Scheduling Real-time Tasks: Algorithms and Complexity*

Sanjoy Baruah
The University of North Carolina at Chapel Hill
Email: baruah@cs.unc.edu

Joël Goossens
Université Libre de Bruxelles
Email: joel.goossens@ulb.ac.be

*Supported in part by the National Science Foundation (Grant Nos. CCR-9988327, ITR-0082866, and CCR-0204312).

Contents

1 Introduction

The use of computers to control safety-critical real-time functions has increased rapidly over the past few years. As a consequence, **real-time systems** — computer systems where the correctness of a computation is dependent on both the logical results of the computation *and* the time at which these results are produced — have become the focus of much study.

Since the concept of **"time"** is of such importance in real-time application systems, and since these systems typically involve the sharing of one or more resources among various contending processes, the concept of **scheduling** is integral to real-time system design and analysis. Scheduling theory as it pertains to a finite set of requests for resources is a well-researched topic. However, requests in real-time environments are often of a recurring nature. Such systems are typically modelled as finite collections of simple, highly repetitive **tasks**, each of which generates **jobs** in a very predictable manner. These jobs have upper bounds upon their worst-case execution requirements, and associated deadlines. For example, in a *periodic* task system [22, 20, 17, 21, 24, 4] each task makes a resource request at regular periodic intervals. The processing time and the time elapsed between the request and the deadline are always the same for each request of a particular task; they may however be different for different tasks. A *sporadic* task system [24, 18, 11, 30, 14, 31, 5] is similar, except that the request times are not known beforehand; thus, sporadic tasks may be used to model event-driven systems. Real-time scheduling theory has traditionally focused upon the development of algorithms for *feasibility analysis* (determining whether all jobs can complete execution by their deadlines) and *run-time scheduling* (generating schedules at run-time for systems that are deemed to be feasible) of such systems.

2 Definitions and Terminology

In hard-real-time systems, there are certain basic units of work known as **jobs**, which need to be executed by the system. Each such job has an associated **deadline**, and it is imperative for the correctness of the system that all such jobs complete by their deadlines. In this survey article, we restrict our attention to a *preemptive* model of scheduling – a job executing on the processor may be interrupted, and its execution resumed at a later point in time. There is no penalty associated with such preemption. For results concerning real-time scheduling in *non*-preemptive environments, see, for example, [13, 14, 12].

For our purposes, each real-time job is characterized by three parameters — a *release time*, an

execution requirement, and a *deadline* — with the interpretation that the job needs to be executed for an amount equal to its execution requirement between its release time and its deadline. We will assume throughout this chapter that *all job execution requirements are normalized to processor computing capacity*, i.e., that job execution-requirements are expressed in units that result in the processor's computing capacity being one unit of work per unit time.

Definition 1 (Jobs and **Instances)** A real-time **job** $j = (a, e, d)$ is characterized by three parameters – an *arrival time* a, an *execution requirement* e, and a *deadline* d, with the interpretation that this job must receive e units of execution over the interval $[a, d)$. A real-time **instance** J is a finite or infinite collection of jobs: $J = \{j_1, j_2, \ldots, \}$.

■

Real-time instances may be generated by collections of periodic or sporadic tasks (see Section 2.1 below).

Definition 2 (Schedule) For any collection of jobs J, a (uniprocessor) *schedule* **S** is a mapping from the Cartesian product of the real numbers and the collection of jobs to $\{0, 1\}$:

$$S \ : \ \mathbb{R} \times J \longrightarrow \{0, 1\} \ ,$$

with $S(t, j)$ equal to one if schedule S assigns the processor to job j at time-instant t, and zero otherwise[1].

Definition 3 (Active job) A job $j = (a, e, d)$ in J is defined to be *active* in some schedule S of J at time instant t if

1. $a \leq t$;
2. $t \leq d$; and
3. $\left(\int_{t'=a}^{t} S(j, t) \right) < e$

That is, an active job is one that has arrived, has not yet executed for an amount equal to its execution requirement, and has not yet had its deadline elapse.

[1]Hence, for all t, there is at most one $j \in J$ for which $S(t, j) = 1$.

2.1 Periodic and Sporadic Tasks

In the **periodic** model of real-time tasks, a task T_i is completely characterized by a 4-tuple (a_i, e_i, d_i, p_i), where

- the **offset** a_i denotes the instant at which the first job generated by this task becomes available for execution;
- the **execution requirement** e_i specifies an upper limit on the execution requirement of each job generated by this task;
- the **relative deadline** d_i denotes the temporal separation between each job's arrival time and deadline – a job generated by this task arriving at time-instant t has a deadline at time-instant $(t + d_i)$; and
- the **period** p_i denotes the temporal separation between the arrival times of successive jobs generated by the task.

That is, $T_i = (a_i, e_i, d_i, p_i)$ generates an infinite succession of jobs, each with execution-requirement e_i, at each instant $(a_i + k \cdot p_i)$ for all integer $k \geq 0$, and the job generated at instant $(a_i + k \cdot p_i)$ has a deadline at instant $(a_i + k \cdot p_i + d_i)$.

Sporadic tasks are similar to periodic tasks, except that the parameter p_i denotes the *minimum*, rather than exact, separation between successive jobs of the same task. A sporadic task is usually characterized by three parameters rather than four: $T_i = (e_i, d_i, p_i)$ with the interpretation that T_i generates an infinite succession of jobs each with an execution requirement equal to e_i and a deadline d_i time-units after its arrival time, and with the arrival-instants of successive jobs being separated by at least p_i time units.

A **periodic task system** is comprised of a finite collection of periodic tasks, while a **sporadic task system** is comprised of a finite collection of sporadic tasks.

The **utilization** $U(T_i)$ of a periodic or sporadic task T_i is defined to be the ratio of its execution requirement to its period: $U(T_i) \stackrel{\text{def}}{=} e_i/p_i$. The utilization $U(\tau)$ of a periodic, or sporadic task system τ is defined to be the sum of the utilizations of all tasks in τ: $U(\tau) \stackrel{\text{def}}{=} \sum_{T_i \in \tau} U(T_i)$.

Special kinds of task systems. Real-time researchers have often found it convenient to study periodic and sporadic task models that are more restrictive than the general models described above. Some of these restricted models include

Implicit-deadline task systems. Periodic and sporadic task systems in which every task has its deadline parameter equal to its period (i.e., $d_i = p_i$ for all tasks T_i).

Constrained-deadline task systems. Periodic and sporadic task systems in which every task has its deadline parameter no larger than its period (i.e., $d_i \leq p_i$ for all tasks T_i).

Synchronous task systems. Periodic task systems in which the offset of all tasks is equal (i.e., $a_i = a_j$ for all tasks T_i, T_j; without loss of generality, a_i can then be considered equal to zero for all T_i). Tasks comprising a synchronous period task system are typically specified by three parameters rather than four – the offset parameter is left unspecified, and assumed equal to zero.

2.2 Static- and Dynamic-priority Algorithms

Most uniprocessor scheduling algorithms operate as follows: at each instant, each active job is assigned a distinct priority, and the scheduling algorithm chooses for execution the currently active job with the highest priority.

Some scheduling algorithms permit that periodic/ sporadic tasks T_i and T_j both have active jobs at times t and t' such that at time t, T_i's job has higher priority than T_j's job while at time t', T_j's job has higher priority than T_i's. Algorithms that permit such "switching" of priorities between tasks are known as **dynamic** priority algorithms. An example of of dynamic priority scheduling algorithm is the earliest deadline first scheduling algorithm [22, 8] (EDF).

By contrast, **static** priority algorithms satisfy the property that for every pair of tasks T_i and T_j, whenever T_i and T_j both have active jobs, it is always the case that the same task's job has higher priority. An example of a static-priority scheduling algorithm for periodic scheduling is the rate-monotonic scheduling algorithm [22].

2.3 Real-time Scheduling Problems

As stated in Section 1, real-time scheduling theory is primarily concerned with obtaining solutions to the following two problems:

The feasibility analysis problem: *Given* the specifications of a task system, and constraints on the scheduling environment (e.g., whether dynamic-priority algorithms are acceptable or

static-priority algorithms are required; whether preemption is permitted; etc.), *determine* whether there exists a schedule for the task system that will meet all deadlines.

Any periodic task system generates exactly one collection of jobs, and the feasibility problem is concerned with determining whether this collection of jobs can be scheduled to meet all deadlines. Since the sporadic task model constrains the *minimum*, rather than the exact, inter-arrival separation between successive jobs of the same task, a sporadic task system, on the other hand, is legally permitted to generate infinitely many distinct collections of real-time jobs. The feasibility-analysis question for a sporadic task system is thus as follows: given the specifications of a sporadic task system determine whether all legal collections of jobs that could be generated by this task system can be scheduled to meet all deadlines.

The run-time scheduling problem: *Given* a task system (and associated environmental constraints) that is known to be feasible, *determine* a scheduling algorithm that schedules the system to meet all deadlines.

For sporadic task systems, the requirement once again is that all deadlines be met for all possible collections of jobs that could legally be generated by the sporadic task system under consideration.

3 Dynamic-priority Scheduling

Below, we first summarize the results concerning dynamic-priority scheduling of periodic and sporadic task systems. Then in Sections 3.1–3.6, we delve into further details about these results.

Dynamic-priority scheduling algorithms place no restrictions upon the manner in which priorities are assigned to individual jobs. Within the context of preemptive uniprocessor scheduling, it has been shown [22, 8] that the earliest deadline first scheduling algorithm (EDF), which at each instant in time chooses for execution the currently-active job with the smallest deadline (with ties broken arbitrarily), is an *optimal* scheduling algorithm for scheduling arbitrary collections of independent real-time jobs in the following sense: If it is possible to preemptively schedule a given collection of independent jobs such that all the jobs meet their deadlines, then the EDF-generated schedule for this collection of jobs will meet all deadlines as well.

Observe that EDF is a dynamic-priority scheduling algorithm. As a consequence of the optimality of EDF for preemptive uniprocessor scheduling, the run-time scheduling problem for preemptive uniprocessor dynamic-priority scheduling is essentially solved (the absence of additional constraints)

— EDF is the algorithm of choice, since any feasible task system is guaranteed to be successfully scheduled using EDF.

The feasibility analysis problem, however, turns out to be somewhat less straightforward. Specifically,

- Determining whether an arbitrary periodic task system τ is feasible has been shown to be intractable — co-NP-complete in the strong sense. This intractability result holds even if the utilization $U(\tau)$ of the task system τ is known to be bounded from above by an arbitrarily small constant.

- An exponential-time feasibility test is known, which consists essentially of simulating the behavior of EDF upon the periodic task system for a sufficiently long interval.

- The special case of *implicit-deadline systems* (recall that these are periodic or sporadic task systems in which $(d_i = p_i)$ for all tasks T_i) is however tractable: a necessary and sufficient condition for any implicit-deadline system τ to be feasible upon a unit-capacity processor is that $U(\tau) \leq 1$.

- The special case of *synchronous systems* is also not quite as difficult as the general problem. The computational complexity of the feasibility-analysis problem for synchronous systems is, to our knowledge, still open; for synchronous periodic task systems τ with $U(\tau)$ bounded from above by a constant strictly less than one, however, a pseudo-polynomial time feasibility-analysis algorithm is known.

- For sporadic task systems, it turns out that system τ is feasible if and only if the synchronous periodic task system τ' comprised of exactly as many tasks as τ is, and with each task in τ' having execution-requirement, deadline, and period exactly the same as those of the corresponding task in τ, is feasible. Thus, the feasibility-analysis problem for sporadic task systems turns out to be equivalent to the feasibility-analysis problem for synchronous periodic task systems. That is, the computational complexity of the general problem remains open; for sporadic task systems τ that have $U(\tau)$ bounded from above by a constant strictly less than one, a pseudo-polynomial time feasibility-analysis algorithm is known.

3.1 The Optimality of EDF

It has been shown [22, 8] that the earliest deadline first scheduling algorithm is an optimal preemptive uniprocessor run-time scheduling algorithm not just for scheduling periodic and sporadic task systems, but rather for scheduling arbitrary real-time instances (recall from Definition 1 that a real-time instance J is a finite or infinite collection of jobs: $J = \{j_1, j_2, \ldots, \}$).

Definition 4 (EDF) The **earliest-deadline first** scheduling algorithm (EDF) is defined as follows: At each time-instant t schedule the job j active at time-instant t whose deadline parameter is the smallest (ties broken arbitrarily).

■

In the remainder of this chapter, let EDF.J denote the schedule generated by EDF upon a real-time collection of jobs J. The following theorem is the formal statement of the optimality of EDF from the perspective of meeting deadlines.

Theorem 1 If a real-time instance J can be scheduled to meet all deadlines upon a preemptive uniprocessor, then EDF.J meets all deadlines of J upon the preemptive uniprocessor.

Proof Sketch: This result is easily proved by induction. Let Δ denote an arbitrarily small positive number. Consider a schedule S for J in which all deadlines are met, and let $[t_o, t_o + \Delta)$ denote the first time interval over which this schedule makes a scheduling decision different from the one made by EDF.J. Suppose that job $j_1 = (a_1, e_1, d_1)$ is scheduled in S over this interval, while job $j_2 = (a_2, e_2, d_2)$ is scheduled in EDF.J. Since S meets all deadlines, it is the case that S schedules j_2 to completion by time-instant d_2. In particular, this implies that S schedules j_2 for an amount equal to Δ prior to d_2. But by definition of the EDF scheduling discipline, $d_2 \leq d_1$; hence, S schedules j_2 for an amount equal to Δ prior to d_1 as well. Now the new schedule S' obtained from S by swapping the executions of j_1 and j_2 of length Δ each would agree with EDF.J over $[0, t + \Delta)$. The proof of optimality of EDF.J now follows by induction on time, with S' playing the role of S in the above argument. ■

3.2 The Intractability of Feasibility-analysis

Our next result concerns the intractability of feasibility-analysis for arbitrary periodic task systems. To obtain this result, we will reduce the *Simultaneous Congruences Problem* (SCP), which has been

shown [21, 4] to be NP-complete in the strong sense, to the feasibility-analysis problem. The SCP is defined as follows

Definition 5 (The Simultaneous Congruences Problem (SCP)) ***Given*** a set $A = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of ordered pairs of positive integers, and an integer k, $1 \leq k \leq n$.

Determine whether there is a subset $A' \subseteq A$ of k ordered pairs and a positive integer z such that for all $(x_i, y_i) \in A'$, $z \equiv x_i \bmod y_i$.

Theorem 2 The feasibility-analysis problem for arbitrary periodic task systems is co-NP-hard in the strong sense.

Proof Sketch: Leung and Merrill [20] reduced SCP to the (complement of the) feasibility-analysis problem for periodic task systems, as follows.

Let $\sigma \stackrel{\text{def}}{=} \langle \{(x_1, y_1), ..., (x_n, y_n)\}, k \rangle$ denote an instance of SCP. Consider the periodic task system τ comprised of n tasks $T_1, T_2, \ldots, T_n$, with $T_i = (a_i, e_i, d_i, p_i)$ for all $i, 1 \leq i \leq n$. For $1 \leq i \leq n$, let

- $a_i = x_i$;
- $e_i = \frac{1}{k-1}$;
- $d_i = 1$; and
- $p_i = y_i$.

Suppose that $\sigma \in$ SCP. Then there is a positive integer z such that at least k of the ordered pairs (x_i, y_i) "collide" on z: i.e., $z \equiv x_i \bmod y_i$ for at least k distinct i. This implies that the k corresponding periodic tasks will each have a job arrive at time-instant z; since each job's deadline is 1 unit removed from its arrival time while its execution requirement is $\frac{1}{k-1}$, not all these jobs can meet their deadlines.

Suppose now that τ is *in*feasible, i.e., some deadline is missed in the EDF-generated schedule of the jobs of τ. Let t_o denote the first time-instant at which this happens. Since all task parameters (except the execution requirements) are integers, it must be the case that t_o is an integer. For t_o to represent a deadline-miss, there must be at least k jobs arriving at time-instant $t_o - 1$. It must be the case that the ordered pairs (x_i, y_i) corresponding to the tasks generating these k jobs all collide at $(t_o - 1)$ — i.e., $(t_o - 1) \equiv x_i \bmod y_i$ for each of these k ordered pairs. Hence, $\sigma \in$ SCP. ■

The utilization $U(\tau)$ of the periodic task system constructed in the proof of Theorem 2 above is equal to $1/((k-1)\cdot\sum_{i=1}^{n} y_i)$. However, observe that the proof would go through essentially unchanged if $d_i = \epsilon$ and $e_i = \frac{\epsilon}{k-1}$, for any positive $\epsilon \leq 1$, and such a task system would have utilization $\epsilon/((k-1)\cdot\sum_{i=1}^{n} y_i)$. By choosing ϵ arbitrarily small, this utilization can be made as small as desired, yielding the result that the intractability result of Theorem 2 holds even if the utlization of the task system being analyzed is known to be bounded from above by an arbitrarily small constant:

Corollary 1 For any positive constant c, the feasibility-analysis problem for periodic task systems τ satisfying the constraint that $U(\tau) < c$ is co-NP-hard in the strong sense.

Observe that every periodic task T_i constructed in the proof of Theorem 2 above has its deadline parameter d_i no larger than its period parameter p_i. Therefore, the result of Theorem 2 holds for the more restricted *constrained-deadline* periodic task systems as well:

Corollary 2 The feasibility-analysis problem for constrained-deadline periodic task systems is co-NP-hard in the strong sense.

3.3 Feasibility-analysis Algorithms

The intractability results of Section 3.2 above make it unlikely that we will be able to perform feasibility-analysis on arbitrary (or even constrained-deadline) periodic task systems in polynomial time. In Section 3.3.1 below, we present an exponential-time algorithm for presenting exact feasibility-analysis; in Section 3.3.2, we outline a *sufficient*, rather than exact, feasibility-analysis test that has a better run-time complexity for periodic task systems with utilization bounded from above — by Corollary 1, exact feasibility analysis of such systems is also intractable.

3.3.1 An Exponential-time Feasibility-analysis Algorithm

In this section, we briefly derive an exponential-time algorithm for feasibility-analysis of arbitrary periodic task systems; the intractability result of Theorem 2 makes it unlikely that we will be able to do much better.

The approach adopted in developing the feasibility-analysis algorithm is as follows:

1. In Definition 6 below, we define the **demand** $\eta_\tau(t_1, t_2)$ of a periodic task system τ over an

interval $[t_1, t_2)$ to be the cumulative execution requirement by jobs of tasks in the system over the specified interval.

2. In Theorem 3 below, we reduce the feasibility determination question to an equivalent one concerning the demand function; specifically we prove that τ is infeasible if and only if $\eta_\tau(t_1, t_2)$ exceeds the length of the interval $[t_1, t_2)$ for some t_1 and t_2.

3. In Lemma 1 below, we obtain a closed-form expression for computing $\eta_\tau(t_1, t_2)$ for any τ, t_1 and t_2. This expression is easily seen to be computable in time linear in the number of tasks in τ.

4. In Lemmas 2–4 below, we derive a series of results that, for any infeasible τ, bound from above the value of some t_2 such that $\eta_\tau(t_1, t_2) > (t_2 - t_1)$. This upper bound $B(\tau)$ (given in Equation 4 below) is linear in the offset and deadline parameters of the tasks in τ as well as the least common multiple of the periods of the tasks in τ, and is hence at most exponential in both the size of the representation of periodic task system τ, and the values of the parameters of τ. The immediate consequence of this is that to determine whether τ is infeasible, we need only check whether $\eta_\tau(t_1, t_2) > (t_2 - t_1)$ for t_1 and t_2 satisfying $0 \leq t_1 < t_2 <$ this upper bound.

5. From the optimality of EDF as a preemptive uniprocessor scheduling algorithm, it follows that periodic task system τ is infeasible if and only if EDF misses some deadline while scheduling τ. We extend this observation to show (Lemma 5 below) that τ is infeasible if and only if the EDF schedule for τ would miss a deadline at or below the bound $B(\tau)$ of Equation 4. Since EDF can be implemented to run in polynomial time per scheduling decision, this immediately yields an exponential-time algorithm for feasibility-analysis of an arbitrary periodic task system τ: generate the EDF schedule for τ out until the bound $B(\tau)$, and declare τ feasible if and only if no deadlines are missed in this schedule.

Definition 6 (demand) For a periodic task system τ and any two real numbers t_1 and t_2, $t_1 \leq t_2$,the **demand** $\eta_\tau(t_1, t_2)$ of τ over the interval $[t_1, t_2)$ is defined to be cumulative execution requirement by jobs generated by tasks in τ that have arrival times at or after time-instant t_1, and deadlines at or before time-instant t_2.

Theorem 3 Periodic task system τ is feasible if and only if for all t_1, t_2, $t_1 < t_2$, it is the case that $\eta_\tau(t_1, t_2) \leq (t_2 - t_1)$.

Proof Sketch: **if:** Suppose that τ is infeasible, and let t_f denote the earliest time-instant at which the EDF schedule for τ misses a deadline. Let $t_b < t_f$ denote the latest time-instant prior to t_f at which this EDF schedule is not executing jobs with deadlines $\leq t_f$ — since 0^- is a time-instant prior to t_f at which this EDF schedule is not executing jobs with deadlines $\leq t_f$, time-instant t_b exists and is well-defined. Thus over $[t_b, t_f)$, the EDF schedule for τ executes only jobs arriving at or after t_b, and with deadlines at or before t_f — these are precisely the jobs whose execution requirements contribute to $\eta_\tau(t_b, t_f)$. Since some job nevertheless misses its deadline at t_f, it follows that $\eta_\tau(t_b, t_f) > t_f - t_b$.

only if: Suppose that τ is feasible; i.e., all jobs of τ complete by their deadlines in some optimal schedule S. For all t_1, t_2, $t_1 < t_2$, it is that case that all jobs generated by τ that both arrive in, and have their deadlines within, the interval $[t_1, t_2)$ are scheduled within this interval (along with, perhaps, parts of other jobs). It therefore follows that $\eta_\tau(t_1, t_2) \leq (t_2 - t_1)$. ■

For the remainder of this section let τ denote a periodic task system comprised of n tasks: $\tau = \{T_1, T_2, \ldots, T_n\}$, with $T_i = (a_i, e_i, d_i, p_i)$ for all i, $1 \leq i \leq n$. Let P denote the least common multiple of the periods of the tasks in τ: $P \stackrel{\text{def}}{=} \mathsf{lcm}\{p_1, p_2, \ldots, p_n\}$.

Lemma 1 For any t_1, t_2 satisfying $t_1 < t_2$,

$$\eta_\tau(t_1, t_2) = \sum_{i=1}^{n} e_i \cdot \max\left(0, \left\lfloor \frac{t_2 - a_i - d_i}{p_i} \right\rfloor - \max\left\{0, \left\lceil \frac{t_1 - a_i}{p_i} \right\rceil\right\} + 1\right) . \tag{1}$$

Proof Sketch: The number of jobs of T_i that lie within the interval $[t_1, t_2)$ is the number of non-negative integers k satisfying the inequalities

$$t_1 \leq a_i + k \cdot p_i$$
$$\text{and} \qquad a_i + k \cdot p_i + d_i \leq t_2 .$$

The Lemma follows, since there are exactly

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

such k's. ■

Lemma 2 For any t_1, t_2, satisfying $\max_{1 \leq i \leq n}\{a_i\} \leq t_1 < t_2$,

$$\eta_\tau(P + t_1, P + t_2) = \eta_\tau(t_1, t_2) . \tag{2}$$

Proof Sketch:

$$\eta_\tau(P+t_1, P+t_2) \;=\; \max\left\{0, \left\lfloor \frac{P+t_2-a_i-d_i}{p_i} \right\rfloor - \max\left\{0, \left\lceil \frac{P+t_1-a_i}{p_i} \right\rceil\right\} + 1\right\}.$$

Since $t_1 \geq a_i$, we may replace $\max\{0, \lceil \frac{P+t_1-a_i}{p_i} \rceil\}$ by $\lceil \frac{t_1+P-a_i}{p_i} \rceil$, and since p_i divides P, we may extract P/p_i from the floor and ceiling terms. Thus,

$$\begin{aligned}
\eta_\tau(P+t_1, P+t_2) &= \max\left\{0, \frac{P}{p_i} + \left\lfloor \frac{t_2-a_i-d_i}{p_i} \right\rfloor - \frac{P}{p_i} - \left\lceil \frac{t_1-a_i}{p_i} \right\rceil + 1\right\} \\
&= \max\left\{0, \left\lfloor \frac{t_2-a_i-d_i}{p_i} \right\rfloor - \left\lceil \frac{t_1-a_i}{p_i} \right\rceil + 1\right\} \\
&= \eta_\tau(t_1, t_2).
\end{aligned}$$

■

Lemma 3 Let $U(\tau) \leq 1$, $t_1 \geq \max_{1\leq i\leq n}\{a_i\}$, and $t_2 \geq (t_1 + \max_{1\leq i\leq n}\{d_i\})$. Then

$$\Big(\eta_\tau(t_1, t_2+P) > t_2 + P - t_1\Big) \Rightarrow \Big(\eta_\tau(t_1, t_2) > t_2 - t_1\Big) \tag{3}$$

Proof Sketch:

$$\begin{aligned}
\eta_\tau(t_1, t_2+P) &= \sum_{i=1}^{n} e_i \cdot \max\left\{0, \left\lfloor \frac{t_2+P-a_i-d_i}{p_i} \right\rfloor - \max\left\{0, \left\lceil \frac{t_1-a_i}{p_i} \right\rceil\right\} + 1\right\} \\
&= \sum_{i=1}^{n} e_i \cdot \max\left\{0, \frac{P}{p_i} + \left\lfloor \frac{t_2-a_i-d_i}{p_i} \right\rfloor - \left\lceil \frac{t_1-a_i}{p_i} \right\rceil + 1\right\}
\end{aligned}$$

as in the proof of Lemma 2. Since $P/p_i \geq 0$, we now have

$$\begin{aligned}
\eta_\tau(t_1, t_2+P) &\leq \sum_{i=1}^{n} \left(e_i \cdot \frac{P}{p_i} + e_i \cdot \max\left\{0, \left\lfloor \frac{t_2-a_i-d_i}{p_i} \right\rfloor - \left\lceil \frac{t_1-a_i}{p_i} \right\rceil + 1\right\}\right) \\
&\leq P\left(\sum_{i=1}^{n} \frac{e_i}{p_i}\right) + \eta_\tau(t_1, t_2) \\
&\leq P + \eta_\tau(t_1, t_2).
\end{aligned}$$

(The last step follows because $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$.) If we now suppose $\eta_\tau(t_1, t_2+P) > t_2 + P - t_1$, we have

$$\begin{aligned}
\eta_\tau(t_1, t_2) &\geq \eta_\tau(t_1, t_2+P) - P \\
&> t_2 + P - t_1 - P \\
&= t_2 - t_1
\end{aligned}$$

■

Lemma 4 Suppose that τ is infeasible and $U(\tau) \leq 1$. There exist t_1 and t_2, $0 \leq t_1 < t_2 < B(\tau)$ such that $\eta_\tau(t_1, t_2) > (t_2 - t_1)$, where

$$B(\tau) \stackrel{\text{def}}{=} 2 \cdot P + \max_{1 \leq i \leq n} \{d_i\} + \max_{1 \leq i \leq n} \{a_i\} \tag{4}$$

Proof Sketch: Follows directly from the above two lemmas, and Theorem 3. ■

Lemma 5 Suppose that τ is infeasible and $U(\tau) \leq 1$. Let t_1, t_2 be as specified in the statement of Lemma 4. The EDF schedule for τ will miss a deadline at or before t_2.

Proof Sketch: Immediate. ■

Given the above lemma as well as the optimality of EDF, we have the following theorem, and the consequent fact that feasibility-analysis of an arbitrary periodic task system τ can be performed in exponential time, by generating the EDF schedule of the system over the interval $[0, B(\tau))$.

Theorem 4 Periodic task system τ is infeasible if and only if the EDF schedule for τ misses a deadline at or before time-instant $B(\tau)$, where $B(\tau)$ is as defined in Equation 4. Hence, feasibility-analysis of periodic task systems can be performed in exponential time.

3.3.2 A More Efficient Sufficient Feasibility-analysis Algorithm

By Corollary 1, exact feasibility analysis is intractable even for arbitrary periodic task systems τ satisfying the constraint that $U(\tau) < c$ for some constant c. However, it turns out that a *sufficient* feasibility-test for such systems can be devised, that runs in time pseudo-polynomial in the representation of τ. This algorithm is based upon the following lemma, which we state without proof; this lemma relates the feasibility of an arbitrary periodic task system to the feasibility of a (different) *synchronous* periodic task system:

Lemma 6 Let $\tau = \{T_1 = (a_1, e_1, d_1, p_1), T_2 = (a_2, e_2, d_2, p_2), \ldots, T_i = (a_i, e_i, d_i, p_i), \ldots, T_n = (a_n, e_n, d_n, p_n)\}$ denote an arbitrary periodic task system. Task system τ is feasible if the synchronous periodic task system $\tau' = \{T'_1 = (e_1, d_1, p_1), T'_2 = (e_2, d_2, p_2), \ldots, T'_i = (e_i, d_i, p_i), \ldots, T'_n = (e_n, d_n, p_n)\}$ is feasible.

■

In Section 3.5, a pseudo-polynomial time algorithm for feasibility-analysis of synchronous periodic task systems is presented; this algorithm, in conjunction with the above lemma, immediately

yields a sufficient feasibility analysis algorithm for arbitrary periodic task systems τ satisfying the constraint that $U(\tau) < c$ for some constant c.

3.4 Implicit-deadline Systems

In implicit-deadline periodic and sporadic task systems, all tasks $T = (a, e, d, p)$ have their deadline parameter d equal to their period parameter p.

In the special case of implicit-deadline periodic task systems, the feasibility-analysis problem is quite tractable:

Theorem 5 A necessary and sufficient condition for implicit-deadline periodic task system τ to be feasible is $U(\tau) \leq 1$.

Proof Sketch: Let τ consist of n implicit-deadline periodic tasks $T_1, T_2, \ldots, T_n$. Consider a "processor-sharing" schedule S obtained by partitioning the time-line into arbitrarily small intervals, and scheduling each task T_i for a fraction $U(T_i)$ of the interval — since $U(\tau) \leq 1$, such a schedule can indeed be constructed. This schedule S schedules each job of T_i for $U(T_i) \times \frac{e_i}{p_i} = e_i$ units between its arrival and its deadline. ■

A minor generalization to Theorem 5 is possible: $U(\tau) \leq 1$ is a necessary and sufficient feasibility condition for for any task system τ in which all tasks $T = (a, e, d, p)$ satisfy the condition that $d \geq p$.

3.5 Synchronous Periodic Task Systems

In synchronous periodic task systems, the offset of all tasks is equal (i.e., $a_i = a_j$ for all tasks T_i, T_j; without loss of generality, a_i can then be considered equal to zero for all T_i).

Observe that the NP-hardness reduction in the proof of Theorem 2 critically depends upon being permitted to assign different values to the offset parameters of the periodic task system being reduced to; hence, this proof does not extend to the special case of synchronous periodic task systems. To our knowledge, the computational complexity of feasibility-determination for synchronous periodic task systems remains open. However, it can be shown that the result of Corollary 1 does not hold for synchronous periodic task systems – if the utilization $U(\tau)$ of synchronous periodic task system τ is bounded from above by a positive constant $c < 1$, then the feasibility-analysis problem for τ can be solved in pseudo-polynomial time.

The crucial observation concerning the feasibility-analysis of synchronous periodic task systems is in the following lemma

Lemma 7 For any synchronous periodic task system τ

$$\forall\ t_1, t_2,\quad 0 \leq t_1 < t_2,\quad \Big(\eta_\tau(t_1, t_2) \leq \eta_\tau(0, t_2 - t_1)\Big)$$

That is, the demand by jobs of synchronous task system τ over any interval of a given length is maximized when the interval starts at time-instant zero.

Proof Sketch: We omit a formal proof of this lemma, and instead provide some intuition as to why it should be true. It is easy to observe that the demand — i.e., the cumulative execution requirement — of any particular periodic task over an interval of length ℓ is maximized if a job of the task arrives at the start of the interval, since this (intuitively speaking) permits the largest number of future jobs of the task to have their deadlines within the interval. In a synchronous task system, however, time-instant zero is one at which all the tasks have a job arrive; i.e., the inteval $[0, \ell)$ is the one of length ℓ with maximum demand for each individual task, and consequently the one with maximum demand for task system τ. ■

As a consequence of Lemma 7, we may "specialize" Theorem 3 for synchronous periodic task systems, by restricting t_1 to always be equal to zero:

Theorem 6 (Theorem 3, for synchronous task systems) Synchronous periodic task system τ is feasible if and only if for all t_o, $0 < t_o$, it is the case that $\eta_\tau(0, t_o) \leq t_o$.

Proof Sketch: By Theorem 3, τ is feasible if and only if for all t_1, t_2, $t_1 < t_2$, it is the case that $\eta_\tau(t_1, t_2) \leq (t_2 - t_1)$. But by Lemma 7 above, $\eta_\tau(t_1, t_2) < \eta_\tau(0, t_2 - t_1)$. The theorem follows, by setting $t_o = t_2 - t_1$. ■

Despite the statement of Theorem 6 (which may seem to indicate that feasibility-analysis of synchronous systems may not be as intractable as for arbitrary systems) there is no synchronous feasibility-analysis algorithm known that guarantees significantly better performance than the exponential-time algorithm implied by Lemma 5. For the special case when the utilization of a synchronous periodic task system is a priori known to be bounded from above by a positive constant strictly less than one, however, Lemma 8 below indicates that pseudo-polynomial time feasibility anaysis is possible (contrast this to the case of arbitrary periodic task systems, where Corollary 1 rules out the existence of pseudo-polynomial time feasibility analysis unless P=NP).

Lemma 8 Let c be a fixed constant, $0 < c < 1$. Suppose that τ is infeasible and $U(\tau) \leq c$. There exists a t_2 satisfying

$$0 < t_2 < \frac{c}{1 - c} \max\Big\{p_i - d_i\Big\},$$

such that $\eta_\tau(0, t_2) > t_2$.

Proof Sketch: Suppose that $\eta_\tau(0, t_2) > t_2$. Let $I = \{i \mid d_i \leq t_2\}$. Substituting $a_i = 0$ for all i, $1 \leq i \leq n$, and $t_1 = 0$ in Lemma 1, we have

$$\begin{aligned}
t_2 &< \eta_\tau(0, t_2) \\
&= \sum_{i=1}^{n} e_i \cdot \max\left\{0, \left\lfloor \frac{t_2 - d_i}{p_i} \right\rfloor + 1\right\} \\
&= \sum_{i \in I} \left(\left\lfloor \frac{t_2 - d_i}{p_i} \right\rfloor + 1 \right) e_i \\
&\leq \sum_{i \in I} \frac{t_2 - d_i + p_i}{p_i} e_i \\
&= \sum_{i \in I} \left(\frac{t_2 e_i}{p_i} + \frac{(p_i - d_i)\, e_i}{p_i} \right) \\
&\leq c\, t_2 + c \max\{p_i - d_i\}.
\end{aligned}$$

Solving for t_2, we get

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

■

For constant c, observe that $\frac{c}{1-c} \max\{p_i - d_i\}$ is pseudo-polynomial in the parameters of the task system τ. A pseudo-polynomial time feasibility analysis algorithm for any synchronous periodic task system τ with bounded utilization now suggests itself, in the manner of the algorithm implied by Lemma 5. Simply generate the EDF schedule for τ out until $\frac{c}{1-c} \max\{p_i - d_i\}$, and declare τ feasible if and only if no deadlines are missed in this schedule.

Theorem 7 Feasibility-analysis of synchronous periodic task systems with utilization bounded by a constant strictly less than one can be performed in time pseudo-polynomial in the representation of the system.

3.6 Sporadic Task Systems

Each periodic task system generates exactly one (infinite) collection of real-time jobs. Since the sporadic task model constrains the *minimum*, rather than the exact, inter-arrival separation between successive jobs of the same task, a sporadic task system, on the other hand, is legally permitted to generate infinitely many distinct collections of real-time jobs. This is formalized in the notion of a *legal sequence of job arrivals* of sporadic task systems:

Definition 7 (Legal sequence of job arrivals) Let τ denote a sporadic task system comprised of n sporadic tasks: $\tau = \{T_1, T_2, \ldots, T_n\}$, with $t_i = (e_i, d_i, p_i)$ for all $i, 1 \leq i \leq n$. A **job arrival of** τ is an ordered pair (i, t), where $1 \leq i \leq n$ and t is a positive real number, indicating that a job of task T_i arrives at time-instant t (and has execution requirement e_i and deadline at $t + d_i$). A **legal sequence of job arrivals of** τ R_τ is a (possibly infinite) list of job arrivals of τ satisfying the following property: if (i, t_1) and (i, t_2) both belong to R_τ, then $|t_2 - t_1| \geq p_i$.

■

The feasibility-analysis question for a sporadic task system is as follows: given the specifications of a sporadic task system τ, determine whether there is any legal sequence of job arrivals of τ which cannot be scheduled to meet all deadlines. (Constrast this to the periodic case, in which *feasibility for a periodic task system* meant exactly the same thing as *feasibility* – the ability to schedule to meet all deadlines – *for the* unique *legal sequence of job arrivals* that the periodic task system generates. In a sense, therefore, the term "feasibility" for a periodic task system is merely shorthand for feasibility of a specific infinite sequence of jobs, while "feasibility" for a sporadic task system means something more – feasibility for infinitely many distinct infinite sequences of jobs.)

For periodic task systems the set of jobs to be scheduled is known *a priori*, during feasibility analysis. For sporadic task systems, however, there is no *a priori* knowledge about which set of jobs will be generated by the task system during run-time. In analyzing sporadic task systems, therefore, every conceivable sequence of possible requests must be considered. Fortunately, it turns out that, at least in the context of dynamic-priority preemptive uniprocessor scheduling, it is relatively easy to identify a unique "worst-case" legal sequence of job arrivals, such that all legal sequences of job arrivals can be scheduled to meet all deadlines if and only if this worst-case legal sequence can. And, this particular worst-case legal sequence of job arrivals is exactly the unique legal sequence of job arrivals generated by the synchronous periodic task task system with the exact same parameters as the sporadic task system:

Lemma 9 Let $\tau = \{T_1 = (e_1, d_1, p_1), T_2 = (e_2, d_2, p_2), \ldots, T_i = (e_i, d_i, p_i), \ldots, T_n = (e_n, d_n, p_n)\}$ denote an arbitrary sporadic task system. Every legal sequence of jobs arrivals R_τ of τ can be scheduled to meet all deadlines if and only if the synchronous periodic task system $\tau' = \{T'_1 = (e_1, d_1, p_1), T'_2 = (e_2, d_2, p_2), \ldots, T'_i = (e_i, d_i, p_i), \ldots, T'_n = (e_n, d_n, p_n)\}$ is feasible.

Proof Sketch: We will not prove this lemma formally here; the interested reader is referred to [5] for a complete proof. The main ideas behind the proof are these:

- Sporadic task system τ is infeasible if and only if there is some legal sequence of job arrivals R_τ and some interval $[t, t+t_o)$ such that the cumulative execution requirement of job arrivals in R_τ that both have their execution requirement and deadlines within the interval exceeds t_o, the length of the interval.

- The cumulative execution requirement by jobs generated by sporadic task system τ over an interval of length t_o is maximized if each task in τ generates a job at the start of the interval, and then generates successive jobs as rapidly as legal (i.e., each task T_i generates jobs exactly p_i time-units apart).

- But this is exactly the sequence of jobs that would be generated by the synchronous periodic task system τ' defined in the statement of the lemma.

■

Lemma 9 above reduces the feasibility-analysis problem for sporadic task systems to the feasibility-analysis problem for synchronous periodic task systems. the following two results immediately follow, from Theorem 4 and Theorem 7 respectively:

Theorem 8 Sporadic task system τ is infeasible if and only if the EDF schedule for τ misses a deadline at or before time-instant $2P$, where P denotes the least common multiple of the periods of the tasks in τ[2]. Hence, feasibility-analysis of sporadic task systems can be performed in exponential time.

Theorem 9 Feasibility-analysis of sporadic task systems with utilization bounded by a constant strictly less than one can be performed in time pseudo-polynomial in the representation of the system.

4 Static-priority Scheduling

Below, we first summarize the main results concerning dynamic-priority scheduling of periodic and sporadic task systems. Then in Sections 4.2.1–4.4, we provide further details about some of these results.

Recall that the *run-time scheduling problem* – the problem of choosing an appropriate scheduling algorithm – was rendered trivial for all the task models we had considered in the dynamic-priority

[2]Although this does not follow directly from Theorem 4, this $2P$ bound can in fact be improved to P.

case due to the proven optimality of EDF as a dynamic-priority run-time scheduling algorithm. Unfortunately, there is no static-priority result analogous to this result concerning the optimality of EDF; hence, the run-time scheduling problem is quite non-trivial for static-priority scheduling.

In the static-priority scheduling of periodic and sporadic task systems, all the jobs generated by an individual task are required to be assigned the same priority, which should be different from the priorities assigned to jobs generated by other tasks in the system. Hence, the run-time scheduling problem essentially reduces to the problem of associating a unique priority with each task in the system. The specific results known are as follows:

- For *implicit-deadline* sporadic and synchronous periodic task systems, the **Rate Monotonic** (RM) priority assignment algorithm, which assigns priorities to tasks in inverse proportion to their period parameters with ties broken arbitrarily, is an optimal priority assignment. That is, if there is any static priority assignment that would result in such a task system always meeting all deadlines, then the RM priority assignment for this task system, which assigns higher priorities to jobs generated by tasks with smaller values of the period parameter, will also result in all deadlines always being met.

- For implicit-deadline periodic task systems that are not synchronous, however, RM is provably not an optimal priority-assignment scheme (Section 4.3.2).

- For *constrained-deadline* sporadic and synchronous periodic task systems, the **Deadline Monotonic** (DM) priority assignment algorithm, which assigns priorities to tasks in inverse proportion to their deadline parameters with ties broken arbitrarily, is an optimal priority assignment. (Observe that RM priority assignment is a special case of DM priority assignment.)

- For constrained-deadline (and hence also arbitrary) periodic task systems which are not necessarily synchronous, however, the computational complexity of determining an optimal priority-assignment remains open. That is, while it is known (see below) that determining whether a constrained-deadline periodic task system is static-priority feasible is co-NP-complete in the strong sense, it is unknown whether this computational complexity is due to the process of assigning priorities, or merely to validating whether a given priority-assignment results in all deadlines being met. In other words, the computational complexity of the following question remains open [21, page 247]: *Given a constrained-deadline periodic task system*

τ that is known to be static-priority feasible, determine an optimal priority assignment for the tasks in τ.

Feasibility analysis. Determining whether an arbitrary periodic task system τ is feasible has been shown to be intractable — co-NP-complete in the strong sense. This intractability result holds even if $U(\tau)$ is known to be bounded from above by an arbitrarily small constant.

Utilization-based feasibility analysis. For the special case of *implicit-deadline* periodic and sporadic task systems (recall from above that rate-monotonic priority assignment is an optimal priority-assignment scheme for such task systems), a simple sufficient utilization-based feasibility test is known: an implicit-deadline periodic or sporadic task system τ is static-priority feasible if its utilization $U(\tau)$ is at most $n(2^{\frac{1}{n}} - 1)$, where n denotes the number of tasks in τ. Since $n(2^{\frac{1}{n}} - 1)$ monotonically decreases with increasing n and approaches $\ln 2$ as $n \to \infty$, it follows that any implicit-deadline periodic or sporadic task system τ satisfying $U(\tau) \leq \ln 2$ is static-priority feasible upon a preemptive uniprocessor, and hence can be scheduled using rate-monotonic priority assignment[3].

This utilization bound is a sufficient, rather than exact, feasibility-analysis test: it is quite possible that an implicit-deadline task system τ with $U(\tau)$ exceeding the bound above be static-priority feasible (as a special case of some interest, it is known that any implicit-deadline periodic or sporadic task system in which the periods are **harmonic** — i.e., for every pair of periods p_i and p_j in the task system it is either the case that p_i is an integer multiple of p_j or p_j is an integer multiple of p_i — is static-priority feasible if and only if its utilization is at most one).

Nevertheless, this is the best possible test using the utilization of the task system, and the number of tasks in the system, as the sole determinants of feasibility. That is, it has been shown that $n(2^{\frac{1}{n}} - 1)$ is the best possible utilization bound for feasibility-analysis of implicit-deadline periodic and sporadic task systems, in the following sense: For all $n \geq 1$, there is an implicit-deadline periodic task system τ with $U(\tau) = n(2^{\frac{1}{n}} - 1) + \epsilon$ that is not static-priority feasible, for ϵ an arbitrarily small positive real number.

[3]Here, $\ln 2$ denotes the natural logarithm of 2 (approximately 0.6931).

4.1 Some Preliminary Results

In this section, we present some technical results concerning static-priority scheduling that will be used later, primarily in Sections 4.3 and 4.4 when the rate-monotonic and deadline-monotonic priority assignments are discussed.

For the remainder of this section, we will consider the static-priority scheduling of a periodic/sporadic task system τ comprised of n tasks: $\tau = \{T_1, T_2, \ldots T_n\}$. We use the notation $T_i \succ T_j$ to indicate that task T_i is assigned a higher priority than task T_j in the (static) priority-assignment scheme under consideration.

The **response time** of a job in a particular schedule is defined to be the amount of time that has elapsed between the arrival of the job and its completion; clearly, in order for a schedule to meet all deadines it is necessary that the response time of each job not exceed the relative deadline of the task that generates the job. The following definition is from [23, page 131]

Definition 8 (critical instant) A critical instant of a task T_i is a time-instant which is such that

1. A job of T_i released at the instant has a maximum response time of all jobs of T_i, if the response time of every job of T_i is less than or equal to the relative deadline of T_i, and
2. the response time of the job of T_i released at the instant is greater than the relative deadline if the response time of some job of T_i exceeds the relative deadline.

The response time of a job of T_i is maximized when it is released at its critical instant.

The following lemma asserts that for synchronous task systems in which the deadline of all tasks are no larger than their periods, a critical instant for all tasks $T_i \in \tau$ occurs at time-instant zero (i.e., when each task in τ simultaneously releases a job). While this theorem is intuitively appealing — it is reasonable that a job will be delayed the most when it arrives simultaneous with a job from each higher-priority task, *and* each such higher-priority task generates successive jobs as rapidly as permitted — the proof turns out to be quite non-trivial and long. We will not present the proof here; the interested reader is referred to a good text-book on real-time systems (e.g., [23, 6]) for details.

Lemma 10 Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a synchronous periodic task system with constrained deadlines (i.e., with $d_i \leq p_i$ for all $i, 1 \leq i \leq n$). When scheduled using a static-priority scheduler under static priority assignment $T_1 \succ T_2 \succ \cdots \succ T_n$, the response time of the first job of task T_i is the largest among all the jobs of task T_i.

■

It was proven [19] that the restriction that τ be comprised of *constrained-deadline* tasks is necessary to the correctness of Lemma 10; i.e., that Lemma 10 does not extend to sporadic or synchronous periodic task systems in which the deadline parameter of tasks may exceed their period parameter.

A schedule is said to be *work-conserving* if it never idles the processor while there is an active job awaiting execution.

Lemma 11 Let τ denote a periodic task system, and S_1 and S_2 denote work-conserving schedules for τ. Schedule S_1 idles the processor at time-instant t if and only if schedule S_2 idles the processor at time-instant t, for all $t \geq 0$.

Proof Sketch: The proof is by induction: we assume that schedules S_1 and S_2 both idle the processor at time-instant t_o, and that they both idle the processor at the same time-instants at all time-instants prior to t_o. The base case has $t_o = 0$.

For the inductive step, let t_1 denote the first instant after t_o at which either schedule idles the processor. Assume without loss of generality that schedule S_1 idles the processor over $[t_1, t_2)$. Since S_1 is work-conserving, this implies that all jobs that arrived prior to t_2 have completed over $[t_1, t_2)$, i.e., the cumulative execution requirement of jobs of τ arriving prior to t_2 is equal to $(t_1 - t_o)$. But since S_2 is also work-conserving, this would imply that S_2 also idles the processor over $[t_1, t_2)$. ■

4.2 The Feasibility-analysis Problem

In Section 4.2.1 below, we show that the static-priority feasibility-analysis problem is intractable for arbitrary periodic task systems. We also show that the problem of determining whether a *specific* priority assignment results in all deadlines being met is intractable, even for the special case of implicit-deadline periodic task systems. However, all these intractability results require asynchronicity: for *synchronous* task systems, we will see (Section 4.2.2) that static-priority feasibility-analysis is no longer quite as computationally expensive.

4.2.1 The Intractability of Feasibility-analysis

Theorem 10 The static-priority feasibility-analysis problem for arbitrary periodic task systems is co-NP-hard in the strong sense.

Proof Sketch: Leung and Whitehead [21] reduced SCP to the complement of the static-priority feasibility-analysis problem for periodic task systems, as follows. (This transformation is identical to the one used in the proof of Theorem 2.)

Let $\sigma \stackrel{\text{def}}{=} \langle \{(x_1, y_1), ..., (x_n, y_n)\}, k \rangle$ denote an instance of SCP. Consider the periodic task system τ comprised of n tasks $T_1, T_2, \ldots, T_n$, with $T_i = (a_i, e_i, d_i, p_i)$ for all $i, 1 \leq i \leq n$. For $1 \leq i \leq n$, let

- $a_i = x_i$;
- $e_i = \frac{1}{k-1}$;
- $d_i = 1$; and
- $p_i = y_i$.

Suppose that $\sigma \in$ SCP. Then there is a positive integer z such that at least k of the ordered pairs (x_i, y_i) "collide" on z: i.e., $z \equiv x_i \bmod y_i$ for at least k distinct i. This implies that the k corresponding periodic tasks will each have a job arrive at time-instant z; since each job's deadline is 1 unit removed from its arrival time while its execution requirement is $\frac{1}{k-1}$, not all these jobs can meet their deadlines regardless of the priority assignment.

Suppose now that $\sigma \notin$ SCP. That is, for no positive integer w is it the case that k or more of the ordered pairs (x_i, y_i) collide on w. This implies that at no time-instant will k periodic tasks each have a job arrive at that instant; since each job's deadline is 1 unit removed from its arrival time while its execution requirement is $\frac{1}{k-1}$, all deadlines will be met with any of the n possible priority assignments. ■

Observe that every periodic task T_i constructed in the proof of Theorem 10 above has its deadline parameter d_i no larger than its period parameter p_i. Therefore, the result of Theorem 10 holds for constrained-deadline periodic task systems as well:

Corollary 3 The static-priority feasibility-analysis problem for constrained-deadline periodic task systems is co-NP-hard in the strong sense.

Theorem 10 above does not address the question of whether the computational complexity of static-priority feasibility-analysis problem for arbitrary periodic task systems arises due to the complexity of **(i)** determining a suitable priority assignment, or **(ii)** determining whether this priority-assignment results in all deadlines being met. The following result asserts that the second

question above is in itself intratable; however, the computational complexity of the first question above remains open.

Theorem 11 Given an implicit-deadline periodic task system τ and a priority assignment on the tasks in τ, it is co-NP-hard in the strong sense to determine whether the schedule generated by a static-priority scheduler using these priority assignments meets all deadlines.

(Since constrained-deadline and arbitrary periodic task systems are generalizations of implicit-deadline periodic task systems, this hardness result holds for constrained-deadline and arbitrary periodic task systems as well.)

Proof Sketch: This proof, too, is from [21].

Let $\sigma \stackrel{\text{def}}{=} \langle \{(x_1, y_1), ..., (x_n, y_n)\}, k \rangle$ denote an instance of SCP. Consider the periodic task system τ comprised of $n+1$ tasks $T_1, T_2, \ldots, T_n, \mathbf{T_{n+1}}$, with $T_i = (a_i, e_i, d_i, p_i)$ for all $i, 1 \leq i \leq n+1$. For $1 \leq i \leq n$, let

- $a_i = x_i$;
- $e_i = \frac{1}{k}$; and
- $d_i = p_i = 1$.

Let $T_{n+1} = (0, \frac{1}{k}, 1, 1)$. The priority-assignment is according to task indices, i.e., $T_i \succ T_{i+1}$ for all $i, 1 \leq i \leq n$. Specifically, T_{n+1} has the lowest priority. We leave it to the reader to verify that all jobs of task T_{n+1} meet their deadlines if and only if $\sigma \notin$ SCP. ■

4.2.2 More Tractable Special Cases

Observe that the NP-hardness reduction in the proof of Theorem 10 critically depends upon being the fact that different periodic tasks are permitted to have different offsets; hence, this proof does not hold for sporadic or for synchronous periodic task systems. In fact, feasibility-analysis is known to be more tractable for sporadic and synchronous periodic task systems in which all task have their deadline parameters no larger than their periods (i.e., deadline-constrained and implicit-deadline task systems):

Theorem 12 The static-priority feasibility-analysis problem for synchronous constrained-deadline (and implicit-deadline) periodic task systems can be solved in time pseudo-polynomial in the representation of the task system.

Proof Sketch: In Section 4.3 and Section 4.4, we will see that the *priority-assignment problem* — determining an assignment of priorities to the tasks of a static-priority feasible task system such that all deadlines is met — has efficient solutions for implicit-deadline and constrained-deadline task systems. By Lemma 10, we can verify that such a synchronous periodic task system is feasible by ensuring that each task meets its first deadline; i.e., by generating the static-priority schedule under the "optimal" priority assignment out until the largest deadline of any task. ■

4.3 The Rate Monotonic Scheduler

The rate-monotonic priority assignment was defined by Liu and Layland [22] and Serlin [28], for sporadic and synchronous periodic implicit-deadline task systems. That is, each task T_i in task system τ is assumed characterized of two parameters: execution requirement e_i and period p_i. The RM priority-assignment scheme assigns tasks priorities in inverse proportion to their period parameter (equivalently, in direct proportion to their *rate* parameter – hence the name), with ties broken arbitrarily.

Computing the priorities of a set of n tasks for the rate monotonic priority rule amounts to ordering the task set according to their periods. Hence the time complexity of the rate monotonic priority assignment is the time complexity of a sorting algorithm, i.e., $\mathcal{O}(n \log n)$.

4.3.1 Optimality for Sporadic and Synchronous Periodic Implicit-deadline Task Systems

Theorem 13 Rate monotonic priority assignment is optimal for sporadic and synchronous periodic task systems with implicit deadlines.

Proof Sketch: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ denote a sporadic or synchronous periodic task system with implicit deadlines. We must prove that if a static priority assignment would result in a schedule for τ in which all deadlines are met, then a rate monotonic priority assignment for τ would also result in a schedule for τ in which all deadlines are met.

Suppose that priority assignment ($T_1 \succ T_2 \succ \cdots \succ T_n$) results in a such a schedule. Let T_i and T_{i+1} denote two tasks of adjacent priorities with $p_i \geq p_{i+1}$. Let us exchange the priorities of T_i and T_{i+1}: if the priority assignment obtained after this exchange results in all deadlines being met, then we may conclude that any rate monotonic priority assignment will also result in all deadlines being met since any rate monotonic priority assignment can be obtained from any priority ordering

by a sequence of such priority exchanges.

To see that the priority assignment obtained after this exchange results in all deadlines being met, observe that

- The priority exchange does not modify the schedulability of the tasks with a higher priority than T_i (i.e., $T_1, \ldots, T_{i-1}$).
- The task T_{i+1} remains schedulable after the priority exchange, since its jobs may use all the free time-slots left by $\{T_1, T_2, \ldots, T_{i-1}\}$ instead of only those left by $\{T_1, T_2, \ldots, T_{i-1}, T_i\}$.
- Assuming that the jobs of T_i remain schedulable (this will be proved below), by Lemma 11 above we may conclude that the scheduling of each task T_k, for $k = i+2, i+3, \ldots, n$ is not altered since the idle periods left by higher priority tasks are the same.
- Hence we need only verify that T_i also remains schedulable. From Lemma 10 we can restrict our attention to the first job of task T_i. Let r_{i+1} denote the response time of the first job of T_{i+1} before the priority exchange: the feasibility implies $r_{i+1} \leq p_{i+1}$. During the interval $[0, r_{i+1})$ the processor (when left free by higher priority tasks) is assigned first to the (first) job of T_i and then to the (first) job of T_{i+1}; the latter is not interrupted by subsequent jobs of T_i since $p_i > p_{i+1} \geq r_{i+1}$. Hence, after the priority exchange, the processor allocation is exchanged between T_i and T_{i+1}, and it follows that T_i ends it computation at time r_{i+1} and meets its deadline since $r_{i+1} \leq p_{i+1} \leq p_i$.

■

As a consequence of Theorem 13 and Lemma 10, it follows that synchronous implicit-deadline periodic task system τ is static-priority feasible if and only if the first job of each task in τ meets its deadline when priorities are assigned in rate-monotonic order. Since this can be determined by simulating the schedule out until the largest period of any task in τ, we have the following corollary:

Corollary 4 Feasibility-analysis of sporadic and synchronous periodic implicit-deadline task systems can be done in pseudo-polynomial time.

4.3.2 Non-optimality for Asynchronous Periodic Implicit-deadline Task Systems

If all the implicit-deadline periodic tasks are not required to have the same initial offset, however, RM priority-assignment is no longer an optimal priority-assignment scheme. This can be seen by

the following example task system [10]:

$$\tau = \{T_1 = (0, 7, 10), T_2 = (4, 3, 15), T_3 = (0, 1, 16)\} .$$

The RM-priority assignment ($T_1 \succ T_2 \succ T_3$) results in the first deadline of T_3 being missed, at time-instant 16. However, the priority-assignment ($T_1 \succ T_3 \succ T_2$) does not result in any missed deadlines: this has been verified [10] by constructing the schedule over the time-interval $[0, 484)$, during which no deadlines are missed, and observing that the state of the system at time-instant 4 is identical to the state at time-instant 484[4].

4.3.3 Utilization Bound

In this section, we restrict our attention to sporadic or synchronous periodic implicit-deadline periodic task systems — hence unless explicitly stated otherwise, all task systems are either sporadic or synchronous periodic, and implicit-deadline.

Definition 9 Within the context of a particular scheduling algorithm, task system $\tau = \{T_1, \ldots, T_n\}$ is said to *fully utilize* the processor if all deadlines of τ are met when τ is scheduled using this scheduling algorithm, and an increase in the execution requirement of any T_i $(1 \leq i \leq n)$ results in some deadline being missed.

Theorem 14 When scheduled using the rate monotonic scheduling algorithm, task system $\tau = \{T_1, \ldots, T_n\}$ fully utilizes the processor if and only if the task system $\{T_1, \ldots, T_{n-1}\}$

1. meets all deadlines, and
2. idles the processor for exactly e_n time units over the interval $[0, p_n)$,

when scheduled using the rate-monotonic scheduling algorithm.

Proof Sketch: Immediately follows from Lemma 10. ■

Let b_n denote the lower bound of $U(\tau)$ among all task systems τ comprised of exactly n tasks which fully utilize the processor under rate-monotonic scheduling.

[4]Leung and Whitehead [21, Theorem 3.5] proved that any constrained-deadline periodic task system τ meets all deadlines in a schedule constructed under a given priority assignment if and only if it meets all deadlines over the interval $(a, a + 2P]$, where a denotes the largest offset of any task in τ, and P the least common multiple of the periods of all the tasks in τ. Hence, it suffices to test this schedule out until $4 + 240 \times 2 = 484$.

Lemma 12 For the subclass of task systems satisfying the constraint the the ratio between the periods of any two tasks is less than 2, $b_n = n(\sqrt[n]{2} - 1)$.

Proof Sketch: Let $\tau = \{T_1, \ldots, T_n\}$ denote an n-task task system, and assume that $p_1 \leq p_2 \leq \cdots \leq p_n$. We proceed in several stages.

§1. We first show that, in the computation of b_n, we may restrict our attention to task systems τ fully utilizing the processor such that $\forall i < n : \ e_i \leq p_{i+1} - p_i$.

Suppose that τ is a task system that fully utilizes the processor and has the smallest utilization from among all task systems that fully utilize the processor. Consider first the case of e_1 and suppose that $e_1 = p_2 - p_1 + \Delta$ ($\Delta > 0$; notice that we must have that $p_2 < 2p_1$, otherwise $e_1 > p_1$ and the task set is not schedulable). Notice that the task system $\tau' = \{T_1', \ldots, T_n'\}$ with $p_i' = p_i \ \forall i$ and $e_1' = p_2 - p_1$, $e_2' = e_2 + \Delta$, $e_3' = e_3$, ..., $e_n' = e_n$ also fully utilizes the processor. Furthermore, $U(\tau) - U(\tau') = \frac{\Delta}{p_1} - \frac{\Delta}{p_2} > 0$, contradicting our hypothesis that the utilization of $T_1, \ldots, T_n$ is minimal.

The above argument can now be repeated for $e_2, e_3, \ldots, e_{n-1}$; in each case, it may be concluded that $e_i \leq p_{i+1} - p_i$.

§2. Next, we show that, in the computation of b_n, we may restrict our attention to task systems τ fully utilizing the processor such that $\forall i < n : \ e_i = p_{i+1} - p_i$.

It follows from Theorem 14, the fact that each task T_i with $i < n$ releases and completes exactly two jobs prior to time-instant p_n (this is a consequence of our previously-derived constraint that $e_i \leq p_{i+1} - p_i$ for all $i < n$) that

$$e_n = p_n - 2\sum_{i=1}^{n-1} e_i$$

— this is since the first $n-1$ tasks meet all deadlines, and over the interval $[0, p_n)$ they together use $\sum_{i=1}^{n-1} e_i$ time units, with $\sum_{i=1}^{n-1} e_i \leq p_n - p_1 < p_1$).

Consider first the case of e_1 and suppose that $e_1 = p_2 - p_1 - \Delta$ ($\Delta > 0$). Notice that the task system $\tau'' = \{T_1'', \ldots, T_n''\}$ with $p_i'' = p_i \ \forall i$ and $e_1'' = e_1 + \Delta = p_2 - p_1$, $e_n'' = e_n - 2\Delta$, $e_i'' = e_i$ for $i = 2, 3, \ldots, n-1$, also fully utilizes the processor. Furthermore, $U(\tau) - U(\tau'') = -\frac{\Delta}{p_1} + \frac{2\Delta}{p_n} > 0$, contradicting our hypothesis that the utilization of $T_1, \ldots, T_n$ is minimal.

The above argument can now be repeated for $e_2, e_3, \ldots, e_{n-1}$; in each case, it may be concluded that $e_i = p_{i+1} - p_i$.

§3. Finally, let $g_i \stackrel{\text{def}}{=} \frac{p_n - p_i}{p_i} \quad (i = 1, \ldots, n-1)$; we get

$$U(\tau) = \sum_{i=1}^{n} \frac{e_i}{p_i} = 1 + g_1(\frac{g_1 - 1}{g_1 + 1}) + \sum_{i=2}^{n-1} g_i(\frac{g_i - g_{i-1}}{g_i + 1})$$

This expression must be minimzed; hence $\frac{\partial U(\tau)}{\partial g_j} = \frac{g_j^2 + 2g_j - g_{j_1}}{g_j + 1^2} - \frac{g_{j+1}}{g_{j+1}+1} = 0, \quad \text{for } j = 1, \ldots, n-1.$ The general solution for this can be shown to be $g_j = 2^{\frac{n-j}{n}} - 1 \quad (j = 1, \ldots, n-1)$, from which it follows that $b_n = n(\sqrt[n]{2} - 1)$. ■

The restriction that the ratio between task periods is less than 2 can now be relaxed, yielding the desired utilization bound.

Theorem 15 ([22]) Any implicit-deadline sporadic or synchronous periodic task system τ comprised of n tasks is successfully scheduled using static-priority scheduling with the rate-monotonic priority assignment, provided

$$U(\tau) \leq n(\sqrt[n]{2} - 1) \ .$$

Proof Sketch: Let τ denote a system of n tasks that fully utilizes the processor. Suppose that for some i, $\left\lfloor \frac{p_n}{p_i} \right\rfloor > 1$, i.e., there exists an integer $q > 1$ such that $p_n = q \cdot p_i + r, \ r \geq 0$. Let us obntain task system τ' from task system τ by replacing the task T_i in τ by a task T_i' such that $p_i' = q \cdot p_i$ and $e_i' = e_i$, and increase e_n by the amount needed to again fully utilize the processor. This increase is at most $e_i(q-1)$, the time within the execution of T_n occupied by T_i but not by T_i' (it may be less than e_i if some slots left by T_i' are used by some T_j with $i < j < n$). We have

$$U(\tau') \leq U(\tau) - \frac{e_i}{p_i} + \frac{e_i}{p_i'} + [(q-1)\frac{e_i}{p_n}] \ ,$$

i.e.,

$$U(\tau') \leq U(\tau) + e_i(q-1)[\frac{1}{q \cdot p_i + r} - \frac{1}{q \cdot p_i}].$$

Since $q - 1 > 0$ and $\frac{1}{q \cdot p_i + r} - \frac{1}{q \cdot p_i} \leq 0$, $U(\tau') \leq U(\tau)$. By repeated applications of the above argument, we can obtain a τ'' in which no two task periods have a ratio greater than two, such that τ'' fully utilizes the processor and has utilization no greater than $U(\tau)$. That is, the bound b_n derived in Lemma 12 represents a lower bound on $U(\tau)$ among <u>all</u> task systems τ comprised of exactly n tasks which fully utilize the processor under rate-monotonic scheduling, and not just those task systems in which the ratio of any two periods is less than two.

To complete the proof of the theorem, it remains to show that if a system of n tasks has an utilization factor less than the upper bound b_n, then the system is schedulable. This immediately follows from the above arguments, and the fact that b_n is strictly decreasing with increasing n. ■

4.4 The Deadline Monotonic Scheduler

Leung and Whitehead [21] have defined the deadline monotonic priority assignment (also termed the *inverse-deadline* priority assignment): priorities assigned to tasks are inversely proportional to the deadline. It may be noticed that in the special case where $d_i = p_i \quad (1 \leq i \leq n)$, the deadline monotonic assignment is equivalent to the rate monotonic priority assignment.

4.4.1 Optimality for Sporadic and Synchronous Periodic Constrained-deadline Task Systems

Theorem 16 The deadline monotonic priority assignment is optimal for sporadic and synchronous periodic task systems with constrained deadlines.

Proof Sketch: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ denote a sporadic or synchronous periodic task system with constrained deadlines. We must prove that if a static priority assignment would result in a schedule for τ in which all deadlines are met, then a deadline-monotonic priority assignment for τ would also result in a schedule for τ in which all deadlines are met.

Suppose that priority assignment ($T_1 \succ T_2 \succ \cdots \succ T_n$) results in a such a schedule. Let T_i and T_{i+1} denote two tasks of adjacent priorities with $d_i \geq d_{i+1}$. Let us exchange the priorities of T_i and T_{i+1}: if the priority assignment obtained after this exchange results in all deadlines being met, then we may conclude that any deadline monotonic priority assignment will also result in all deadlines being met since any deadline-monotonic priority assignment can be obtained from any priority ordering by a sequence of such priority exchanges. To see that the priority assignment obtained after this exchange results in all deadlines being met, observe that

- The priority exchange does not modify the schedulability of the tasks with a higher priority than T_i (i.e., $T_1, \ldots, T_{i-1}$).
- The task T_{i+1} remains schedulable after the priority exchange, since its jobs may use all the free time-slots left by $\{T_1, T_2, \ldots, T_{i-1}\}$ instead of merely those left by $\{T_1, T_2, \ldots, T_{i-1}, T_i\}$.
- Assuming that the jobs of T_i remain schedulable (this will be proved below), by Lemma 11 above we may conclude that the scheduling of each task T_k, for $k = i+2, i+3, \ldots, n$ is not altered since the idle periods left by higher priority tasks are the same.
- Hence we need only verify that T_i also remains schedulable. From Lemma 10 we can restrict

our attention to the first job of task T_i. Let r_{i+1} denote the response time of the first job of T_{i+1} before the priority exchange: the feasibility implies $r_{i+1} \leq d_{i+1}$. During the interval $[0, r_{i+1})$ the processor (when left free by higher priority tasks) is assigned first to the (first) job of T_i and then to the (first) job of T_{i+1}; the latter is not interrupted by subsequent jobs of T_i since $p_i \geq d_i \geq d_{i+1} \geq r_{i+1}$. Hence after the priority exchange, the processor allocation is exchanged between T_i and T_{i+1}, and it follows that T_i ends its computation at time r_{i+1} and meets its deadline since $r_{i+1} \leq d_{i+1} \leq d_i$.

■

As a consequence of Theorem 16 and Lemma 10, it follows that synchronous periodic task system τ with constrained deadlines is static-priority feasible if and only if the first job of each task in τ meets its deadline when priorities are assigned in deadline-monotonic order. Since this can be determined by simulating the schedule out until the largest deadline of any task in τ, we have the following corollary:

Corollary 5 Feasibility-analysis of sporadic and synchronous periodic constrained-deadline task systems can be done in pseudo-polynomial time.

4.4.2 Non-optimality for Sporadic and Synchronous Periodic Arbitrary-deadline Task Systems

We have already stated that the computational complexity of optimal priority-assignment for constrained-deadline periodic task systems that are not necessarily synchronous is currently unknown; in particular, DM is *not* an optimal priority-assignment scheme for such systems.

Even for *synchronous* periodic (as well as sporadic) task systems which are not constrained-deadline (i.e., in which individual tasks T_i may have $d_i > p_i$), however, the deadline monotonic rule is no longer optimal. This is illustrated by the following example given by Lehoczky [19].

Example 1 Let $\tau = \{T_1 = (e_1 = 52, d_1 = 110, p_1 = 100), T_2 = (e_2 = 52, d_2 = 154, p_2 = 140,)\}$. Both the RM and the DM priority assignment schemes would assign higher priority to T_1: $T_1 \succ T_2$. With this priority assignment, however, the first job of T_2 misses its deadline at time 154 With the priority assignment $T_2 \succ T_1$, it may be verified that all deadlines are mt. ■

5 Conclusions and Further Reading

In this chapter, we have presented what we consider to be some of the more important results that have been published over the past 30 years, on the preemptive uniprocessor scheduling of recurring real-time tasks. Without a doubt, our decisions regarding which results to present have been driven by our own limited knowledge, our preferences and biases, and the page-limits and time-constraints that we have worked under. Below, we list some other ideas that we consider very important to a further understanding of preemptive uniprocessor scheduling of recurring tasks that we have chosen to not cover in this chapter.

More efficient feasibility tests. We chose to restrict ourselves to showing the computational complexity of the various feasibility-analysis problems we discussed here. Much work has been done on obtaining more efficient implementations of some of the algorithms described here; although these more efficient implementations have about the same worst-case computational complexity as the algorithms described here, they typically perform quite a bit better on average. Important examples of such algorithms include the work of Ripoll, Crespo and Mok [27] and the recent sufficient feasibility test of Devi [9] on dynamic-priority scheduling, and the large body of work on *response time analysis* [16, 15, 1] on static-priority scheduling.

More general task models. Recent research has focused upon obtaining more general models for recurring tasks than the simple periodic and sporadic models discussed in this chapter. Some interesting new models include the *multiframe* model [25, 26, 3], which was proposed to accurately model multimedia traffic, and the *recurring real-time task* model [2], which allows for the modelling of conditional real-time code. (Another interesting recurring task model – the *pinwheel* model – is the subject of another chapter [7] in this handbook.)

Resource sharing. Real-time systems typically require the sharing of some other resources as well as the processor; often, a process or task needs exclusive access to such resources. Issues that arise when such resources must be shared by recurring tasks have been extensively studied, and form the subject of another chapter [29] of this handbook.

References

[1] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995.

[2] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 24(1):99–128, 2003.

[3] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 17(1):5–22, July 1999.

[4] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2:301–324, 1990.

[5] S. Baruah, A. Mok, and L. Rosier. The preemptive scheduling of sporadic, real-time tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997.

[7] D. Chen and A. Mok. The pinwheel: A real-time scheduling problem. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2003.

[8] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[9] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the Euromicro Conference on Real-time Systems*, Porto, Portugal, 2003. IEEE Computer Society Press.

[10] J. Goossens and R. Devillers. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, 13(2):107–126, 1997.

[11] K. Hong and J. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41:1326–1331, 1992.

[12] R. Howell and M. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle times. Technical Report TR-CS-91-6, Kansas State University, Department of Computer and Information Sciences, 1991.

[13] K. Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with preemption constraints. In *Proceedings of the 10th Real-Time Systems Symposium*, Santa Monica, California, December 1989. IEEE Computer Society Press.

[14] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139, San Antonio, Texas, December 1991. IEEE Computer Society Press.

[15] M. Joseph, editor. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, Oct. 1986.

[17] E. Lawler and M. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12:9–12, 1981.

[18] J. Lehoczky, L. Sha, and J. Stronider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the Real-Time Systems Symposium*, pages 261–270, San Jose, CA, December 1987. IEEE.

[19] J. P. Lehoczky. Fixed priority scheduling of periodic tasks with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–209, Dec. 1990.

[20] J. Leung and M. Merrill. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11:115–118, 1980.

[21] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[22] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[23] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2000.

[24] A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[25] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th Real-Time Systems Symposium*, Washington, DC, 1996. IEEE Computer Society Press.

[26] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, October 1997.

[27] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 11:19–39, 1996.

[28] O. Serlin. Scheduling of time critical processes. In *Proceedings of The 1972 Spring Joint Computer Conference*, volume 40 of AFIPS *Conference Proceedings*, 1972.

[29] L. Sha. Synchronization issues in real-time task systems. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2003.

[30] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the Real-Time Systems Symposium*, pages 251–258, Huntsville, Alabama, December 1988. IEEE.

[31] B. Sprunt, L. Sha, and J. Lehoczky. Scheduling sporadic and aperiodic events in a hard real-time system. Technical Report ESD-TR-89-19, Carnegie Mellon University, 1989.