Real-Time Systems, 30, 105–128, 2005

Feasibility Analysis of Real-Time Periodic Tasks with Offsets

RODOLFO PELLIZZONI rodolfo@sssup.it
GIUSEPPE LIPARI lipari@sssup.it
Scuola Superiore S. Anna, Pisa, Italy

Abstract. The problem of feasibility analysis of asynchronous periodic task sets, where tasks can have an initial offset, is known to be co-NP-complete in the strong sense. A sufficient pseudo-polynomial test has been proposed by Baruah, Howell and Rosier, which consists in analyzing the feasibility of the corresponding synchronous task set (i.e. all offsets are set equal to 0). If the test gives a positive result, then the original asynchronous task set is feasible; else, no definitive answer can be given. In many cases, this sufficient test is too pessimistic, i.e. it gives no response for many feasible task sets.

In this paper, we present a new sufficient pseudo-polynomial test for asynchronous periodic task sets. Our test reduces the pessimism by explicitely considering the offsets in deriving a small set of critical arrival patterns. We show, trough a set of extensive simulations, that our test outperforms the previous sufficient test.

Keywords: real-time systems, scheduling

1. Introduction

In critical real-time systems it is necessary to guarantee *a priori* that all timing constraints are respected. Usually, these systems are modelled as sets of concurrent periodic real-time tasks. Each task τ_i is characterized by an initial offset ϕ_i, a worst-case execution time C_i, a relative deadline D_i and a period T_i. In general, a task's relative deadline can be different from its period. At run time, tasks are executed on the system by a real-time operating system. The scheduling algorithm selects which task to execute at each time instant, generating a *schedule*.

A *feasible schedule* is a schedule in which all tasks meet their deadlines and any additional specified constraint. A *feasibility test* is an algorithm that, given a task set, returns a positive answer if there exists a feasible schedule. In case of negative answer, no algorithm can generate a feasible schedule. A *schedulability test* is an algorithm that, given a task set and a scheduling algorithm, returns a positive answer if the algorithm generates a feasible schedule.

In single preemptive processor systems, the Earliest Deadline First (EDF) scheduling algorithm is optimal (Dertouzos, 1974), in the sense that: if a task set is feasible, then it is schedulable by EDF. Therefore, the feasibility problem on single processor systems can be reduced to the problem of testing the schedulability with EDF.

The feasibility problem for a set of independent periodic tasks to be scheduled on a single processor has been proven to be co-NP-complete in the strong sense (Leung and Merril, 1980; Baruah et al., 1990b). In Leung and Merril (1980) the authors showed that it

is necessary to analyze all deadlines from 0 to $\Phi + 2H$ where Φ is the largest task offset and H is the hyperperiod (i.e. the least common multiple of all task periods). In Baruah et al. (1990b) it is proved that, when the system utilization $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$ (where N is the number of tasks) is strictly less than 1, the Leung and Merril's condition is also sufficient.

Under certain assumptions, the problem becomes more tractable. For example, if deadlines are equal to periods, a simple polynomial test has been proposed (Liu and Layland, 1973). If the system utilization U is less than or equal to 1, the system is schedulable:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

If the deadlines are less than or equal to the periods and the task set is *synchronous* (i.e. all tasks have initial offset equal to 0), then a pseudo-polynomial test has been proposed (Baruah et al., 1990a, 1990b).

In the case of asynchronous periodic task sets, any known necessary and sufficient feasibility test requires an exponential time to run. However, it is possible to obtain a sufficient test by ignoring the offsets and considering the task set as synchronous. In Baruah et al. (1990b) the authors showed that, given an asynchronous periodic task set $\mathcal{T}$, if the corresponding synchronous task set $\mathcal{T}'$ (obtained by considering all offsets equal to 0) is feasible, then $\mathcal{T}$ is feasible too. However, if $\mathcal{T}'$ is not feasible, no definitive answer can be given on $\mathcal{T}$. In some cases this sufficient test is quite pessimistic, as we will show in Section 5.

The basic idea behind Baruah's result is based on the concept of *busy period*. A busy period is an interval of time where the processor is never idle. If all tasks start synchronously at the same time t, the first deadline miss (if any) must happen in the longest busy period starting from t. When tasks have offsets, it may not be possible for them to start at the same time. Hence, we do not know where the deadline miss might happen in the schedule.

In this paper, a new sufficient pseudo-polynomial feasibility test for asynchronous task sets is proposed. Our idea is based on the observation that the patterns of arrivals of the tasks depend both on the offsets and on the periods of the tasks. By computing the minimum possible distance between the arrival times of any two tasks, we are able to select a small group of critical arrival patterns that generate the worst-case busy period. Our arrival patterns are pessimistic, in the sense that some of these patterns may not be possible in the schedule. Therefore, our test is only sufficient. However, experiments show that our test greatly reduces pessimism with respect to previous sufficient tests.

2. System Model

In this paper, the problem of feasibility analysis of a periodic task set on a single processor is considered. We assume that all task parameters are expressed by integer numbers. Time is divided into slots, numbered starting from 0: $t \in \mathsf{N}$.

The system consists of a set $\mathcal{T}$ of N periodic real time tasks $\tau_1, \ldots, \tau_N$. A task τ_i is characterized by a tuple (ϕ_i, C_i, D_i, T_i) where ϕ_i is the offset, C_i is the worst case computation time, D_i is the relative deadline and T_i is the period. Each task τ_i generates an

infinite number of jobs τ_{ij}, $j \geq 0$. Each job is assigned a release time $r_{ij} = \phi_i + jT_i$ and an absolute deadline $d_{ij} = r_{ij} + D_i$.

A task set $\mathcal{T}$ is said to be *synchronous* if all offsets are equal to zero. A task set is said to be *asynchronous* if it isn't synchronous. Given an asynchronous task set $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, we define the corresponding synchronous task set as $\mathcal{T}' = \{\tau'_1, \ldots, \tau'_N\}$, where for every task τ'_i: $\phi'_i = 0, C'_i = C_i, D'_i = D_i, T'_i = T_i$.

We further define:

1. $U_i = \frac{C_i}{T_i}$ is the utilization of task τ_i;
2. $U = \sum_{i=1}^{N} U_i$ is the total utilization of task set $\mathcal{T}$;
3. $\Phi = \max\{\phi_1, \ldots, \phi_N\}$ is the largest offset;
4. function gcd (T_i, T_j) is the greatest common divisor between two periods T_i and T_j;
5. function lcm $(T_1, \ldots, T_j)$ is the least common multiple among all periods $T_1, \ldots, T_j$;
6. $H = \text{lcm}\{T_1, \ldots, T_N\}$ is the hyperperiod;
7. $\eta_i(t_1, t_2) = (\lfloor \frac{t_2 - \phi_i - D_i}{T_i} \rfloor - \lceil \frac{t_1 - \phi_i}{T_i} \rceil + 1)_0$ is the number of jobs of task τ_i with release time greater than or equal to t_1 and deadline less then or equal to t_2 (Baruah et al., 1990b).

Notation $(x)_0$ is an abbreviation of $\max(x, 0)$.

A schedule is a function $\sigma : \mathsf{N} \rightarrow \mathcal{T} \cup \{\emptyset\}$ that assigns to each time slot t a task τ_i, or the symbol $\emptyset$ to indicate that the processor is idle. A *busy period* $[t_1, t_2)$ is an interval of time in which the processor is always busy.

Synchronization among tasks is carried out through critical sections of code, that uses *shared resources*. The usage of critical sections ensures that all resources are accessed in exclusive mode. To simplify our presentation, only single-unit resources are considered, although there are ways to consider the case of multi-unit resources (Baker, 1991). We consider a set $\mathcal{R}$ of R shared resources $\rho_1, \ldots, \rho_R$. Each task τ_i may access N_i different critical sections. Each critical section ξ_{ij} is described by a 3-ple $(\rho_{ij}, \psi_{ij}, C_{ij})$, where:

1. $\rho_{ij} \in \mathcal{R}$ is the resource being accessed;
2. ψ_{ij} is the earliest time, relative to the release time of job τ_{ij}, that the task can enter ξ_{ij};
3. C_{ij} is the worst-case computation time of the critical section.

Critical sections can be properly nested in any arbitrary way, as long as their earliest entry time and worst-case computation time is known. Note that our model, first proposed in Lipari and Buttazzo (2000), is actually slightly different from the classic one used in the

literature in that it requires earliest entry times to be known. Note that if earliest entry times are unknow, they can simply be set to zero, although this will lead to increased pessimism in the analysis.

In all the figures showing a schedule, we represent the execution of each task on a separate horizontal line. Upward arrows represent release times while downward arrows represent absolute deadlines.

3. Feasibility Analysis

In this section, we will first show the fundamental results for the problem of feasibility analysis of periodic task sets on single processor systems. Then we present our idea and prove it correct. Throughout this section we suppose that tasks are independent, i.e. no resource other than the processor is shared among tasks. An extension to the case of shared resources will be presented in Section 4.

Our analysis is based on the *processor demand criterion* (Baruah et al., 1990b; Buttazzo, 1997). The *processor demand function* is defined as

$$df(t_1, t_2) = \sum_{i=1}^{N} \eta_i(t_1, t_2) C_i.$$

It is the amount of time demanded by the tasks in interval $[t_1, t_2)$ that the processor must execute to ensure that no task misses its deadline. Intuitively, the following is a necessary condition for feasibility:

$$\forall\, 0 \leq t_1 < t_2 :\ df(t_1, t_2) \leq t_2 - t_1.$$

In plain words, the amount of time demanded by the task set in any interval must never be larger than the length of the interval.

3.1. Existing Results on Feasibility Analysis

We report two fundamental results on the schedulability analysis of a periodic task set with EDF. The proofs of these results are different from the original ones. They have been rewritten to clarify our methodology.

Lemma 1 (Baruah et al., 1990b)**.** *Task set $\mathcal{T}$ is feasible on a single processor if and only if*:

1. $U \leq 1$, *and*
2. $\forall\, 0 \leq t_1 < t_2 \leq \Phi + 2H : df(t_1, t_2) \leq t_2 - t_1$.

Proof: Both conditions are clearly necessary. By contradiction. Suppose both conditions hold but $\mathcal{T}$ is not feasible. Consider the schedule generated by EDF. It can be proven (Baruah et al., 1990b; Leung and Merril, 1980) that since $U \leq 1$ and the task set is not feasible, some

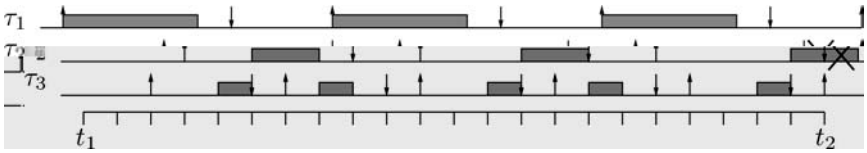


Figure 1. Example of busy period.

deadline in $(0, \Phi + 2H]$ is missed. Let t_2 be the first instant at which a deadline is missed, and let t_1 be the last instant prior to t_2 such that either no jobs or a job with deadline greater than t_2 is scheduled at $t_1 - 1$. By choice of t_1, it follows that $[t_1, t_2)$ is a busy period and all jobs that are scheduled in $[t_1, t_2)$ have arrival times and deadlines in $[t_1, t_2]$ (see Figure 1). Also, at least one job with deadline no later than t_2 is released exactly at t_1, otherwise either a job with deadline greater than t_2 or no job would be scheduled at t_1. Since there is no idle time in $[t_1, t_2)$ and the deadline at t_2 is missed, the amount of work to be done in $[t_1, t_2)$ exceeds the length of the interval. By definition of df, it follows that $df(t_1, t_2) > t_2 - t_1$, which contradicts condition 2. □

By looking at the proof, it follows that it is sufficient to check the values of $df(t_1, t_2)$ for all times t_1 that correspond to the release time of some job. In the same way, we can check only those t_2 that correspond to the absolute deadline of some job.

We will now prove in the following theorem that for a synchronous task set the first deadline miss, if any, is found in the longest busy period starting from $t_1 = 0$ (usually referred to as the *critical instant*). Thus, it suffices to check all deadlines from 0 to the first idle time. The idea is that, given any busy period starting at t_1, we can always produce a "worst-case" busy period by "pulling back" the release times of all tasks that are not released at t_1, so that all tasks are released at the same time. Figures 2 and 3 shows the idea for a task set of $N = 3$ tasks with $\tau_1(C = 4, D = 6, T = 10)$, $\tau_2(C = 4, D = 12, T = 12)$, $\tau_3(C = 4, D = 10, T = 14)$. Figure 2 shows a busy period starting at t_1 with the release of task τ_1, while tasks τ_2 and τ_3 are released at $t_1 + 2$ and $t_1 + 3$ respectively; task τ_3 misses its deadline at $t_2 = t_1 + 27$. Figure 3 shows the busy period obtained by "pulling back" τ_2 and τ_3 until their release times coincide with that of τ_1. Note that the newly obtained busy period

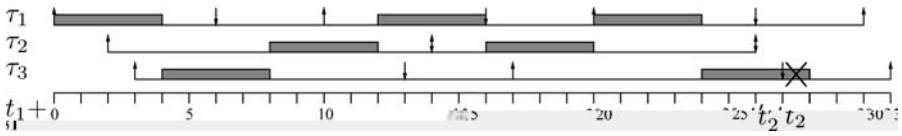


Figure 2. Example: asynchronous task set.

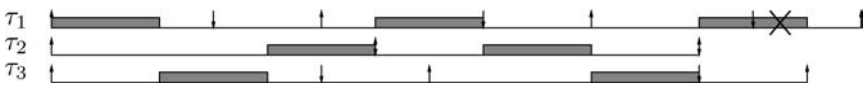


Figure 3. Example: corresponding synchronous task set.

corresponds to the synchronous busy period starting at 0, and that while the second job of τ_3 now finishes at its deadline, there is still a task, in this case τ_1, that misses its deadline at $t_2' = t_1 + 26$.

Theorem 1 (Baruah et al., 1990b)**.** *A synchronous task set $\mathcal{T}$ is feasible on a single processor if and only if*:

$$\forall L \leq L^\star, \quad df(0, L) \leq L$$

where L is an absolute deadline and $L^\star$ is the first idle time in the schedule.

Proof: The condition is clearly necessary, so it remains to prove the sufficiency by contradiction. Consider the schedule generated by EDF. Suppose that a deadline is missed, and let $[t_1, t_2)$ be a busy period as in the previous lemma. We already proved that there is at least one task that is released exactly at t_1. Let τ_i be one such task, so that $t_1 = r_{im}$ for some m, and τ_k be the task whose deadline d_{kp} is not met (note that it could be $i = k$). By following the same reasoning as in Lemma 1, we obtain $df(t_1, t_2) > t_2 - t_1$.

Now consider a task τ_j, $j \neq i, k$, and let r_{jl} be the first release time of a job of τ_j such that $r_{jl} > t_1$. The new schedule generated by "pulling back" all releases of task τ_j of $r_{jl} - t_1$ is still unfeasible. In fact, since all absolute deadlines of task τ_j are now located earlier, the number of jobs of τ_j in $[t_1, t_2]$ could be increased: $\eta_j'(t_1, t_2) \geq \eta_j(t_1, t_2)$. Thus, $\sum_{i=1, i\neq j}^{N} \eta_i(t_1, t_2)C_i + \eta_j'(t_1, t_2)C_j \geq \sum_{i=1}^{N} \eta_i(t_1, t_2)C_i > t_2 - t_1$ (see task τ_2 in Figures 2 and 3). Now consider task τ_k, and suppose that $k \neq i$. Let r_{kl} be the first release time of τ_k after t_1. By moving all releases back of $r_{kl} - t_1$, we also move back its deadlines. Let $d_{kp}' \leq d_{kp}$ be the new deadline. We shall consider two possible cases. First, suppose that for each deadline $d_{jq} \leq d_{kp}$, $j \neq k$, it still holds $d_{jq} \leq d_{kp}'$. Then $df(t_1, t_2' = d_{kp}') = df(t_1, t_2) > t_2 - t_1 > t_2' - t_1$ and the new task set is not feasible. Second, suppose that d_{jq} is the largest deadline in $[t_1, t_2]$ such that $d_{kp}' < d_{jq} \leq d_{kp}$. Consider the new busy period $[t_1, t_2' = d_{jq})$. Then $df(t_1, t_2') = df(t_1, t_2)$, with $t_2' \leq t_2$, thus the new task set is not feasible (see Figures 2 and 3, where $k = 3$ and $j = 1$).

Therefore, by moving back all tasks such that their first release time is at t_1 we obtain an unfeasible schedule where all tasks are simultaneously released at t_1. Thus, if a deadline is missed inside any busy period, then a deadline is missed inside the busy period starting at 0. Moreover, $df(0, t_2 - t_1) \geq df(t_1, t_2) > t_2 - t_1$. This contradicts the hypothesis, hence the theorem holds. □

It can be proved that for $U < 1$ the busy period length is bounded by $\frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\}$ (Baruah et al., 1990b). Therefore, the analysis has complexity $O\left(N \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\}\right)$.

The previous theorem does not hold in the case of asynchronous task sets. It still gives a sufficient condition, in the sense that if the hypothesis holds for the corresponding synchronous task set, than the original asynchronous task set is feasible. However the condition is no longer necessary. Consider the feasible task set in Figure 4, composed of tasks $\tau_1(\phi = 1, C = 2, D = 3, T = 4)$ and $\tau_2(\phi = 0, C = 2, D = 3, T = 6)$. Since we are showing the schedule in a full hyperperiod, it should be easy to see that no instant t_1 exists such that both tasks are released simultaneously. We can still use a pessimistic analysis

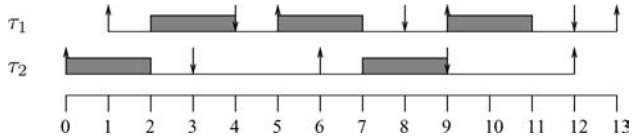


Figure 4. Example task set.

by considering the corresponding synchronous task set, but in the case of Figure 4 this would not work since it can be easily seen that the corresponding synchronous task set is not feasible; in fact, if we move back the deadline of the first job of τ_1 to 3, we obtain $df(0, 3) = 4 > 3$ and therefore a deadline will be missed. Checking all busy periods in $[0, \Phi + 2H]$ is possible but would imply an exponential complexity.

3.2. Improving the Analysis for Asynchronous Tasks

The intuition behind our idea is as follow. In the proofs of Lemma 1 and Theorem 1 we note that there is always an initial task that is released at t_1, the start of the "critical" busy period. However, we do not know which task is it. Therefore, we build a new task set $\mathcal{T}_i'$ for each possible initial task τ_i, $1 \leq i \leq N$. Since τ_i is released at the beginning of the busy period, we fix $\phi_i' = 0$ in $\mathcal{T}_i'$ and check the busy period starting from 0 instead of t_1. We can then "pull back" each other task τ_j by setting ϕ_j' to the minimum time distance between any activation of τ_i and the successive activation of τ_j in the original task set $\mathcal{T}$. We will use the following Lemma:

Lemma 2. *Given two tasks τ_i and τ_j, the minimum time distance between any release time of task τ_i and the successive release time of task τ_j is equal to*:

$$\Delta_{ij} = \phi_j - \phi_i + \left\lceil \frac{\phi_i - \phi_j}{\gcd(T_j, T_i)} \right\rceil \gcd(T_j, T_i)$$

Proof: Note that for each possible job τ_{im} and τ_{jl}, $r_{jl} - r_{im} = \phi_j - \phi_i + lT_j - mT_i$. Thus $\forall l \geq 0, \forall m \geq 0, \exists K \in \mathbb{Z}, r_{jl} - r_{im} = \phi_j - \phi_i + K \gcd(T_j, T_i)$. By imposing $r_{jl} \geq r_{im}$, we obtain $K \geq \lceil \frac{\phi_i - \phi_j}{\gcd(T_j, T_i)} \rceil$. By simple substitution, we obtain the lemma. □

De nition 1. Given task set $\mathcal{T}$, $\mathcal{T}_i'$ is the task set with the same tasks as $\mathcal{T}$ but with offsets:

$$\phi_i' = 0$$
$$\phi_j' = \Delta_{ij} \qquad \forall j \neq i, 1 \leq j \leq N$$

Consider the example of Figure 5. By setting $i = 1$ we obtain $\phi_1' = 0$, $\phi_2' = r_{23} - r_{14} = 0$ and $\phi_3' = r_{31} - r_{11} = 2$.

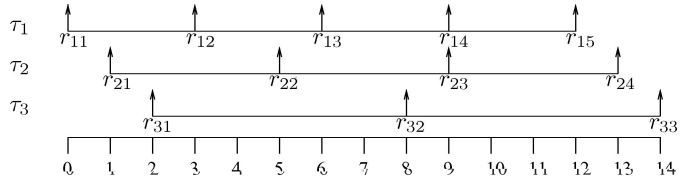


Figure 5. Example task set, $\tau_1(\phi = 0, T = 3)$, $\tau_2(\phi = 1, T = 4)$, $\tau_3(\phi = 2, T = 6)$.

We will now prove that, to assess the feasibility of $\mathcal{T}$, it suffices to check that, for every task set $\mathcal{T}_i'$, $1 \leq i \leq N$, all deadlines are met inside the busy period starting from time 0.

Theorem 2. *Given task set $\mathcal{T}$ with $U \leq 1$, scheduled on a single processor, if $\forall\ 1 \leq i \leq N$ all deadlines in task set $\mathcal{T}_i'$ are met until the first idle time, then $\mathcal{T}$ is feasible.*

Proof: By contradiction. Consider the schedule generated by EDF. Suppose that a deadline is not met for task set $\mathcal{T}$, and let $[t_1, t_2)$ be the busy period as defined in Lemma 1. We already proved that there is at least one task that is released at t_1, let it be τ_i. From Lemma 2, it follows that for every τ_j, $j \neq i$, the successive release time is $r_{jl} \geq t_1 + \Delta_{ij}$. By following the same reasoning as in Theorem 1, we can "pull back" every task so that its first release time coincides with its minimum distance from t_1, and the resulting schedule is still unfeasible. Let $\sigma_i(t)$ be the new resulting schedule.

Now, observe that, from t_1 on, the new schedule $\sigma_i(t)$ is coincident with the schedule $\sigma_i'(t)$ generated by task set $\mathcal{T}_i'$ from time 0: $\forall t \geq t_1 : \sigma_i(t) = \sigma_i'(t - t_1)$. Therefore, there is a deadline miss in the first busy period in the schedule generated by $\mathcal{T}_i'$, against the hypothesis. Hence, the theorem follows. □

Note that Theorem 2 gives us a less pessimistic feasibility condition that Theorem 1. As an example, consider the task set in Figure 4. According to Theorem 2 the task set is feasible (the minimum distance between either τ_1 and τ_2 or viceversa is equal to 1, therefore there is always idle time at 4), while Theorem 1 gives no result.

However, Theorem 2 gives only a sufficient condition. For example, consider the following task set: $\tau_1(\phi = 0, C = 1, D = 2, T = 5)$, $\tau_2(\phi = 1, C = 1, D = 2, T = 4)$, $\tau_3(\phi = 2, C = 1, D = 2, T = 6)$. By analyzing the schedule, it can be seen that it is feasible, but Theorem 2 fails to give any result. The reason can be easily explained. When we "pull back" the tasks to their minimum distance from τ_i, we are not considering the cross relations between them. In other words, it may be possible that the pattern of release times analyzed with Theorem 2 are not found in the original schedule of $\mathcal{T}$.

3.3. Generalization

In order to reduce the pessimism in the analysis, we can generalize Theorem 2 in the following way. Instead of fixing just the initial task τ_i, we can also fix the position of other tasks with respect to τ_i and then minimize the offsets of all the remaining tasks with respect to the fixed ones. The following lemma provides deeper insight.

Lemma 3. *The time distance between any release time r_{il} of task τ_i and the successive release time r_{jp} of task τ_j assumes values inside the following set*:

$$\left\{ \Delta_{ij}(k) \mid \forall\, 0 \le k < \frac{T_j}{\gcd(T_i, T_j)} \right\}$$

where

$$\Delta_{ij}(k) = \left\lceil \frac{\phi_i + kT_i - \phi_j}{T_j} \right\rceil T_j + (\phi \qquad kT$$

where

$$\Delta_{i_1 i_2 i_3}(k_2) = \left\lceil \frac{\phi_{i_1} + k_1 T_{i_1} + k_2 \text{lcm}(T_{i_1}, T_{i_2}) - \phi_{i_3}}{T_{i_3}} \right\rceil T_{i_3} + \\ - \left(\phi_{i_1} + k_1 T_{i_1} + k_2 \text{lcm}(T_{i_1}, T_{i_2}) - \phi_{i_3}\right)$$

Proof: Since the time difference between $r_{i_1 l_1}$ and the successive release time $r_{i_2 l_2}$ of τ_{i_2} must be equal to $\Delta_{i_1 i_2}(k_1)$, not all values of l_1 are acceptable. Indeed, it must hold $l_1 \equiv k_1 \text{mod}(\frac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})})$.

Let $\tau_{i_1 i_2}$ be a task with period $T_{i_1 i_2} = T_{i_1} \frac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})} = \text{lcm}(T_{i_1}, T_{i_2})$ and offset $\phi_{i_1 i_2} = \phi_{i_1} + k_1 T_{i_1}$. All acceptable release times $r_{i_1 l_1}$ correspond to the release times of task $\tau_{i_1 i_2}$. We can then apply Lemma 3 to $\tau_{i_1 i_2}$ and τ_{i_3} obtaining $\Delta_{i_1 i_2 i_3}$. □

Lemma 5. *The time distance between any release time $r_{i_1 l_1}$ of task τ_{i_1} and the successive release time of task τ_{i_p}, given $r_{i_2 l_2} - r_{i_1 l_1} = \Delta_{i_1 i_2}(k_1), \ldots, r_{i_{p-1} l_{p-1}} - r_{i_1 l_1} = \Delta_{i_1 \ldots i_{p-1}}(k_{p-2})$, assumes values inside the following set:*

$$\left\{ \Delta_{i_1 \ldots i_p}(k_{p-1}) \mid \forall\, 0 \leq k_{p-1} < \frac{T_{i_p}}{\gcd\left(T_{i_p}, \text{lcm}(T_{i_1}, \ldots, T_{i_{p-1}})\right)} \right\}$$

where

$$\Delta_{i_1 \ldots i_p}(k_{p-1}) = \left\lceil \frac{\phi_{i_1} + \sum_{q=1}^{p-1} k_q \text{lcm}(T_{i_1}, \ldots, T_{i_q}) - \phi_{i_p}}{T_{i_p}} \right\rceil T_{i_p} + \\ - \left(\phi_{i_1} + \sum_{q=1}^{p-1} k_q \text{lcm}(T_{i_1}, \ldots, T_{i_q}) - \phi_{i_p} \right)$$

Proof: The proof can be obtained by induction, reasoning in the same way as in Lemma 4. □

Lemma 6. *The minimum time distance between any release time $r_{i_1 l_1}$ of task τ_{i_1} and the successive release time of task τ_j, given $r_{i_2 l_2} - r_{i_1 l_1} = \Delta_{i_1 i_2}(k_1), \ldots, r_{i_M l_M} - r_{i_1 l_1} = \Delta_{i_1 \ldots i_M}(k_{M-1})$, is equal to*:

$$\Delta_{i_1 \ldots i_M j} = \phi_j - \phi_{i_1} - \sum_{q=1}^{M-1} k_q \text{lcm}(T_{i_1}, \ldots, T_{i_q}) \\ + \left\lceil \frac{\phi_{i_1} + \sum_{q=1}^{M-1} k_q \text{lcm}(T_{i_1}, \ldots, T_{i_q}) - \phi_j}{\gcd\left(T_j, \text{lcm}(T_{i_1}, \ldots, T_{i_M})\right)} \right\rceil \cdot \gcd\left(T_j, \text{lcm}(T_{i_1}, \ldots, T_{i_M})\right)$$

Proof: Reasoning in the same way as in Lemmas 4 and 5, all acceptable release times $r_{i_1 l_1}$ must correspond to the release times of a task $\tau_{i_1 \ldots i_M}$ with period $T_{i_1 \ldots i_M} = \text{lcm}(T_{i_1}, \ldots, T_{i_M})$ and offset $\phi_{i_1 \ldots i_M} = \phi_{i_1} + \sum_{q=1}^{M-1} k_q \text{lcm}(T_{i_1}, \ldots, T_{i_q})$. We can then apply Lemma 2 to $\tau_{i_1 \ldots i_M}$ and τ_j obtaining $\Delta_{i_1 \ldots i_M j}$. □

Following the same line of reasoning as in Theorem 2, we now define task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ in the same way as $\mathcal{T}'_i$ before.

De nition 2. Given task set $\mathcal{T}$, $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ is the task set with the same tasks as $\mathcal{T}$ but with offsets:

$$
\begin{aligned}
\phi'_{i_1} &= 0 \\
&\vdots \ \vdots \ \vdots \\
\phi'_{i_p} &= \Delta_{i_1 \ldots i_p}(k_{p-1}) \\
&\vdots \ \vdots \ \vdots \\
\phi'_{i_M} &= \Delta_{i_1 \ldots i_M}(k_{M-1}) \\
\phi'_j &= \Delta_{i_1 \ldots i_M j} \quad \forall j \neq i_1, \ldots, i_M, 1 \leq t \leq N
\end{aligned}
$$

Finally, we generalize Theorem 2 to the case of M fixed tasks.

Theorem 3. *Given task set $\mathcal{T}$ with $U \leq 1$, to be scheduled on a single processor, let M be a number of tasks, $1 \leq M < N$. If:*

$$
\begin{aligned}
&\forall \tau_{i_1}, 1 \leq i_1 \leq N, \forall \tau_{i_2} \neq \tau_{i_1}, 1 \leq i_2 \leq N, \ldots, \\
&\forall \tau_{i_M} \neq \tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_{M-1}}, 1 \leq i_M \leq N, \\
&\forall k_1, 0 \leq k_1 < \frac{T_{i_2}}{\gcd\left(T_{i_1}, T_{i_2}\right)}, \forall k_2, 0 \leq k_2 < \frac{T_{i_3}}{\gcd\left(T_{i_3}, \text{lcm}\left(T_{i_1}, T_{i_2}\right)\right)}, \ldots, \\
&\forall k_{M-1}, 0 \leq k_{M-1} < \frac{T_{i_M}}{\gcd\left(T_{i_M}, \text{lcm}\left(T_{i_1}, \ldots, T_{i_{M-1}}\right)\right)},
\end{aligned}
$$

all deadlines in task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ are met until the rst idle time, then $\mathcal{T}$ is feasible.

Proof: By contradiction. Consider the schedule generated by EDF. Suppose that a deadline is missed and let $[t_1, t_2)$ be the busy period as in the proof of Lemma 1. We choose i_1 such that $t_1 = r_{i_1 l_1}$ for some l_1. Next, we choose any distinct indexes $i_2, \ldots, i_M$ and any values $k_1, \ldots, k_{M-1}$ as in Lemmas 3, 4, 5, and compute the corresponding distances $\Delta_{i_1 i_2}(k_1), \ldots, \Delta_{i_1 \ldots i_M}(k_{M-1})$ from t_1. These tasks are fixed and will not be "pulled back". For the remaining tasks, we "pull back" their release times as much as it is possible: for every non-fixed task τ_j we set the distance from t_1 equal to $\Delta_{i_1 \ldots i_M j}$. By following the same reasoning as in Theorem 2, it can be easily proven that a deadline is still missed in the new

generated schedule $\sigma(t)$. Note that, from t_1 on, the schedule $\sigma(t)$ is coincident with the schedule $\sigma'(t)$ generated by task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ from time 0: $\forall t \geq t_1 : \sigma(t) = \sigma'(t - t_1)$. Hence, a deadline is missed in the first busy period of $\sigma'(t)$, against the hypothesis. □

3.4. Analysis of Mixed Sporadic and Periodic Task Sets

In the previous section we considered task sets consisting only of periodic tasks. Although many real-time activities can be modelled by periodic tasks, there are many others that are usually modelled by using *sporadic* tasks, i.e. aperiodic tasks with a minimum inter-arrival time. Therefore, we now extend our analysis to the case of sporadic tasks.

First note that the concept of offset is meaningless for a sporadic task, since it does not follow a periodic activation pattern; a sporadic job can be activated anywhere in the schedule at the same time as any other task. Therefore, our asynchronous analysis does not yield any result for task set composed of sporadic tasks only. We are instead interested in analyzing task sets composed of both periodic and sporadic tasks.

We will consider a mixed periodic-sporadic task set $\mathcal{T}$ composed of N periodic tasks $\tau_1, \ldots, \tau_N$ and M sporadic tasks $\tau_{N+1}, \ldots, \tau_{N+M}$. Each task is defined by the same parameters as in Section 2, except that for a sporadic task τ_i, T_i represents its minimum inter-arrival time and offset ϕ_i is meaningless.

We can easily extend out test with one fixed task from Theorem 2 to mixed period-sporadic task sets, simply considering that the minimum distance between the activation of a sporadic task and the successive activation of any other task is always zero. The extension is more formally defined as follows.

De nition 3. Given a mixed periodic-sporadic task set $\mathcal{T}$, $\mathcal{T}'_i$ is the task set with the same tasks as $\mathcal{T}$ but with offsets:

$$\begin{aligned} \phi'_i &= 0 \\ \phi'_j &= \Delta_{ij} \quad \forall j \neq i, 1 \leq j \leq N \\ \phi'_j &= 0 \quad \forall j, N < j \leq N + M \end{aligned}$$

Theorem 4. *Given a mixed periodic-sporadic task set $\mathcal{T}$ with $U \leq 1$, scheduled on a single processor, if $\forall\ 1 \leq i \leq N$ all deadlines in task set $\mathcal{T}'_i$ are met until the rst idle time, then $\mathcal{T}$ is feasible.*

Proof: By contradiction. Suppose that a deadline is not met for task set $\mathcal{T}$, and let $[t_1, t_2)$ be the busy period as defined in Lemma 1. We must consider two cases. First, suppose that a periodic task τ_i is released at t_1. We can then follow the same proof as in Theorem 2, "pulling back" all sporadic tasks so that their first release time coincides with t_1.

If no periodic task is released at t_1, than a sporadic task τ_j must surely be released at t_1. Let τ_i be the periodic task with the smallest activation time (suppose r_{ik}) inside the busy period. If we pull back all periodic tasks of an amount of time units equal to $r_{ik} - t_1$, the resulting schedule is still unfeasible as proven in Theorem 1. Let $\sigma_i(t)$ be the new resulting

schedule after pulling back all other sporadic tasks so that their first release time coincides with t_1. The new schedule $\sigma_i(t)$ is coincident with the schedule $\sigma_i'(t)$ generated by task set $\mathcal{T}_i'$ from time 0: $\forall t \geq t_1 : \sigma_i(t) = \sigma_i'(t - t_1)$; therefore, there is a deadline miss in the first busy period in the schedule generated by $\mathcal{T}_i'$, against the hypothesis. Hence, the theorem follows. □

It is possible to extend Theorem 3 in the same straightforward way; the extension is not shown due to space constraint.

3.5. Algorithms

Theorem 2 gives us a new feasibility test for asynchronous task sets with $U < 1$ on single processor systems. For each initial task τ_i we first compute the minimal offset ϕ_j for each $j \neq i$ and the length L^* of the busy period. Then, we check that each deadline L less than or equal to L^* is met. The pseudo code is given in Figure 6.

Note that the recurrence over the length of the busy period $L^*(t+1) = \sum_{i=1}^{N} (\lceil \frac{L^\star(t)-\phi_i}{T_i} \rceil)_0 C_i$ converges in pseudo-polynomial time if $U < 1$ (Spuri, 1996a).

Since we must execute the algorithm for each initial task τ_i, the test has a computational complexity that is N times that of Barauh's synchronous test: $O(N^2 \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\})$.

We can obtain a less pessimistic test, at the cost of an increased computation time, by using Theorem 3. As the number of fixed tasks M increases, we can expect to obtain higher percentages of feasible task sets, but the computation complexity rises quickly. If we select M fixed tasks, the complexity is bounded by $O(N^{M+1} \max_{i,j=1}^{N}\{\frac{T_i}{\gcd(T_i,T_j)}\}^{M-1} \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\})$. The pseudo code for $M = 2$ is given in Figure 7.

```
for each i = 1 ... N
    φ_i = 0
    for each j = 1 ... N, j ≠ i
        compute φ_j (according to Lemma 2)
    next j
    L* = C_i
    while L* changes
        L* = Σ_{i=1}^{N} ( ⌈(L* − φ_i)/T_i⌉ )_0 C_i
    repeat
    for each deadline L ≤ L*
        if df(0, L) > L return unknown
    next L
next i
return feasible
```

Figure 6. Pseudocode for the feasibility test, 1 fixed task.

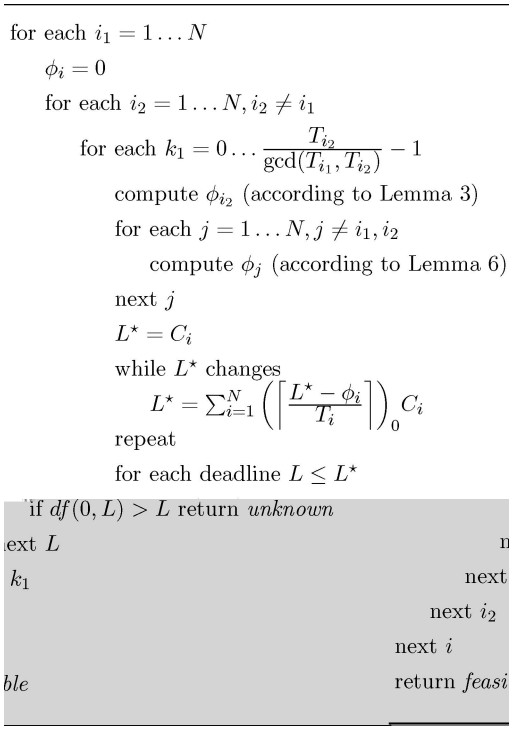


Figure 7. Pseudocode for the feasibility test, 2 fixed tasks.

4. Taking into Account Shared Resources

In this section, we will extend the test with 1 fixed task to cover the problem of blocking times and synchronization on shared resources. In order to achieve predictability, a *resource access protocol* must be introduced in order to bound the maximum blocking time experienced by tasks due to mutual exclusion. Many resource access protocols have been proposed in literature (Chen and Lin, 1990; Jeffay, 1992); we base our discussion on the Stack Resource Protocol (Baker, 1991).

In the remainder of this section, we briefly introduce the SRP and the processor demand criterion with blocking time; our test is a modified version of the one presented in Lipari and Buttazzo (2000). Finally we will extend the analysis in order to account for offsets.

4.1. SRP

Under SRP, each task is assigned a static preemption level $\pi_i = \frac{1}{D_i}$ in addition to its dynamic priority defined by EDF. The following fundamental property holds:

Property 1. *Task τ_i can preempt task τ_j only if $\pi_i > \pi_j$.*

To ease further definitions, we will also define an additional preemption level π_s as a preemption level that is strictly greater than the preemption level of every task.

Furthermore, each resource ρ_k is assigned a static ceiling $\text{ceil}(\rho_k) = \max_i\{\pi_i | \exists \xi_{ij}, \rho_{ij} = \rho_k\}$. A dynamic system ceiling is then defined as follows:

$$\Pi_s(t) = \max(\{\text{ceil}(\rho_k) | \rho_k \text{ is busy at time } t\} \cup 0)$$

The scheduling rule is the following: a job is not allowed to start execution until its priority is the highest among the active jobs and its preemption level is strictly higher then the system ceiling.

Among the many useful properties of SRP, we are mainly interested in two of them:

Property 2. ([Baker, 1991]). *Under SRP, a job can only be blocked before it starts execution; once started, it can only be preempted by higher priority jobs.*

Property 3. ([Baker, 1991]). *A job can be blocked only once by one lower priority job.*

In the following Section 4.2, we will briefly introduce the processor demand criterion with blocking time; our test is a modified version of the one presented in Lipari and Buttazzo (2000). In Section 4.3 we will then extend the analysis in order to account for offsets.

4.2. Processor Demand Analysis with Blocking Time

Since tasks can now share resources, we need to account for blocking times inside busy periods. We will start by proving than no more than one job may cause blocking time inside a busy period:

Lemma 7. *Jobs that are completely executed in any busy period* $[t_1, t_2)$, *where* t_2 *corresponds to a deadline and* t_1 *is the last instant prior to* t_2 *such that either no jobs or a job with deadline greater than* t_2 *is scheduled at* $t_1 - 1$, *may be blocked only once by a single lower priority job.*

Proof: Without blocking times, all jobs completely executed inside the busy period must be released at or after t_1 and have deadline at or before t_2 as in Lemma 1. We will call $\mathcal{A}$ the set of such jobs. However, when blocking times are introduced it is possible for a job of some task τ_j with deadline greater than t_2 to be executed inside the busy period. For this to be possible, the job must be inside a critical section at time t_1, since it must block some higher priority job in $\mathcal{A}$. However, there can only be one such job; otherwise, some job in $\mathcal{A}$ would be blocked by at least two lower priority jobs, which is impossible due to Property 3. □

Due to Lemma 7, it makes sense to define a *dynamic maximum blocking time* $B(t)$ as the maximum blocking time that can be experienced by any task inside a busy period of length t.

De nition 4. Given a synchronous task set $\mathcal{T}$, we define the dynamic maximum blocking time for $\mathcal{T}$ as follows:

$$B(t) = \max_{jk} \left(\left\{ C_{jk} - 1 | D_j > (t + \psi_{jk} + 1) \wedge \text{ceil}(\rho_{jk}) \geq \frac{1}{t} \right\} \cup 0 \right)$$

Lemma 8. *$B(t_2 - t_1)$ is an upper bound to the maximum blocking time experienced by any task completely executed in a busy period $[t_1, t_2)$ as de ned in Lemma 7.*

Proof: As in Lemma 7, let $\mathcal{A}$ be the set of jobs that are released at or after t_1 and have deadline at or before t_2. Since jobs in $\mathcal{A}$ can only be blocked by a single lower priority job, the maximum blocking time can be no longer than the length of some critical section $C_{jk} - 1$; in fact, the blocking job can enter ξ_{jk} at worst at $t_1 - 1$. Furthermore, since the job cannot enter ξ_{jk} before ψ_{jk} time units have elapsed since its activation, and its deadline must be greater than t_2, it must also hold $D_j > t_2 - t_1 + \psi_{jk} + 1$. Finally, resource ρ_{ij} must be able to block some job in $\mathcal{A}$; since no job in $\mathcal{A}$ can have a deadline longer than $t_2 - t_1$, it must hold $\text{ceil}(\rho_{ij}) \geq \frac{1}{t}$. Since $B(t)$ considers the critical section of maximum length among the ones that respect the previous conditions, it is surely an upper bound to the maximum blocking time experienced by tasks in $[t_1, t_2)$. □

Note that the computed bound is clearly pessimistic; in particular, it may be impossible for the blocking job to enter the critical section exactly at $t_1 - 1$.

Once $B(t)$ has been defined, it is trivial to prove the following theorem:

Theorem 5. *A synchronous task set $\mathcal{T}$, using mutually exclusive critical sections, is feasible on a single processor if:*

$$\forall L \leq L^\star, \; df(0, L) + B(L) \leq L$$

where L is an absolute deadline and $L^\star$ is the rst idle time in the schedule.

Proof: The theorem is a direct extension of Theorem 1, since tasks executed inside any busy period of length L cannot be blocked for more than $B(L)$ time units as proven in Lemma 8. □

4.3. Extension to Offsets

We will now extend the analysis to account for asynchronous task sets. When tasks have non zero offsets, we can exploit two different behaviors in order to reduce the computed maximum blocking time. First of all, it may be impossible for the blocking task to be released exactly $\psi_{jk} + 1$ time units before the beginning of the busy period. In order to capture this behavior, we need to compute a new minimum time distance between tasks.

Lemma 9. *Given two tasks τ_i and τ_j, the minimum time distance between any release time of task τ_i and the successive release time of task τ_j that is greater or equal to some value $q+1$ is equal to*:

$$\Delta_{ij}^{q} = \phi_j - \phi_i + \left\lceil \frac{\phi_i + q + 1 - \phi_j}{\gcd(T_j, T_i)} \right\rceil \gcd(T_j, T_i)$$

Proof: The proof is a simple extension of Lemma 2; it is sufficient to see that the condition on q is equivalent to considering τ_i being released $q+1$ time units later. □

Second, due to the offset, it may be impossible for a task τ_p to be executed inside a busy period of length t even if $t \geq D_p$. To account for this behavior we need to define a new *dynamic preemption level*. In the following definition, $\mathcal{T}_i'$ is the task set from Definition 1.

De nition 5. Given task set $\mathcal{T}_i'$, we define the following dynamic preemption level:

$$\pi_i(t) = \min_j \left(\left\{ \pi_j | \phi_j' + D_j \leq t \right\} \cup \pi_s \right)$$

We can finally define the maximum dynamic blocking time for asynchronous task sets.

De nition 6. Given task set $\mathcal{T}_i'$, the maximum dynamic blocking time is defined as:

$$B_i(t) = \max_{jk} \left(\left\{ C_{jk} - 1 | D_j > \left(t + \Delta_{ji}^{\psi_{jk}} \right) \wedge \text{ceil}(\rho_{jk}) \geq \pi_i(t) \right\} \cup 0 \right)$$

Lemma 10. *$B_i(t_2 - t_1)$ is an upper bound to the maximum blocking time experienced by tasks in a busy period $[t_1, t_2)$, where t_1 corresponds to an activation time of task τ_i and t_2 to the deadline of some task.*

Proof: The proof is an extension of the proof of Lemma 8. As before, suppose that a job of task τ_j blocks higher priority tasks executed inside the busy period (set $\mathcal{A}$).

Once again, τ_j can enter some critical section ξ_{jk} at worst at $t_1 - 1$, so that the maximum blocking time can be no longer than $C_{jk} - 1$. Furthermore, τ_j must be activated at least $\psi_{jk} + 1$ time units before t_1 and its deadline must be greater than t_2, thus it must hold $D_j > t_2 - t_1 + \Delta_{ji}^{\psi_{jk}}$. Finally, since τ_j must be able to block some job in $\mathcal{A}$, it must also hold $\text{ceil}(\rho_{ij}) \geq \pi_i(t)$. In fact, no task τ_p in $\mathcal{A}$ can have a preemption level smaller than $\pi_i(t)$, since ϕ_p' is the smallest time distance after t_1 at which τ_p can be activated. Therefore $B_i(t_2 - t_1)$ is an upper bound to the maximum blocking time. □

We can now introduce the main theorem.

Theorem 6. *Given task set $\mathcal{T}$ with $U \leq 1$, if $\forall\ 1 \leq i \leq N, \forall\ L \leq L^\star$:*

$$df(0, L) + B_i(L) \leq L$$

where L is an absolute deadline of task set $\mathcal{T}_i'$ and $L^\star$ is the first idle time in the schedule of $\mathcal{T}_i'$, then $\mathcal{T}$ is feasible.

Proof: The theorem is a straightforward extension of Theorem 2. □

4.4. Busy Period Length

Note that the recurrence over the length of the busy period for task set $\mathcal{T}_i'$:

$$L^\star = \sum_{j=1}^{N} \left(\left\lceil \frac{L^\star - \phi_j'}{T_j} \right\rceil \right)_0 C_j$$

is no longer valid when blocking times are considered. In fact, a task τ_j whose offset is greater than or equal to the maximum length computed in this way, may still contribute to the busy period by adding a blocking time. The recurrence must thus be modified by adding the blocking time contribution:

$$L^\star = \sum_{j=1}^{N} \left\lceil \frac{L^\star - \phi_j'}{T_j} \right\rceil C_j + B_j^\star(L^\star)$$

where

$$B_i^\star(t) = \max(\{C_{jk} - 1 | \phi_j' \geq t \wedge \text{ceil}(\rho_{jk}) \geq \pi_i^\star(t)\} \cup 0)$$

and

$$\pi_i^\star(t) = \min\left(\left\{\pi_j | \phi_j' < t\right\} \cup \pi_s\right)$$

In other words, we must consider the blocking time introduced by tasks with offset greater or equal to the currently computed length, due to critical sections that may introduce blocking on tasks with offsets smaller than the busy period length.

5. Experimental Evaluation

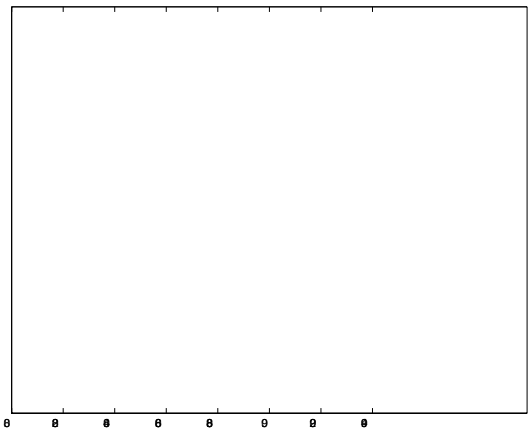
In this section, we evaluate the effectiveness of the proposed tests against Baruah's sufficient synchronous test described by Theorem 1 and against the exponential test executed by checking the demand function for every deadline until $\Phi + 2H$. For each experiment, we generated 2000 synthetic periodic task sets consisting of 6, 10 and 20 tasks, respectively, and with total utilization ranging from 0.8 to 1. No resource dependency is considered.

Each task set was generated in the following way. First, utilizations U_i were randomly generated according to a uniform distribution, so that the total utilization summed up to the desired value. Then periods were generated uniformly between 10 and 200 and the worst-case computation time of each task was computed based on utilization and period. Finally, relative deadlines were assigned to be either between 0.3 and 0.8 times the task's period or between half period and the period, and offsets were randomly generated between 0 and the period.

We experimented with two types of task sets. In the first case, we generated the periods so that the greatest common divisor between any two of them were a multiple of 5. In the second case, we chose the gcds as multiples of 10. The basic idea is that, if the gcd between two periods is 1, the distance between the release times of the two tasks can assume any value, 0 included. In the limit case in which all periods are relatively prime, it is possible to show that the synchronous test is necessary and sufficient also for asynchronous task sets. Note that in the real world, a situation in which the task periods are relatively prime is not very common.

In each experiment we computed the percentage of detected feasible task sets out of the total number of feasible task sets using Baruah's synchronous test and our test with one, two and three fixed tasks. In the following we will denote these tests with *sync*, *1- xed*, *2- xed*, *3- xed*, respectively. The results are presented in Figures 8–13.

Figures 8–10 shows the results for 6 tasks, with a minimum gcd of 10 in Figures 8 and 9 and a minimum gcd of 5 in Figure 10. In the first two figures, 1-fixed accepts a number of task sets up to 20% higher then the sync test. Performances are clearly lower with gcd $= 5$, as shown in Figure 10. Also note that the 2-fixed test does not achieve significant improvements over the 1-fixed test, while the 3-fixed accepts up to 10% more task sets. Results are partially different in Figures 11 and 12 where we show the results for 10 tasks and gcd $= 10$. The 1-fixed test again achieves good results, but the 2-fixed and particularly the 3-fixed tests seem much less beneficial. In fact, increasing the number



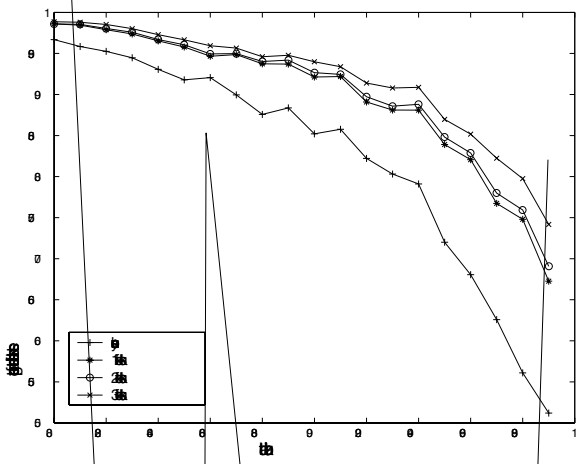
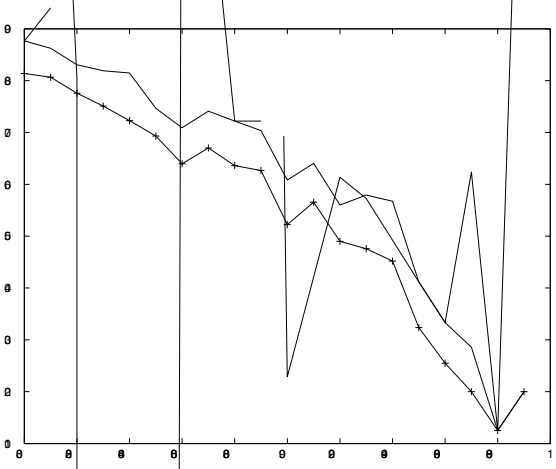


Figure 9. 6 tasks, gcd $= 10$, deadline $\in [0.5, 1.0]T$.



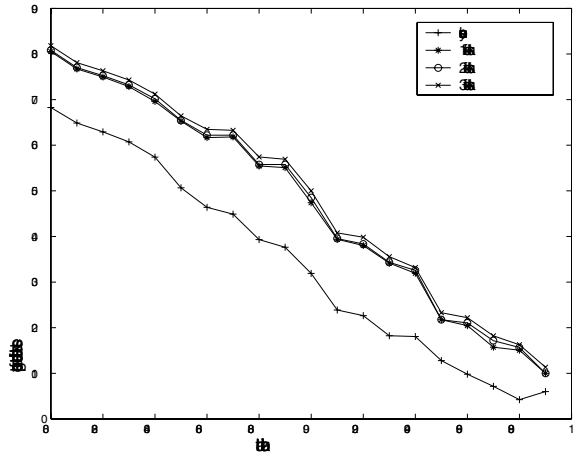


Figure 11. 10 tasks, gcd = 10, deadline $\in [0.3, 0.8]T$.

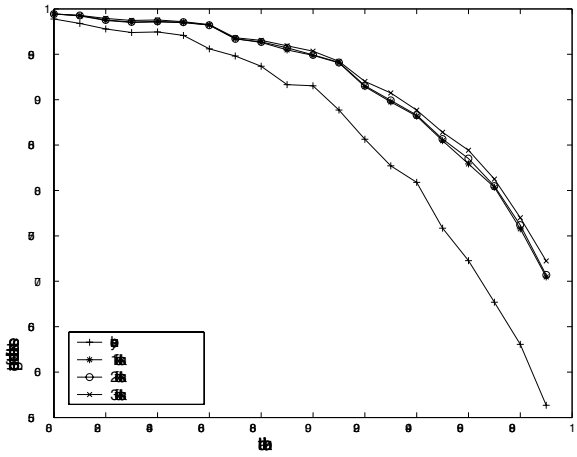


Figure 12. 10 tasks, gcd = 10, deadline $\in [0.5, 1.0]T$.

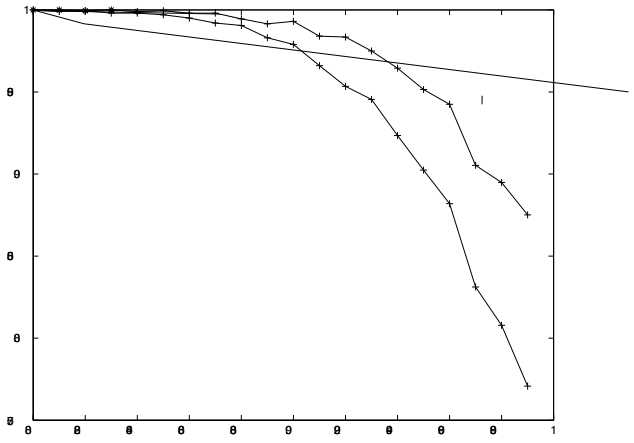
test is only one order of magnitude greater than the sync test. As the number of tasks increases we can appreciate that the growth in the number of cycles for the 1-fixed test is still acceptable compared to the corresponding growth of the exponential test.

6. Related work

The problem of feasibility analysis of asynchronous task sets can be found in many practical applications. For example, in distributed systems a *transaction* consists of a chain of precedence constrained tasks that must execute one after the other and each task can be allocated to a different processor. A transaction is usually modeled as a set of tasks with

Table 1. Mean number of cycles, gcd $= 10$, deadline $\in [0.3, 0.8]T$.

	Synchronous	1 fixed task	Exponential
6 tasks	40	67	2233
10 tasks	122	387	461356
20 tasks	639	6341	42781200



7. Conclusions and Future Work

In this paper, we presented a new sufficient feasibility test for asynchronous task sets and proved it correct. Our test tries to take into account the offsets by computing the minimum distances between the release times of any two tasks. By analyzing a reduced set of critical arrival patterns, the proposed test keeps the complexity low and reduces the pessimism of the synchronous sufficient test. We showed, with an extensive set of experiments, that our test outperforms the synchronous sufficient test.

As future work, we are planning to extend our test to the case of relative deadline greater than the period. We would also like to explore the application of our test to the problem of minimizing the *output jitter* of a set of tasks. The output jitter is defined as the distance between the response time of two consecutive instances of a periodic task. Minimizing the output jitter is a very important issue in control systems (Cervin, 1999). In Baruah et al. (1999) a method to reduce the output jitter is presented, consisting in reducing as much as it is possible the relative deadlines of the tasks without violating the feasibility of the task set. However, in Baruah's work tasks are considered synchronous. In our opinion, the method can be improved by introducing appropriate offsets, with a beneficial effect on the output jitter.

Acknowledgments

This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems", and by the European Community within the IST project 2001-34820 ARTIST.

References

Baker, T. 1991. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems* 3.

Baruah, S., Buttazzo, G., Gorinsky, S., and Lipari, G. 1999. Scheduling periodic task systems to minimize output jitter. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*. Hong Kong, pp. 62–69.

Baruah, S., Mok, A., and Rosier, L. 1990a. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*. pp. 182–190.

Baruah, S., Rosier, L., and Howell, R. 1990b. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems* 2.

Buttazzo, G. 1997. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers.

Cervin, A. 1999. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*. York, UK, pp. 4–10.

Chen, M., and Lin, K. 1990. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems* 2.

Dertouzos, M. L. 1974. Control robotics: The procedural control of physical processes. *Information Processing*.

Jeffay, K. 1992. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*. Phoenix, pp. 89–99.

Leung, J.-T., and Merril, M. 1980. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters* 3(11).

Lipari, G. and Buttazzo, G. 2000. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *System Architecture* 46: 327–338.

Liu, C. and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery* 20(1).

Palencia, J., and González Harbour, M. 2003. Offset-based response time analysis of distributed systems scheduled under EDF. In *15th Euromicro Conference on Real-Time Systems*. Porto, Portugal.

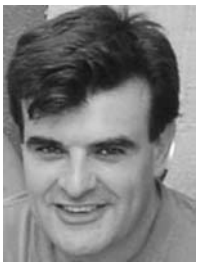
Pellizzoni, R. and Lipari, G. 2005. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Francisco, California.

Spuri, M. 1996a. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, France.

Spuri, M. 1996b. Holistic analysis for deadline scheduled real-time distributed systems. Technical Report RR-2873, INRIA, France.

Tindell, K., Burns, A., and Wellings, A. 1994. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems* 6(2): 133–151.

Rodolfo Pellizzoni received the Laurea degree in Computer Engineering from the "Università di Pisa" and the Diploma degree from the Scuola Superiore Sant'Anna, in 2004. He is presently a Ph.D. student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His main research interests are in real-time operating systems, scheduling theory and resource-allocation in distributed and multiprocessor systems.



Giuseppe Lipari graduated in Computer Engineering at the University of Pisa in 1996, and received the Ph.D. degree in Computer Engineering from Scuola Superiore Sant'Anna in 2000. During 1999, he was a visiting student at University of North Carolina at Chapel Hill, collaborating with professor S.K. Baruah and professor K. Jeffay on real-time scheduling. Currently, he is assistant professor of Operating Systems with Scuola Superiore Sant'Anna. His main research activities are in real-time scheduling theory and its application to real-time operating systems, soft real-time systems for multimedia applications and component-based real-time systems.