Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor*

Sanjoy K. Baruah
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Rodney R. Howell
Dept. of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Louis E. Rosier
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

We investigate the preemptive scheduling of periodic, real-time task systems on one processor. First, we show that when all parameters to the system are integers, we may assume without loss of generality that all preemptions occur at integer time values. We then assume, for the remainder of the paper, that all parameters are indeed integers. We then give as our main lemma both necessary and sufficient conditions for a task system to be feasible on one processor. Although these conditions cannot, in general, be tested efficiently (unless P = NP), they do allow us to give efficient algorithms for deciding feasibility on one processor for certain types of periodic task systems. For example, we give a pseudo-polynomial-time algorithm for synchronous systems whose densities are bounded by a fixed constant less than 1. This algorithm represents an exponential improvement over the previous best algorithm. We also give a polynomial-time algorithm for systems having a fixed number of distinct types of tasks. Furthermore, we are able to use our main lemma to show that the feasibility problem for task systems on one processor is co-NP-complete in the strong sense. In order to show this last result, we first show the Simultaneous Congruences Problem to be NP-complete in the strong sense. Both of these last two results answer questions that have been open for ten years. We conclude by showing that for incomplete task systems, i.e., task systems in which the start times are not specified, the feasibility problem is Σ_2^P -complete.

1 Introduction

The study of scheduling theory has recently become a rapidly expanding area of research (see, e.g., [Bla87]). One specific area of scheduling theory which has been studied for a number of years involves preemptive scheduling of periodic, real-time task systems with deadlines [LL73, Lab74, LM80, LM81, LW82, Leu89]. A task system consists of a finite number of tasks, each of which is released at regular periodic intervals. In general, the initial release of individual tasks may occur at different times. Each time a task is released, it

^{*}This work was supported in part by National Science Foundation Grant No. CCR-8711579.

must be scheduled on a processor for some specified number of time units before its deadline is reached. The execution time requirement and the amount of time before the deadline is reached are both unchanged for each subsequent release of the task, but may be different for different tasks. We are interested in schedules in which a task may be preempted and resumed at a later time, or if we are considering a multiprocessor environment, resumed by a different processor. However, we require that no single task be simultaneously scheduled on more than one processor. There is no penalty associated with a preemption.

In most of the work done in the past (e.g., [LL73, LM80, LM81, LW82, Leu89]), the parameters to the problems have not been restricted to integer values, and preemptions have been allowed at any time. Furthermore, in schedules constructed by several algorithms in the literature, preemptions may occur at noninteger time values even if all start times, execution times, and periods are integers (see e.g., [C+76, Leu89]). We will refer to a schedule in which preemptions may occur at arbitrary time values as a continuous schedule. In Section 2, we argue in favor of considering only discrete schedules; i.e., schedules in which preemptions may only occur at specified discrete intervals. In particular, we first restrict all inputs to be integers, then allow preemptions to occur only at integer time values. We believe this work is the first to assume restrictions of this type [Mok89]. There are two issues involved in assuming these restrictions. The first issue is whether it is reasonable to restrict the inputs to integer values. We discuss this issue in Section 2, concluding that integer inputs provide the proper abstraction to the underlying physical problem. The second issue is whether, once the inputs have been restricted to integer values, it is reasonable to restrict preemptions to integer values. (Of course, if the inputs are allowed to be nonintegers, restricting preemptions to integer values necessarily causes a loss of generality.) Concerning this issue, we again conclude that the restriction is preferable. Furthermore, we show that once the inputs have been restricted to integer values, we can assume without loss of generality that preemptions occur only at integer values; i.e., we prove that valid discrete schedules exist whenever valid continuous schedules exist, provided the parameters are all integers. Although we show this fact specifically for single processor schedules, the proof is general enough to extend to task systems on several identical processors.

The ultimate goal regarding task systems is to find an algorithm to mechanically synthesize online scheduling algorithms. Such an algorithm would first determine the feasibility of an instance — i.e., whether there exists a valid schedule — and then construct a suitable scheduling algorithm. The Deadline Algorithm (see [LL73, Lab74]) is known to be an optimal scheduling algorithm for task systems on one processor; that is, it produces a valid schedule for every feasible task system. This algorithm generates a cyclic schedule with a potentially exponential-length period at the rate of $O(\log n)$ steps per time unit of the schedule, and can therefore be considered an exponential-time feasibility test (see [LM80]). However, an exponential-time feasibility test is usually unacceptable. On the other hand, if we could couple the Deadline Algorithm with a polynomial-time (or even a pseudo-polynomial-time) feasibility test, we would have made significant progress toward the realization of a viable algorithm for mechanical synthesis.

Let us review the current state of affairs concerning the feasibility problem — the problem of determining whether a given task system is feasible. The most efficient algorithm known for the general feasibility problem for task systems on one processor was given by Leung and Merrill [LM80]; this algorithm operates in a time exponential in the number of tasks. They also showed this problem to be co-NP-hard. However, they have

left as open questions whether the problem is co-NP-complete or co-NP-hard in the strong sense. Nothing along these lines appears to be known regarding synchronous systems — those task systems in which all start times are identical. However, Leung and Whitehead [LW82] have given a pseudo-polynomial-time algorithm for deciding feasibility with respect to fixed priority schedules. This last fact might be taken as evidence that a pseudo-polynomial-time algorithm exists for the feasibility problem for general synchronous systems.

In Sections 3 and 4 of this paper, we examine the feasibility problem for various types of task systems on one processor. The types of task systems we consider fall into two main classes: *complete* task systems and *incomplete* task systems. In a complete task system, the start times, execution times, deadlines, and periods for each task are given as input; in an incomplete task system, the start times are omitted. All of the results given in the above paragraph are with respect to complete task systems. Recall that a complete task system is said to be feasible if a valid schedule exists. Similarly, an incomplete task system is said to be feasible if there exist start times such that the resulting complete task system is feasible.

We first show a main lemma that provides simple necessary and sufficient conditions for testing the feasibility of complete task systems on one processor. These conditions cannot, in general, be tested efficiently unless P = NP. We can show, however, that they yield efficient algorithms with respect to several significant subclasses of complete task systems. Consider, for example, synchronous task systems in which the sum of the ratios of the execution times to the periods, or the *density*, is bounded above by a fixed constant less than 1 (say, 0.9 or 0.99, for example). This restriction is fairly minor, since all task systems that are feasible on one processor have a density no greater than 1 (see, e.g., [LM80]). The lemma yields a simple algorithm for testing the feasibility of such systems on one processor in a time proportional to the product of the number of tasks and the largest period — an exponential improvement over the algorithm given by Leung and Merrill [LM80] for general complete task systems. Our algorithm is pseudo-polynomial, since the value of the largest period is not necessarily polynomial in the size of the problem description. While this result does not provide a pseudo-polynomial-time algorithm for all synchronous systems, it is the next best thing.

The obvious next question is whether such an algorithm can be developed for asynchronous task systems. Unfortunately, we show that such an algorithm is not possible unless P = NP; i.e., we show that the feasibility problem for general task systems on one processor is co-NP-hard in the strong sense even if we restrict ourselves to instances whose densities are bounded above by any fixed positive constant. This seemingly destroys any hope for obtaining pseudo-polynomial-time algorithms for solving the general feasibility problem for complete task systems on one processor. In showing this last result, we also show the Simultaneous Congruences Problem to be NP-complete in the strong sense; both of these results answer questions that have been open for ten years [LM80, LW82]. We then use our main lemma to answer a third open question [LM80] by showing the feasibility problem for complete task systems on one processor to be co-NP-complete in the strong sense.

Another significant subclass of task systems for which our main lemma proves useful is the set of task systems having a fixed number of distinct types of tasks. For this subclass, we derive a polynomial-time algorithm for deciding feasibility on one processor. Even in the case of incomplete task systems, this result

yields a pseudo-polynomial-time algorithm, in spite of the fact that we show the general feasibility problem for incomplete task systems on one processor to be Σ_2^P -complete.

Throughout the paper, we show how several of our results can be extended to multiprocessor scheduling. We conclude in Section 5 with a discussion of the possibility of further extensions. In particular, we discuss the possibility of extending our results to continuous schedules for noninteger inputs.

2 Discrete Schedules vs. Continuous Schedules

In this section, we argue in favor of considering preemptive scheduling problems with respect to discrete schedules as opposed to the more common continuous schedules. In particular, we first discuss the proper type of inputs to the problem; we conclude that integer inputs provide the proper abstraction. We then show that no loss in generality with respect to feasibility results from restricting schedules to be discrete, once the inputs are assumed to be integers.

In order to make a formal comparison of the two types of schedules, we must define them more formally than in the introduction. For discrete schedules, such a definition is straightforward. Let N denote the natural numbers, and let $N^+ = N - \{0\}$. For a task system T of n tasks, a discrete schedule on one processor is a mapping $S: N \to \{0, 1, ..., n\}$. Intuitively, if $i \neq 0$, then S(t) = i means that task i is scheduled at time t; if S(t) = 0, then no task is scheduled at time t. For each task T_i in T, let $s_i \in N$ denote the start time, $e_i \in N^+$ denote the execution time, $d_i \in N^+$ ($d_i \geq e_i$) denote the deadline, and $p_i \in N^+$ ($p_i \geq d_i$) denote the period. Then S is valid if for all $k \in N$, $i \in \{1, ..., n\}$, there are at least e_i times t such that S(t) = i and $s_i + kp_i \leq t < s_i + kp_i + d_i$. These definitions may be extended naturally to m identical processors by making $S: N \to \{0, 1, ..., n\}^m$ such that for all $t \in N$, all nonzero components of S(t) are distinct.

Formally defining continuous schedules is a bit more difficult. The problem occurs in allowing preemptions to occur "at any time." Taken literally, this definition could allow us to schedule some task T_1 at all rational points between a and b, and some other task T_2 at all irrational points between a and b. In such a schedule, it is unclear how much time was spent in processing each task. Furthermore, it does not seem to be very natural to consider such a function to be a schedule because in each interval $[t_1, t_2)$, $t_1 < t_2$, both T_1 and T_2 are scheduled. Restricting the time values to the rational numbers is of no help, either, as we may schedule T_1 at exactly those points p/q between a and b such that p/q is a reduced fraction and p is odd. Somewhat informally, suppose we have some function χ such that for any t, $\chi(t)$ is either 0 or 1, as shown in Figure 1 (we will leave the domain of χ unspecified for the time being). Specifically, $\chi(t) = 1$ iff some particular task T_i is scheduled at t. We want some way of describing the amount of time between points a and b at which $\chi(t) = 1$. Note that this amount is the same as the area bounded by the t axis on the bottom, $\chi(t)$ on the top, t = a on the left, and t = b on the right (the shaded area of Figure 1). More formally, if χ is a Riemann integrable function on [a, b], then $\int_a^b \chi(t) \, dt$ describes the amount of time between a and b during which χ has a value of 1. Therefore, let $\chi_i : N \to \{0, 1\}$ be defined by

- $\chi_i(i) = 1$; and
- $\chi_i(j) = 0$ if $i \neq j$

for $i \in N$. Let R^+ denote the nonnegative real numbers. We define a *continuous schedule* of T on one processor to be a function $S: R^+ \to \{0, 1, ..., n\}$ such that for all $i \in \{1, ..., n\}$, $(\chi_i \circ S)$ is Riemann integrable on R^+ (or equivalently, such that S is Riemann integrable on R^+). Intuitively, S(t) gives the job scheduled at t (or 0 if no job is scheduled), and $\chi_i(S(t)) = 1$ iff task T_i is scheduled at time t. S is valid if for all $k \in N$, $i \in \{1, ..., n\}$,

$$\int_{s_i+kp_i}^{s_i+kp_i+d_i} \chi_i(S(t)) \ dt \ge e_i.$$

These definitions also may be extended to m identical processors in a straightforward manner.

As an example, let us consider the function $S_1: \mathbb{R}^+ \to \{0,1,2\}$ defined by

- $S_1(t) = 1$ if t is rational; and
- $S_1(t) = 2$ if t is irrational.

Since S_1 is not integrable, it is not a schedule. On the other hand, consider the function $S_2: \mathbb{R}^+ \to \{0, 1, 2\}$ defined by

- $S_2(t) = 1$ if $k 2^{-2i} \le t < k 2^{1-2i}$ for some $i \in N, k \in N^+$; and
- $S_2(t) = 2$ otherwise.

 S_2 can be partitioned into intervals [a,b), a < b, such that for each $t \in [a,b)$, $S_2(t)$ has the same value; however, there are finite intervals which must be partitioned into infinitely many such intervals. The question then arises as to whether S_2 is a schedule. Under the above definition, S_2 qualifies as a continuous schedule, since it is Riemann integrable on R^+ . (For example, $\int_0^1 \chi_1(S_2(t)) dt = 2/3$.)

It is clear that any function $S: R^+ \to \{0, 1, ..., n\}$ for which all intervals $[t_1, t_2)$ can be partitioned into a finite number of subintervals $I_1, I_2, ..., I_m$ such that S is constant over each I_i will be Riemann integrable, and will therefore satisfy our definition of a continuous schedule. Since we have also included schedules like S_2 above in our definition, this definition appears to be quite general. The definition could be generalized even further to include functions like S_1 if we were to use the Lebesgue integral rather than the Riemann integral; however, our definition seems to be general enough for all practical purposes¹. This very general definition was chosen in order to make a strong case for the claim that discrete schedules should be considered rather than continuous schedules. We give two arguments to support this claim. First, we will show that discrete schedules are the most general types of schedules that remain "realistic." Second, we will show that for task systems on one processor, a continuous schedule exists iff a discrete schedule exists. The generality of our

¹ All of our results hold for the Lebesgue integral as well as the Riemann integral.

definition of continuous schedules makes this second claim very strong. Furthermore, our proof extends to complete task systems on several identical processors.

First, let us consider what types of schedules are realistic. For the sake of argument, we will temporarily relax our definitions so that the inputs to the problems do not need to be integers. Clearly, any scheduler is ultimately limited to scheduling in multiples of some discrete time unit due to timing constraints involved in computation. Even the inputs to the problem must be coerced to be multiples of this time unit. Furthermore, even if we wish to abstract away this limitation, all inputs to the problem must be expressed as rational numbers which may all be multiplied by a common denominator. Since this transformation can be done in polynomial time, and since we will show the feasibility problem to be co-NP-complete in the strong sense, we do not change the complexity by restricting the inputs to be integers. Finally, as we will show below, the resulting task system has a discrete schedule iff the original one has a continuous schedule.

We will now show that for task systems on one processor, a discrete schedule exists iff a continuous schedule exists. By the above argument, we will assume that all inputs are integers. However, in order to be completely general, we will not restrict the start times for incomplete task systems to be integers. Thus, we will first show that if an incomplete task system has a valid schedule in which the start times are not required to be integers, then it has a valid schedule in which the start times are integers. This proof makes use of the fact that the Deadline Algorithm is an optimal schedule for complete task systems on one processor; i.e., a valid schedule exists iff the Deadline Algorithm produces a valid schedule [LL73, Lab74]. In order to describe the Deadline Algorithm, we must first introduce some terminology. Suppose we have a complete task system T with start times s_i , execution times e_i , deadlines d_i , and periods p_i for $1 \le i \le n$. For a given schedule S, a task T_i is said to be active beginning at any release $s_i + kp_i$ until the next time b such that $\int_a^b \chi_i(S(t))dt = e_i$, where a is the last release prior to b. The Deadline Algorithm schedules, at each time t, that active task with the earliest deadline. If no task is active at t, no task is scheduled at t, and we assume that ties are broken by giving priority to the task with the lowest subscript.

Lemma 2.1: Let T be an incomplete periodic task system in which all inputs are integers. There is a valid schedule for T on one processor such that all start times are integers iff there is a valid schedule for T on one processor in which start times may be any nonnegative real numbers.

Proof: For each task T_i , let e_i , d_i , and p_i be the corresponding execution time, deadline, and period, respectively. Suppose there exist real start times s_i for each T_i such that the resulting complete task system T'' is feasible on one processor. Without loss of generality, assume the earliest start time is 0, and that if $s_i - \lfloor s_i \rfloor < s_j - \lfloor s_j \rfloor$, then i < j. Let S be the schedule generated by the Deadline Algorithm for T''. Since the Deadline Algorithm is an optimal scheduling algorithm for one processor, S is valid. For each task T_i , we define a new start time $s_i' = \lfloor s_i \rfloor$. Let T' be the resulting complete task system, and let S' be the schedule generated by the Deadline Algorithm for T'. (Note that by our assumption on the ordering of the tasks, the Deadline Algorithm uses the same priorities in S' as in S.) We claim that S' is valid.

Suppose, to the contrary, that some deadline in S' is not met, and let t_0 be the first such deadline. Let T'_{i_0} be the task with smallest subscript whose deadline at t_0 is not met. We construct nonperiodic task systems

U and U' such that U (U') consists of tasks $U_{i,k}$ ($U'_{i,k}$) defined by release time $s_i + kp_i$ ($s'_i + kp_i$), execution time e_i , and deadline $s_i + kp_i + d_i$ ($s'_i + kp_i + d_i$), for all i and k such that either $s'_i + kp_i + d_i < t_0$, or $s'_i + kp_i + d_i = t_0$ and $i \le i_0$. The double subscripts are merely a notational convenience; where convenient, we will also refer to $U_{i,k}$ ($U'_{i,k}$) as U_j (U'_j) if it has the jth earliest deadline (ties are broken according to the value of i). Thus, U' consists of all instances of tasks from T' whose deadlines occur before t_0 , and all instances of tasks from T' whose deadlines occur at t_0 and whose subscripts are no greater than i_0 ; U consists of the corresponding instances of tasks from T. By our assumptions, the Deadline Algorithm produces a valid schedule for U, but an invalid schedule for U'. We will show that this last conclusion does not hold, thereby arriving at a contradiction.

We will construct a series of nonperiodic task systems $U^0, ..., U^n$ such that n is the number of tasks in U (or equivalently U'), $U^0 = U$, $U^n = U'$, and the Deadline Algorithm produces a valid schedule for each U^j . For $0 \leq j \leq n$, we define U^j as containing $U'_1, ..., U'_i, U_{j+1}, ..., U_n$. Clearly, $U^0 = U$ and $U^n = U'$. We will show by induction on j that the Deadline Algorithm produces a valid schedule for U^{j} . Since we already know this fact for U^0 , we assume that for some j > 0, the Deadline Algorithm produces a valid schedule for U^{j-1} . Now U^j contains all the tasks in U^{j-1} except U_j , which is replaced by U'_j . Let S^{j-1} be the schedule produced by the Deadline Algorithm for U^{j-1} . We will show that S^j , the schedule produced by the Deadline Algorithm for U^j , is valid. Let r be the release time of U_j , r' be the release time of U'_j , d be the deadline of U_j , and d' be the deadline of U'_i . If r=r', then U^j is the same as U^{j-1} . Therefore, assume $0 \le r' < r < d' < d < t_0$. If $0 \le t < r'$, then $S^j(t) = S^{j-1}(t)$; clearly, all deadlines are met up to r'. Suppose $r' \leq t < r$. If $S^{j-1}(t) = 0$ or $S^{j-1}(t) > j$, then $S^{j}(t) = j$; otherwise, $S^{j} = S^{j-1}$ (note that U'_{i} 's execution time is at least 1 > r - r'). From r until U'_i becomes inactive, S^j is the same as S^{j-1} ; therefore, all deadlines for tasks U'_i , i < j, are met. Let f be the time at which U'_i finishes in S^j , and suppose, in order to derive a contradiction, that f > d'. Since U'_i is active from s' until f, $\sum_{i=1}^{j} \int_{r'}^{f} \chi_i(S^j(t)) dt = f - r'$. Recall that all release times and execution times for tasks U'_i , i < j, are integers. Thus, if i < j, U'_i cannot be preempted or finish except at an integer time value. Since r' is also an integer, this means that $\sum_{i=1}^{j-1} \int_{r'}^{f} \chi_i(S^j(t)) dt$ is an integer. Furthermore, since $\int_{r'}^{f} \chi_j(S^j(t)) dt$ is the execution time of U'_j , also an integer, f - r', and hence, f, must also be an integer. Since f > d', $f \ge d' + 1 > d$. Since S^j is identical to S^{j-1} from r to f, a deadline is violated in S^{j-1} — a contradiction. Thus, the deadline for U'_i is met in S^j . Finally, consider the tasks U_i such that i > j. Prior to f the only times allotted to U_i in S^{j-1} that are not allotted to U_i in S^j have been instead allotted to U'_i . Thus, the same amount of time is now available for allotment between f and d. Since U_i has a deadline no earlier than d, its deadline is not violated. We have now shown that for $0 \le j \le n$, the Deadline Algorithm produces a valid schedule for U^{j} ; in particular, the Deadline Algorithm produces a valid schedule for $U^n = U'$ — a contradiction. Therefore, S' is a valid schedule.

We are now ready to show that for any task system, if a valid continuous schedule on one processor exists, then a valid discrete schedule on one processor exists. Perhaps the simplest way of showing this fact is to observe that when all inputs are integers, the Deadline Algorithm produces a schedule with preemptions only at integers. Thus, since the Deadline Algorithm is optimal for one processor, the result follows from Lemma 2.1. However, since the Deadline Algorithm is not optimal for more than one processor, this strategy does not extend beyond the single processor case. The proof we give below is more general. (Throughout

the remainder of the paper, we assume all inputs to problems are integers.)

Theorem 2.1: For any (complete or incomplete) periodic task system T with integer-valued parameters, there is a valid discrete schedule for T on one processor iff there is a valid continuous schedule for T on one processor.

Proof: From Lemma 2.1, we may assume without loss of generality that T is a complete task system. Using a technique similar to that given by Horn [Hor74], we will construct, from T, a flow network G and an integer K such that valid schedules of T on one processor correspond to maximum flows of K in G. Furthermore, it will be the case that all capacities in G will be integers. Since there are maximal network flow algorithms that never introduce nonintegers if all capacities are integers (e.g., the Ford-Fulkerson Algorithm [FF62]), we will be able to generate a discrete schedule from an integer-valued maximum flow.

Let each task T_i , $1 \le i \le n$, consist of start time s_i , execution time e_i , deadline d_i , and period p_i . We first make use of the fact, shown by Leung and Merrill [LM80], that if T has a valid schedule on one processor, then it has a valid cyclic schedule with period $P = \text{lcm}(p_1, ..., p_n)$; i.e., for all t, S(t) = S(t + P). Using P, we can construct our network G (see Figure 2). G contains four types of nodes:

- 1. a source node a;
- 2. for all natural numbers $j \leq P 1$, a node t_j ;
- 3. for each task T_i and each $k \in \{0, 1, ..., (P/p_i) 1\}$, a node $q_{i,k}$; and
- 4. a sink node z.

G contains three types of edges:

- 1. from a to each Type 2 node, an edge with capacity 1 (to extend this proof to m processors, these capacities are changed to m);
- 2. from each Type 2 node t_j to each Type 3 node $q_{i,k}$ such that $s_i + kp_i \le k'P + t_j < s_i + kp_i + d_i$ for some $k' \in N$, an edge with capacity 1; and
- 3. from each Type 3 node $q_{i,k}$ to z, an edge with capacity e_i .

Finally, let K be the sum of the capacities of the Type 3 edges.

Now suppose there is a valid schedule for T on one processor. Then there is a cyclic schedule S with period P. Based on S, we will construct a flow for G. To each Type 3 edge $(q_{i,k},z)$ we assign a flow of e_i . In what follows, we will only be concerned with providing enough flow on the remaining edges to support the flow assigned to the Type 3 edges; if the assignment causes more flow to enter a node than the amount of flow leaving that node, we can always decrease the flow on incoming edges. To each Type 2 edge $(t_j, q_{i,k})$ we assign a flow of $\int_j^{j+1} \chi_i(S(t)) dt$, which is clearly no more than 1. Now each Type 3 node $q_{i,k}$ has incoming

edges from all Type 2 nodes t_j such that $s_i + kp_i \le k'P + t_j < s_i + kp_i + d_i$ for some $k' \in N$. Since S is cyclic with period P, the total flow into $q_{i,k}$ is

$$\sum_{j=s_i+kp_i}^{s_i+kp_i+d_i-1} \int_j^{j+1} \chi_i(S(t))dt = \int_{s_i+kp_i}^{s_i+kp_i+d_i} \chi_i(S(t))dt$$

$$\geq e_i$$

since S is valid. Finally, to each Type 1 edge (a, t_j) we assign a flow of $\sum_{i=1}^n \int_j^{j+1} \chi_i(S(t)) dt$, which is no more than 1 since $\sum_{i=1}^n \chi_i(S(t))$ never exceeds 1. From each Type 2 node, there is an edge to at most one Type 3 node $q_{i,k}$ for each i. Thus, the total flow out of t_j is $\sum_{i=1}^n \int_j^{j+1} \chi_i(S(t)) dt$. We have therefore shown how to construct a flow F in G in which all Type 3 edges are filled to capacity. Clearly, F represents a maximum flow of K.

Since there are algorithms that find a maximum flow consisting entirely of integers whenever all capacities are integers, G has a maximum flow F' that is entirely made up of integers. We now describe a valid discrete schedule S' as follows. Since each Type 2 node t_i can have an incoming flow of at most 1, it has at most one outgoing edge for which F' has a positive (=1) flow. If this edge is $(t_j, q_{i,k})$, let S(k'P + j) = i for all $k' \in N$. Now suppose some task T_i is released at time $s_i + k'P + kp_i$ for $k' \in N$ and $k \in \{0, 1, ..., (P/p_i) - 1\}$. (Clearly, k and k' can be chosen to yield any release time of T_i .) The number of time units assigned to this release of T_i in S is exactly the number of incoming edges to node $q_{i,k}$ having positive flow in F'. Since this number must equal the capacity of the single outgoing edge of $q_{i,k}$, or e_i , the schedule is valid.

As mentioned earlier, the above proof can be extended to complete task systems on m processors. Lawler and Martel [LM81] have shown that if a task system has a valid schedule on m processors, then it has a valid cyclic schedule of length P on m processors, where P is as in the above proof. Therefore, by changing the capacities of the Type 1 edges in the above proof to m, we show the theorem for complete task systems on m processors. In order to be able to extend the proof to incomplete task systems on m processors, it seems we must be able to extend Lemma 2.1 to m processors (since the start times are incorporated into the construction of the network G); however, we do not know at this time whether Lemma 2.1 extends to m processors.

3 Complete Task Systems

In this section, we examine the feasibility problem for complete task systems on one processor. We first note that Theorem 2.1 gives an algorithm for deciding the problem in general; however, the algorithm generates a network whose size is exponential in the number of tasks. In order to generate a more efficient algorithm, we give in our main lemma both necessary and sufficient conditions for a complete task system to be feasible. Although these conditions cannot, in general, be tested efficiently (unless P = NP), they are useful in constructing algorithms for certain special types of task systems. For example, we will use this lemma to give an algorithm for deciding feasibility of synchronous task systems for which the density is bounded above by some fixed constant strictly less than 1; this algorithm will run in a time proportional to the product of the number of tasks and the value of the largest period. We will then show that unless

P = NP, such a pseudo-polynomial-time algorithm cannot extend to asynchronous task systems; i.e., we show that the general feasibility problem on one processor is co-NP-complete in the strong sense even if the density is bounded above by any fixed positive constant. Finally, we give a polynomial-time algorithm for deciding the feasibility problem for complete tasks systems on one processor when the number of distinct types of tasks is fixed.

In what follows, we will develop necessary and sufficient conditions for a complete task system to be feasible. It is a well-known fact (see, e.g. [LM80]) that the density must be no more than 1 in order for a task system to be feasible on one processor. Our strategy is to show that if the density is no more than 1 and the system is not feasible, then there must be a time interval in which too much execution time is required. In other words, we can ignore specific release times and deadlines, and simply show that the total amount of execution time needed by jobs which must be scheduled entirely within the interval exceeds the amount of time available in the interval. Furthermore, we are able to show that the endpoints of this interval can be encoded in a polynomial number of bits, and that the total amount of execution time needed within the interval may be computed efficiently.

Before we formally present this result, we will introduce some notation. In what follows, T will denote a task system of n tasks such that for each task T_i , s_i is the start time, e_i is the execution time, d_i is the deadline, and p_i is the period. Let $P = \text{lcm}\{p_1, ..., p_n\}$ and $s = \text{max}\{s_1, ..., s_n\}$. For a given discrete schedule S of T, let $C_S(T, t) = (e_{1,t}, ..., e_{n,t})$, where $e_{i,t}$ is the number of times T_i is scheduled in S before t, beginning with its last request before t. Finally, given times t_1 and $t_2, 0 \le t_1 < t_2$, let $\eta_i(t_1, t_2)$ denote the total number of natural numbers k such that

- 1. $t_1 \leq s_i + kp_i$, and
- $2. \ s_i + kp_i + d_i \le t_2.$

Thus, $\eta_i(t_1, t_2)$ is the number of times T_i must be completely scheduled in $[t_1, t_2)$.

We will eventually show that if T is not schedulable, then for some "small" t_1 and t_2 , $\sum_{i=1}^n \eta_i(t_1, t_2) \cdot e_i > t_2 - t_1$. In order to avoid the need to reason about all possible schedules in showing this result, we will make use of the fact that the Deadline Algorithm is optimal for one processor [LL73, Lab74]. This fact allows us to focus our attention solely on schedules produced by the Deadline Algorithm. Furthermore, due to Theorem 2.1, we will only consider discrete schedules, which the Deadline Algorithm always produces on integer input. We now reproduce the following facts shown by Leung and Merrill [LM80].

Lemma 3.1: (From [LM80].) Let S be the schedule of T on one processor constructed by the Deadline Algorithm. Then for each task T_i and for each time $t_1 \geq s_i$, we have $e_{i,t_1} \geq e_{i,t_2}$, where $t_2 = t_1 + P$.

Lemma 3.2: (From [LM80].) Let S be the schedule of T on one processor constructed by the Deadline Algorithm. T is feasible on one processor iff

1. all deadlines in the interval $(0, t_2]$ are met in S, where $t_2 = s + 2P$, and

2. $C_S(T, t_1) = C_S(T, t_2)$, where $t_1 = s + P$.

Lemma 3.2 gives two conditions, one of which must fail if T is not feasible. We need something stronger, namely, that the first condition must fail. As we show in the following lemma, we can reach this conclusion provided the density is no more than 1. (Note also that the following lemma holds whether or not T is feasible.)

Lemma 3.3: Let S be the schedule of T on one processor constructed by the Deadline Algorithm. If $\sum_{i=1}^{n} e_i/p_i \leq 1$, then $C_S(T, t_1) = C_S(T, t_2)$, where $t_1 = s + P$ and $t_2 = s + 2P$.

Proof: Suppose $C_S(T, t_1) \neq C_S(T, t_2)$. ¿From Lemma 3.1, there must be a task T_j for which $e_{j,t_1} > e_{j,t_2}$. We first show that the schedule has no empty time slots in $[t_1, t_2)$ (i.e., S(t) is defined for $t_1 \leq t < t_2$). Suppose there is some empty slot $t_1 + \Delta$, $0 \leq \Delta < P$. This implies that no tasks are active at that time; i.e., for all $i \in \{1, ..., n\}$, $e_{i,t_1+\Delta} = e_i$, its maximum value. By Lemma 3.1, we have $C_S(T, s + \Delta) = C_S(T, t_1 + \Delta)$. Since the task requests in the intervals $[s + \Delta, t_1 + \Delta)$ and $[t_1 + \Delta, t_2 + \Delta)$ are the same $(t_1 + \Delta = s + \Delta + P)$, the schedules in these intervals are identical. This means that $C_S(T, t_1) = C_S(T, t_2)$ — a contradiction. Therefore, S has no empty time slots in $[t_1, t_2)$. Since $e_{i,t_1} \geq e_{i,t_2}$ for $1 \leq i \leq n$ (Lemma 3.1) and since $e_{j,t_1} > e_{j,t_2}$, the total execution time needed for tasks released in the interval $[t_1, t_2)$ is strictly larger than $t_2 - t_1 = P$. Therefore, $\sum_{i=1}^n (P/p_i)e_i > P$, and $\sum_{i=1}^n e_i/p_i > 1$ — a contradiction.

We are now ready to give our main lemma.

Lemma 3.4: A complete task system T is feasible on one processor iff

- 1. $\sum_{i=1}^{n} e_i/p_i \le 1$, and
- 2. $\sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot e_i \le t_2 t_1$ for all $0 \le t_1 < t_2 \le s + 2P$.

Proof: Both conditions are clearly necessary. Suppose both conditions hold, and suppose T is not feasible. Let S be a schedule of T constructed by the Deadline Algorithm. From Lemma 3.3, $C_S(T, s + P) = C_S(T, s + 2P)$. From Lemma 3.2, some deadline in (0, s + 2P] must not be met by S. Let t_2 be this deadline. Let t_1 be the last time prior to t_2 such that there is no task with deadline less than or equal to t_2 scheduled at $t_1 - 1$ in S. Since t_2 is a deadline, $t_2 > 0$, so t_1 is well-defined. Furthermore, since the deadline t_2 is not met, there is an active task at $t_2 - 1$, so some task is scheduled at $t_2 - 1$. It follows that there is a task scheduled at every time in $[t_1, t_2)$. Since all tasks scheduled in $[t_1, t_2)$ have deadlines no later than t_2 , and since no task having a deadline less than or equal to t_2 is scheduled at $t_1 - 1$, every task scheduled in $[t_1, t_2)$ must have been released no earlier than t_1 . Since there is a task scheduled at every time in $[t_1, t_2)$ and the deadline t_2 is not met, $\sum_{i=1}^n \eta_i(t_1, t_2) \cdot e_i > t_2 - t_1$ — a contradiction.

In order for the above lemma to be more useful, we will now show that $\eta_i(t_1, t_2)$ can be efficiently computed. This fact follows from the following lemma.

Lemma 3.5:

$$\eta_i(t_1, t_2) = \max \left\{ 0, \left| \frac{t_2 - s_i - d_i}{p_i} \right| - \left[\frac{t_1 - s_i}{p_i} \right] + 1 \right\}.$$

Proof: From the definition of η_i , $\eta_i(t_1, t_2)$ is the number of ks satisfying two inequalities. Solving the first inequality, $t_1 \leq s_i + kp_i$, for k, we get $k \geq (t_1 - s_i)/p_i$. Thus, the smallest natural number satisfying this inequality is $\lceil (t_1 - s_i)/p_i \rceil$. Solving the second inequality, $s_i + kp_i + d_i \leq t_2$, for k, we get $k \leq (t_2 - s_i - d_i)/p_i$. The largest natural number satisfying this inequality is $\lfloor (t_2 - s_i - d_i)/p_i \rfloor$. Therefore, the total number of ks satisfying both inequalities is

$$\max\left\{0, \left\lfloor \frac{t_2 - s_i - d_i}{p_i} \right\rfloor - \left\lceil \frac{t_1 - s_i}{p_i} \right\rceil + 1\right\}.$$

We will now apply Lemma 3.4 to synchronous systems (i.e., systems in which all start times are 0) in which the density is bounded above by a fixed constant strictly less than 1 (for example, 0.9 or 0.99). We will show that the feasibility problem for such systems can be solved in $O(n \max\{p_i - d_i\})$ time — an exponential improvement over the algorithm for general complete task systems given by Leung and Merrill [LM80]. Our result is noteworthy for at least two reasons. First, the complexity of the feasibility problem for synchronous systems is unknown. Second, such a result does not hold for asynchronous systems unless P = NP; in particular, we will show that the feasibility problem for complete task systems on one processor is co-NP-hard in the strong sense even if the density is bounded by any fixed positive constant.

Theorem 3.1: Let c be a fixed constant, 0 < c < 1. The feasibility problem for synchronous systems on one processor is solvable in $O(n \max\{p_i - d_i\})$ time if $\sum_{i=1}^n e_i/p_i \le c$.

Proof: Let T be a synchronous task system such that $\sum_{i=1}^{n} e_i/p_i \leq c$. From Lemma 3.4, T is feasible on one processor iff for all $0 \leq t_1 < t_2 \leq s + 2P$, $\sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot e_i \leq t_2 - t_1$. We will show that if there exist t_1 and t_2 such that this inequality does not hold, then t_1 may be chosen to be 0, and t_2 may be chosen to be less than

$$\frac{c}{1-c}\max\{p_i-d_i\}.$$

Since c is a fixed constant, the above value is linear in $\max\{p_i - d_i\}$. Thus, the following algorithm solves the feasibility problem in time $O(n \max\{p_i - d_i\})$:

$$\begin{array}{l} t \leftarrow 1; \\ tlim \leftarrow \frac{c}{1-c} \max\{p_i - d_i\}; \\ \mathbf{while} \ t < tlim \ \mathbf{and} \ t \geq \sum_{i=1}^n \left(\left\lfloor \frac{t-d_i}{p_i} \right\rfloor + 1 \right) e_i \ \mathbf{do} \\ t \leftarrow t+1; \\ \mathbf{if} \ t = tlim \ \mathbf{then} \\ \mathbf{return} \ feasible \\ \mathbf{else} \\ \mathbf{return} \ not \ feasible \end{array}$$

We first show that t_1 may be chosen to be 0. Suppose we have $\sum_{i=1}^n \eta_i(t_1, t_2) \cdot e_i > t_2 - t_1$. ¿From Lemma 3.5, we have that for $1 \le i \le n$,

$$\eta_{i}(0, t_{2} - t_{1}) = \max \left\{ 0, \left\lfloor \frac{t_{2} - t_{1} - d_{i}}{p_{i}} \right\rfloor + 1 \right\}$$

$$\geq \max \left\{ 0, \left\lfloor \frac{t_{2} - d_{i}}{p_{i}} \right\rfloor - \left\lceil \frac{t_{1}}{p_{i}} \right\rceil + 1 \right\}$$

$$= \eta_{i}(t_{1}, t_{2}).$$

Thus, $\sum_{i=1}^{n} \eta_i(0, t_2 - t_1) \cdot e_i > t_2 - t_1$.

We will now show that if t_1 is chosen to be 0, then t_2 cannot be too large. Suppose $\sum_{i=1}^n \eta_i(0, t_2) \cdot e_i > t_2$. Since $\lfloor (t_2 - d_i)/p_i \rfloor \geq -1$, we have

$$t_{2} < \sum_{i=1}^{n} \eta_{i}(0, t_{2}) \cdot e_{i}$$

$$= \sum_{i=1}^{n} \left(\left\lfloor \frac{t_{2} - d_{i}}{p_{i}} \right\rfloor + 1 \right) e_{i}$$

$$\leq \sum_{i=1}^{n} \frac{t_{2} - d_{i} + p_{i}}{p_{i}} e_{i}$$

$$= \sum_{i=1}^{n} \left(\frac{t_{2}e_{i}}{p_{i}} + \frac{(p_{i} - d_{i})e_{i}}{p_{i}} \right)$$

$$\leq ct_{2} + c \max\{p_{i} - d_{i}\}.$$

Solving for t_2 , we get

$$t_2 < \frac{c}{1-c} \max\{p_i - d_i\}.$$

The above proof might be a bit more intuitive if we consider a graphical interpretation. The function $g(t) = \sum_{i=1}^{n} (\lfloor (t-d_i)/p_i \rfloor + 1)e_i$ is a step function, but if we eliminate the floor operations, we get a linear function $f \geq g$. The slope of f is equal to the density, and $f(0) = \sum_{i=1}^{n} (p_i - d_i)e_i/p_i$ (see Figure 3). If the density is less than 1, f must cross the line f(t) = t. The above proof gives an upper bound on where this intersection must occur.

The algorithm given in the above proof is a pseudo-polynomial-time algorithm, since the value of $\max\{p_i\}$ is not necessarily polynomial in the size of the problem description. At this time, we do not know whether the above problem can be solved in polynomial time, nor whether the feasibility problem for general synchronous systems can be solved in pseudo-polynomial time. However, we can address the open question (see [LM80]) of whether the feasibility problem for asynchronous systems can be solved in pseudo-polynomial time. Leung and Merrill [LM80] have shown this problem to be co-NP-hard, but only in the weak sense. In what follows, we sharpen this result by showing the problem to be co-NP-complete in the strong sense. Thus, there is no pseudo-polynomial-time algorithm for this problem unless P = NP. Furthermore, we show that the problem remains co-NP-complete in the strong sense even if the density is bounded above by any fixed positive constant.

In the proof that the feasibility problem is co-NP-hard, Leung and Merrill used a reduction from the complement of the Simultaneous Congruences Problem (SCP). This problem was shown to be NP-complete in the weak sense by Leung and Whitehead [LW82]. It was left as an open question whether SCP was NP-complete in the strong sense [LM80, LW82]. In what follows, we answer this question in the affirmative. It then follows from the proof of Leung and Merrill that the feasibility problem is co-NP-hard in the strong sense. Finally, Lemma 3.4 immediately implies that the problem is co-NP-complete.

We will now introduce the Simultaneous Congruences Problem. Let $A = \{(a_1, b_1), ..., (a_n, b_n)\} \subseteq N \times N^+$ and $2 \le k \le n$ be given. The Simultaneous Congruences Problem is to determine whether there is a subset $A' \subseteq A$ of k pairs and a natural number x such that for every $(a_i, b_i) \in A'$, $x \equiv a_i \pmod{b_i}$. In what follows, we show this problem to be NP-complete in the strong sense. Our proof uses the Generalized Chinese Remainder Theorem (see, e.g., [Knu81]), which we now reproduce.

Lemma 3.6: (The Generalized Chinese Remainder Theorem.) Let $A = \{(a_1, b_1), ..., (a_k, b_k)\} \subseteq N \times N^+$. There is an x such that for all $(a_i, b_i) \in A$, $x \equiv a_i \pmod{b_i}$ iff for all $1 \le i < j \le k$, $a_i \equiv a_j \pmod{\gcd(b_i, b_j)}$.

Thus, we can conclude that if each pair of distinct pairs (a_i, b_i) and (a_j, b_j) "collides" (i.e., there is an x such that $x \equiv a_i \pmod{b_i}$ and $x \equiv a_j \pmod{b_j}$, then there is a simultaneous collision of all pairs. This fact is very useful in the construction that follows.

The proof by Leung and Whitehead [LW82] that SCP is NP-hard (in the weak sense) consisted of a reduction from the CLIQUE problem [Kar72]. Given an undirected graph G = (V, E) such that $V = \{v_1, ..., v_n\}$, a set of pairs $\{(a_1, b_1), ..., (a_n, b_n)\}$ was constructed such that $a_i \equiv a_j \pmod{\gcd(b_i, b_j)}$ iff $(v_i, v_j) \in E$. Thus, there is a simultaneous collision of k items iff G has a clique of size k. However, since each a_i and b_i were the product of $O(n^2)$ distinct prime numbers, the values of a_i and b_i were not polynomial in the size of the description of G. In order to overcome this problem, we give an entirely different reduction, from 3-SAT rather than CLIQUE.

Theorem 3.2: SCP is NP-complete in the strong sense.

Proof: Leung and Whitehead [LW82] have shown SCP to be in NP, so we need only show NP-hardness. The proof is via a polynomial-time reduction from 3-SAT [Coo71]. Let C be an instance of 3-SAT over variables x_1, \ldots, x_n ; without loss of generality let $C = \bigwedge_{j=1}^m C_j$, $C_j = \bigvee_{k=1}^3 c_{jk}$, where each c_{jk} is a distinct literal — i.e., either a variable or the negation of a variable. Also without loss of generality, we assume that no clause contains both a variable and its negation.

The intuitive idea of our construction is as follows. We will construct an instance of SCP such that a collision of m+n pairs will give a satisfying assignment for C. For each variable x_i , we will construct a pair (a_{x_i}, b_{x_i}) corresponding to an assignment of true to x_i , and a pair $(a_{\bar{x}_i}, b_{\bar{x}_i})$ corresponding to an assignment of false to x_i ; these two pairs will be constructed so that they do not collide. On the other hand, we will force pairs corresponding to two different variables to collide with each other. Thus, from Lemma 3.6, any simultaneous collision of n of these pairs will correspond to some assignment to all n variables. Now for each clause $c_{j1} \vee c_{j2} \vee c_{j3}$, we will construct three pairs (a_{jk}, b_{jk}) , $1 \leq k \leq 3$, one corresponding to each of the

three literals in the clause. (a_{jk}, b_{jk}) will be constructed to collide with all other pairs except those pairs corresponding to literals within the same clause and the pair $(a_{\bar{c}_{jk}}, b_{\bar{c}_{jk}})$. Thus, a collision of m + n pairs will occur iff there is an assignment that makes at least one literal in each clause true.

More formally, let $p_1, ..., p_{m+n}$ denote the first m+n primes greater than 2. ¿From the Prime Number Theorem (see, e.g., [Knu81]), p_{m+n} is bounded by a polynomial in m+n. Now for each $i, 1 \le i \le n$, let

- $\bullet \ a_{x_i} = p_i;$
- $a_{\bar{x}_i} = p_i 1$; and
- $\bullet \ b_{x_i} = b_{\bar{x}_i} = p_i.$

Note that since each $p_i > 2$, $a_{\bar{x}_i} < a_{x_i} < a_{\bar{x}_{i+1}}$ for $1 \le i < n$. Now for each $j, 1 \le j \le m$ and each $k, 1 \le k \le 3$, let

- $a_{jk} = a_{c_{jk}}$ (defined above); and
- $\bullet \ b_{jk} = b_{c_{jk}} p_{n+j}.$

Clearly, the values of all of the integers defined above are bounded by a polynomial in m + n. We will now show that there is a collision of m + n pairs iff C is satisfiable.

 \Rightarrow : Suppose there is a collision of m+n pairs at some y. For any variable x_i , $\gcd(b_{x_i},b_{\bar{x}_i})=p_i$. Since $p_i>2$, $p_i\not\equiv p_i-1\pmod{p_i}$. Therefore, from Lemma 3.6, (a_{x_i},b_{x_i}) and $(a_{\bar{x}_i},b_{\bar{x}_i})$ do not collide; hence, there is at most one pair corresponding to each variable in the collision at y. For any literal c_{jk} , if $k'\neq k$, $\gcd(b_{jk},b_{jk'})=p_{n+j}$ since c_{jk} and $c_{jk'}$ are distinct and noncontradictory. Since $a_{c_{jk}}< p_{n+j}$, $a_{c_{jk'}}< p_{n+j}$, and $a_{c_{jk}}\neq a_{c_{jk'}}$ for all j,k,k', two pairs corresponding to literals in the same clause do no collide. Thus, there is at most one pair corresponding to each clause in the collision at y. Since there are m+n pairs in the collision, there are exactly n pairs corresponding to variables, and m pairs corresponding to clauses. Consider the assignment in which x_i is true iff $y\equiv a_{x_i}\pmod{b_{x_i}}$, for $1\leq i\leq n$. Let c_{jk} be some literal such that $y\equiv a_{jk}\pmod{b_{jk}}$. (Note that there is exactly one such literal in each clause.) Since $a_{jk}=a_{c_{jk}}$ and b_{jk} is a multiple of $b_{c_{jk}}$, $y\equiv a_{c_{jk}}\pmod{b_{c_{jk}}}$, so c_{jk} is true in this assignment. Hence, C is satisfiable.

 \Leftarrow : Suppose C is satisfiable, and consider some satisfying assignment. Let $A' = \{(a_{x_i}, b_{x_i}) \mid x_i = true\} \cup \{(a_{\bar{x}_i}, b_{\bar{x}_i}) \mid x_i = false\} \cup \{(a_{jk}, b_{jk}) \mid c_{jk} = true \text{ and for all } k' < k, c_{jk'} = false\}$. It is easily seen by inspection that any two pairs in A' collide. Therefore, from Lemma 3.6, there is a simultaneous collision of the m+n pairs in A'.

Leung and Merrill [LM80] have given a reduction from SCP to the complement of the feasibility problem that causes only a polynomial increase in the values of the integers involved; it therefore follows from Theorem 3.2 that the feasibility problem is co-NP-hard in the strong sense. For completeness, we now reproduce their reduction.

Lemma 3.7: The feasibility problem for complete task systems on one processor is co-NP-hard in the strong sense.

Proof: Let $\{(a_1, b_1), ..., (a_n, b_n)\}$ and k represent an instance of SCP. Leung and Merrill [LM80] have given the following polynomial-time reduction to the complement of the feasibility problem.² For $1 \le i \le n$, let

- $s_i = (k-1)a_i$;
- $e_i = 1$;
- $d_i = k 1$; and
- $p_i = (k-1)b_i$.

Leung and Merrill [LM80] have shown that this task system is feasible iff there is no simultaneous collision of k pairs from the instance of SCP. Furthermore, all of the values in the task system are bounded by a polynomial in the values in the instance of SCP.

The above lemma implies that there is no pseudo-polynomial-time algorithm for the general feasibility problem unless P = NP. In fact, we can show the following corollary, which implies that unless P = NP, Theorem 3.1 does not extend to asynchronous systems.

Corollary 3.1: The feasibility problem for complete task systems on one processor is co-NP-hard in the strong sense even if the systems are restricted to have density not greater than ϵ , where ϵ is any fixed positive constant.

Proof: Consider the construction in the proof of Lemma 3.7. If we multiply all of the start times and periods in this construction by the same positive integer, the proof still holds. In particular, if we multiply all of the start times and periods by some positive integer $c \ge n/(\epsilon(k-1)\min\{b_i\})$, then $\sum_{i=1}^n e_i/p_i \le \epsilon$. \square

As a matter of theoretical interest, the following theorem now follows immediately from Lemmas 3.4, 3.5, and 3.7, thus answering another open question posed by Leung and Merrill [LM80].

Theorem 3.3: The feasibility problem for complete periodic task systems on one processor is co-NP-complete in the strong sense.

We now use Lemma 3.4 to show one last result concerning synchronous systems.

Theorem 3.4: Let T be a synchronous task system with $\min\{p_i - d_i\} > P(1 - \sum_{i=1}^n e_i/p_i)$, where P is the least common multiple of the periods. Then T is not feasible on one processor.

Proof: We will show that $\sum_{i=1}^{n} \eta_i(0, P - \min\{p_i - d_i\}) \cdot e_i > P - \min\{p_i - d_i\}$; it will then follow from Lemma 3.4 that the system is not feasible on one processor. We first note that since $\min\{p_i - d_i\}$

²This reduction is actually slightly different from the one by Leung and Merrill, but the basic idea is the same.

$$P(1 - \sum_{i=1}^{n} e_i/p_i)$$
, we have $P \sum_{i=1}^{n} e_i/p_i > P - \min\{p_i - d_i\}$. Now

$$\begin{split} \sum_{i=1}^n \eta_i(0,P-\min\{p_i-d_i\}) \cdot e_i &= \sum_{i=1}^n \left(\left\lfloor \frac{P-\min\{p_i-d_i\}-d_i}{p_i} \right\rfloor + 1 \right) e_i \\ &\geq \sum_{i=1}^n \left(\left\lfloor \frac{P-p_i}{p_i} \right\rfloor + 1 \right) e_i \\ &= \sum_{i=1}^n \frac{P}{p_i} e_i \\ &= P \sum_{i=1}^n \frac{e_i}{p_i} \\ &> P - \min\{p_i-d_i\}. \end{split}$$

A special case of the above theorem occurs when the density is exactly 1 and no deadline is equal to its corresponding period. We also note that Theorem 3.4 does not hold in general for asynchronous systems, even in this special case (the system in which $s_1 = 0$, $s_2 = 1$, $e_1 = e_2 = d_1 = d_2 = 1$, and $p_1 = p_2 = 2$ is a counterexample).

Lemma 3.4 yields one final result. This result concerns complete task systems with a fixed number of distinct types of tasks; i.e., we are given m tuples, (s_i, e_i, d_i, p_i) , and m integers $n_1, ..., n_m$, where m is a fixed constant, and we wish to determine whether the task system having n_i tasks described by (s_i, e_i, d_i, p_i) , $1 \le i \le n$, is feasible. This problem can be decided in polynomial time.

Theorem 3.5: For complete task systems with a fixed number of distinct types of tasks, the feasibility problem for one processor can be solved in polynomial time.

Proof: Suppose we have a task system T with m distinct types of tasks as described above. We first decide in polynomial time whether $\sum_{i=1}^{m} (n_i e_i)/p_i \leq 1$. If so, we construct the following system of 2m+1 linear inequalities in 2m+2 unknowns:

- $p_i k_i + s_i > t_1$ for 1 < i < m
- $p_i k_i + p_i k'_i + s_i + d_i < t_2$ for 1 < i < m
- $\sum_{i=1}^{m} n_i e_i k_i' > t_2 t_1$

The variables in the above systems are t_1, t_2, k_i , and k'_i , for $1 \le i \le m$. Since m is a fixed constant, we have a fixed number of inequalities in a fixed number of unknowns; we can therefore determine, in polynomial time, whether there exists a nonnegative integer solution to this system [Len83]. In any nonnegative integer solution, k'_i gives a lower bound on the number of times each of the tasks of type i must be executed completely in $[t_1, t_2)$. Thus, if there is a solution, the last inequality guarantees that T is not feasible. \Box

4 Incomplete Task Systems

In this section, we extend our results of the last section to incomplete task systems. We first show that the feasibility problem for incomplete task systems on one processor is Σ_2^P -complete. We then give a pseudo-polynomial time algorithm for solving this problem with a fixed number of distinct types of tasks.

Recall that an incomplete task system is feasible iff there is a choice of start times such that the resulting complete task system is feasible. Since the feasibility problem for complete task systems on one processor is co-NP-complete, it follows that if it suffices to consider only integer start times that can be written in a polynomial number of bits, then the feasibility problem for incomplete task systems is in Σ_2^P . To this end, we give the following lemma.

Lemma 4.1: Let T be an incomplete task system. If T is feasible on one processor, then there exist integer start times $s_1, ..., s_n$, where $s_i < p_i$ for $1 \le i \le n$, such that the resulting complete task system is feasible.

Proof: Suppose T is feasible and from Lemma 2.1, let $s'_1, ..., s'_n$ be integer start times for which the resulting task system T' is feasible. Let S' be a valid schedule for T', and let $s' = \max\{s'_1, ..., s'_n\}$. We construct new start times $s_1, ..., s_n$ and a valid schedule for the resulting complete task system as follows. First, let r_i be the first release of T_i no earlier than s', then let $s_i = r_i - s'$. Since $s' \le r_i < s' + p_i$, $0 \le s_i < p_i$. Finally, let S(t) = S'(t + s'). Since S' is valid, S is clearly valid.

¿From the above lemma, we can conclude that the feasibility problem for incomplete task systems on one processor is in Σ_2^P . In order to show the problem to be Σ_2^P -complete, we will give a reduction from a periodic maintenance problem shown to be Σ_2^P -complete in [BRTV89]. The periodic maintenance scheduling problem is a restriction of the feasibility problem for incomplete task systems on m processors, where m is given as input, such that for each task T_i , $e_i = d_i = 1$. This problem was shown to be Σ_2^P -complete in [BRTV89]. Our next theorem gives a nice contrast to the complexity of the periodic maintenance scheduling problem, which is NP-complete when restricted to one processor [BRTV89].

Theorem 4.1: The feasibility problem for incomplete task systems on one processor is Σ_2^P -complete.

Proof: It follows from Lemma 4.1 and Theorem 3.3 that the problem is in Σ_2^P . In order to show the lower bound, we give a reduction from the periodic maintenance scheduling problem. Let T be an incomplete task system such that for $1 \le i \le n$, $e_i = d_i = 1$, and let m be an arbitrary positive integer. We will construct an incomplete task system T' such that T' is feasible on one processor iff T is feasible on m processors. The idea is to make m time units for T' correspond to one time unit for T. In order to force proper alignment of the start times for T', we will construct one task that must be scheduled at all times t in the range $2am \le t < (2a+1)m$ for all $a \in N$. For each $a \in N$, the m time slots t such that $(2a+1)m \le t < 2(a+1)m$ will correspond to the m processors available at time a for T. We will then construct a task T'_i to correspond with each task T_i . More formally, let

• $p'_i = 2mp_i \text{ for } 1 \le i \le n;$

- $p'_{n+1} = 2m$;
- $d'_i = m \text{ for } 1 \le i \le n+1;$
- $e'_i = 1 \text{ for } 1 \le i \le n;$
- $e'_{n+1} = m$.

We claim that T' is feasible on one processor iff T is feasible on m processors.

 \Rightarrow : Suppose there exist start times $s'_1, ..., s'_{n+1}$ for which T' is feasible. Using a similar strategy as in Lemma 4.1, we can assume without loss of generality that $s'_{n+1} = 0$ and $0 \le s'_i < p'_i$ for $1 \le i \le n$. For any time t in any valid schedule S' of T', S'(t) = n + 1 iff $t \mod 2m < m$. Thus, it is easily seen that if some task T'_i , $1 \le i \le n$, is released within the interval [2am, 2(a+1)m), it must finish execution no later than 2(a+1)m. For $1 \le i \le n$, let $s_i = \lfloor s'_i/(2m) \rfloor$. Since $e_i = d_i = 1$ for $1 \le i \le n$, T is feasible on m processors for these start times iff no more than m tasks are ever simultaneously released. Suppose that at some time t, at least m+1 task are simultaneously released; that is, there are at least m+1 k_i s such that $s_i + k_i p_i = t$. Multiplying both sides of this equation by 2m and substituting $\lfloor s'_i/(2m) \rfloor$ for s_i , we get $s'_i - (s'_i \mod 2m) + 2mk_i p_i = 2mt$. Thus, in S', we have at least m+1 tasks besides T'_{n+1} released within the interval $\lfloor 2mt, 2m(t+1) \rfloor$. Since all of these tasks must finish execution no later than 2m(t+1), S' must fit 2m+1 units of execution time into 2m time units — a contradiction.

 \Leftarrow : Suppose S is a valid schedule for T on m processors with start times $s_1, ..., s_n$. For $1 \le i \le n$, let $s'_i = 2ms_i + m$, and let $s'_{n+1} = 0$. It is easily seen that the complete task system resulting from these start times for T' is feasible.

We now show that for a fixed number of distinct types of systems, the feasibility problem for incomplete task systems on one processor is solvable in pseudo-polynomial time. At this time, we do not know whether the problem is NP-hard.

Theorem 4.2: For a fixed number of distinct types of tasks, the feasibility problem for incomplete task systems on one processor can be solved in pseudo-polynomial time.

Proof: Let T be an incomplete task system with m distinct types of tasks. From Lemma 4.1, the total number of possible sets of start times that need to be checked is at most $\prod_{i=1}^{m} p_i$. Since m is a fixed constant, this number is polynomial in the values of the periods. We can therefore solve the problem by applying the algorithm of Theorem 3.5 for each choice of start times such that for all $i, s_i < p_i$.

5 Conclusion

In Section 2, we showed that for any (complete or incomplete) task system T, there is a valid discrete schedule for T on one processor iff there is a valid continuous schedule for T on one processor. Although this result extends to complete task systems on several identical processors, we have left open the question

of whether it extends to incomplete task systems on several processors. We might also mention that we have said nothing concerning the existence of algorithms for finding discrete schedules. For example, it might be possible to decide very quickly, for some particular type of periodic task system, whether a system is feasible, but at the same time not be able to efficiently generate a valid discrete schedule. In this case, even an efficient algorithm for generating a continuous schedule would not necessarily lead to an efficient algorithm for generating a discrete schedule.

Although we have made the claim that using integer inputs constitutes a better abstraction that using noninteger inputs, there may still be some interest in how our results might extend to noninteger inputs. The most natural extension would be to the rational numbers. (Irrational inputs raise a number of questions involving the computational model; we would rather avoid these issues here.) As we have already suggested in Section 2, all polynomial-time, co-NP, and Σ_2^P results can be extended to rational numbers by multiplying all inputs by a common denominator; however, when such an extension is made to a pseudo-polynomial-time algorithm, the resulting algorithm is potentially exponential, since the value of the least common multiple of n integers is potentially exponential in n. To avoid this problem in extending Theorem 3.1, we must first modify the algorithm to compute the summation for only those times t at which a deadline occurs. The resulting algorithm not only works for noninteger inputs, it also operates in time $O(n^2 \max\{p_i\}/\min\{p_i\})$, which is slightly better than the original algorithm on integer inputs, as long as every period is at least n. On the other hand, we do not know at this time whether Theorem 4.2 extends to rational inputs.

A better understanding of synchronous systems seems to be quite desirable. For example, Baruah, Mok, and Rosier [BMR90] have recently used our techniques from Theorem 3.1 to attack the feasibility problem for sporadic task systems. Based on these results, it seems quite likely that any algorithm for synchronous systems can be extended to sporadic systems. We have left several questions open regarding synchronous systems. On the one hand, we have given no lower bounds at all for synchronous systems; thus, we do not know whether the pseudo-polynomial time algorithm of Theorem 3.1 can be improved upon. On the other hand, we do not give any upper bound, aside from the co-NP upper bound of Theorem 3.3, for general synchronous systems. The techniques given in this paper do not appear to extend to these questions.

Another natural question to consider next is whether our main lemma extends to the multiprocessor case; i.e., are the conditions

1.
$$\sum_{i=1}^{n} e_i/p_i \le k$$
, and

2.
$$\sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot e_i \le k(t_2 - t_1)$$
 for all $0 \le t_1 < t_2 \le s + 2P$

sufficient to conclude that the complete task system T is feasible on k processors? Unfortunately, these conditions are easily seen to be insufficient. Consider, for example, the following task system T on two processors:

	e_i	d_i	p_i	s_i
T_1	1	1	2	0
T_2	1	1	3	0
T_3	2	3	4	0
T_4	2	3	4	2

It is easily seen by inspection that conditions 1 and 2 above both hold. Clearly, in any schedule, T_1 must be scheduled at 0, 2, 4, and 6, and T_2 must be scheduled at 0, 3, and 6. Since both processors are occupied at times 0 and 6, T_3 must then be scheduled at 1, 2, 4, and 5. Since both processors are now occupied at times 2 and 4, T_4 cannot be scheduled twice before its deadline. It would appear that new insights are needed to find useful necessary and sufficient conditions for feasibility of task systems on more than one processor.

References

- [Bla87] J. Blazewicz. Selected topics in scheduling theory. Annals of Discrete Mathematics, 31:1–60, 1987.
- [BMR90] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. To be presented at the 11th Real-Time Systems Symposium, Orlando, Florida, December 1990.
- [BRTV89] S. Baruah, L. Rosier, I. Tulchinsky, and D. Varvel. The complexity of periodic maintenance. Submitted for publication, 1989.
- [C⁺76] E. Coffman, Jr. et al. Computer and Job-Shop Scheduling Theory. Wiley, 1976.
- [Coo71] S. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Ann. ACM Symp. on Theory of Computing*, pages 151–158, 1971.
- [FF62] L. Ford and D. Fulkerson. Flows in Networks. Princeton University Press, Princeton, NJ, 1962.
- [Hor74] W. Horn. Some simple scheduling algorithms. Naval Research Logistics Quarterly, 21:177–185, 1974.
- [Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, Complexity of Computer Computations, pages 85–103. Plenum Press, 1972.
- [Knu81] D. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Programming. Addison Wesley, second edition, 1981.
- [Lab74] J. Labetoulle. Some theorems on real time scheduling. In E. Gelenbe and R. Mahl, editors, Computer Architecture and Networks, pages 285–293. North-Holland, 1974.
- [Len83] H. Lenstra, Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

- [Leu89] J. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.
- [LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20:46–61, 1973.
- [LM80] J. Leung and M. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11:115–118, 1980.
- [LM81] E. Lawler and C. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12:9–12, 1981.
- [LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [Mok89] A. Mok, 1989. Personal communication.