3

FUNDAMENTALS OF EDF SCHEDULING

This chapter discusses the basic results for the EDF scheduling algorithm, regarding optimality and feasibility analysis, respectively. The optimality of a real-time scheduling algorithm means that whenever a task set can be scheduled to meet all its deadlines, then it will be feasibly scheduled by the optimal algorithm. Usually, a real-time scheduling algorithm must guarantee *a priori* that all deadlines of a particular task set are met. The problem is thus to establish whether a given task set can be feasibly scheduled by the chosen algorithm. In the literature, a solution to this problem is termed *feasibility analysis*. In this chapter, the optimality of the EDF algorithm, and a feasibility analysis for task sets, when EDF scheduling is assumed, are described in detail.

The description in this chapter and the next is meant to present a basic, but comprehensive theory of the EDF algorithm for independent tasks. The results are presented for task models of increasing complexity, but always retaining the notion of independent tasks. Three different concepts used in the analysis are described: processor utilization, processor demand, and busy period. Each concept is valuable in different contexts. In the remainder of the book more sophisticated task modles are presented including task sets with dependencies of various types.

One practical result of the basic theory presented in chapters 3 and 4 is an algorithm that real-time system designers can use to analyze the feasibility of EDF scheduled systems.

3.1 OPTIMALITY ON UNI-PROCESSOR SYSTEMS

The first result concerning the optimality of the EDF scheduling algorithm was originally given in [13]. In this work the reference (system) model is quite simple: there are n independent jobs in the system (*i.e.*, n jobs with a single instance each), all ready at time $t = 0$, with each job J_i having a *deadline* d_i. For any given scheduling sequence, the *lateness* of a job i is defined as $l_i = f_i - d_i$, where f_i is its completion time. If the goal is to minimize the maximum lateness of all jobs, assuming the schedule is non-preemptive, a simple solution is the *earliest-deadline first* algorithm:

Theorem 3.1 (Jackson's Rule) *Any sequence is optimal that puts the jobs in order of non-decreasing deadlines.*

The proof of the theorem can be given by a simple interchange argument, however it is not presented here, since it is similar to other arguments described later, and applicable to more general optimality results.

Now consider some changes to the reference model. When *release times* are introduced in the model, the problem becomes NP-hard [18]. However, if preemption is allowed, even with release times the scheduling problem remains easy, and, in particular, the EDF algorithm is one possible solution. Note that release times are complications to task models which make scheduling problems difficult and that allowing preemption tends to make the scheduling problems easier.

The optimality of the EDF preemptive scheduling algorithm was first described for a set of synchronous periodic tasks (i.e., all tasks share the same start (release) time) by Liu and Layland [21], whose paper is considered a milestone in the field of real-time scheduling. In particular, any synchronous periodic task set, with deadlines equal to their respective periods, is feasibly scheduled by EDF *if* and *only if* the *processor utilization* $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is not larger than 1. [1] The optimality is given by the fact that the condition is necessary for any algorithm. The condition is also sufficient for feasibility under EDF scheduling.

The optimality result (not the feasibility condition, though) was later extended to asynchronous periodic task sets, with $D_i \leq T_i$ for any i, by Labetoulle [17]

[1] The proof is given in a later section.

where D_i is a relative deadline. Another result is the proof of Dertouzos [7], in which the optimality of the EDF scheduling algorithm (Theorem 3.2) is shown for tasks with:

- arbitrary release times and deadlines, and
- arbitrary and unknown (to the scheduler) execution times.

Theorem 3.2 (Dertouzos) *The EDF algorithm is optimal in that if there exists any algorithm that can build a valid (feasible) schedule on a single processor, then the EDF algorithm also builds a valid (feasible) schedule.*

Proof. By using a "time slice swapping" technique, it can be shown that any valid schedule for the task set can be transformed into a valid EDF schedule. In particular, by induction on t, the transformation is shown for any interval $[0, t)$ (note that all the parameters of the problem are assumed to be integers).

The theorem is trivially true for $t = 0$. Assume now that it is true for the interval $[0, t)$, that a task's instance with absolute deadline d_j is executed in the interval $[t, t+1)$, and that the earliest deadline among all instances pending at time t is $d_i < d_j$. Let t' be the first time at which the instance with deadline d_i is executed after t. By definition, $t < t'$. Furthermore, since this is a valid schedule, $t' < d_i < d_j$. It follows that by swapping the executions in the intervals $[t, t+1)$ and $[t', t'+1)$, a valid EDF schedule is obtained in the interval $[0, t+1)$. □

Note that with a similar "time slice swapping" technique, the Least Laxity First (LLF) algorithm was also proven optimal by Mok [23]. However, the LLF algorithm has the disadvantage of a potentially very large number of preemptions, and it is no longer optimal if preemption is not allowed [10].

When preemption is not allowed the scheduling problem is known to be NP-hard. However, if only *non-idling* schedulers are considered, the problem is again tractable. Namely, a scheduler is non-idling if it is not allowed to leave the processor idle whenever there are pending jobs. In this subclass of non-preemptive schedulers, the EDF algorithm is optimal.

This result was first shown by Kim and Naghibzadeh [16], who term the non-preemptive non-idling EDF algorithm as the *relative urgency non-preemptive* (RUNP) strategy.

Theorem 3.3 (Kim and Naghibzadeh) *The RUNP strategy is an optimal non-preemptive scheduling strategy in the sense that if a system runs without a task overrun under any non-preemptive scheduling strategy, the system also runs without a task overrun under the RUNP strategy.*

The theorem is proven assuming systems of sporadic tasks, with relative deadlines equal to the respective minimum interarrival times ($D_i = T_i$ in our notation). The generalization to more general task sets, that is, the equivalent of Dertouzos' theorem, is due to George et. al. [9].

Theorem 3.4 (George et. al.) *Non-preemptive non-idling EDF is optimal.*

Proof. By using a "swapping" technique, it is shown that any finite valid schedule can be transformed into a valid EDF schedule. In particular, let t_1 and t_2 be the execution start times of two successive jobs in the valid schedule. Assume both jobs were pending at t_1, and $d_1 > d_2$, where d_1 and d_2 are the absolute deadlines of the two jobs, respectively. That is, at t_1 the scheduler has not chosen the job with the earliest deadline.

If the executions of the two jobs are swapped, the resulting schedule is still valid, since due to the condition $d_1 > d_2$ the two jobs are still completed by their respective deadlines, while the rest of the schedule remains unchanged. In a finite number of such swaps the schedule is transformed into a valid EDF schedule. □

What Theorems 3.2 and 3.4 show is that the EDF algorithm theoretically dominates any other in the field of real-time uni-processor scheduling when there is no system overload and all jobs are independent. Other practical considerations may reduce its advantages, but its optimality still makes it one of the best choices for real-time system designers.

It is also worth remarking that the EDF algorithm is not only optimal in the sense of Theorems 3.2 and 3.4, but also "under various stochastic conditions" [28, 11]; but these are not treated in this book. Furthermore, because of EDF optimality, in the following sections the terms "feasibility" and "feasibility under EDF scheduling" are used interchangeably.

3.2 FEASIBILITY ANALYSIS

This section focuses on techniques for the assessment of task set feasibility under EDF scheduling. To assess the feasibility of a task set means to establish whether the task deadlines are going to always be met.

Historically, the first feasibility analysis for synchronous periodic task sets was given by Liu and Layland [21]. Afterwards, new approaches have been introduced in order to relax some of the assumptions and to analyze more complex task sets.

At present, sophisticated analysis procedures exist that are able to precisely assess the feasibility of "almost" any task set in pseudo-polynomial time. Whether the problem can be solved in fully polynomial time is still an open question [2].

In this section, the most important results concerning the feasibility analysis of real-time task sets are described. The discussion spans from the relatively simple model of synchronous periodic task sets, with relative deadlines equal to their respective periods, to more complex models in which extensions like deadlines not related to task periods, release jitter, and system overheads are taken into account. The analysis of non-preemptive non-idling systems is also described.

3.2.1 The Notion of Loading Factor

Two concepts are particularly helpful when analyzing the feasibility of real-time task (job) sets: the *processor demand* and the *loading factor*. The processor demand is a focused measure of how much computation is requested, with respect to timing constraints, in a given interval of time, while the loading factor is the maximum fraction of processor time possibly demanded by the task (job) set in any interval of time.

Definition 3.1 *Given a set of real-time jobs and an interval of time* $[t_1, t_2)$, *the* processor demand *of the job set on the interval* $[t_1, t_2)$ *is*

$$h_{[t_1,t_2)} = \sum_{t_1 \le r_k, d_k \le t_2} C_k.$$

$u_{[3,18)}$	=	$\frac{4+4+6}{15} = \frac{14}{15}$
$u_{[5,12)}$	=	$\frac{4}{7}$
$u_{[5,14)}$	=	$\frac{4+6}{9} = \frac{10}{9}$
$u_{[6,14)}$	=	$\frac{6}{8}$
u	=	$\frac{10}{9}$

Table 3.1 Loading factor computation for the job set of Figure 3.1.

Namely, the processor demand on $[t_1, t_2)$ represents the amount of computation time that is requested by all jobs with release time at or after t_1 and deadline before or at t_2.

Definition 3.2 *Given a set of real-time jobs, its* loading factor *on the interval* $[t_1, t_2)$ *is the fraction of the interval needed to execute its job, that is,*

$$u_{[t_1,t_2)} = \frac{h_{[t_1,t_2)}}{t_2 - t_1}.$$

and its

Definition 3.3 Absolute loading factor, *or simply* loading factor, *is the maximum of all possible intervals, that is,*

$$u = \sup_{0 \le t_1 < t_2} u_{[t_1,t_2)}.$$

In other words, a job set (*i.e.*, a set of task instances) has loading factor u if in each interval of time $[t_1, t_2)$ the maximum amount of demanded cpu time is at most $u(t_2 - t_1)$. For example, the job set of Figure 3.1 has loading factor $u = \frac{10}{9}$, as shown in Table 3.1. Note that only the computation of the loading factor on the most significant intervals is shown. It is easy to verify that the loading factor on any other interval is less than those shown in Table 3.1.

Intuitively, a condition necessary for the feasibility under any scheduling algorithm is that the loading factor is not greater than 1. In fact, not only is this claim true, but the condition is also sufficient for feasibility under EDF scheduling [25].

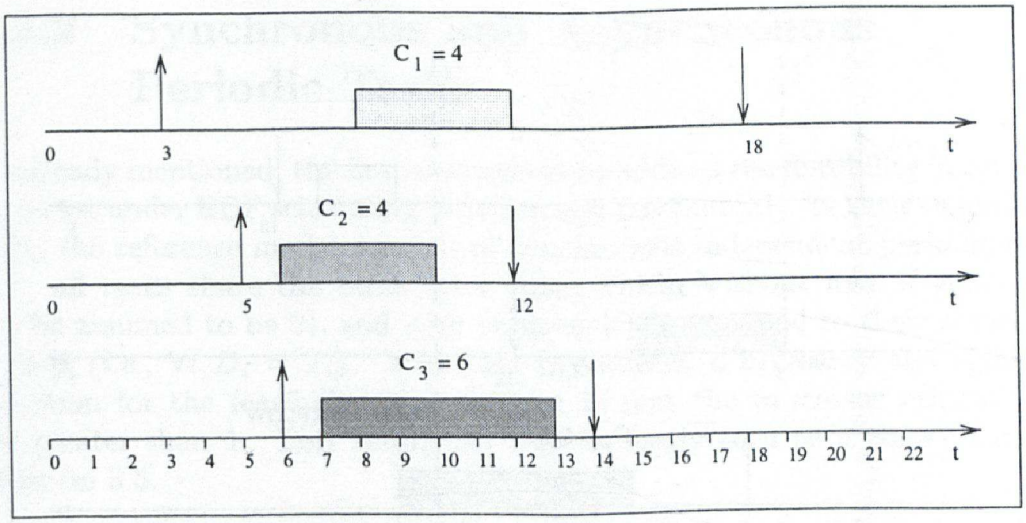

Figure 3.1 A set of three real-time jobs.

Theorem 3.5 (Spuri) *Each set of real-time jobs is feasibly scheduled by EDF* if *and* only if

$$u \leq 1.$$

Proof. "If": Assume there is an overflow, that is, a deadline miss, at time t. The overflow must be preceded by a cpu busy period, that is, a period of continuous processor utilization, in which only jobs with deadlines less than t are executed. Let $t_2 = t$ and t_1 be the last instant preceding t such that there are no pending execution requests of jobs released before t_1 and having deadlines less than or equal to t. Both t_1 and t_2 are well defined. See Figure 3.2 for an example. In particular, after t_1, which must be the release time of some job, the processor is allocated to jobs released after t_1 and having deadlines less than t_2. Since there is an overflow at t_2, the amount of cpu time demanded in the interval $[t_1, t_2)$ must be greater than the interval itself, that is

$$\sum_{t_1 \leq r_k, d_k \leq t_2} C_k > (t_2 - t_1).$$

It follows that

$$u_{[t_1,t_2)} > 1,$$

hence

$$u > 1,$$

a contradiction.

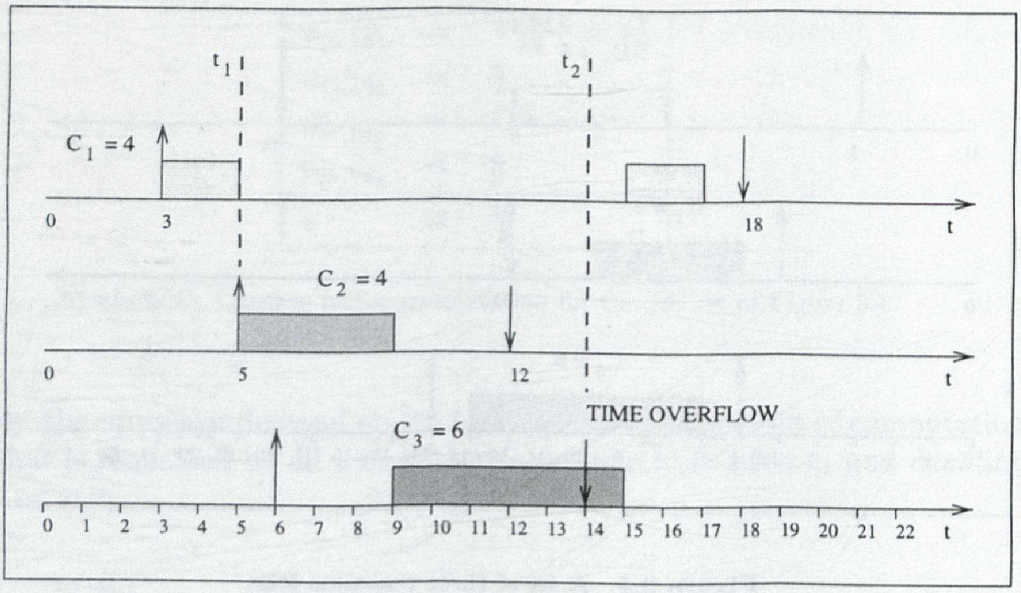

Figure 3.2 EDF schedule (with time overflow) of the job set of Figure 3.1.

"Only If": Since the schedule is feasible, the amount of time demanded in each interval of time must be less than or equal to the length of the interval, that is

$$\forall [t_1, t_2), \sum_{t_1 \leq r_k, d_k \leq t_2} C_k \leq (t_2 - t_1).$$

It follows that

$$u_{[t_1,t_2)} \leq 1,$$

hence

$$u \leq 1.$$

□

Note that the result just shown also confirms the optimality of the EDF algorithm for uni-processor systems. From this point of view the theorem is equivalent to that by Dertouzos, previously described in Section 3.1. The main outcome of the theorem, however, is that the problem of assessing the feasibility of a task set is equivalent to the problem of computing the loading factor of the same task set. This fact is used in the following sections to show several other results.

3.2.2 Synchronous and Asynchronous Periodic Tasks

As already mentioned, the first researchers to address the feasibility analysis of a task set under EDF scheduling were Liu and Layland [21]. In their remarkable work, the reference model consists of synchronous independent periodic tasks (*i.e.*, all tasks share the same start time, which, without loss of generality, can be assumed to be 0), and with relative deadlines equal to their respective periods (*i.e.*, $\forall i, D_i = T_i$). With this hypothesis, a necessary and sufficient condition for the feasibility of a task set is that the processor utilization is not greater than 1. This result can now be easily seen as a consequence of Theorem 3.5.

Corollary 3.1 (Liu and Layland) *Any set of n synchronous periodic tasks with processor utilization* $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ *is feasibly scheduled by EDF* if *and* only if

$$U \leq 1.$$

Proof. It is sufficient to show that the loading factor u of the task set (*i.e.*, the loading factor of the instances generated by the task set) is equal to U. By Theorem 3.5 the thesis then follows.

For any interval $[t_1, t_2)$:

$$\sum_{t_1 \leq r_k, t_2 \leq d_k} C_k \leq \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1) \sum_{i=1}^{n} \frac{C_i}{T_i},$$

that is,

$$u_{[t_1, t_2)} \leq U.$$

Now, let $t_1 = 0$ and $t_2 = \mathrm{lcm}(T_1, \ldots, T_n)$:

$$u_{[t_1, t_2)} = \frac{\sum_{i=1}^{n} \frac{t_2}{T_i} C_i}{t_2} = U.$$

It follows

$$u = U.$$

□

In the previous corollary it is assumed that all the periodic tasks have a null initial phasing, that is, for each task τ_i, the first instance is released at time

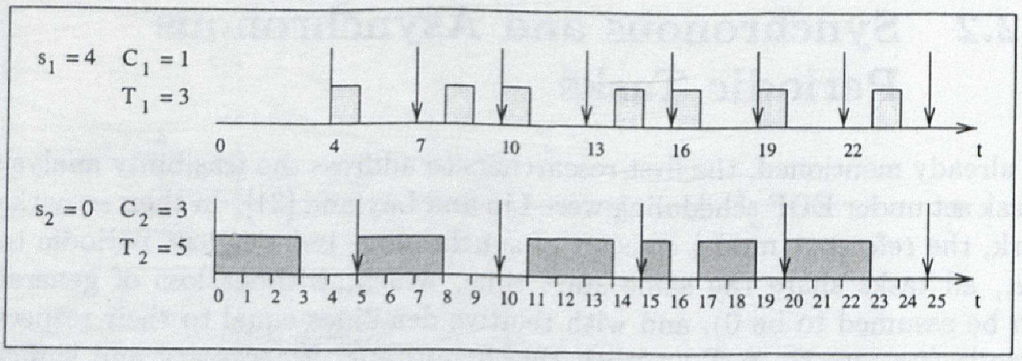

Figure 3.3 EDF schedule of two periodic tasks with different initial phasing.

$t = 0$. Liu and Layland's result is very simple and efficient to use, however, in actual systems it may not be practical to start all periodic tasks at the same time. A first relaxation to their reference model is thus to let each task have its own *start time*. Namely, each task τ_i is allowed to have a start time s_i, the time at which the first job of the task is released. In this way, the k^{th} job of τ_i, $J_{i,k}$, has release time

$$r_{i,k} = s_i + (k-1)T_i$$

and deadline

$$d_{i,k} = s_i + (k-1)T_i + D_i,$$

where d_i is the deadline of task i. Again, such task sets where the first instance has a non-zero start time are termed *asynchronous*. It is still assumed that $\forall i, D_i = T_i$. See Figure 3.3 for an example of EDF scheduling with different initial phasing.

Coffman [6] showed that the condition of having a processor utilization not exceeding 1 is still sufficient for the feasibility under EDF of asynchronous periodic task sets. His result, too, can be seen as a consequence of Theorem 3.5.

Corollary 3.2 (Coffman) *Any set of n asynchronous periodic tasks with processor utilization* $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$, *is feasibly scheduled by EDF* if *and* only if

$$U \leq 1.$$

Proof. Once again, it is sufficient to show that $u = U$. For any interval $[t_1, t_2)$:

$$\sum_{t_1 \leq r_k, t_2 \geq d_k} C_k \leq \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1) \sum_{i=1}^{n} \frac{C_i}{T_i},$$

that is,

$$u_{[t_1,t_2)} \leq U.$$

Consider now the interval $[0, s + mH)$, where m is an integer greater than 0, $s = \max\{s_1, \ldots, s_n\}$, and $H = \text{lcm}(T_1, \ldots, T_n)$:

$$h_{[0,s+mH)} \geq \sum_{i=1}^{n} \frac{mH}{T_i} C_i = mH \sum_{i=1}^{n} \frac{C_i}{T_i} = mHU.$$

It follows

$$u_{[0,s+mH)} \geq \frac{mH}{s + mH} U,$$

and for arbitrary m

$$u \geq U,$$

from which it can be finally concluded that $u = U$. By Theorem 3.5, the thesis follows. □

The importance of this result is that the feasibility condition introduced by Liu and Layland can be efficiently used in asynchronous systems. That is, as long as deadlines are equal to periods, the feasibility of task set can be assessed by simply computing its processor utilization, whatever the start times of the tasks are.

A further natural relaxation for the reference model thus far introduced is to let the relative deadline of each task be different from the task period. That is, deadlines and periods may not be necessarily related. In fact, in actual systems it may be useful to specify deadlines shorter than periods, in order to improve the responsiveness of a given task, or to enforce a minimum time gap between two consecutive instances of the same task. Other times, it may be useful to assign deadlines longer than periods, if responsiveness is not a requirement and a maximum processor utilization is needed. If deadlines are longer than periods note that two or more instances of the task may be active at the same time. The runtime system must be designed to handle this situation.

Unfortunately, these relatively simple extensions to the reference model make the feasibility analysis much more difficult. Indeed, even if attention is restricted to task sets in which $\forall i, D_i \leq T_i$, Leung and Merrill [19] proved that the analysis becomes NP-hard.

Theorem 3.6 (Leung and Merrill) *Deciding if an asynchronous periodic task set, when deadlines are less than the periods, is feasible on one processor is NP-hard.*

The proof of the theorem is given by showing a polynomial reduction of the Simultaneous Congruences Problem to the given feasibility decision problem. The Simultaneous Congruences Problem is shown to be NP-complete by Leung and Whitehead [20].

Further, Baruah et. al. [2] show that the Simultaneous Congruences Problem is NP-complete in the strong sense which means that the feasibility decision problem is even harder than initially proven. This extra difficulty is shown in the following theorem.

Theorem 3.7 (Baruah et. al.) *The problem of deciding whether an asynchronous periodic task set when deadlines are less than the periods is feasible on one processor is NP-hard in the strong sense.*

The importance of this negative result is that it precludes the existence of pseudo-polynomial time algorithms for the solution of this feasibility decision problem, unless P=NP. In fact, the problem remains NP-hard in the strong sense even if the task sets are restricted to have processor utilization bounded above by any fixed positive constant.

Asynchronous task sets (which are defined as having known start times for the tasks) are termed *complete* by Baruah et. al. [2]. Complete task sets are in contrast to *incomplete* task sets where start times are not specified. According to Baruah et. al., an incomplete task set is feasible if there is some choice of start times such that the resulting complete task set is feasible.

So far, it has been shown that the feasibility analysis of complete task sets is a very difficult problem. As can be easily guessed, the analysis of incomplete task sets is exponentially more difficult.

Theorem 3.8 (Baruah et. al.) *The problem of deciding whether an incomplete task set is feasible on one processor is $\sum_2^P$-complete.*

According to the notation of Garey and Johnson [8], the class $\sum_2^P$ is the class of problems $\mathrm{NP}^{\mathrm{NP}}$.

The difficulty of this problem can be circumvented if the feasibility of incomplete task sets is defined in different terms. In fact, due to the strictness of timing constraints, real-time system design is usually based on a worst case

analysis. If it is assumed that the actual start times are fixed by the run-time system, and hence, they are not known *a priori*, it seems reasonable to determine the feasibility of the task set in any possible scenario.

Definition 3.4 *An incomplete task set is feasible if for any choice of start times the resulting complete task set is feasible.*

With this new definition, the feasibility problem for incomplete task sets is slightly simplified. In fact, it turns out that as our intuition suggests, the most constraining scenario is when all tasks share the same start time. This permits the restriction of attention to synchronous task sets. The proof of this fact, which is also common to *sporadic* task sets, is given in the following section.

3.2.3 Sporadic Tasks

Not all the activities of a real-time system can be modeled with strictly periodic tasks. Some tasks can be activated by external events or anomalous situations, which do not necessarily occur at a fixed rate. Thus, it is necessary to introduce into the reference model some form of *aperiodic* task, that is, a task released irregularly.

The introduction of aperiodic tasks in a hard real-time system must be subject to some form of restriction such as a maximum rate. If a guarantee on the execution of periodic task instances is still desired, as well as a deterministic responsiveness of the aperiodic workload, the computational bandwidth demanded by the aperiodic tasks must be restricted in some way. The goal is achieved by using the notion of *sporadic* tasks, a term introduced by Mok [23], although the concept was already known earlier (see for example [16]).

Without ambiguity, the minimum interarrival time of a sporadic task τ_i is denoted T_i, as is the period of a periodic task. Similarly, C_i and D_i denote its worst case execution time and its relative deadline, respectively. Unless otherwise stated, no particular relation is assumed between periods and corresponding deadlines, which are thus arbitrary.

According to the definition, the k^{th} instance of τ_i has release time

$$r_{i,k} \geq r_{i,k-1} + T_i$$

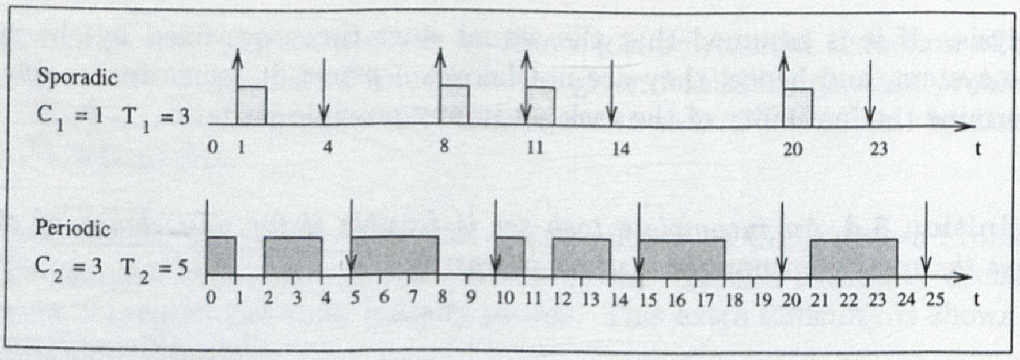

Figure 3.4 EDF schedule of two tasks: a sporadic one and a periodic one.

and deadline

$$d_{i,k} = r_{i,k} + D_i.$$

See Figure 3.4 for an example of an EDF schedule with one sporadic task and one periodic task.

Since the release time of any sporadic task instance is not known *a priori*, in order to guarantee the feasibility under any possible scenario, the definition of feasibility must take all situations into account.

Definition 3.5 *A sporadic task set is feasible if for any choice of release times compatible with the specified minimum interarrival times, the resulting job set is feasible.*

Note that this definition is quite similar to Definition 3.4, in that both refer to all possible situations allowed by the problem specifications. Note also that according to Dertouzos' Theorem, the EDF algorithm is also optimal for sporadic task sets.

Even if the feasibility of a sporadic task set may seem hard to study, it is not difficult to see that the analysis can be limited to a particular set of task instances [3]. In particular, it turns out that the worst possible scenario occurs when each sporadic task τ_i is activated synchronously (*i.e.*, all tasks share the same start time, 0 without loss of generality) and at its maximum rate, that is, it behaves like a synchronous periodic task with period T_i. This is stated formally in the following lemma.

Lemma 3.1 *Given a set of n sporadic tasks τ, and the corresponding synchronous periodic task set τ', for any set of instances of tasks in τ:*

$$u \leq u'.$$

Proof. It is sufficient to prove that for any set of instances and $\forall [t_1, t_2)$, there exists an interval $[t_1', t_2')$ such that $h_{[t_1,t_2)} \leq h'_{[t_1',t_2')}$. For any interval $[t_1, t_2)$:

$$h_{[t_1,t_2)} \leq \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) C_i.$$

Let $t_1' = 0$ and $t_2' = t_2 - t_1$:

$$\begin{aligned} h'_{[t_1',t_2')} &= \sum_{D_i \leq t_2' - t_1'} \left(1 + \left\lfloor \frac{t_2' - t_1' - D_i}{T_i} \right\rfloor\right) C_i \\ &= \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) C_i. \end{aligned}$$

It follows:

$$h_{[t_1,t_2]} \leq h'_{[t_1',t_2')}.$$

□

The lemma proves that among all sets of instances possibly issued by a sporadic task set, those with synchronous periodic releases have the maximum loading factor. By Theorem 3.5, the feasibility of a sporadic task set is thus equivalent to the feasibility of the corresponding synchronous periodic task set, as previously addressed. This proves the following lemma.

Lemma 3.2 *A sporadic task set is feasible* if *and* only if *the corresponding synchronous periodic task set is feasible.*

Note that Lemma 3.1 applies also to incomplete periodic task sets, whose feasibility, as defined in Definition 3.4, can thus be similarly analyzed by looking at the corresponding synchronous complete periodic task sets.

Lemma 3.3 *An incomplete task set is feasible* if *and* only if *the corresponding synchronous periodic task set is feasible.*

According to the previous lemmas, the feasibility analysis of periodic and sporadic task sets can be unified, since in both cases it is reduced to the feasibility analysis of synchronous periodic task sets. In the following sections, all results are thus discussed with respect to *hybrid* task sets (*i.e.*, containing both periodic and sporadic tasks).

3.2.4 The Processor Demand Approach

Unfortunately, whether the problem of deciding the feasibility of a hybrid task set is tractable, is still an open question. In fact, while polynomial or pseudo-polynomial time solutions are known for particular cases, it is also not clear whether the problem in its general formulation is NP-hard [2, 3].

A clear and intuitive necessary condition for the feasibility of any hybrid task set is that the processor utilization is not larger than 1 [3].

Theorem 3.9 (Baruah et. al.) *If a given hybrid task set is feasible under EDF scheduling, then*

$$U \leq 1.$$

Proof. Consider the synchronous periodic task set corresponding to the given hybrid task set. If the processor utilization is shown to be not greater than its loading factor, by Theorem 3.5 the thesis follows.

Let $H = \text{lcm}(T_1, \ldots, T_n)$ and m be an integer greater than zero. In the interval $[0, mH + D_{\max})$, where $D_{\max}$ is the maximum relative deadline, the processor demand is

$$h_{[0,mH+D_{\max})} \geq \sum_{i=1}^{n} \frac{mH}{T_i} C_i = mHU,$$

from which

$$u_{[0,mH+D_{\max})} \geq \frac{mH}{mH + D_{\max}} U.$$

With m arbitrary, it follows

$$u \geq U.$$

□

It is not difficult to see that the condition of the previous theorem is no longer sufficient for the feasibility of generic hybrid task sets. But, it is sufficient when relative deadlines are not shorter than the corresponding periods [3].

Theorem 3.10 (Baruah et. al.) *Given any hybrid task set τ with $D_i \geq T_i, \forall i$, τ is feasible under EDF scheduling* if *and* only if $U \leq 1$.

Proof. Consider the synchronous periodic task set corresponding to τ, and let $[t_1, t_2)$ be any interval of time. The processor demand in the interval is

$$\begin{aligned} h_{[t_1,t_2)} &\leq \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) C_i \\ &\leq \sum_{D_i \leq t_2 - t_1} \left(\frac{t_2 - t_1}{T_i} - \frac{D_i - T_i}{T_i}\right) C_i \\ &\leq (t_2 - t_1) \sum_{D_i \leq t_2 - t_1} \frac{C_i}{T_i} \\ &\leq t_2 - t_1, \end{aligned}$$

from which $u \leq 1$. By Theorem 3.5 the thesis follows. □

When dealing with generic hybrid task sets, the approach of the previous theorem leads only to a sufficient (but not necessary) condition for feasibility under EDF scheduling.

Theorem 3.11 *Given any hybrid task set τ, if*

$$\sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \leq 1,$$

then τ is feasible under EDF scheduling.

Proof. Consider the synchronous periodic task set corresponding τ, and let $[t_1, t_2)$ be any interval of time. The processor demand in the interval is

$$\begin{aligned} h_{[t_1,t_2)} &\leq \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) C_i \\ &\leq \sum_{D_i \leq t_2 - t_1} \left(1 + \frac{t_2 - t_1 - \min\{D_i, T_i\}}{\min\{D_i, T_i\}}\right) C_i \\ &\leq \sum_{D_i \leq t_2 - t_1} \frac{t_2 - t_1}{\min\{D_i, T_i\}} C_i \end{aligned}$$

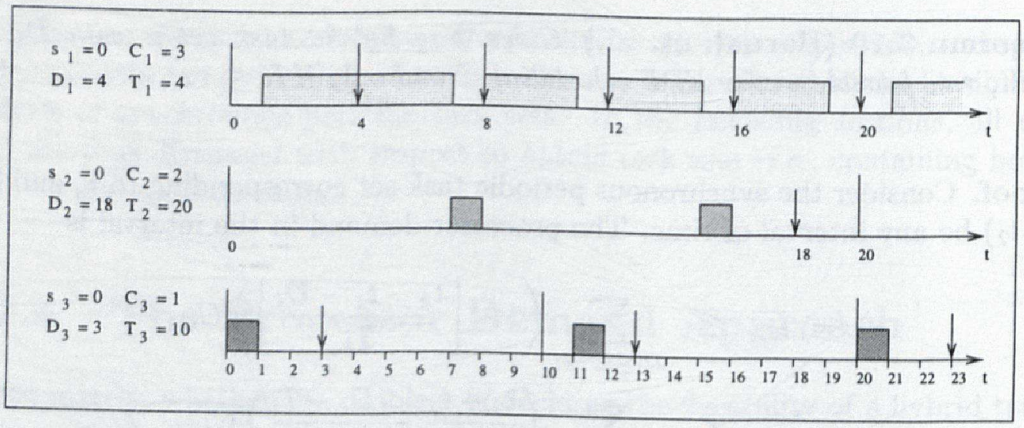

Figure 3.5 EDF schedule of periodic tasks with deadlines different from their periods.

$$\leq (t_2 - t_1) \sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}}$$
$$\leq t_2 - t_1,$$

from which $u \leq 1$. By Theorem 3.5 the thesis follows. □

An example of an EDF schedule with deadlines different from the corresponding periods is depicted in Figure 3.5. Note that in the example, the condition of the previous theorem does not hold, since $\sum_i \frac{C_i}{\min\{D_i,T_i\}} = \frac{3}{4} + \frac{2}{18} + \frac{1}{3} = \frac{43}{36} > 1$, however, the task set is feasible, as can be clearly seen.

In order to develop a procedure for the feasibility assessment of generic hybrid task sets, Theorem 3.5 can be reformulated. It has been shown that the feasibility of a hybrid task set is equivalent to the feasibility of the corresponding synchronous periodic task set. For any such set, the ratio between the processor demand and the length of the relative interval is maximized in the intervals $[0, t)$, for any t greater than zero[2].

The processor demand of a synchronous periodic task set in the interval $[0, t)$, denoted for simplicity as $h(t)$ from now on, is

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i.$$

[2]As in Lemma 3.1, it is simple to see that $h_{[t_1,t_2)} \leq h_{[0,t_2-t_1)}$, for any interval $[t_1, t_2)$.

The condition $u \leq 1$, necessary and sufficient for the feasibility of the task set, is thus equivalent to $h(t) \leq t, \forall t$. The condition is formally stated in the following theorem.

Theorem 3.12 *Any given hybrid task set is feasible under EDF scheduling* if *and* only if

$$\forall t, h(t) \leq t.$$

Unfortunately, testing the processor demand on any interval $[0, t)$ is not practical. However, Baruah et. al. [2, 3] show that it is a valuable approach to find pseudo-polynomial solutions for the feasibility problem. In fact, they show that it is sufficient to test the processor demand for a finite number of intervals, which gives a pseudo-polynomial complexity in "most of the cases."

Theorem 3.13 (Baruah et. al.) *If the given hybrid task set is not feasible and* $U < 1$, *then* $h(t) > t$ *implies* $t < D_{\max}$ *or* $t < \frac{U}{1-U} \max_{i=1,\ldots,n}\{T_i - D_i\}$.

Proof. Assume $h(t) > t$ and $t \geq D_{\max}$.

$$\begin{aligned} t < h(t) &= \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i \\ &\leq \sum_{i=1}^{n} \left(\frac{t + T_i - D_i}{T_i}\right) C_i \\ &= t \sum_{i=1}^{n} \frac{C_i}{T_i} + \sum_{i=1}^{n} \frac{C_i}{T_i}(T_i - D_i) \\ &\leq tU + \max_{i=1,\ldots,n} \{T_i - D_i\} U, \end{aligned}$$

from which

$$t(1 - U) < U \max_{i=1,\ldots,n} \{T_i - D_i\}.$$

It follows

$$t < \frac{U}{1 - U} \max_{i=1,\ldots,n} \{T_i - D_i\}.$$

□

A valuable consequence of this theorem is that whenever the processor utilization of the hybrid task set is less than or equal to a fixed positive constant c

smaller than 1, then the complexity of evaluating the feasibility of the task set is pseudo-polynomial. In fact, $U \leq c$ implies

$$\frac{U}{1-U} \leq \frac{c}{1-c}.$$

According to the result just shown, the condition of Theorem 3.12 can then be efficiently tested in $O(n \max_{i=1,\ldots,n}\{T_i - D_i\})$. Note that when $U = 1$ the upper bound for t is $H + D_{\max}$ [3], which can lead to an exponential complexity.

A suggestion for a further practical improvement in testing Theorem 3.12 condition is given by Zheng and Shin [31]. Accordingly, since the value of $\lfloor (t - D_i)/T_i \rfloor$ changes only on the set $\{mT_i + D_i : m = 0, 1, \ldots\}$, the inequality $h(t) \leq t$ needs only to be checked on the set $S = \bigcup_{i=1}^{n} S_i$, where $S_i = \{mT_i + D_i : m = 0, 1, \ldots, \lfloor (t_{\max} - D_i)/T_i \rfloor\}$, and $t_{\max}$ is the upper bound on the values to be checked. In their work, $t_{\max}$ is found by stopping the algebraic manipulation of Theorem 3.13 a step earlier, thus obtaining a potentially smaller value

$$t_{\max} = \max \left\{ D_{\max}, \frac{\sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i}\right) C_i}{1-U} \right\}.$$

A third upper bound is similarly obtained by George et. al. [10]:

$$t_{\max} = \frac{\sum_{D_i \leq T_i} \left(1 - \frac{D_i}{T_i}\right) C_i}{1-U}.$$

In all cases, however, the resulting complexity of the feasibility analysis is pseudo-polynomial only when $U \leq c < 1$. Note that this is enough to make the processor demand analysis a practical tool in many situations.

3.2.5 Busy Period Analysis

By using a completely different argument, a further upper bound on the values of t for which the condition of Theorem 3.12 must be evaluated can be determined. Its rationale is found again in the work of Liu and Layland [21], in which it is proven that if a synchronous periodic task set is not feasible, then a deadline is missed in the first period of processor activity, that is, before any processor idle time.

Later, Spuri [25] and Ripoll et. al. [24] independently found that the result also applies to synchronous periodic task sets with $D_i \leq T_i, \forall i$, while the case of generic synchronous periodic task sets is discussed by Spuri [26].

Theorem 3.14 (Liu and Layland) *If a synchronous periodic task set is not feasible under EDF scheduling, then in its schedule there is an overflow without idle time prior to it.*

Proof. The same argument given by Liu and Layland can be applied with the new model. Assume there is an overflow at time t. Let t' be the end of the last processor idle period before t, or 0 if there are none. t' must be the arrival time of at least one instance. If all instances arriving after t' are "shifted" left up to t', the processor workload in the interval $[t', t)$ cannot decrease. Since there was no processor idle time between t' and t, there is no processor idle time after the shift. Furthermore, an overflow still occurs at or before t. By considering the new pattern only from time t' on, the thesis follows. □

The immediate consequence of this theorem is that when checking the feasibility of a hybrid task set, the evaluation of Theorem 3.12's condition can be limited to the interval of time preceding the first processor idle time in the schedule of the corresponding synchronous periodic task set.

Any interval of time in which the processor is not idle is termed a *busy period.* The interval of time preceding the first processor idle time[3] in the schedule of a synchronous periodic task set is termed a *synchronous busy period.* The new upper bound mentioned previously is the length L of the synchronous busy period.

Given a hybrid task set and its corresponding synchronous periodic task set, L can be computed by means of a simple procedure. Given any interval $[0, t)$, the idea is to compare the cumulative workload $W(t)$, i.e., the sum of the computation times of the task instances arriving before time t, with the length of the interval: if $W(t)$ is greater than t then the duration of the busy period is at least $W(t)$. The argument is then recursively applied to $W(t), W(W(t)), \ldots$, until two consecutive values are found equal. Formally, L is the fixed point of the following iterative computation:

$$\begin{cases} L^{(0)} & = & \sum_{i=1}^{n} C_i, \\ L^{(m+1)} & = & W(L^{(m)}), \end{cases} \tag{3.1}$$

[3]Note that a processor idle period can have zero duration, if the last pending instance completes and at the same time a new one is released.

$L^{(0)}$	=	$3+2+1=6$
$L^{(1)}$	=	$W(6)=2\cdot 3+1\cdot 2+1\cdot 1=9$
$L^{(2)}$	=	$W(9)=3\cdot 3+1\cdot 2+1\cdot 1=12$
$L^{(3)}$	=	$W(12)=3\cdot 3+1\cdot 2+2\cdot 1=13$
$L^{(4)}$	=	$W(13)=4\cdot 3+1\cdot 2+2\cdot 1=16$
$L^{(5)}$	=	$W(16)=4\cdot 3+1\cdot 2+2\cdot 1=16$
L	=	16

Table 3.2 Computation of the synchronous busy period length for the task set of Figure 3.5.

where

$$W(t)=\sum_{i=1}^{n}\left\lceil\frac{t}{T_i}\right\rceil C_i,$$

and $L^{(m)}$ is the value computed at the mth step. The computation in Equation (3.1) is stopped when two consecutive values are found equal, that is, $L^{(m+1)}=L^{(m)}$. L is then set to $L^{(m)}$. Accordingly, the value found for L is the smallest positive solution of the equation

$$x=W(x).$$

In Table 3.2 the computation of L for the task set of Figure 3.5 is shown. Note that L is correctly assigned the value 16 and not 19. The reason is that at time $t=16$ all instances arriving earlier are completed. Even if at the same time another instance is released, the situation is like having an idle period of length 0.

Note that the value of L does not depend on the scheduling algorithm, as long as it is non-idling, since neither its definition nor its computation are related to any particular algorithm. Furthermore, the synchronous busy period turns out to be the longest one [26].

It can be easily proven that the sequence $L^{(m)}$ converges to L in a finite number of steps if the overall processor utilization of the task set does not exceed 1 [26] (recall that if this condition does not hold the task set is not feasible).

Lemma 3.4 *If $U\leq 1$, then the computation (3.1) converges in a finite number of steps.*

Proof. Let $H = \text{lcm}(T_1, \ldots, T_n)$.

$$U \leq 1 \quad \Rightarrow \quad W(H) = \sum_{i=1}^{n} \left\lceil \frac{H}{T_i} \right\rceil C_i = H \sum_{i=1}^{n} \frac{C_i}{T_i} \leq H.$$

It follows that $L \leq H$, since $W(t)$ is a non-decreasing function and $W(0^+) > 0$. Furthermore, at each step $L^{(m)}$ is either increased by at least $C_{\min}$ or is unchanged. The final value is thus achieved in a finite number of steps. □

As reported by Ripoll et. al. [24], the new bound L can speed up the feasibility analysis of a task set. However, the worst case complexity of the analysis is not improved. In particular, even with this different approach, the analysis has pseudo-polynomial complexity if the processor utilization is $U \leq c$, with c a fixed positive constant smaller than 1 [26]. In this case:

$$L = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil C_i \leq \sum_{i=1}^{n} \left(1 + \frac{L}{T_i}\right) C_i = \sum_{i=1}^{n} C_i + L \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \sum_{i=1}^{n} C_i + Lc,$$

from which

$$L \leq \frac{\sum_{i=1}^{n} C_i}{1 - c}.$$

Hence, the feasibility analysis has time complexity $O(n \sum_{i=1}^{n} C_i)$, which is, as claimed, pseudo-polynomial. Furthermore, since each step of the iterative formula 3.1 takes $O(n)$ time, also the computation of L takes $O(n \sum_{i=1}^{n} C_i)$ time. This leaves the question of whether there exists a fully polynomial time solution for the feasibility problem as an open question.

3.2.6 Feasibility Analysis Algorithm

A practical algorithm for the feasibility analysis of hybrid task sets can be developed by collecting the results described in the previous sections. The algorithm, whose pseudo-code is reported in Figure 3.6, first checks whether the processor utilization of the given task set is greater than 1. If this is the case, according to Theorem 3.9 the task set is not feasible. Otherwise, the analysis continues by checking the condition of Theorem 3.12, $h(t) \leq t$, on any interval $[0, t)$, with t limited by the minimum among the three upper bounds previously discussed. Only the values corresponding to actual deadlines of the synchronous periodic arrival pattern are taken into account. Recall that if $U \leq c$, with a fixed positive constant smaller than 1, the algorithm has pseudo-polynomial complexity.

Analyze(τ):
  if $U > 1$ **then**
    return("Not Feasible");
  endif
  $t_1 = \max\left\{D_{\max}, \frac{\sum_{i=1}^{n}(1-D_i/T_i)C_i}{1-U}\right\}$; [Zheng and Shin bound]
  $t_2 = \frac{\sum_{D_i \le T_i}(1-D_i/T_i)C_i}{1-U}$; [George et. al. bound]
  L = synchronous busy period length;
  $t_{\max} = \min\{t_1, t_2, L\}$;
  $S = \bigcup_{i=1}^{n}\{mT_i + D_i : m = 0, 1, \ldots\} = \{e_1, e_2, \ldots\}$;
  $k = 1$;
  while $e_k < t_{\max}$
    if $h(e_k) > e_k$ **then**
      return("Not Feasible");
    endif
    $k = k + 1$;
  endwhile
  return("Feasible");

Figure 3.6 Pseudo-code of the algorithm for the feasibility analysis of hybrid task sets.

3.2.7 Extended Task Models

In the reference model of hybrid task sets studied thus far, the tasks are characterized by three timing parameters, namely maximum execution time, relative deadline, and period, or minimum interarrival time if the task is sporadic. However, some systems may require even more complex models (such as release jitter and sporadic periodic tasks) in order to be analyzed. Furthermore, taking into account the additional costs of an actual system implementation is required and in other situations analyzing the feasibility of systems in which preemption is not allowed may be necessary. These aspects are briefly covered in this section.

Release Jitter

Given a hybrid task set, task instances may arrive at any time, assuming periodicity or minimum distance (time between two consecutive arrivals) constraints are respected. However, in an actual system, an arrival must be recognized by a run-time dispatcher, which then places the instance in some run-time queue. The instance is then said to be *released*. The time between an instance arrival and its release is known as *release jitter*. Note that the release of a task instance can also be delayed by other factors, such as communication of tasks executing on nodes in a distributed system.

So far, the analysis has been described with the implicit assumption of null release jitter. In this section this assumption is removed. It is now assumed that after each arrival, any instance of a task τ_i may be delayed for a maximum time J_i before actually being released.

When a task experiences jitter, there can be arrival patterns in which two consecutive *releases* of the same task are separated by an interval of time shorter than T_i. Thus, intuitively, the worst case arrival pattern is one in which all tasks experience their shortest inter-release times at the beginning of the schedule. That is, the first instance of each task is released at time $t = 0$, all others are then released at time $t = \max\{kT_i - J_i, 0\}$, $\forall i$ and $\forall k > 0$. See Figure 3.7 for an example.

Indeed, it can be shown that the maximum loading factor is obtained with this arrival pattern, since the maximum ratio between processor demand and interval length is obtained in the intervals $[0, t)$ of this scenario. The processor

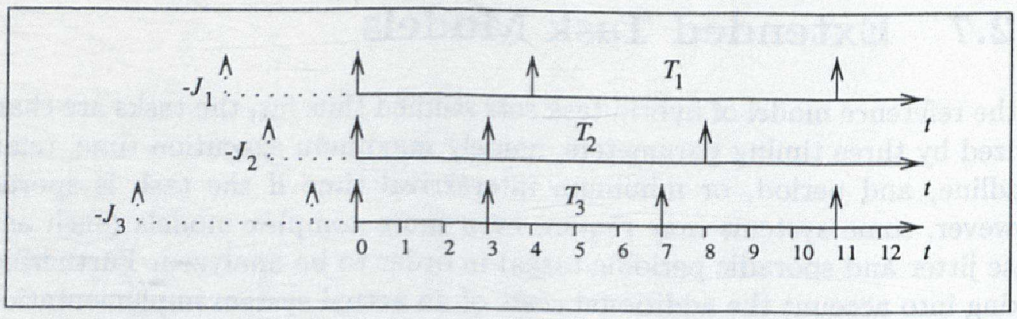

Figure 3.7 Worst case arrival pattern for a task set with release jitter.

demand on such intervals becomes

$$h(t) = \sum_{D_i \leq t+J_i} \left(1 + \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor\right) C_i,$$

since the situation is like having the first instance arrival of any task τ_i at time $t = -J_i$, and all others equally spaced by T_i: all instances arriving at time $t < 0$ are actually released at time $t = 0$. By applying the argument of Theorem 3.13, Baruah, et. al.'s upper bound becomes

$$t_{\max} = \max\left\{ \max_{i=1,\ldots,n} \{D_i - J_i\}, \frac{U}{1-U} \max_{i=1,\ldots,n} \{T_i + J_i - D_i\} \right\}.$$

The upper bounds given by Zheng and Shin, and George, et. al. change in a similar way.

The argument of Theorem 3.14 can still be applied to show that if the task set is not feasible, then a time overflow is found in the initial busy period of the EDF schedule, when the described arrival pattern is considered [26]. The equations concerning the busy period analysis must be modified accordingly.

In particular, the iterative computation (3.1) must be modified in the definition of the cumulative workload $W(t)$, which becomes

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i.$$

The complexity of the feasibility analysis is not affected by the introduction of release jitter in the task model.

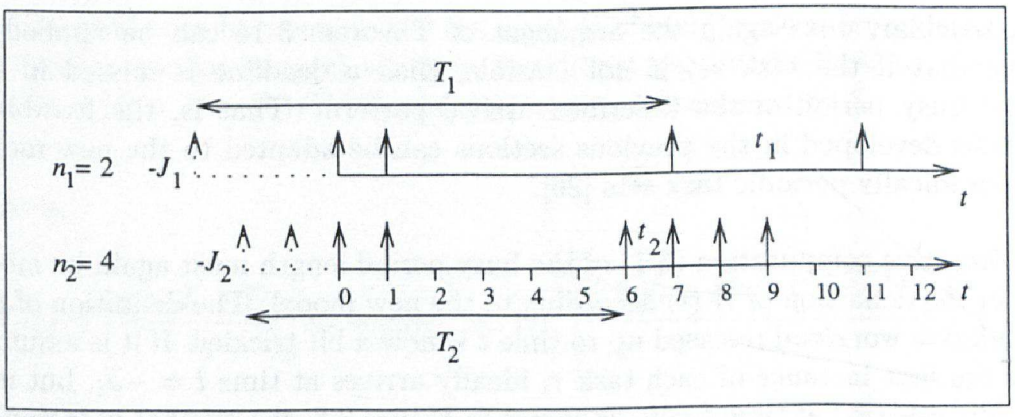

Figure 3.8 Worst arrival pattern of sporadically periodic tasks.

Sporadically Periodic Tasks

The reference model can also be extended by introducing the notion of *sporadically periodic* tasks [1]. These sorts of tasks are intended to model the behavior of events which may arrive at a certain rate for a number of times, and then not re-arrive for a longer time. For example, there are interrupts which behave in this way (they are also termed bursty sporadics). Sporadically periodic tasks are assigned two periods: an *inner period* (t) and an *outer period* (T). The outer period is the worst case inter-arrival time between bursts. The inner period is the worst case inter-arrival time between task instances within a burst. There is a bounded number of arrivals to each burst (n).

It is assumed that for each task τ_i, the total time for any burst (*i.e.*, $n_i t_i$, the number of inner arrivals multiplied by the inner period) must be less than or equal to the outer period T_i. Each instance may suffer a maximum release jitter J_i. "Ordinary" periodic and sporadic tasks, which are not bursty, are simply modeled by assigning inner periods equal to the corresponding outer periods, and by allowing at most one inner arrival.

It can be easily realized that for any given sporadically periodic task set, the worst arrival pattern in terms of processor loading factor is obtained by "packing" as much as possible, the releases of task instances at the beginning of the schedule. An example of such arrival pattern is depicted in Figure 3.8. As previously, it is such that the first instances of all tasks are released at time $t = 0$, and are ideally experiencing their maximum jitter. All the following instances are then released as soon as possible.

In particular, once again the argument of Theorem 3.14 can be applied to prove that if the task set is not feasible, then a deadline is missed in the initial busy period of the described arrival pattern. That is, the feasibility analysis developed in the previous sections can be adapted to the new model of sporadically periodic task sets [26].

The iterative computation (3.1) of the busy period length must again be modified in the definition of $W(t)$ according to the new model. The definition of the cumulative workload released up to time t is now a bit trickier. If it is assumed that the first instance of each task τ_i ideally arrives at time $t = -J_i$, but it is actually released at time $t = 0$, as shown in Figure 3.8, the number of instances of τ_i arrived and released by time t, $I_i(t)$, can be computed as the sum of:

- n_i times the number of outer periods entirely fitting within an interval of $t + J_i$ units of time, and
- the minimum between n_i and the number of inner periods (rounded to the smallest larger integer) which fit in the last part of the interval ($t + J_i - \lfloor (t + J_i)/T_i \rfloor T_i$ wide) preceding t.

That is,

$$I_i(t) = \left\lfloor \frac{t + J_i}{T_i} \right\rfloor n_i + \min \left\{ n_i, \left\lceil \frac{t + J_i - \left\lfloor \frac{t+J_i}{T_i} \right\rfloor T_i}{t_i} \right\rceil \right\}.$$

$W(t)$ then becomes[4]

$$W(t) = \sum_{i=1}^{n} I_i(t) C_i.$$

With a similar argument, the number of instances of task τ_i with deadline before or at t, $H_i(t)$, can also be determined, which is thus the sum of:

- n_i times the number of outer periods entirely fitting within an interval of $t + J_i - D_i$ units of time, and

[4]It is not difficult to see that since the new model is more general than the previous one, the computation of $W(t)$ reduces to those previously shown when used with simpler models, in which tasks do not have bursty behavior nor release jitter.

- the minimum between n_i and the number of inner periods (rounded to the largest smaller integer) which fit in the last part of the interval ($t + J_i - D_i - \lfloor (t + J_i - D_i)/T_i \rfloor T_i$ wide) preceding t, increased by 1.

That is,

$$H_i(t) = \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor n_i + \min \left\{ n_i, 1 + \left\lfloor \frac{t + J_i - D_i - \left\lfloor \frac{t+J_i-D_i}{T_i} \right\rfloor T_i}{t_i} \right\rfloor \right\}.$$

Finally, the processor demand on any interval $[0, t)$ becomes

$$h(t) = \sum_{D_i \leq t+J_i} H_i(t) C_i.$$

The condition found in Theorem 3.12, evaluated in the interval $[0, L)$, can still be utilized to test the feasibility of sporadically periodic task sets. The resulting algorithm is basically similar to that described in Section 3.2.6.

Tick Scheduling

Feasibility analysis can be further extended in order to take into account the costs of an actual EDF preemptive scheduler implementation. From this point of view, the considerations are very similar to those made for fixed priority scheduling. Thus, in what follows the approach is the same as that described by Tindell, et. al. in [29].

According to [29] "Tick scheduling is a common way of implementing a priority preemptive scheduler: a periodic clock interrupt runs a scheduler which polls for the arrivals of tasks; any arrived tasks are placed in a priority ordered run queue. The scheduler then dispatches the highest priority task on the run queue." In the most general case, task instances can arrive at any time, and hence can suffer a worst case release jitter of T_{tick}, the period of the tick scheduler (unless there are periodic tasks with periods which are multiple of T_{tick}).

Normally, the tick scheduler uses two queues: the *pending queue*, which holds a deadline ordered list of tasks awaiting their start conditions, and the *run queue*, a priority-ordered list of runnable tasks. "At each clock interrupt the scheduler scans the pending queue for tasks which are now runnable and transfers them to the run queue." The system overhead that must be taken into account is

the time needed to handle the two queues, and more precisely the time needed to move tasks from one queue to another one.

In particular, like in the work of Tindell, et. al., the following implementation costs are considered:

C_{tick} The worst case computation time of the periodic timer interrupt.

C_{QL} The cost to take the first task from the pending queue.

C_{QS} The cost to take any possible subsequent task from the pending queue.

According to Tindell, et. al.'s analysis, the tick scheduling overheads over a window of width w are

$$OV(w) = T(w)C_{tick} + \min\{T(w), K(w)\}C_{QL} + \max\{K(w) - T(w), 0\}C_{QS}, \tag{3.2}$$

where $T(w)$ is the number of timer interrupts within the window:

$$T(w) = \left\lceil \frac{w}{T_{tick}} \right\rceil$$

and $K(w)$ is the worst case number of times tasks move from the pending queue to the run queue:

$$K(w) = \sum_{i=1}^{n} \left\lceil \frac{w + J_i}{T_i} \right\rceil .$$

In order to extend the feasibility analysis to include this overhead, Theorem 3.14 must be generalized. Again, the generalization is achieved by looking at the paper of Liu and Layland [21]: it is simply sufficient to reformulate their theorem in order to fulfill our needs. This theorem is proven with respect to a task set scheduled by the deadline driven algorithm, in a system in which the processor time is accumulated by a certain *availability function*, that is, only a fraction of the processor time is devoted to the task schedule. The attention of Liu and Layland is on sublinear functions, that is, functions for which for all t and T

$$f(T) \leq f(t+T) - f(t).$$

The reason is that when there is a task set scheduled by fixed priority scheduling and another task set scheduled when the processor is not occupied by tasks of the first set (*i.e.*, in background), then the availability function for the second task set can be shown to be sublinear. The new model, in which all tasks are

scheduled when the processor is not busy executing tick scheduler code, fits perfectly in this description.

Theorem 3.15 (Liu and Layland) *When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor whose availability function is sublinear, if there is an overflow for a certain arrival pattern, then there is an overflow without idle time prior to it in the pattern in which all task instances are released as soon as possible.*

Proof. Similar to that of Theorem 3.14. □

The feasibility analysis must be modified accordingly. The computation of the busy period length must take into account the additional load due to the tick scheduler. Thus the workload arriving by time t becomes[5]

$$W(t) = OV(t) + \sum_{i=1}^{n} \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i.$$

Equation (3.2) can be used to evaluate the availability function:

$$a(t) \geq \max\{t - OV(t), 0\}.$$

A sufficient condition for the feasibility of the task set is then

$$t - OV(t) \geq \sum_{D_i \leq t + J_i} \left(1 + \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor\right) C_i$$

for all absolute deadlines in the initial busy period. Note that the condition is a generalization of Theorem 9 of [21].

Jeffay and Stone [15] extended the analysis to account for interrupt handling costs. In this work, the authors analyze the feasibility of a set of hard deadline tasks which execute in the presence of h interrupt handlers. The interrupt handlers are treated as sporadic tasks running at the highest priority.

Non-Preemptive Non-Idling EDF Scheduling

When preemption is not allowed in the schedule of a task set, the problem becomes much more difficult. If a priority based scheduler like EDF is utilized,

[5]The analysis is described by assuming hybrid task sets with release jitter. The argument can be similarly applied to sporadically periodic task sets [26].

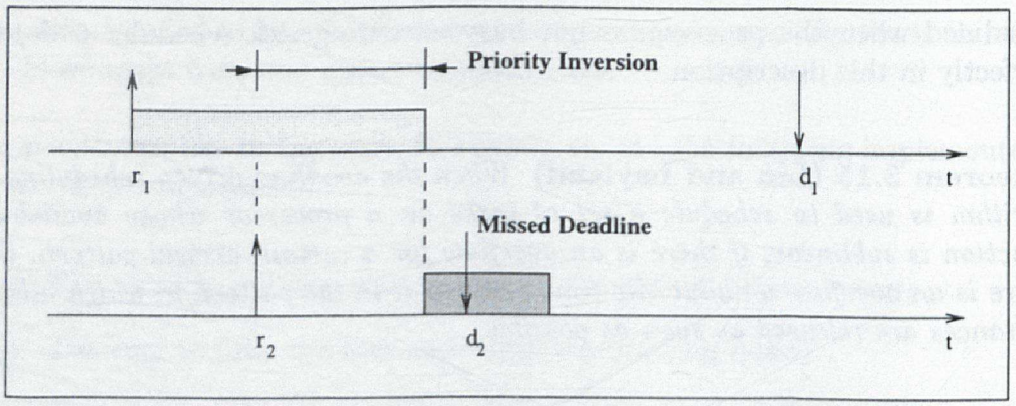

Figure 3.9 Non-preemptive non-idling EDF schedule of two jobs.

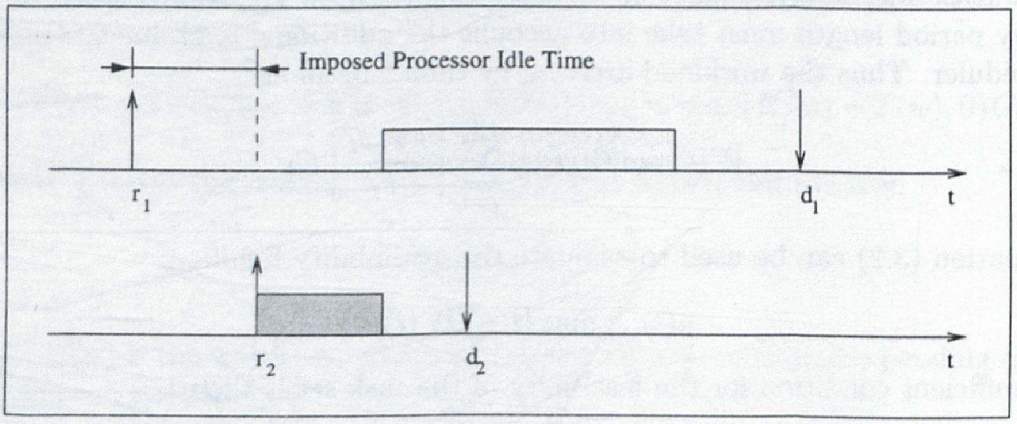

Figure 3.10 Non-preemptive idling schedule of two jobs.

non-preemption may be the source of undesired priority inversions. In Figure 3.9, a typical situation is depicted. When the second job is released at time r_2, the execution of the first job cannot be preempted. The second job is forced to wait for the completion of the first one, even if it has a shorter deadline. In this case the effect of the priority inversion is so bad as to invalidate the schedule, since the second job misses its deadline.

The situation can be improved if the scheduler is allowed to keep the processor idle, even when there are pending jobs. In the example, if the processor is left idle between r_1 and r_2, the second job can be executed first. As shown in Figure 3.10, if such idling decision is taken, no deadline is missed, that is, the schedule is valid.

Unfortunately, as mentioned in Section 3.1, the feasibility problem for idling systems, which are more general than non-idling ones, is NP-hard. Further, Howell and Venkatrao [12] show on one hand that "the decision problem of determining whether a periodic task system is schedulable for all start times with respect to the class of idling algorithms is NP-hard in the strong sense, even when the deadlines are equal to the periods." On the other hand, they also formally prove that there cannot exist an optimal on-line idling algorithm for scheduling sporadic tasks, and that if a sporadic task set is schedulable by an on-line idling algorithm, then it is also schedulable by an on-line non-idling algorithm. These arguments, as well as practical reasons, motivate the choice of restricting the attention only to the suboptimal class of non-idling scheduling algorithms, among which EDF is optimal.

The analysis of task sets scheduled by the non-preemptive non-idling EDF algorithm (termed simply non-preemptive EDF in the following) is not much more difficult than that discussed thus far for preemptive systems. The processor demand and busy period approaches are still useful and can be adapted without much effort. The impact of non-preemption, and hence of priority inversions is, in fact, limited and can be easily taken into account in the equations.

It turns out that any task instance may be subject to at most only a single priority inversion, which can be easily bounded. Assume there is an overflow in the schedule at time t, as shown in Figure 3.11. An argument similar to that of Theorem 3.14 can be applied. Let t' be the last time before t such that there are no pending instances with arrival time earlier than t' and absolute deadline before or at t. By choice, t' must be the arrival time of a task instance, and there is no idle time between t' and t.

A lower priority instance (*i.e.*, with deadline after t) may be executing before t'. If it is not completed by t', owing to its non-preemptability, a priority inversion occurs. However, after its completion only task instances arrived at t' or later, and having deadline before or at t are executed. In the schedule of such instances there may be several priority inversions, but only the first one has an impact on the deadline missed at time t.

If all "higher priority" instances are "packed" to the left, so that the corresponding tasks are released synchronously from time t', an overflow is still found at t or earlier. This proves the following lemma [10].

Lemma 3.5 (George, et. al.) *If a hybrid task set is not feasible under non-preemptive EDF scheduling, then there is an overflow at time t in the initial*

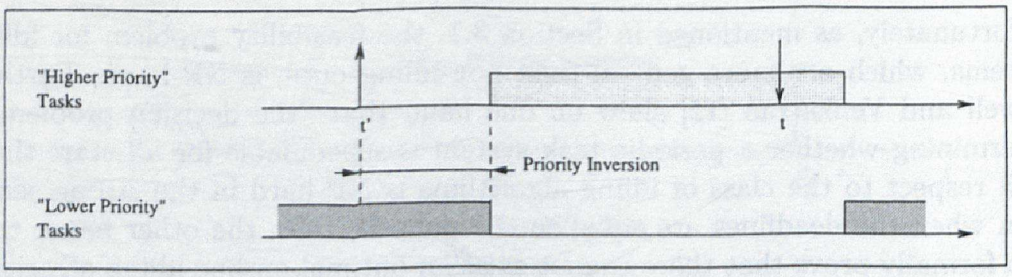

Figure 3.11 Busy period preceding an overflow in a non-preemptive non-idling EDF schedule.

busy period when all tasks with relative deadline less than or equal to t are released synchronously from time 0, and all others, if any, are released one unit of time earlier.

Accordingly, having assumed a discrete scheduling model, for any deadline at time t the maximum penalty introduced by the non-preemption is

$$\max_{D_j>t}\{C_j - 1\},$$

with the convention that the value is zero if $\nexists j : D_j > t$. The processor demand approach must then be corrected by adding this value to $h(t)$. The condition of Theorem 3.12 now becomes

$$\forall t, h(t) + \max_{D_j>t}\{C_j - 1\} \leq t.$$

A similar feasibility condition is shown by Kim and Naghibzadeh [16] for sporadic task sets with deadlines equal to periods, although the assumed scheduling model is continuous. Jeffay, et. al. [14] also prove similar results, but assume periodic and sporadic task sets within a discrete scheduling model. Zheng and Shin [31] extend their feasibility condition for preemptive EDF, described in Section 3.2.4, to non-preemptive EDF. All these results are finally generalized by George, et. al. [10] in the following theorem.

Theorem 3.16 (George, et. al.) *Any hybrid task set with processor utilization $U \leq 1$ is feasible under non-preemptive EDF* if *and* only if

$$\forall t \in S, h(t) + \max_{D_j>t}\{C_j - 1\} \leq t,$$

where

$$S = \bigcup_{i=1}^{n} \left\{ kT_i + D_i, k = 0, \ldots, \left\lfloor \frac{t_{\max} - D_i}{T_i} \right\rfloor \right\},$$

and $t_{\max} = \min\{L, t_1, t_2\}$.

As for the preemptive case, L is the length of the initial busy period in a synchronous arrival pattern. According to Lemma 3.5, any possible deadline miss is found in a busy period. Since L is the length of the longest busy period, it is an upper bound on the absolute deadlines to check. The other upper bounds, t_1 and t_2, are obtained by algebraic manipulations of the condition

$$h(t) + \max_{D_j > t}\{C_j - 1\} > t.$$

In particular,

$$t_1 = \max\left\{ D_{\max}, \frac{\sum_{i=1}^{n} (1 - D_i/T_i)\, C_i}{1 - U} \right\},$$

$$t_2 = \frac{\sum_{D_i \le T_i} (1 - D_i/T_i)\, C_i + \max_{i=1,\ldots,n}\{C_i - 1\}}{1 - U}.$$

The algorithm described in Section 3.2.6 can then be used to check the feasibility of any task set according to Theorem 3.16. Similar to the preemptive model, the complexity is pseudopolynomial whenever $U \le c$, with c a fixed positive constant less than 1.

3.3 SUMMARY

A theory for EDF feasibility analysis has been described in this chapter. In the simplest case, it has been shown that the feasibility of an independent periodic task set can be established by just computing the task set processor utilization U: the task set is then recognized as feasible if and only if $U \le 1$.

If asynchronous periodic tasks with deadlines not necessarily equal to their periods are considered, the problem becomes intractable, since it has been shown to be NP-Hard in the strong sense, which excludes even the existence of a pseudo-polynomial solution, unless P=NP.

In the general case, $U \le 1$ is only a necessary condition, as expected. New interesting techniques based on processor demand and busy period approaches have

been developed to analyze more complex task sets. The result is a pseudo-polynomial solution for the analysis of hybrid task sets, that is task sets including both periodic and sporadic tasks. Whether a fully polynomial solution exists is still an open question.

The algorithm for the pseudo-polynomial solution has been described. Extensions necessary to handle models including release jitter, sporadically periodic tasks and tick schedulers have also been discussed. The description of how to apply the approach to non-preemptive non-idling EDF schedulers has been given.

REFERENCES

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, September 1993.

[2] S.K. Baruah, L.E. Rosier and R.R. Howell, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems* 2, 1990.

[3] S.K. Baruah, A.K. Mok, and L.E. Rosier, "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," *Proc. of IEEE Real-Time Systems Symposium*, 1990.

[4] G.C. Buttazzo and J.A. Stankovic, "RED: A Robust Earliest Deadline Scheduling Algorithm," *Proc. of 3rd Int. Workshop on Responsive Computing Systems*, 1993.

[5] G.C. Buttazzo and J.A. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling," in *Responsive Computer Systems: Toward Integration of Fault Tolerance and Real-Time*, Kluwer Press, 1994.

[6] E.G. Coffman, Jr., "Introduction to Deterministic Scheduling Theory," in E.G. Coffman, Jr., Ed., *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[7] M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing* 74, North-Holland Publishing Company, 1974.

[8] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[9] L. George, P. Muhlethaler, and N. Rivierre, "Optimality and Non-Preemptive Real-Time Scheduling Revisited," Rapport de Recherche RR-2516, INRIA, Le Chesnay Cedex, France, 1995.

[10] L. George, N. Rivierre, and M. Spuri, "Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling," Rapport de Recherche RR-2966, INRIA, Le Chesnay Cedex, France, 1996.

[11] J. Hong, X. Tan, and D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Transactions on Computers*, Vol. 38, No. 12, Dec. 1989.

[12] R.R. Howell and M.K. Venkatrao, "On Non-Preemptive Scheduling of Recurring Tasks Using Inserted Idle Times," *Information and Computation* 117, 1995.

[13] J.R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[14] K. Jeffay, D.F. Stanat, and C.U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. of IEEE Real-Time Systems Symposium*, 1991.

[15] K. Jeffay and D. L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," *Proc. of IEEE Real-Time Systems Symposium*, 1993.

[16] K.H. Kim and M. Naghibzadeh, "Prevention of Task Overruns in Real-Time Non-Preemptive Multiprogramming Systems," *Proc. of Performance 80*, Association for Computing Machinery, 1980.

[17] J. Labetoulle, "Some Theorems on Real-Time Scheduling," *Computer Architectures and Networks*, E. Gelembe and R. Mahl (Eds.), North Holland Publishing Company, 1974.

[18] J.K. Lenstra and A.H.G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Math.* 5, 1977.

[19] J.Y.-T. Leung and M.L. Merrill, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters* 11(3), 1980.

[20] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation* 2, 1982.

[21] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery* 20(1), 1973.

[22] Locke C.D., Vogel D.R., and Mesler T.J., "Building a Predictable Avionics Platform in Ada: A Case Study," *Proc. of IEEE Real-Time Systems Symposium*, 1991.

[23] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. Dissertation, MIT, 1983.

[24] I. Ripoll, A. Crespo, and A.K. Mok, "Improvement in Feasibility Testing for Real-Time Tasks," *Real-Time Systems* 11, 1996.

[25] M. Spuri, "Earliest Deadline Scheduling in Real-Time Systems," Doctorate Dissertation, Scuola Superiore S.Anna, Pisa, Italy, 1995.

[26] M. Spuri, "Analysis of Deadline Scheduled Real-Time Systems," Rapport de Recherche RR-2772, INRIA, Le Chesnay Cedex, France, 1996.

[27] M. Spuri, "Holistic Analysis for Deadline Scheduled Real-Time Distributed Systems," Rapport de Recherche RR-2873, INRIA, Le Chesnay Cedex, France, 1996.

[28] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, June 1995.

[29] K. Tindell, A. Burns, and A.J. Wellings, "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems* 6(2), 1994.

[30] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessors and Microprogramming* 40, 1994.

[31] Q. Zheng and K.G. Shin, "On the Ability of Establishing Real-Time Channels in Point-to-Point Packet-Switched Networks," *IEEE Trans. on Communications* 42(2/3/4), 1994.