j.

We motivate our algorithm by asking two key questions. Suppose a "birdie" were to tell us the induced deadline of each job in an optimal schedule. Having only this information, can we construct an optimal schedule? Second, how do we get the information that the "birdie" has? It is clear that the answers to these two questions immediately provide an algorithm that yields an optimal schedule for our problem.

We proceed with answering the first question. The next lemma shows that if a short job has an induced deadline that is no later than that of a long job, then the short job is completed no later than the long job (and possibly earlier). Lemma 1 can be proved via a standard interchange argument.

LEMMA 1. *Suppose we have two jobs j and k such that $p_j \leq p_k$. If there is a schedule σ such that $d_j(\sigma) \leq d_k(\sigma)$ and $C_j > C_k$, then there exists another schedule σ' with total completion time $\sum C_j$ not larger than that of σ and with job j completing no later than job k (and possibly earlier).*

PROOF. We omit the proof since it is based on a simple interchange argument. □

Lemma 1 yields a completion sequence in an optimal schedule. That is, jobs with induced deadlines equal to D_1 finish first, in ascending order of their processing requirements, followed by jobs with induced deadlines equal to D_2, in ascending order of their processing requirements, and so on. Since the jobs finish their processing in this order, they should also be scheduled in the same order. Thus, we consider a procedure to schedule the jobs using the list L where the jobs are ordered according to the given completion sequence. However, there are a huge number of different schedules that could be generated by such procedure. Our procedure will schedule the next job in L in such a way that it is completed as early as possible, with the provision that the remaining jobs in L can meet their induced deadlines. In Theorem 1, we show that this strategy generates an optimal schedule. We refer to this procedure as the Scheduling Procedure *SP*. The input to *SP* is the ordering L in which the procedure will try to complete the n jobs while adhering to the current induced deadlines. The output of *SP* is a complete schedule (if one exists) with starting times, preemptions and completion times of all n jobs. However, as we shall see later on, when L is an optimal ordering procedure *SP* generates an optimal schedule in which the job completion sequence will be identical to L.

In this article, we use an interchange argument which we refer to as the *massive interchange argument*. Let us give an example on how this argument works by applying it to show Lemma 1 in another way. First, delete jobs j and k from the schedule. Wherever those jobs were scheduled, there is idle time now. The idle time previously assigned to job j from time C_k to time C_j is assigned to job k and the idle time prior to time C_k is partitioned into two disjoint sets of idle times that are called *general virtual machines*. As we shall establish later on, one can show that the two jobs can be scheduled on these virtual machines if their remaining processing requirements are such that they each fit on the faster one of the two virtual machines. The beauty of this argument is that we actually do not need to construct a schedule, we just have to show that one such schedule exists.

Before describing the scheduling procedure, we have to introduce additional notation. The input of the *SP* procedure is the list L as well as the current induced deadlines. Assume that jobs $1, \ldots, j-1$ have already been scheduled and jobs $j, j+1, \ldots, n$ need to be scheduled. Let S denote the partial schedule already in place (see crossed lines in Figure 1) during the time interval $[0, r]$, where r is the smallest induced deadline among the jobs $j, j+1, \ldots, n$. In this partial schedule, we partition all the idle time during the time interval $[0, r]$ into m groups and refer to them as *virtual machines* $\mathcal{VM}$. Virtual machine μ_l is defined as a series of consecutive idle time intervals on machines $m, m-1, \ldots, 1$ (an idle time interval may have length zero). Virtual machine μ_l comprises the idle time interval $[b_{i,l}, e_{i,l}]$ on machine i. The various time segments belonging to μ_l satisfy the property $e_{i,l} = b_{i-1,l}$, $e_{1,l} = r$, and if $e_{i,l} > b_{i,l}$, then for every $x \in [b_{i,l}, e_{i,l}]$ virtual machine μ_k for $k < l$ must include time x on machine k' for some $k' < i$. The processing power of μ_l is defined as

$$W(\mu_l) = \sum_{i=1}^{m} v_i(e_{i,l} - b_{i,l}).$$

From the above definitions, it follows that for all $l < k$ virtual machine μ_l has at least as much processing power as virtual machine k and for every machine i,

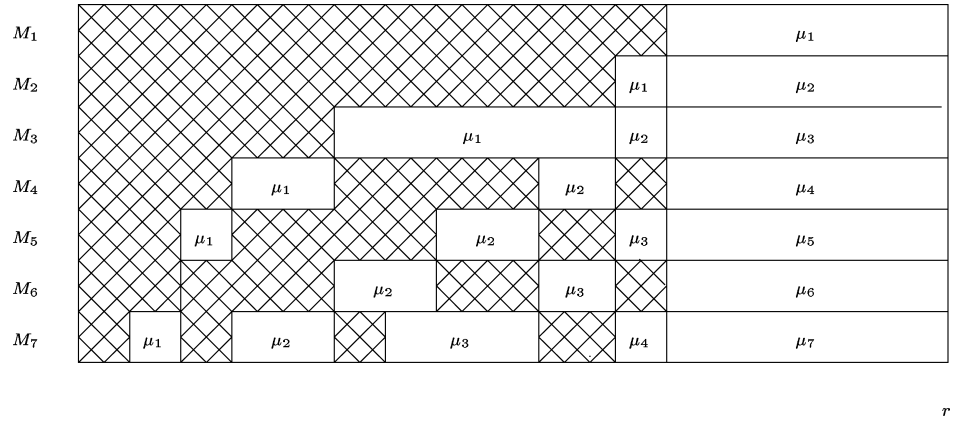

FIG. 1. Virtual machines from time 0 to time r.

$e_{i,l} \leq b_{i,k}$ for all $l < k$. Our virtual machines are similar to the disjoint processors introduced by Gonzalez and Sahni [1978] and Gonzalez [1978] for uniform machines.

Consider a preemptive scheduling problem with m virtual machines; virtual machine μ_l has processing power $W(\mu_l)$. Jobs $j, j+1, \ldots, n$ have processing requirements $p'_j, p'_{j+1}, \ldots, p'_n$, and all jobs have an induced deadline r. By using the Longest Remaining Processing Time on the Fastest Machine (LRPT-FM) rule, or equivalently, by using the preemptive scheduling rule similar to the level algorithm developed by Horvath et al. [1976], which we will refer to in what follows as the *Highest Level algorithm*, one can establish the following lemma. The inequalities in the necessary and sufficient conditions are similar to those established by Liu and Yang [1974] for feasible schedules on uniform machines.

LEMMA 2. *Consider the preemptive scheduling problem with virtual machine μ_l having processing power $W(\mu_l)$, jobs $j, j+1, \ldots, n$ having processing requirements $P'_j = (p'_j, p'_{j+1}, \ldots, p'_n)$, and all jobs having an induced deadline r. This partial scheduling problem has a feasible schedule if and only if*

$$\sum_{i=1}^{l} x_i(P'_j) \leq \sum_{i=1}^{l} W(\mu_i)$$

for $1 \leq l \leq m-1$ *and*

$$\sum_{i=j}^{n} p'_i \leq \sum_{i=1}^{m} W(\mu_i),$$

where $x_i(P'_j)$ is the ith largest value of $p'_j, p'_{j+1}, \ldots, p'_n$.

PROOF. The proof is omitted since it is a simple application of the Highest Level algorithm. □

We now define some important terms and establish a property that will be used in subsequent proofs. For a problem instance that satisfies the conditions of Lemma 2, a virtual machine μ_q is said to be *tight* when $\sum_{i=1}^{q} x_i(P'_j) = \sum_{i=1}^{q} W(\mu_i)$ for

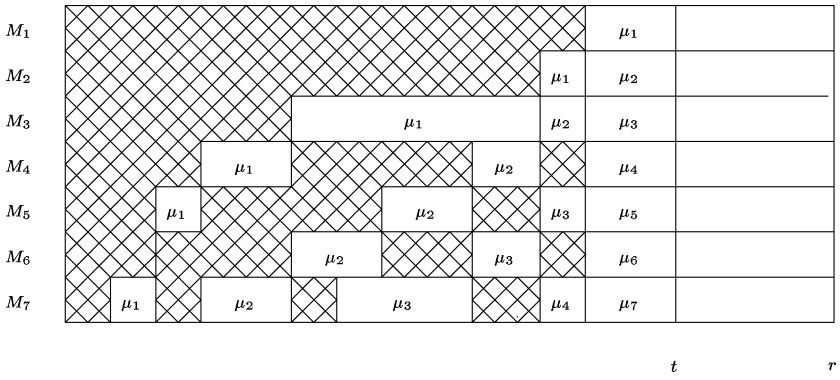

FIG. 2. Virtual machines from time 0 to time t.

some $q < m$, or $\sum_{i=j}^{n} p'_i = \sum_{i=1}^{m} W(\mu_i)$. All the jobs involved in a summation where equality holds are said to be *critical*. Let virtual machine μ_q be the smallest indexed virtual machine that is tight. We claim that either $x_q(P'_j) > x_{q+1}(P'_j)$, or $x_q(P'_j) = x_{q+1}(P'_j)$ and $q = 1$. Note that $q > 1$ and $x_q(P'_j) = x_{q+1}(P'_j)$ can be used to contradict the assumption that

$$\sum_{i=1}^{q+1} x_i(P'_j) \leq \sum_{i=1}^{q+1} W(\mu_i).$$

Consider again the original problem with the virtual machines and take now into account that the jobs that still need processing may be subject to different deadlines. As we shall prove later on, an optimal schedule for our original problem is a feasible schedule in which job j is completed as early as possible, with the provision that all jobs are completed before or at their deadlines. Our Scheduling Procedure (*SP*) schedules job j so that it finishes as early as possible, provided that all remaining jobs are completed before or at their deadlines. Let the induced deadline of job j be D_q. We note that the induced deadline of job j is the smallest among the jobs $j, j+1, \ldots, n$. Let t be the earliest completion time of job j. Since job j has induced deadline D_q, it must be that $t \leq D_q$ or there is no feasible schedule. The value of t is determined by the procedure $MC_j(S)$ which finds the earliest possible completion time for job j given the partial schedule S and the induced deadlines for jobs $j+1, \ldots, n$. Procedure $MC_j(S)$ finds this minimum value of t by considering a scheduling problem that is reversed in time: the deadlines act as release dates and the jobs are scheduled backwards in time. The preemptive scheduling rule that is used for scheduling the jobs backwards is the Highest Level algorithm discussed above. If there is a feasible schedule, then procedure $MC_j(S)$ returns the value of t; otherwise, it returns ∞. When t is determined to be smaller than or equal to D_q, Procedure *SP* defines the virtual machines from time 0 to t (see Figure 2). It then schedules job j either on virtual machine μ_m if $p_j = W(\mu_m)$, or by using two adjacent virtual machines such that job j cannot be processed only by the slower one of these two machines, but it can be processed by the faster one (see Figure 3 where we use μ_2 and μ_3). At the next iteration, the virtual machines from time 0 to r will be defined as in Figure 4.

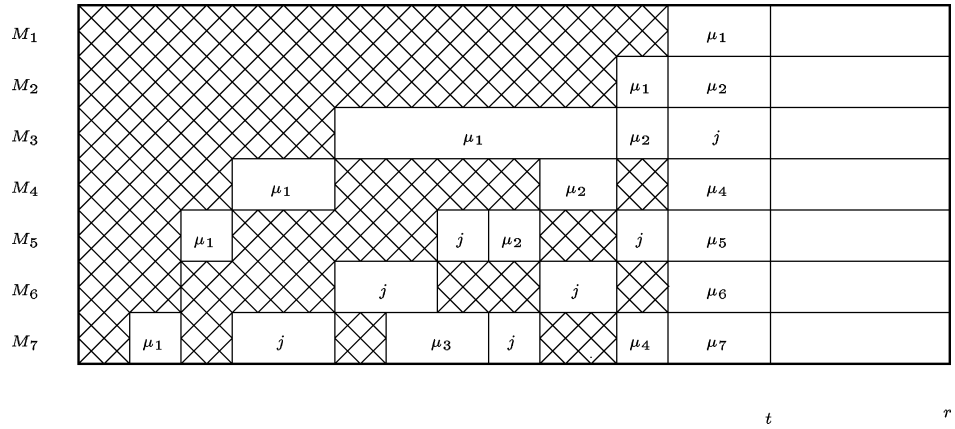

FIG. 3. Assignment of job j to μ_2 and μ_3.

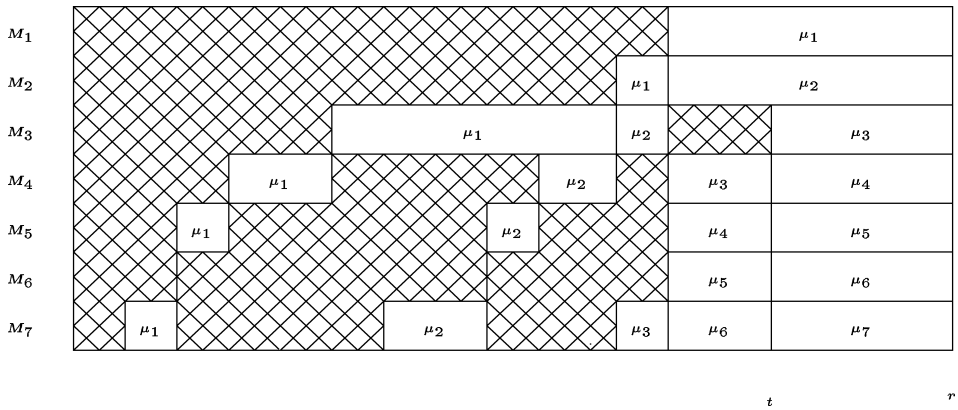

FIG. 4. Resulting virtual machines.

The *SP* procedure consists of seven steps.

Scheduling Procedure *(SP)*

Step 1 (*Initialization*). Reindex the jobs so that job 1 is supposed to finish first, job 2 second, and so on. So $L = (1, 2, \ldots, n)$. Let $j = 1$ and $S = \emptyset$.

Step 2 (*Find least completion time for job* j). Let the induced deadline of job j be D_q. Procedure $MC_j(S)$ finds and returns the earliest possible completion time $t \leq D_q$ for job j in schedule S, if it exists, such that it is possible to schedule the remaining jobs to finish by their induced deadlines. Otherwise, the procedure returns ∞.

Step 3 (*Feasibility test*). If $t = \infty$, then exit;

Step 4 (*Define virtual machines*). Define the virtual machines for schedule S from time 0 to time t.

Step 5 (*Determine the virtual machines where job* j *is to be scheduled*). Determine the value of i such that $i = m$ and $p_j = W(\mu_m)$, or if job j would be processed only on μ_i, it will be completed by time t, but job j cannot be processed only on μ_{i+1} to complete by time t.

Step 6 (*Schedule job* j *to finish by time* t). If $i = m$, then job j is scheduled only on μ_m where it fits exactly. Otherwise, determine the value x such that job j fits exactly on virtual machine μ_i from time 0 to time x and on virtual machine μ_{i+1} from time x to time t. Schedule job j accordingly (on μ_i and μ_{i+1}).

Step 7 (*Stopping Criterion*). If $j < n$, increase j by 1 and go back to Step 2.

End Scheduling Procedure *(SP)*

This completes the description of the Scheduling Procedure. In the third section, we show that, given the completion sequence of an optimal schedule, *SP* always generates an optimal schedule.

The $MC_j(S)$ procedure consists of eleven steps. Our approach to find the earliest completion time for job j is to consider a scheduling problem that is reversed in time: the deadlines act as release dates and the jobs that have not been scheduled yet are scheduled backwards in time. The preemptive scheduling rule that is used for scheduling the jobs backwards is the Highest Level algorithm mentioned above. After finding the value for t, the temporary schedule that had been generated, which we call S_T, is discarded. If there is a feasible schedule, then procedure $MC_j(S)$ returns the value of t; otherwise, it returns ∞. Note that, for the purpose of procedure *SP* the temporary schedule S_T does not have to be specified (it suffices to verify that such a schedule exists). However, in some of our proofs, we refer to the schedule generated by procedure $MC_j(S)$ which is simply schedule S_T.

Procedure ($MC_j(S)$)

Step 1 (*Initialization*). Initialize schedule S_T to be identical to partial schedule S. Let D_q be the induced deadline of job j, and let $D_q < D_{q+1} < \cdots < D_z$ be the induced deadlines of the jobs $j, j+1, \ldots, n$. Let $i = z$. Let $p'_j, p'_{j+1}, \ldots, p'_n$ be the processing requirements of jobs $j, j+1, \ldots, n$.

Step 2 (*Construct the schedule for the time interval* $[D_{i-1}, D_i]$). Repeat this step while $i > q$. Generate schedule S_T (in reverse time) by using the Highest Level algorithm over the time interval $[D_{i-1}, D_i]$ for the jobs with induced deadline greater than or equal to D_i. Let $p'_j, p'_{j+1}, \ldots, p'_n$ be the remaining processing requirements of jobs $j, j+1, \ldots, n$. Decrease i by one.

Step 3 (*Define virtual machines*). Define the virtual machines for schedule S_T from time 0 to time D_q.

Step 4 (*Feasibility test*). If it is not feasible to schedule all the remaining jobs on the virtual machines in schedule S_T, then set t equal to ∞ and return.

Step 5 (*Define virtual machine break points*). Let $B_1 < B_2 < \cdots < B_y$ be the minimum set of distinct points in time such that the same (nonempty) set of virtual machines are defined over the same set of (real) machines in between every pair of break points. Let $i = y$.

Step 6 (*Construct the schedule for the time interval* $[B_{i-1}, B_i]$). If we were to schedule jobs $j+1, j+2, \ldots, n$ in the time interval $[B_{i-1}, B_i]$ using the Highest Level procedure and the resulting jobs plus job j can be scheduled in the resulting virtual machines from time $t = 0$ to time $t = B_{i-1}$, then schedule jobs $j+1, j+2, \ldots, n$ in $[B_{i-1}, B_i]$ as indicated (in reverse time), decrease i by 1 and repeat this step.

Step 7 (*Initialize s*). Let $s = B_i$.

Step 8 (*Find points where the remaining execution requirements of jobs becomes identical*). Let $s' \geq B_{i-1}$ be the largest value such that if we were to schedule jobs $j+1, j+2, \ldots, n$ in the time interval $[s', s]$ using the Highest Level procedure, then two jobs (among jobs $j, j+1, \ldots, n$) that had different remaining processing times before will now have identical remaining processing times. If this condition is not met for any $B_{i-1} \leq s' < s$, then let $s' = B_{i-1}$. If it is possible to schedule jobs $j, j+1, \ldots, n$ on the virtual machines from time 0 to time s', then construct the schedule (for jobs $j+1, j+2, \ldots, n$) from time s' to s as indicated above (but in reverse time), let $s = s'$ and repeat this step.

Step 9 (*Find the value of t*). Find the least value of t in the interval $[s', s]$ such that if we were to schedule jobs $j+1, j+2, \ldots, n$ using the Highest Level algorithm in the interval $[t, s]$, it is possible to schedule jobs $j, j+1, \ldots, n$ on the virtual machines from time 0 to time t. Construct the schedule from time t to time s (in reverse time) for jobs $j+1, j+2, \ldots, n$ using the Highest Level algorithm.

Step 10 (*Construct the schedule from time* 0 *to time t*). The schedule for jobs $j, j+1, \ldots, n$ from time 0 to time t is generated by the Highest Level algorithm. Note that this operation is not actually needed by procedure *SP*, however it will be useful in our correctness proofs.

Step 11 (*Return t*). Return the value of t.

End Procedure ($MC_j(S)$)

This completes the description of the Procedure $MC_j(S)$. In Section 3, we show that this procedure finds the correct value for t.

We now focus on the second question—How do we get the information the "birdie" has? Our approach is to begin by using the SRPT-FM rule, ignoring the deadline constraints. After all, if the SRPT-FM schedule does not have any deadline violations, then the schedule is already optimal. In general, however, the SRPT-FM schedule may have some deadline violations. We need a mechanism to avoid deadline violations while maintaining as much of an SPT-type structure as possible. In what follows we describe the procedure that generates the optimal ordering and we refer to this procedure as the *Job-Ordering Procedure* (*JOP*) (this ordering procedure invokes the *SP* procedure as a subprocedure). *JOP* consists of an initialization step and a main step. The input of this procedure is the list of jobs L which orders the jobs in increasing order of their processing requirements and, when there are ties, in increasing order of their original deadlines. The output of the procedure is a list $\bar{L}$ that specifies the order in which the jobs are completed in an optimal schedule.

Job-Ordering Procedure (*JOP*)

Step 1 (*Initialization*). Reindex the jobs in ascending order of their processing requirements and in ascending order of their original deadlines for identical processing requirements. Let $L = (1, 2, \ldots, n)$ be the list of jobs in ascending order of their indexes. For each job j, initialize its induced deadline to be its original deadline $\bar{d}_j$. (This is a slight abuse of notation since induced deadlines are defined with respect to a schedule. As we shall see later, the induced deadlines will be updated to correspond to the induced deadlines in an optimal schedule.) Set $k = 1$ and $\bar{L} = L$.

Step 2 (*Main*). In what follows, we reorder the jobs in $\bar{L}$ to form an ordering of the completion sequence in an optimal schedule. We consider each job in turn, starting with the first job in $\bar{L}$. Suppose we have fixed the position of the first $k-1$ jobs and we are considering the kth job. Let $\bar{L} = (i_1, i_2, \ldots, i_n)$ and let $R(i_{k+1}, i_{k+2}, \ldots, i_n)$ denote the list obtained by reordering the last $n-k$ jobs in $\bar{L}$ in ascending order of their induced deadlines and in ascending order of their processing requirements for identical induced deadlines. Set the induced deadline of job i_k to be the smallest induced deadline among the jobs $i_{k+1}, i_{k+2}, \ldots, i_n$. We now construct a complete schedule σ by applying *SP* to the list

$$\hat{L} = (i_1, i_2, \ldots, i_k) \,||\, R(i_{k+1}, i_{k+2}, \ldots, i_n).$$

The outcome of the application of *SP* may fall into one of the following two cases:

Case (i). σ is a feasible schedule and t is finite. This means that job i_k is completed no later than any one of the jobs in $(i_{k+1}, i_{k+2}, \ldots, i_n)$. In this case, job i_k is fixed in position k. $\bar{L}$ will be the same as before (i.e., $\bar{L} = (i_1, i_2, \ldots, i_n)$) and the above process will be repeated with k increased by 1.

Case (ii). $t = \infty$ and σ is infeasible. In this case, we move all the jobs in $(i_{k+1}, i_{k+2}, \ldots, i_n)$ with the smallest induced deadline (among all the jobs in $(i_{k+1}, i_{k+2}, \ldots, i_n)$) ahead of job i_k (but behind job i_{k-1}). Let $i_{j_1}, i_{j_2}, \ldots, i_{j_l}$ be those jobs in ascending order of their processing requirements. Fix job i_{j_1} in the kth position, job i_{j_2} in the $(k+1)$st position, $\ldots$, and job i_{j_l} in the $(k+l-1)$st position. Reset the induced deadline of job i_k to be its original deadline $\bar{d}_{i_k}$. Set k to be $k+l$. Let

$$L' = (i_{k+1}, i_{k+2}, \ldots, i_n) - (i_{j_1}, i_{j_2}, \ldots, i_{j_l}).$$

We set $\bar{L}$ to be

$$\bar{L} = (i_1, i_2, \ldots, i_{k-1}) \,||\, (i_{j_1}, i_{j_2}, \ldots, i_{j_l}) \,||\, (i_k) \,||\, L'.$$

The above process will be repeated with the new $\bar{L}$ and the new k.

Step 3 (*Stopping Criterion*). If $k = n$, then STOP; otherwise, go back to Step 2.

End Job-Ordering Procedure (*JOP*)

When *JOP* stops, $\bar{L}$ specifies the completion sequence of an optimal schedule. In Section 4, we show that *JOP* generates an optimal completion ordering. The results in Sections 3 and 4 yield our main result that $Qm \mid prmt, \bar{d}_j \mid \sum C_j$ can be solved in polynomial time for each $m \geq 2$.

3. *The Scheduling Procedure*

Before we can establish that procedure *SP* generates an optimal schedule, we need to show that procedure $MC_j(S)$ finds the earliest possible completion time t for job j so that the remaining jobs complete their processing by their induced deadlines. To prove this result, we need Lemma 3, where we establish that if it is possible to schedule job j in the partial schedule S to complete by time t, then one can schedule job j to complete at the same time in schedule S_T. Schedule S_T is just schedule S plus a temporary schedule from time t till the latest deadline D_z generated by procedure $MC_j(S)$ using the Highest Level scheduling rule in reverse.

Suppose that Procedure *SP* is given the list L as well as the current induced deadlines. Without loss of generality, we may assume that jobs are ordered in increasing order of their induced deadlines and jobs with common deadlines are ordered in increasing order of their processing requirements. Assume that jobs $1, \ldots, j-1$ have already been scheduled by procedure *SP* and the schedule is given in S. For $j \leq i \leq n$, let p_i be the processing requirement of job i.

In the following lemma we assume that a partial schedule S for jobs $1, 2, \ldots, j-1$ is already in place (generated by procedure *SP*) and that job j can be completed by time t while allowing all remaining jobs to be finished by their induced deadlines.

LEMMA 3. *Let S be the schedule for jobs $1, \ldots, j-1$ generated by procedure SP. Suppose there exists a complete schedule σ_Y that includes S and has job j finishing at time t. Then, there exists a schedule with job j finishing at time t that includes the temporary schedule S_T, where S_T includes schedule S plus the schedule from time t till D_z generated by procedure $MC_j(S)$ using the Highest Level scheduling procedure in reverse.*

PROOF. To prove the lemma, we use the massive interchange argument to show the existence of a feasible schedule S_F that includes S_T and in which job j finishes by time t. Initially, let S_F be schedule S_T. Let l be such that jobs $l+1, l+2, \ldots, n$ are all the jobs that have completion times greater than t in σ_Y. Since jobs $j, j+1, \ldots, l$ complete by time t in σ_Y and schedule S_T (and also S_F) includes S, it follows that one can schedule in S_F exactly as in σ_Y all the jobs $j, j+1, \ldots, l$ without introducing any conflicts.

Let

$$P''_{l+1} = (p''_{l+1}, p''_{l+2}, \ldots, p''_n)$$

denote the processing time for jobs $l+1, l+2, \ldots, n$ in schedule σ_Y from time 0 to time t. Let

$$P'_{l+1} = (p'_{l+1}, p'_{l+2}, \ldots, p'_n)$$

denote the remaining processing time for jobs $l+1, l+2, \ldots, n$ in schedule S_F.

Since schedule S_F includes schedule S_T, which uses the Highest Level preemptive scheduling procedure in reverse, it follows that for each $1 \leq k \leq n - l$

$$\sum_{i=1}^{k} x_i(P'_{l+1}) \leq \sum_{i=1}^{k} x_i(P''_{l+1}),$$

where $x_i(P'_{l+1})$ is the ith largest processing time of P'_{l+1}. The meaning of $x_i(P''_{l+1})$ is the same as $x_i(P'_{l+1})$, but using the processing times given by P''_{l+1}.

Let μ_i, $1 \leq i \leq m$, be the virtual machines for all the idle time in S_F[1] from time 0 to time t. Let $W(\mu_i)$ be the processing power of virtual machine μ_i. Since schedule σ_Y exists and the virtual machines in S_F correspond to the times when jobs $l + 1, l + 2, \ldots, n$ are scheduled in σ_Y from time 0 to time t or idle time in schedule σ_Y, we know that

$$\sum_{i=1}^{k} x_i(P''_{l+1}) \leq \sum_{i=1}^{k} W(\mu_i) \qquad 1 \leq k \leq \min\{n - l, m - 1\}$$

and

$$\sum_{i=1}^{n-l} x_i(P''_{l+1}) \leq \sum_{i=1}^{m} W(\mu_i).$$

Since for each $1 \leq k \leq n - l$

$$\sum_{i=1}^{k} x_i(P'_{l+1}) \leq \sum_{i=1}^{k} x_i(P''_{l+1}),$$

it follows that

$$\sum_{i=1}^{k} x_i(P'_{l+1}) \leq \sum_{i=1}^{k} W(\mu_i) \qquad 1 \leq k \leq \min\{n - l, m - 1\}$$

and

$$\sum_{i=1}^{n-l} x_i(P'_{l+1}) \leq \sum_{i=1}^{m} W(\mu_i).$$

Using a lemma similar to Lemma 2 we can show that it is possible to schedule the remaining processing times for all the jobs $l + 1, l + 2, \ldots, n$ in the virtual machines defined from S_F to generate a complete schedule that includes S_T and in which job j completes by time t. Therefore, a schedule with the required properties exists. This completes the proof of the lemma. □

We are now ready to establish in Lemma 4 the correctness of procedure $MC_j(S)$.

[1] Strictly speaking, the idle time in schedule S_F is not necessarily a set of virtual machines. However, one can easily modify the definition of virtual machines to cover this more general case. The more general definition assigns to virtual machine μ_1 at each time x the fastest machine that is unused, to virtual machine μ_2 the second fastest machine, and so on. These more general virtual machines have the same properties as the virtual machines (the properties discussed around Lemma 2 and also Lemma 2).

LEMMA 4. *Given the partial schedule S for jobs* $1, 2, \ldots, j-1$ *generated by procedure SP, procedure* $MC_j(S)$ *determines whether or not there is a feasible schedule for* $j, j+1, \ldots, n$ *and if so, it returns the earliest possible completion time t for job j such that the remaining jobs still can be scheduled by their induced deadlines.*

PROOF. Lemma 3 shows that the scheduling strategy used in Steps 2, 6 and 8 does not increase earliest possible completion time for job j. Therefore, it is simple to verify that procedure $MC_j(S)$ determines the earliest possible time job j can be scheduled to complete provided that jobs $1, 2, \ldots, j-1$ are scheduled as in S and the remaining jobs can be scheduled by their induced deadlines. This completes the proof of the lemma. □

The following theorem is instrumental in proving that *SP* yields an optimal schedule.

THEOREM 1. *Given the order of completions in an optimal schedule, there exists an optimal schedule* (*i.e., a schedule with minimum* $\sum C_j$) *in which each job is finished as early as possible, provided that all the jobs that follow can meet their induced deadlines.*

PROOF. Let $L = (1, 2, \ldots, n)$ denote the completion sequence in an optimal schedule. Let σ denote a schedule in which each job is completed as early as possible (provided that all the jobs following it can meet their induced deadlines) and the jobs complete in the order given by L. We prove by contradiction that the schedule σ is optimal with respect to $\sum C_j$. Suppose σ is not optimal. Let σ^* be a schedule with minimum $\sum C_j$ in which the jobs finish in the order L and which has the largest value of j such that the first $j-1$ jobs finish exactly at the same time as in σ. If there are several schedules with the above property, select one in which job j finishes at the earliest possible time, which we refer to as t. By Lemma 4, we know that the completion time of job j in σ^* is greater than its completion time in σ.

We now use the massive interchange arguments to show that either σ^* is not a schedule with minimum $\sum C_j$, or that there is a schedule with the same value for $\sum C_j$ in which the first $j-1$ jobs are finishing at the same time as in schedule σ, but job j finishes before time t. In both cases, we contradict how σ^* was selected.

Now apply the $MC_j(S_X)$ procedure to schedule S_X which consists of the first $j-1$ jobs scheduled as in σ^* and the induced deadlines for jobs $j, j+1, \ldots, n$ being their completion times in schedule σ^*. Let schedule σ_T denote the schedule generated by $MC_j(S_X)$. (Note that this definition of induced deadline is different from the one defined in Section 2. We use the same term here because we do not want to introduce additional terminology. This new definition of induced deadline applies only to the proof of this theorem.) Clearly, schedule σ_T can be constructed by procedure $MC_j(S_X)$ since σ^* is a feasible schedule, and it includes schedule S_X. If the $\sum C_j$ for schedule σ_T is less than that for σ^*, then clearly σ^* is not optimal, which contradicts our assumption. So it must be that the $\sum C_j$ for schedule σ_T is equal to the one for σ^* and job j finishes at time t. If for some integer $i > 1$, there is at least one job with an induced deadline equal to D_i and this job is not being processed continuously in σ_T during the interval D_{i-1} and D_i, then we can swap in the schedule some δ units within that interval with the last δ units just before

D_i. Clearly, at least one job will have its completion time reduced by δ units in σ_T without increasing the completion time of the other jobs. This contradicts the fact that σ^* is an optimal schedule.

Our approach to find a contradiction begins by showing the existence of a job i_d, which is scheduled just before time t in σ_T and whose induced deadline is less than its original deadline. Let D_α be the induced deadline of job i_d. Then, we define a value of δ that is small enough to satisfy several important properties. We define the partial schedule T^i as schedule σ_T after deleting all the assignments from time 0 to time $t - \delta$ for jobs $j, j+1, \ldots, n$, that is, these assignments in σ_T are idle time in T^i. We then interchange job i_d and some jobs scheduled in the time interval D_α and $D_\alpha + \delta$ and the time interval $t - \delta$ and t in T^i. As a result, the completion time of job i_d increases by δ. But instead of scheduling job j in the time interval from $t - \delta$ to t, we schedule job i_d. This decreases the completion time for job j by δ units since we can show that it is possible to schedule the remaining processing times for the jobs $j, j+1, \ldots, n$ to finish by time $t - \delta$ in the virtual machines defined from T^i. Therefore, we have constructed another optimal schedule in which jobs $1, 2, \ldots, j-1$ are scheduled as in σ but job j completes by time $t - \delta$, which contradicts the way we selected schedule σ^*. Before we establish this result it is convenient to define other partial schedules.

Let T_t be schedule σ_T after deleting the assignments of jobs $j, j+1, \ldots, n$ from time 0 to time t, that is, these assignments in σ_T are idle time in T_t. Let the total time jobs $j, j+1, \ldots, n$ need to be processed in the interval $[0, t]$ in T_t be $P'_j = (p'_j, p'_{j+1}, \ldots, p'_n)$, respectively. Also, define the virtual machines (see footnote 1) from time 0 to t in T_t. Let $W'(\mu_i)$ be the total processing power of the ith virtual machine. Clearly,

$$\sum_{i=1}^{k} x_i(P'_j) \leq \sum_{i=1}^{k} W'(\mu_i) \qquad 1 \leq k \leq \min\{n - j + 1, m - 1\}$$

and

$$\sum_{i=1}^{n-j+1} x_i(P'_j) \leq \sum_{i=1}^{m} W'(\mu_i).$$

We claim that either μ_m is the only tight machine for which job j is critical and the number of nonzero values $p'_j, p'_{j+1}, \ldots, p'_n$ is at most m; or job j is critical for at least one virtual machine $l < m$. If none of the virtual machines is tight for job j, or if μ_m is the only tight machine for which job j is critical and the number of nonzero values $p'_j, p'_{j+1}, \ldots, p'_n$ is larger than m, then procedure $MC_j(S_X)$ would have found an earlier termination time for job j when procedure $MC_j(S_X)$ constructed σ_T. So let l be the smallest integer such that job j is critical for virtual machine l. Let jobs $i_1, i_2, \ldots, i_{l-1}$ and j be the critical jobs because of virtual machine l. Let ϵ be a small enough value such that if we were to increase the execution time requirement of job j by ϵ none of the virtual machines $\mu_1, \mu_2, \ldots, \mu_{l-1}$ would be tight. This value of ϵ will be used later on. We assert that one of the jobs in $\{i_1, i_2, \ldots, i_{l-1}\}$ satisfies the conditions for job i_d identified above; that is, job i_d is scheduled just before time t in σ_T and its induced deadline is less than its original deadline. Otherwise, schedule σ does not exist because jobs $1, 2, \ldots, j, i_1, i_2, \ldots, i_{l-1}$ do not have a feasible schedule that includes S and in which job j finishes before

time t. Note that jobs $i_1, i_2, \ldots, i_{l-1}$ and job j use the fastest machines available (excluding the ones used for S) in schedule σ_T from time zero till they reach their induced deadlines.

Let us assume that all remaining processing requirements (p') in T_t for jobs $i_1, i_2, \ldots, i_{l-1}$ and j are different. Later on, we explain how to handle the more general case. The value of δ is defined in such a way that in σ_T the time intervals $[t-\delta, t]$ and $[D_\alpha, D_\alpha + \delta]$ satisfy the following conditions:

(1) At most, one job is executed in σ_T on each of the machines in each of the two intervals;
(2) If $p'_i > p'_k$, then, even if job i is scheduled for the next two intervals of length δ on the fastest machine, it is the case that the remaining processing time for job i is strictly greater than the one for job k. This is true even if job k was not processed in these two intervals. That is, $p'_i - 2\delta v_1 > p'_k$; and
(3) δ is less than ϵ / v_1.

Clearly, jobs $i_1, i_2, \ldots, i_{l-1}$ and j must each be scheduled on only one machine in σ_T during the time interval $[t-\delta, t]$. Jobs j and i_d are not scheduled in the interval $[D_\alpha, D_\alpha + \delta]$.

Let $T_{t-\delta}$ be schedule σ_T after deleting all the assignments from time 0 to time $t-\delta$ for jobs $j, j+1, \ldots, n$, that is, these assignments in σ_T are idle time in $T_{t-\delta}$. Let $P''_j = (p''_j, p''_{j+1}, \ldots, p''_n)$ be the remaining processing time requirements for jobs $j, j+1, \ldots, n$ in $T_{t-\delta}$. Also, define the virtual machines (see footnote 1) for schedule $T_{t-\delta}$ from time 0 to time $t-\delta$. Let $W''(\mu_i)$ be the total processing power of the ith virtual machine. Clearly,

$$\sum_{i=1}^{k} x_i(P''_j) \leq \sum_{i=1}^{k} W''(\mu_i) \qquad 1 \leq k \leq \min\{n-j+1, m-1\}$$

and

$$\sum_{i=1}^{n-j+1} x_i(P''_j) \leq \sum_{i=1}^{m} W''(\mu_i).$$

We define schedule T^i as schedule $T_{t-\delta}$ except that we interchange job i_d in $[t-\delta, t]$ with the job scheduled on the same machine (j_1) in the interval $[D_\alpha, D_\alpha + \delta]$. If the new job in the interval $[t-\delta, t]$ is already scheduled in this interval, then interchange that assignment in machine j_2 with the corresponding one in $[D_\alpha, D_\alpha + \delta]$. Repeat this until we obtain an interchange without conflicts, that is, the new job in the interval $[t-\delta, t]$ is either one that is not assigned in that time interval, or it is just idle time. Figure 5 shows a sequence of interchanges including the additional one for jobs j and i_d. Now instead of executing job j in schedule T^i during the interval $[t-\delta, t]$ on machine j_0, we execute job i_d (a portion of job i_d that was executed before time $t-\delta$ in σ_T). The net effect of this interchange is that the completion time of job i_d increases by δ units, and, as we will show, it is possible to schedule job j in such a way that it has a completion time of at most $t-\delta$.

Let $P^i_j = (p^i_j, p^i_{j+1}, \ldots, p^i_n)$ be the remaining processing time for jobs $j, j+1, \ldots, n$ in schedule T^i Also, define the virtual machines (see footnote 1) in T^i. Let $W^i(\mu_i)$ be the total processing power of the ith virtual machine. Note that

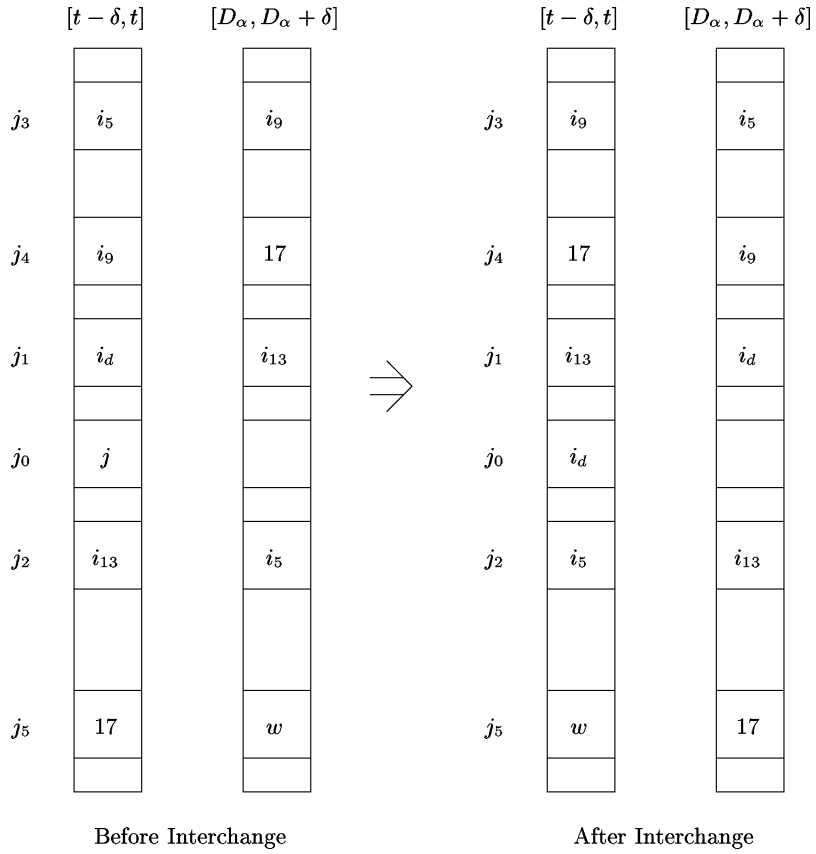

FIG. 5. Job interchange.

$W^i(\mu_i) = W''(\mu_i)$. Furthermore, $p^i_j = p''_j + \delta v_{j_0}$ and $p^i_{i_d} = p''_{i_d} - \delta v_{j_0}$, where j_0 is the machine that processes job j in the time interval $t - \delta$ and t in σ_t. For all other jobs x, $p^i_x = p''_x$.

From the definition of δ, we know that the ordering of the jobs $j, i_1, i_2, \ldots, i_{l-1}$ with respect to the processing requirements given by p' is the same as the one for p'', which is also the same as the one for p^i. Let Q be any proper subset of $j, i_1, i_2, \ldots, i_{l-1}$ of job indices of the largest remaining processing times jobs with respect to p^i. We now establish that

$$\sum_{q \in Q} p^i_q \leq \sum_{k=1}^{|Q|} W^i(\mu_k).$$

By the definition of ϵ, we know that

$$\epsilon + \sum_{q \in Q} p'_q \leq \sum_{k=1}^{|Q|} W'(\mu_k).$$

Subtracting from both sides the total processing of these jobs that takes place during the interval $[t - \delta, t]$ in σ_T then (since the ordering of the jobs is the same before

and after the interval) the expression becomes

$$\epsilon + \sum_{q \in Q} p''_q \leq \sum_{k=1}^{|Q|} W''(\mu_k).$$

The right-hand side is equal to $\sum_{k=1}^{|Q|} W^i(\mu_k)$. Now,

$$\sum_{q \in Q} p^i_q = \sum_{q \in Q} p''_q,$$

except when the subset Q includes job j but not job i_d. In this case,

$$\sum_{q \in Q} p^i_q = \delta v_{j_0} + \sum_{q \in Q} p''_q.$$

But, by definition, $\delta < \epsilon / v_1$. So it follows that

$$\sum_{q \in Q} p^i_q \leq \delta v_{j_0} + \sum_{q \in Q} p''_q < \epsilon + \sum_{q \in Q} p''_q \leq \sum_{k=1}^{|Q|} W''(\mu_k) = \sum_{k=1}^{|Q|} W^i(\mu_k).$$

Since for $R = \{i_1, i_2, \ldots, i_{l-1}, j\}$,

$$\sum_{q \in R} p^i_q = \sum_{k=1}^{|R|} W^i(\mu_k),$$

by proving a lemma similar to Lemma 2 we know it is possible to schedule the remaining processing times of the jobs $j, j+1, \ldots, n$ in T^i. This contradicts the way σ^* was defined, since the new schedule has the same $\sum C_j$ and jobs $1, 2, \ldots, j-1$ finish at the same times as in σ^*; however, job j has a smaller completion time. This completes the proof of the theorem for the case when the processing requirements of jobs $i_1, i_2, \ldots, i_{l-1}$ and j have different processing requirements.

To complete the proof of the theorem, we need to show how we can arrive at a contradiction when some of the jobs $i_1, i_2, \ldots, i_{l-1}$ and j have identical processing requirements. The value of δ is defined exactly as before. But now, we need to alter the schedule in the interval $[t - \delta, t]$ for jobs that have identical execution times. What we will do is to apply the Highest Level algorithm in the interval and the jobs with identical processing requirements will use processor sharing. As a result of this, every set of k jobs with identical execution times will have the same remaining processing requirements before and after the δ interval because of the processor sharing. We cannot claim that only one job will be scheduled in each machine during the entire interval. However, the interval can be partitioned into several subintervals in which one job will be scheduled on each machine. Now we find job i_d as before. But instead of making one set of interchanges for the two intervals of length δ we will make one set of interchanges in each subinterval. The proof now follows similar arguments as those described above. This completes the proof of the theorem. □

THEOREM 2. *Given the completion sequence in an optimal schedule, the SP procedure always generates an optimal schedule.*

PROOF. The *SP* procedure schedules each job to complete as early as possible. By Theorem 1, it produces the minimum $\sum C_j$. When a job is scheduled, the *SP*

procedure ensures that the remaining jobs can meet their induced deadlines. Thus, the schedule produced by *SP* is feasible. □

4. *The Job-Ordering Procedure*

The following theorem shows that the *JOP* procedure does generate an optimal completion ordering. The proof of this theorem is similar to the one given by Leung and Pinedo [2003].

THEOREM 3. *The JOP procedure yields an optimal completion ordering* $\bar{L}$.

PROOF. Let $L' = (1', 2', \ldots, n')$ be an optimal ordering and σ' be an optimal schedule with completion sequence $1', 2', \ldots, n'$. By Theorem 2, we may assume that σ' is constructed by the *SP* procedure using the list L'. Let $\bar{L} = (1, 2, \ldots, n)$ be the ordering obtained by the JOP procedure. Let k be the smallest index such that $k' \neq k$, that is, $i' = i$ for each $1 \leq i < k$ but $k' \neq k$. We differentiate among three cases, depending upon the processing requirements of jobs k' and k.

Case I. $p_{k'} < p_k$. Since $p_{k'} < p_k$, job k' appears before job k in the initial ordering of $\bar{L}$ but appears after job k in the final ordering. This means that job k' was considered in the job ordering process before job k, but was overtaken by job k. In the *JOP* procedure, a job is overtaken by other jobs only if it fails to produce a feasible schedule. That is, it is infeasible to schedule jobs $1, 2, \ldots, k-1, k'$, but feasible to schedule jobs $1, 2, \ldots, k-1, k$. This is impossible since $p_{k'} < p_k$.

Case II. $p_{k'} = p_k$. If the original deadline of job k' is greater than or equal to that of job k, that is, $\bar{d}_{k'} \geq \bar{d}_k$, then we can swap k' with k in L' and the new ordering will produce a feasible schedule with total completion time equal to that of σ'. Thus, we may assume that $\bar{d}_{k'} < \bar{d}_k$. In this case job k' appears before job k in the initial ordering of $\bar{L}$, but it appears after job k in the final ordering. Again, this means that job k' was overtaken by job k in the job ordering process. But this is impossible, since if it is feasible to complete job k before job k', it must also be feasible to complete job k' before job k. (Recall that $p_{k'} = p_k$ and $\bar{d}_{k'} < \bar{d}_k$.)

Case III. $p_{k'} > p_k$. If $\bar{d}_{k'} \geq \bar{d}_k$, then we can swap k' with k in L' and the new ordering will produce a feasible schedule with total completion time less than that of σ'. Thus, we may assume that $\bar{d}_{k'} < \bar{d}_k$. If job k completes by $\bar{d}_{k'}$ in σ', then we can swap k' with k in L' and the new ordering will produce a feasible schedule with total completion time less than that of σ'. Thus, we may assume that job k completes after $\bar{d}_{k'}$ (but at or before $\bar{d}_k$) in σ'. Let $\bar{d}_{k'} = D_x$ and the induced deadline of job k in σ' be D_y. By our assumption, $D_x < D_y \leq \bar{d}_k$.

Consider the jobs that follow job k' in L' up until job k. We assert that there is a job j with its original deadline $\bar{d}_j > D_{y-1}$ and job j starts before D_{y-1} in σ'. This is because $\bar{L}$ (with job k in the kth position) indicates that it is possible to complete job k by D_x along with all the jobs whose original deadlines are less than or equal to D_x. But $D_x \leq D_{y-1}$. Since job k does not complete by D_{y-1} in σ', there must be another job in its place. We consider two cases, depending upon the processing requirements of jobs j and k.

If $p_j < p_k$, then job j was considered in the job ordering process before job k, but it was overtaken by job k. This is impossible since σ' indicates that it is feasible to complete job j before job k.

If $p_j \geq p_k$, then we can swap j with k in L' and the new ordering will produce a feasible schedule with total completion time less than or equal to that of σ'. We can repeat the above argument until job k completes by $\bar{d}_{k'}$, at which time we can swap job k' with job k. □

5. *Time Complexity And Refinements*

THEOREM 4. *Given m machines and $n \geq m$ jobs, the procedure SP generates a schedule in $O(n^2m^3 + n^3m)$ time with at most $O(nm)$ preemptions.*

PROOF. Step 1 in Procedure *SP* takes $O(n)$ time and Step 3 takes constant time. Step 2 takes $t(n-j+1, m)$ time, where $t(k, m)$ is the time complexity of procedure $MC_j(S)$ for k jobs and m machines. Determining the virtual machines takes $O(m^2)$ time when performed independently from the previous iteration. If it is performed in conjunction with the previous step it just takes $O(m)$ time. Steps 5, 6, and 7 take $O(m)$ time. Since Steps 2–9 is repeated at most n times the overall time complexity is $O(nm^2 + nt(n-j+1, m))$.

Let us consider procedure $MC_j(S)$ when it determines the earliest possible completion time for job j. Suppose there are n jobs that need to be scheduled and there are m machines. Step 1 takes $O(n)$ time. When using the Highest Level scheduling procedure, two or more jobs may end up with identical remaining processing time even though they previously had different execution time requirements. Such an event is called *decreasing the number of different processing requirements* or simply *decreasing-dpr events*. Clearly, during the scheduling process there may be at most $n-1$ decreasing-dpr events. When there are k jobs to be scheduled in an interval, Step 2 is executed once, and there are $h \geq 0$ decreasing-dpr events, then Step 2 takes $O((h+1)km)$ time since there are $h+1$ regions where km different time slots are needed to schedule the jobs under processor sharing. Since Step 2 may be repeated n times, k is at most n, and the total number of decreasing-dpr events is at most n, it follows that the overall time complexity for this step is $O(n^2m)$ time. Step 3 takes $O(m^2)$ time and Step 4 takes $O(n+m)$ time. The number of different break points in Step 5 is $O(m^2)$. For each adjacent pair of break points, Step 6 takes $O(nm)$ time plus $O(hnm)$ time, where h is the total number of decreasing-dpr events over all the break point intervals. So the overall time for this step is $O(nm^3 + n^2m)$, since the total number of decreasing-dpr events is at most n. Step 7 takes constant time. Steps 8 and 10 take $O(nm)$ time for every decreasing-dpr events. Since there are at most n such events, the overall time complexity for these steps is $O(n^2m)$. Step 9 takes $O(n+m)$ time. The overall time complexity for procedure $MC_j(S)$ (or $t(n, m)$) is $O(nm^3 + n^2m)$. Therefore, the overall time complexity for procedure *SP* is $O(n^2m^3 + n^3m)$.

The number of preemptions introduced for each job is at most $2(m-1)$ since when we schedule job j it is scheduled on two virtual machines and the two virtual machines have at most $2(m-1)$ blocks of nonzero time. Since there are n jobs, the total number of preemptions introduced is at most $n(2m-2)$. □

One can speed-up procedure $MC_j(S)$ by making simple changes. The modifications are based on the observation that the schedule constructed by this procedure is a temporary one whose purpose is to find the earliest possible completion time t for job j in schedule S provided it is possible to complete all jobs by their induced

deadline. Instead of generating nm time slots for the scheduling of n tasks, we just need to update the remaining processing time for the jobs. This can be done in $O(m)$ time. We just need a list of the jobs being executed in an interval. All the jobs with identical remaining execution time are listed together with an integer specifying the number of jobs with that remaining processing time. The list will have at most m entries. One can easily compute in $O(m)$ time the remaining processing time of the tasks when scheduled in an interval that ends when we reach the end of the interval or an decreasing-dpr event takes place. Instead of the processing taking $O(hnm)$ (or $O(n^2m)$) time it just takes $O(hm)$ (or $O(nm)$) time. Therefore, the overall time complexity is reduced to $O(nm^3 + n^2m)$. We call this procedure *FastSP*.

THEOREM 5. *Given m machines and $n \geq m$ jobs, the procedure FastSP generates a schedule in $O(nm^3 + n^2m)$ time with at most $O(nm)$ preemptions.*

PROOF. Follows from the discussion above. □

The JOP procedure involves at most n iterations, since each iteration of the procedure fixes the position of at least one job. Thus, we have the following main result of this article.

THEOREM 6. *$Qm \mid prmt, \bar{d}_j \mid \sum C_j$ can be solved in $O(n^2m^3 + n^3m)$ time with at most $O(nm)$ preemptions.*

6. *Extensions and Conclusions*

In this article, we presented a polynomial-time algorithm for $Qm \mid prmt, \bar{d}_j \mid \sum C_j$. This algorithm can be used to solve other scheduling problems as well. Suppose that, instead of a deadline, each job j has a due date d_j, and the objective is to minimize the maximum lateness, where the lateness of a job is defined to be the difference between its completion time and its due date. (In the 3-field notation, this problem is denoted by $Qm \mid prmt \mid L_{\max}$.) This problem can be solved as follows. Parametrize on the maximum lateness. Assume $L_{\max} = z$ and create for all jobs the deadlines $d_j + z$. We then check if there is a feasible schedule with this set of deadlines. The optimal value for the maximum lateness can be obtained by conducting a binary search of z in a range between a lower and an upper bound. Once the minimal value of z has been obtained, say z^*, we can use the algorithm described in this article to find a schedule that minimizes $\sum C_j$. In this way, we can solve the problem of minimizing $\sum C_j$, subject to the constraint that $L_{\max}$ is minimum. Of course, the algorithm will also work for any $L_{\max}$ greater than or equal to z^*.

The algorithm presented in this article can also form a basis for a polynomial-time algorithm for a multi-objective scheduling problem with the same machine environment and the objective $\alpha_1 \sum C_j + \alpha_2 L_{\max} + \alpha_3 C_{\max}$ with α_1, α_2 and α_3 being the weights of the three objectives. Again, the algorithm can be developed by parametrizing on both $L_{\max}$ and $C_{\max}$.

Polynomial-time algorithms that check whether a set of jobs is feasible do exist, even when the jobs have release dates and deadlines. Federgruen and Groenevelt [1986] showed that the problem of determining feasibility can be reduced to a network flow problem. Faster algorithms exist if the jobs have identical release dates or identical deadlines (see Cho and Sahni [1980]).

More efficient algorithms exist for a single machine. Smith [1956] showed that this problem can be solved in $O(n \log n)$ time. Smith's rule schedules the jobs backward, starting at time $t = \sum_{j=1}^{n} p_j$. From among all the jobs that can complete at time t (i.e., jobs whose deadline is greater than or equal to t), choose the one with the largest processing time. This reduces the problem to a set of $n - 1$ jobs to which the same rule applies. Preemption is not necessary for a set of jobs with the same release date. Thus, Smith's rule solves $1 \mid prmt, \bar{d}_j \mid \sum C_j$ as well as $1 \mid \bar{d}_j \mid \sum C_j$.

If each job j has, instead of a deadline, a release date r_j (before which it cannot start its processing), then minimizing $\sum C_j$ is NP-hard for a single machine in the nonpreemptive case but solvable in polynomial time in the preemptive case. Lenstra [1977] showed that $1 \mid r_j \mid \sum C_j$ is NP-hard and Baker [1974] presented an $O(n \log n)$ algorithm for $1 \mid prmt, r_j \mid \sum C_j$. Baker's rule schedules, at each point in time, the job with the smallest remaining processing time from among all the available jobs.

Lawler [1982] posed a single machine problem with release dates and deadlines, that is, $1 \mid prmt, r_j, \bar{d}_j \mid \sum C_j$, and asked whether it can be solved in polynomial time. This question has been answered in the negative by Du and Leung [1993] who showed that the problem is NP-hard. Thus, as far as polynomial-time algorithms are concerned, we cannot have both release dates and deadlines in the problem. Hence, we are only able to solve problems with either release dates or deadlines, but not both.

Consider first the problem $Pm \mid prmt, r_j \mid \sum C_j$. Lawler [1982] asked whether this problem can be solved in polynomial time. Du et al. [1990] showed that it is NP-hard, even for two identical and parallel machines; that is, $P2 \mid prmt, r_j \mid \sum C_j$ is NP-hard in the ordinary sense. Thus, there is no hope in developing a polynomial-time algorithm for multimachine when jobs have different release dates.

The situation is more promising when the jobs have different deadlines. Leung and Pinedo [2003] gave a polynomial-time algorithm for $Pm \mid prmt, \bar{d}_j \mid \sum C_j$ for each $m \geq 2$. In this article, we show that a more general version of the problem, that is, $Qm \mid prmt, \bar{d}_j \mid \sum C_j$, can also be solved in polynomial time for every $m \geq 2$. As noted before, Sitters [2001] has already shown that $Rm \mid prmt, \bar{d}_j \mid \sum C_j$ is NP-hard in the strong sense, for arbitrary m and $\bar{d}_j = \infty$ for all j. The only question that remains open is whether $Rm \mid prmt, \bar{d}_j \mid \sum C_j$ can be solved in polynomial time for fixed $m \geq 2$.

REFERENCES

BAKER, K. R. 1974. *Introduction to Sequencing and Scheduling*, Wiley, New York.

CHO, Y., AND SAHNI, S. 1980. Scheduling independent tasks with due times on a uniform processor system. *J. ACM. 20*, 550–563.

DU, J., AND LEUNG, J. Y.-T. 1993. Minimizing mean flow time with release time and deadline constraints. *J. Algor. 14*, 45–68.

DU, J., LEUNG, J. Y.-T., AND YOUNG, G. H. 1990. Minimizing mean flow time with release time constraint. *Theoret. Comput. Sci. 75*, 347–355.

FEDERGRUEN, A., AND GROENEVELT, H. 1986. Preemptive scheduling of uniform machines by ordinary network flow techniques. *Manage. Sci. 32*, 341–349.

GONZALEZ, T. F. 1978. Minimizing the mean and maximum finishing time on uniform processors. Tech. Rep. CS-78-22. Dept. Comput. Sci. The Pennsylvania State University, University Park, PA.

GONZALEZ, T. F., AND SAHNI, S. 1978. Preemptive scheduling of uniform processor systems. *J. ACM. 25*, 92–101.

GRAHAM, R. L., LAWLER, E. L., LENSTRA, J. K., AND RINNOOY KAN, A. H. G. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math. 5*, 287–326.

HORVATH, E. C., LAM, S., AND SETHI, R. 1976. A level algorithm for preemptive scheduling. *J. ACM. 23*, 317–327.

LAWLER, E. L. 1982. Recent results in the theory of machine scheduling. In *Mathematical Programming: The State of the Art*. A. Bachem, M. Grotschel, and B. Korte, Eds. Springer-Verleg, Berlin, Germany.

LENSTRA, J. K. 1977. Sequencing by Enumerative Methods. Mathematical Centre Tracts 69, Mathematisch Centrum, Amsterdam, the Netherlands.

LEUNG, J. Y.-T., AND PINEDO, M. 2003. Minimizing total completion time with parallel machines with deadline constraints. *SIAM J. Comput. 32*, 5, 1370–1388.

LIU, J. W. S., AND YANG, A. 1974. Optimal scheduling of independent tasks on heterogeneous computing systems. In *Proceedings of the ACM Annual Conference* (San Diego, CA, Nov.). ACM, New York, pp. 38–45.

MCCORMICK, S. T., AND PINEDO, M. 1995. Scheduling *n* independent on *m* uniform machines with both flow time and makespan objectives: A parametric analysis. *ORSA J. Comput. 7*, 63–77.

PINEDO, M. 2002. *Scheduling: Theory, Algorithms and Systems*. Prentice-Hall, Englewood Cliffs, NJ.

SITTERS, R. A. 2001. Two NP-hardness results for preemptive minsum scheduling of unrelated parallel machines. In *Proceedings of the 8th International IPCO Conference*, Lecture Notes in Computer Science, vol. 2081. Springer-Verleg, New York, pp. 396–405.

SMITH, W. E. 1956. Various optimizers for single-stage production. *Nav. Res. Log. Quart. 3*, 59–66.

RECEIVED AUGUST 2004; REVISED JULY 2005; ACCEPTED JULY 2005