An Improved Schedulability Test for Uniprocessor Periodic Task Systems *

UmaMaheswari C. Devi
Department of Computer Science, The University of North Carolina, Chapel Hill, NC

Abstract

We present a sufficient linear-time schedulability test for preemptable, asynchronous, periodic task systems with arbitrary relative deadlines, scheduled on a uniprocessor by an optimal scheduling algorithm. We show analytically and empirically that this test is more accurate than the commonly-used density condition. We also present and discuss the results of experiments that compare the accuracy and execution time of our test with that of a pseudo-polynomial-time schedulability test presented previously for a restricted class of task systems in which utilization is strictly less than one.

1 Introduction

We consider the problem of determining the schedulability of preemptable, asynchronous, periodic hard-real-time task sytems with arbitrary relative deadlines, scheduled on a single processor by an online scheduling algorithm. This problem has been studied extensively [9][8][3], and this work has resulted in a number of well-known online scheduling algorithms and associated schedulability tests. It is also well-known that it is difficult to determine efficiently, *i.e.,* in polynomial-time, whether a task system can be correctly scheduled by some scheduling algorithm if tasks exist that have relative deadlines less than their periods [8][3]. Hence, when the amount of time available to determine shedulability is limited, such as in online admission-control tests, polynomial-time sufficient schedulability tests are generally used. However, sufficient tests are not capable of identifying every schedulable task set and thus result in schedulability loss. In this paper, we propose an improved linear-time sufficient schedulability test that can identify a larger percentage of task sets than other known polynomial-time tests.

2 Background and Related Work

A *periodic task system* consists of a set of tasks $\tau = \{T_1,\ T_2, \ldots, T_n\}$, where each task T_i is a sequential program defined by a four-tuple $(\phi_i,\ p_i,\ e_i,\ D_i)$, where $\phi_i \geq 0$, $p_i > 0$, $e_i > 0$, and $D_i > 0$ are the task's *phase*, *period*, *execution time*, and *relative deadline*, respectively. Each task T_i issues periodic requests for $e_i \leq \min(p_i, D_i)$ time units of execution, separated by p_i time units. Every such request, known as a *job*, must complete within D_i time units from the time it was issued. The phase, ϕ_i, denotes the time when T_i makes its first request. Such a task system is *synchronous* if all tasks release their initial jobs at the same time, and *asynchronous*, otherwise. In this paper we assume that task systems are specified by arranging tasks in order of non-decreasing relative deadlines.

The ratio of the execution cost of T_i to its period, e_i/p_i, is defined as its *utilization*. If each task's relative deadline is at least its period, then schedulability can be determined exactly by checking that total utilization is at most one, *i.e.,* by checking that $\sum_{i=1}^{n} e_i/p_i \leq 1$ [9]. However, if relative deadlines are less than periods, then this linear-time utilization-based test is not sufficient. One method for exactly determining schedulability in such a case is by checking that no deadlines are missed in a schedule constructed using an optimal scheduling algorithm. The minimum time over which a schedule has to be constructed is known to be $2P + \max_{1\leq i\leq n}\{D_i\} + max_{1\leq i\leq n}\{\phi_i\}$, where P is the *least common multiple* of the task periods, which is exponential in n [8]. The other known method is to use *response time analysis*, which consists of computing the *worst-case response time* (WCRT) of all tasks in a system and ensuring that each task's WCRT is less than its relative deadline. A job of a task experiences its WCRT when interferences from higher-priority jobs is the maximum that is possible. Though it is possible to compute WCRTs for tasks with the *earliest-deadline-first* scheduler [11], which is *optimal* [4], the time required is again exponential in n. Thus, both exact tests described here run in exponential time.

This problem of determining the schedulability of periodic task systems with *arbitrary relative deadlines* (ARD), is known to be co-NP-complete in the strong sense for asynchronous periodic task systems (*ARD-Async*), and in co-NP for synchronous periodic task systems (*ARD-Sync*) [3]. Hence, no exact polynomial-time utilization-based test or exact pseudo-polynomial-time demand-based test is possible for *ARD-Async* unless $P = NP$. It is currently unknown whether *ARD-Sync* is co-NP-hard or whether it has a polynomial-time or pseudo-polynomial-time solution. Checking that total *density* is at most one, *i.e.*, $\sum_{i=1}^{n} e_i/\min(p_i, D_i) \leq 1$, where $e_i/\min(p_i, D_i)$ is the *density* of task T_i, is the most commonly-used linear-time test, but it is only sufficient. Task systems with relative deadlines less than periods are not uncommon in practice, and

*Work supported by NSF grants ITR 0082866 and CCR 0204312.

Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)
0-7695-1936-9/03 $17.00 © 2003 IEEE

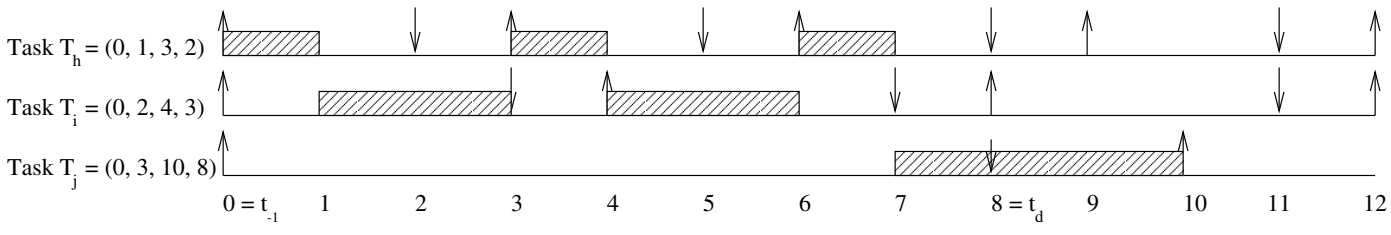

Figure 1. Schedule with a deadline miss.

hence it is desirable that this sufficient condition be tightened. In this paper, we present one such schedulability test, which is tighter than the density condition and can be evaluated in time that is linear in the number of tasks. Note that the density test and our proposed test may not be applicable if the scheduling algorithm being used is not *optimal*. Our focus in this paper is to determine schedulability under optimal scheduling algorithms.

Baruah *et al.* showed that *ARD-Sync* can be solved in pseudo-polynomial-time if total utilization is capped to some fixed positive constant c, where $0 \leq c < 1$ [3]. Capping utilization limits the time interval over which demand has to be checked to $(c/(1-c)) \max_{1 \leq i \leq n}(p_i - D_i)$. Since $c/(1-c)$ is a constant, this test runs in $O(n \max_{1 \leq i \leq n}(p_i - D_i))$ time. We shall hereafter refer to this test as the PPT test (pseudo-polynomial-time test).

To determine how well our test performs in comparison to the PPT test, we conducted a series of experiments involving randomly-generated task sets. These experiments showed that the accuracy of our test was usually comparable to that of the PPT test. However, at high utilizations, the PPT test was more accurate. On the other hand, at such high utilizations the running time of the PPT test was three orders of magnitude greater than our test. Hence, the choice of which test to use is largely a trade-off between schedulability and efficiency.

The rest of the paper is organized as follows. In Sec. 3, we present our more accurate schedulability test, prove it correct, and show that it is tighter than the density condition. In Sec. 4, we show how to adapt the test to account for context-switching overheads, nonpreemptivity of tasks, contention for resources among tasks, interrupt handlers, self-suspensions, and priority-space limitations. Then, in Sec. 5, we present the experimental results summarized above. We conclude in Sec. 6.

3 Improved Schedulability Test

Our proposed schedulability test is given by the following theorem.

Theorem 1 *Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a system of n preemptable, asynchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines. τ is schedulable using an optimal scheduling algorithm if*

$$(\forall k : 1 \leq k \leq n :: \sum_{i=1}^{k} \frac{e_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - min(p_i, D_i)}{p_i} \right) e_i \leq 1) \qquad (1)$$

Proof: Since an *optimal* scheduling algorithm correctly schedules any schedulable task system, it is sufficient that one such algorithm be used in determining if a task system τ is schedulable. The *earliest-deadline-first* (EDF), which prioritizes jobs with earlier deadlines over those with later deadlines [9], is known to be optimal for scheduling the class of task systems considered in the theorem [4]. It is the optimal scheduling algorithm assumed in our proof.

We prove the above theorem by proving the contrapositive, *i.e.*, by showing that if τ is not schedulable by EDF, then (1) is false. Let t_d be the first instant at which a job of some task T_j misses its deadline under EDF, as shown in Fig. 1. Let t_{-1} be the latest instant before t_d at which the processor was idle or was executing a job whose deadline is after t_d. Since T_j misses its deadline at t_d, the total demand placed on the processor in the interval $[t_{-1}, t_d)$ is greater than the length of this interval. The maximum demand placed on the processor in the interval is given by

$$\sum_{1 \leq i \leq n \wedge D_i \leq (t_d - t_{-1})} \left(\left\lfloor \frac{t_d - t_{-1} - D_i}{p_i} \right\rfloor + 1 \right) \cdot e_i.$$

If k $(1 \leq k \leq n)$ is the highest index of any task whose relative deadline is not greater than the length of the interval $[t_{-1}, t_d)$, then the total demand placed on the processor in the interval is bounded from above by

$$\sum_{i=1}^{k} \left(\left\lfloor \frac{t_d - t_{-1} - D_i}{p_i} \right\rfloor \cdot e_i \; + \; e_i \right).$$

Because T_j $(j \leq k)$ misses its deadline at t_d, we have the following:

$$\begin{aligned} t_d - t_{-1} &< \sum_{i=1}^{k} \left(\left\lfloor \frac{t_d - t_{-1} - D_i}{p_i} \right\rfloor \cdot e_i + e_i \right) \\ &\leq \sum_{i=1}^{k} \left(\left(\frac{t_d - t_{-1} - D_i}{p_i} \right) \cdot e_i + e_i \right) \\ &\leq \sum_{i=1}^{k} \left(\left(\frac{t_d - t_{-1} - min(p_i, D_i)}{p_i} \right) \cdot e_i + e_i \right) \\ &= \sum_{i=1}^{k} \frac{e_i}{p_i} \left(t_d - t_{-1} + p_i - min(p_i, D_i) \right). \end{aligned}$$

Dividing both sides of the above inequality by $t_d - t_{-1}$, we have

$$\begin{aligned} 1 &< \sum_{i=1}^{k} \frac{e_i}{p_i} \left(1 + \frac{p_i - min(p_i, D_i)}{t_d - t_{-1}} \right) \\ &= \sum_{i=1}^{k} \frac{e_i}{p_i} + \sum_{i=1}^{k} \frac{e_i}{p_i} \left(\frac{p_i - min(p_i, D_i)}{t_d - t_{-1}} \right) \\ &\leq \sum_{i=1}^{k} \frac{e_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \frac{e_i}{p_i} \left(p_i - min(p_i, D_i) \right). \end{aligned}$$

The final step follows because $D_k \leq (t_d - t_{-1})$. Thus, (1) is false as claimed. □

This condition can be checked in $O(n)$ time, if tasks are sorted by relative deadline. Otherwise, $\Omega(n \log n)$ time is required.

A task system satisfying (1) may not satisfy the density condition, but if it satisfies the density condition, then (1) will necessarily be satisfied, as the following theorem implies.

Theorem 2 *The schedulability test in Theorem 1 is tighter than the density condition,* i.e., *the set of conditions in Theorem 1 is weaker than the density condition.*

Proof: We prove this by showing that the left-hand side of Inequality (1) is less than or equal to the density of the system, as follows.

$$\begin{aligned} &\sum_{i=1}^{k} \frac{e_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - min(p_i, D_i)}{p_i} \right) \cdot e_i \\ &= \sum_{i=1}^{k} \frac{e_i}{p_i} \left(1 - \frac{min(p_i, D_i)}{D_k} \right) + \frac{1}{D_k} \sum_{i=1}^{k} e_i \\ &\leq \sum_{i=1}^{k} \frac{e_i}{min(p_i, D_i)} \left(1 - \frac{min(p_i, D_i)}{D_k} \right) + \frac{1}{D_k} \sum_{i=1}^{k} e_i \\ &= \sum_{i=1}^{k} \frac{e_i}{min(p_i, D_i)} \end{aligned}$$

□

4 Extensions to Account for Practical Overheads

We now show how to extend the test in Theorem 1 to account for context-switching costs, nonpreemptivity of tasks, contention for resources among tasks, interference by interrupt handlers, self-suspensions, and priority-space limitations, when tasks are scheduled using EDF. Due to space constraints, we omit proofs for the extended conditions. (Proofs can be found in an extended version [5].)

The terms *blocking*, *blocking time*, and *priority inversion* are used in this section with their usual meanings. A *priority inversion* is said to occur when a low-priority job executes while a ready high-priority job waits. The waiting high-priority job is said to be *blocking* [7]. *Blocking time* is the duration for which a high-priority job is prevented from executing due to priority inversions.

4.1 Context Switches

In a preemptable system in which the priority of a job does not change over its lifetime, as in EDF, a job that does not self-suspend preempts at most one lower-priority job. Thus a job J is responsible for at most two context switches, one of which may occur when J starts executing and another when it completes executing. Hence, it is sufficient to assess J the cost of two context switches. Under this accounting scheme, context switches due to preemptions that J may suffer will be charged to the preempting higher-priority jobs. Hence, overhead due to context switches can be fully accounted for if the execution cost of each job is increased by twice the worst-case context-switching time of the system. Thus, we have the following theorem.

Theorem 3 *A preemptable, asynchronous, periodic task system consisting of n tasks with arbitrary relative deadlines arranged in order of non-decreasing relative deadlines, and a worst-case context-switching time CS is schedulable under EDF if*

$$(\forall k : 1 \leq k \leq n :: \sum_{i=1}^{k} \frac{e_i + 2 \cdot CS}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - min(p_i, D_i)}{p_i} \right) \cdot (e_i + 2 \cdot CS) \leq 1).$$

4.2 Nonpreemptivity and Resource Contentions

Nonpreemptable sections and the use of shared locks can cause priority inversions. Though the relative priorities of tasks vary with their instances (jobs) under EDF, the set of tasks with jobs that can block T_i is static; specifically, this set consists of only tasks with relative deadlines greater than D_i, as the following theorem proved in [2] shows.

Theorem 4 [2] *Under EDF scheduling, a job J_k with relative deadline D_k can block a job J_i with relative deadline D_i only if D_k is larger than D_i.*

In the absence of self-suspensions, any job can be blocked by at most one lower-priority job due to nonpreemptivity. Hence, when tasks are arranged in order of non-decreasing relative deadlines (as we have assumed), the maximum blocking time of T_i due to nonpreemptivity, $b_i(np)$, is given by

$$b_i(np) = \max_{i<k\leq n} \theta_k, \quad (2)$$

where θ_k is the maximum execution time of the longest nonpreemptable section of T_k [10].

Blocking times of tasks due to contention for shared resources depend upon the *synchronization mechanism* and the *resource access-control protocol* used. In this subsection, we assume that jobs are synchronized using *lock-based* schemes. (We consider *lock-free* schemes in the next subsection.) We also assume that resources are allocated using the *priority-ceiling protocol* (PCP) [7] or the *stack-based resource policy* (SRP) [2]. These protocols avoid deadlocks and limit the number of times a job is blocked to at most one for the duration of the longest outermost critical section of a lower-priority job.

The maximum blocking time, $b_i(rc)$, of T_i due to resource contention is then given by

$$b_i(rc) = \max_{i<k\leq n} \gamma_{i,k}, \quad (3)$$

where

$$\gamma_{i,k} = \begin{cases} 0, & T_k \text{ has no resource conflicts with } T_1, \ldots, T_i \\ \gamma_k, & \text{otherwise} \end{cases}$$

In the above equation, γ_k is the maximum execution time of the longest outermost critical section of T_k. (Here again, tasks are assumed to be arranged in order of non-decreasing relative deadlines.) To avoid deadlocks, a job of T_i may be blocked by a job of T_k, $k > i$, even if T_i and T_k do not have common resource requirements, but if T_k shares a resource with T_j, $j < i$.

A schedulability test that accounts for blocking times due to nonpreemptivity and resource contention described above is then given by the following theorem. (We assume that a nonpreemptable section does not access a shared resource.)

Theorem 5 *Let τ be an asynchronous, periodic task system consisting of n tasks with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines. Let the tasks in τ share resources using either the PCP or SRP and contain nonpreemptive sections that do not access shared resources, and let their blocking times due to nonpreemptivity and resource contention be given by Equations* (2) *and* (3)*, respectively. τ is schedulable under EDF if*

$$(\forall k : 1 \leq k \leq n :: \frac{b_k(np) + b_k(rc)}{D_k} + \sum_{i=1}^{k} \frac{e_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - min(p_i, D_i)}{p_i} \right) \cdot e_i \leq 1).$$

```
typedef Qtype: record data: valtype; next: pointer to Qtype
shared var Head, Tail: pointer to Qtype
private var old, new: pointer to Qtype; input: valtype;
            addr: pointer to pointer to Qtype

procedure Enqueue(input)
    *new := (input, nil);
    do old := Tail;
       if old ≠ nil then addr := &((*old).next)
          else addr := &Head fi
    while ¬CAS2(&Tail, addr, old, nil, new, new)
```

Figure 2. Lock-free enqueue implementation.

Under the PCP, a job may be blocked after it begins execution, and hence may preempt one more lower-priority job, and thus cause two more context switches, one when it is blocked and a second when it resumes execution (in addition to the two when it begins and finishes executing).

4.3 Lock-free Synchronization

An alternative to lock-based protocols when implementing shared data objects is to use the *lock-free* algorithms. Lock-free algorithms work particularly well for simple objects like buffers, queues, and lists. In such algorithms, object calls are implemented using "retry loops." Fig. 2 depicts a lock-free enqueue operation that is implemented in this way. An item is enqueued in this implementation by using a *two-word compare-and-swap* (CAS2) instruction[1] to atomically update a tail pointer and either the "next" pointer of the last item in the queue or a head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the CAS2 instruction succeeds. An important property of lock-free implementations such as this is that operations may *interfere* with each other. An interference results in this example when a successful CAS2 by one task causes another task's CAS2 to fail.

Anderson *et al.* showed that the number of interferences that a job suffers is bounded under priority-based scheduling algorithms like EDF [1]. Assuming a worst-case execution cost of s for each retry loop, they derived sufficient conditions for scheduling with lock-free shared objects under the EDF, rate-monotonic, and deadline-monotonic scheduling algorithms. When lock-free shared objects are included in our task model, a schedulability test under EDF is given by the following theorem:

Theorem 6 *A preemptable, asynchronous, periodic task system consisting of n tasks with arbitrary relative deadlines, arranged in order of non-decreasing realtive deadlines, and sharing lock-free objects is schedulable under EDF if*

[1] The first two parameters of CAS2 specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed. Although CAS2 is uncommon, it makes for a simple example here.

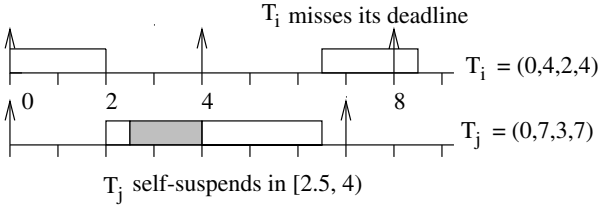

Figure 3. Schedule with a deadline miss due to self-suspension.

$$(\forall k : 1 \leq k \leq n :: \sum_{i=1}^{k} \frac{e_i + s}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - \min(p_i, D_i)}{p_i} \right) \cdot (e_i + s) \leq 1), \quad (4)$$

where s is the worst-case execution time of a retry-loop in the implementation of a lock-free object.

The schedulability test given in [1] requires checking that demand does not exceed the available processor time over any interval whose length, l, is at least the least common multiple of the task periods. If total utilization is known to be strictly less than *one*, then l can be made much shorter by using the ideas in the PPT test of Baruah *et al.* [3]. However, the test would still run in pseudo-polynomial-time, and hence, the sufficient test in Inequality (4) may be preferable if efficiency is a concern.

4.4 Interrupt Handling Costs

Interrupt handlers execute in response to external events at priorities higher than the real-time tasks in the system. Jeffay *et al.* presented a pseudo-polynomial-time exact schedulability test for deadline-driven systems consisting of tasks with relative deadlines equal to periods when interrupt handlers are included and the total system utilization (including interrupt handlers) is less than one [6]. To account for m interrupt handlers, with an interrupt handler I formally modeled as a tuple (c, a), where c is the maximum execution time of I and a is the minimum interval between invocations of I [6], our schedulability test is extended as given in the theorem that follows. We assume that all interrupts are of the same priority, they have no associated deadlines, and the order in which outstanding interrupts are handled is not significant.

Theorem 7 *Let τ be a system of n preemptable, asynchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines and m interrupt handlers $\{(c_1, a_1) \ldots (c_m, a_m)\}$. τ is schedulable using EDF if*

$$(\forall k : 1 \leq k \leq n :: \sum_{i=1}^{k} \frac{e_i}{p_i} + \sum_{j=1}^{m} \frac{c_j}{a_j} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - \min(p_i, D_i)}{p_i} \right) \cdot e_i + \frac{1}{D_k} \sum_{j=1}^{m} c_j \leq 1).$$

4.5 Self-suspensions

A self-suspension by a job, unlike blocking, affects both that job and lower-priority jobs. When a job is self-suspended, it is denied processor time, which reduces the time available for its completion. This leads to a task not behaving like a periodic task, in that it may demand more execution time over some interval than a periodic task. This may also cause a delay in the execution of lower-priority jobs, which may in turn miss their deadlines [10]. It is shown in [10] how to account for self-suspensions in fixed-priority systems by considering delays due to self-suspensions as blocking factors. We adopt a similar approach here to account for self-suspensions under EDF.

If ξ_k is the maximum self-suspension time of a job $J_{k,l}$ of T_k, then the maximum delay in the execution time of jobs of priority lower than $J_{k,l}$ due to self-suspensions by $J_{k,l}$ is equal to $\min(\xi_k, e_k)$ [10]. Since the relative priorities of tasks vary with their instances (jobs) under EDF, a self-suspension by a task can impact all other tasks (unlike blocking due to nonpreemtivity or critical sections, in which a task can be blocked only by those with larger relative deadlines). Fig. 3 shows a schedule in which self-suspension in T_j causes T_i to miss its deadline, where $D_j > D_i$. The maximum delay in a job of T_j, due to self-suspensions by jobs of tasks $T_1, \ldots, T_k$, $1 \leq j \leq k$, is given by

$$s_j = \sum_{1 \leq i \leq k \wedge i \neq j} \min(\xi_i, e_i) + \xi_j.$$

Let
$$S_k = \sum_{i=1}^{k} \min(\xi_i, e_i) \quad (5)$$

and
$$S'_k = \max_{1 \leq i \leq k} (max(0, \xi_i - e_i)). \quad (6)$$

Then, the maximum time that a job of tasks $T_1, \ldots, T_k$ could be delayed for in an interval in which only jobs of $T_1, \ldots, T_k$ execute is upper-bounded by $S_k + S'_k$. ($S_k + S'_k \geq s_j$, for all $j, 1 \leq j \leq k$.) The following theorem gives a schedulability test to account for self-suspensions.

Theorem 8 *Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a system of n preemptable, asynchronous, periodic tasks, with arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines. Let ξ_i be the maximum self-suspension time of T_i. τ is schedulable using EDF if*

$$\forall k : 1 \leq k \leq n :: \frac{S_k + S'_k}{D_k} + \sum_{i=1}^{k} \frac{e_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - min(p_i, D_i)}{p_i} \right) \cdot e_i \leq 1),$$

where S_k and S'_k are given by Equations (5) and (6), respectively.

Self-suspensions also result in an increase in the number of context switches and blockings due to nonpreemptivity and resources contentions. If the maximum number of times

each task self-suspends is the same, say M, then increasing the overhead due to context switches by a factor of $2M$ and blocking overheads by a factor of M would suffice. However, if the maximum number of self-suspensions varies, then determining blocking overheads accurately becomes complicated.

4.6 Limited Priorities

The number of priorities that an actual system supports may be less than the number of distinct priorities of the tasks that are to be scheduled in that system. As a result, jobs that would otherwise be assigned distinct priorities may be assigned the same priority, leading to a loss in schedulable utilization.

Let n denote the number of distinct task priorities required. Then, under EDF, the number of distinct relative deadlines among the tasks is equal to n. One way to implement EDF is using n FIFO queues, one for each distinct relative deadline [10]. Ready jobs with the same relative deadline are appended to the queue associated with that deadline, which ensures that these jobs are ordered among themselves according to their absolute deadlines. Hence, the job with the earliest deadline can be found by examining the job, if any, at the head of each queue. Let m be the number of system priorities. If $m < n$, then the number of FIFO queues is less than n and EDF ceases to support naturally occurring priorities. In such a case, jobs of tasks with different relative deadlines are made to share a single FIFO queue. This artificially advances the absolute deadlines of some jobs, thereby increasing their priorities and scheduling them in preference to other jobs that would be scheduled if there were n queues. The jobs with higher priorities in the unlimited-priority domain could miss their deadlines, as a result. Hence, if $m < n$, then the schedulability test of Theorem 1 is not sufficient. We now show how to extend it in a straightforward manner.

Some notation is in order. If $m < n$, then the n task priorities (or relative deadlines, in the case of EDF) have to be mapped onto m system priorities (or relative deadlines). Let $\pi_1, \pi_2, \ldots, \pi_m$ denote the m system deadlines, where $1 \leq \pi_k \leq n$, $1 \leq k \leq m$, and $\pi_j < \pi_k$, if $j < k$. The set $\{\pi_1, \pi_2, \ldots, \pi_m\}$ can be thought of as a deadline mapping grid [10]. The tasks are mapped onto the grid such that tasks with indices less than or equal to π_1 (tasks $T_1, \ldots, T_{\pi_1}$) are assigned to the first FIFO queue and are given a relative deadline equal to that of T_1, and all tasks with indices in the range $[\pi_{k-1} + 1, \pi_k]$ are assigned to the k^{th} FIFO queue and are given a relative deadline equal to that of $T_{\pi_{k-1}+1}$. π_0 is set to 0 and π_m to n.

A sufficient test for scheduling tasks with n distinct priorities on a system supporting $m < n$ priorities is given by the theorem that follows.

Theorem 9 *A system of n preemptable, asynchronous, periodic tasks, with distinct arbitrary relative deadlines, arranged in order of non-decreasing relative deadlines and mapped onto $m < n$ relative deadlines supported by the system per the mapping grid $\{\pi_1, \pi_2, \ldots, \pi_m\}$, is schedulable under EDF if*

$$(\forall g : 1 \leq g \leq m :: \sum_{i=1}^{\pi_g} \frac{e_i}{p_i} + \frac{1}{D_{\pi_{g-1}+1}} \sum_{j=1}^{g} \sum_{i=\pi_{j-1}+1}^{\pi_j} \left(\frac{p_i - min(p_i, D_{\pi_{i-1}+1})}{p_i} \right) \cdot e_i \leq 1).$$

The time complexity of the test in Theorem 1 is not altered when it is extended to account for one or more of the practical factors described in this section.

5 Experimental Evaluation

In this section, we describe experiments that were conducted to evaluate our proposed test.

5.1 Description of Test Data and Experiments

We compared the density test, our proposed test, and the PPT test (pseudo-polynomial-time test) of Baruah *et al.* [3] by generating groups of independent random task sets and subjecting the task sets in all groups to all three tests. Over 16,000 random task sets were generated for each group. The number of tasks in a task set varied between five and a thousand, but was fixed for all task sets in a group. Thus, the groups differed in the number of tasks in their task sets. The total utilization of each task set was uniformly distributed between 1% and 100%. For conciseness, we shall refer to the difference between a task's period and relative deadline as its *gap*. Each task set's average gap was selected using a uniform distribution to be between 0% and 80% of its period. The number of task sets declared schedulable by each test and its running time were calculated. Results are shown in the plots in Figs. 4 and 5.

5.2 Comparison with the Density Test

The variations in the percentage of task sets declared schedulable for both increasing utilizations and increasing gap values for the density test and the proposed test, for all groups, are shown in Fig. 4. The figure also depicts these variations for the PPT test, which we discuss in the next subsection. Graphs of individual groups (not shown in the paper) also showed trends similar to those shown in Fig 4. This is in accordance with the intuituion that schedulability is a function of utilization and gap values, and is independent of the number of tasks in a task set. Hence, we do not plot the variation in the percentage of task sets schedulable against the number of tasks (i.e., the groups).

When utilization is less than 20%, both the density and proposed tests admit almost the same percentage of task sets, as can be observed in Fig. 4(a). However, with increasing utilizations, the difference in effectiveness between the two tests increases, with our proposed test finding a larger percentage of task sets to be schedulable. This difference reaches a peak of around 20% when utilization is 50%, and then starts decreasing slowly. The plot normalized with respect to the PPT test is shown in Fig. 4(b).

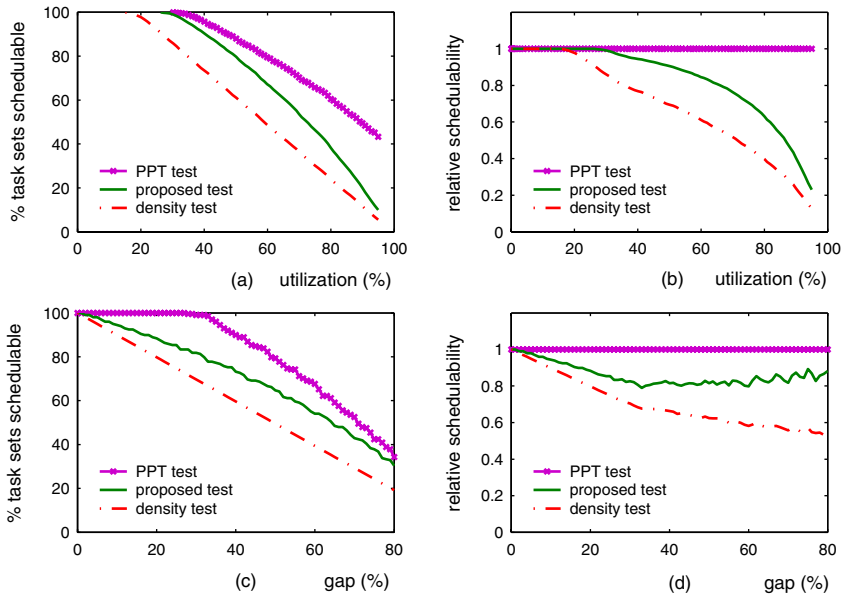

Figure 4. (a) Schedulability by average gap. (b) Relative schedulability by average gap. (c) Schedulability by utilization. (d) Relative schedulability by utilization.

Similarly, when the gap is negligible, the difference in effectiveness of the two tests is not appreciable, as can be observed from Fig. 4(c). Here again, as the difference increases, the new test starts admitting a larger percentage of task sets. The difference reaches a peak of 15% when the average gap is 60%, and decreases very slowly after that. The same plot normalized with respect to the PPT test is shown in Fig. 4(d).

From the plots we can infer that when utilization is high and the gap is large, the new test correctly predicts the schedulability of a significantly higher percentage of task sets (than the density test).

5.3 Comparison with the Pseudo-polynomial-time Test

Fig. 4 shows the variation in the percentage of task sets declared schedulable for increasing utilizations and increasing gap values for the PPT test also (in addition to those for the density test and the proposed test). The plots show that the proposed test admits more than 80% of the schedulable task sets at all gap values. However, it does not perform well at high utilization, *e.g.,* it admits only slightly over 20% of the schedulable task sets when utilization is higher than 95%. The tests were also timed on a 933MHz machine. The average running time for the PPT and the proposed tests are plotted against utilization and average gap for Group 100, the group with a hundred tasks per task set, in Figs. 5(a) and 5(b), and for Group 1000, the group with a thousand tasks per task set, in Figs. 5(c) and 5(d). The running time of the proposed test was of the order of microseconds for both groups, and hence its graph coincides with the horizontal axis. For Group 100, the running time of the PPT test was of the order of hundreds of milliseconds, at high utilizations. The running time was higher by more than two orders of magnitude (tens of seconds) for Group 1000. On the other hand, the proposed test took less than a tenth of a millisecond for Group 100 and less than a millisecond for Group 1000. High test execution times may not be acceptable in online admission control tests with stringent timing requirements, and hence, we can conclude that the choice of which test to use is largely a trade-off between schedulability and efficiency. If the practical overheads described in Sec. 4 have to be accounted for, then the time complexity of the PPT test deteriorates to $O(n^2 \max(p_i - Di))$ and it also ceases to be an exact schedulability test. Using our proposed test under such conditions may be advantageous.

6 Concluding Remarks

In this paper, we have presented a new sufficiency test for determining the schedulability of preemptable, asynchronous, periodic task systems with arbitrary relative deadlines. We have also shown that this test is more accurate than the density condition, both analytically and empirically. We have also empirically compared our test to the pseudo-polynomial-time test of Baruah *et al.* (PPT test) [3] for a restricted class of task systems in which utilization is strictly less than one. We have found that at high utilizations the relative schedulability of the proposed test is low while the execution time of the PPT test is higher by more than three orders of magnitude. Hence, we conclude that the choice of which test to use is a trade-off between accurate and efficient

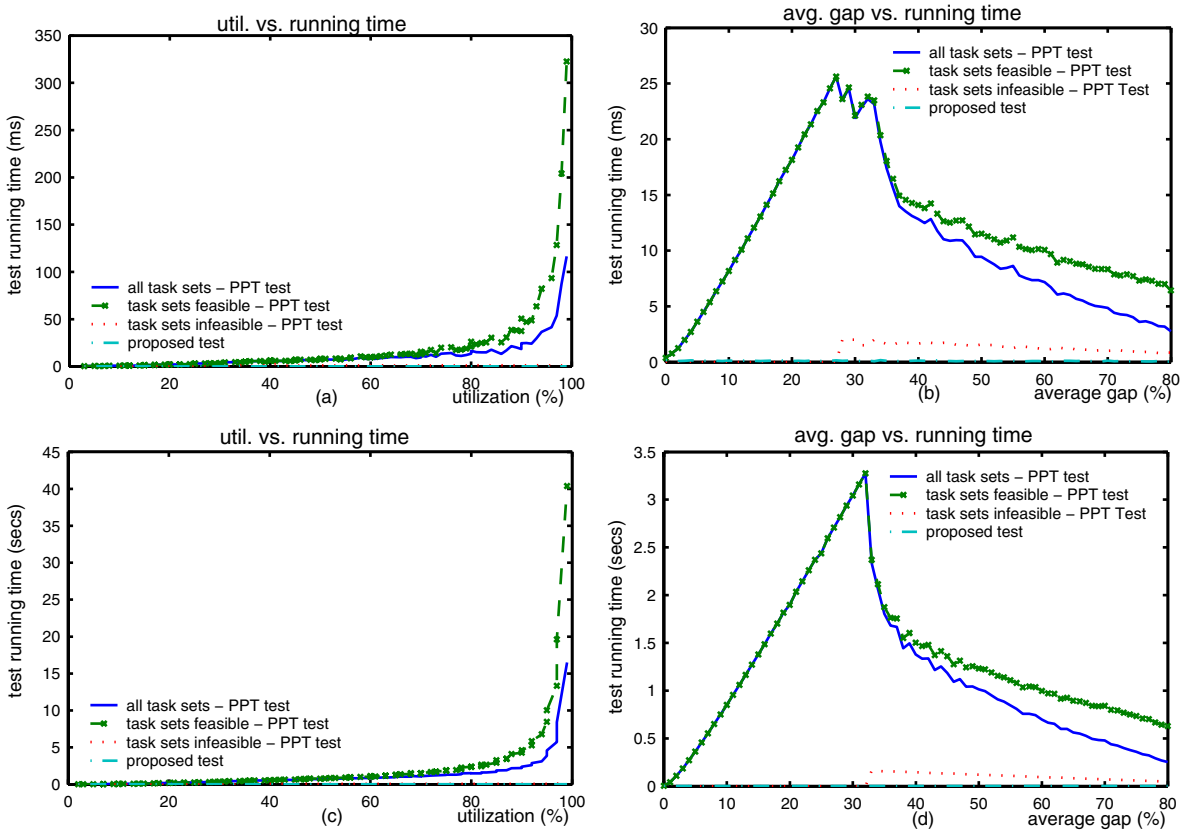

Figure 5. (a) & (b) Average running times for PPT test with 100 tasks per task set. (a) Running time by utilization. (b) Running time by average gap. (c) & (d) Average running times for PPT test with 1000 tasks per task set. (c) Running time by utilization. (d) Running time by average gap.

determination of schedulability. In addition, the running time of the PPT test may be quite high if overheads due to practical factors have to be accounted for.

Acknowledgements: I am grateful to Jim Anderson for his encouragement and helpful suggestions with this work, and for his comments on earlier drafts of this paper.

References

[1] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[2] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1):64–96, March 1991.

[3] S.K. Baruah, R.R. Howell, and L.E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 20(3):3–20, 1993.

[4] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of IFIP Cong.*, pages 807–813, 1974.

[5] U. Devi. An improved schedulability test for uniprocessor periodic task systems. Unpublished Manuscript. Available at `http://www.cs.unc.edu/~uma/papers.html`.

[6] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 212–221, 1993.

[7] Sha L., R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39, 1990.

[8] J.Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[9] 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):46–61, 1973.

[10] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[11] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, 1996.