Schedulability Analysis of Non-Preemptive Recurring Real-time Tasks

Sanjoy K. Baruah
University of North Carolina at Chapel Hill
E-mail: baruah@cs.unc.edu

Samarjit Chakraborty
National University of Singapore
E-mail: samarjit@comp.nus.edu.sg

Abstract

The recurring real-time task model was recently proposed as a model for real-time processes that contain code with conditional branches. In this paper, we present a necessary and sufficient condition for uniprocessor non-preemptive *schedulability analysis for this task model. We also derive a polynomial-time approximation algorithm for testing this condition. Preemptive schedulers usually have a larger schedulability region compared to their non-preemptive counterparts. Further, for most realistic task models, schedulability analysis for the non-preemptive version is computationally more complex compared to the corresponding preemptive version. Our results in this paper show that (surprisingly) the recurring real-time task model does not fall in line with these intuitive expectations, i.e. there exists polynomial-time approximation algorithms for both preemptive and non-preemptive versions of schedulability analysis. This has important implications on the applicability of this model, since fully preemptive scheduling algorithms often have significantly larger runtime overheads.*

1 Introduction

Many real-time embedded systems contain concurrently executing blocks of code that run in an infinite loop and get triggered by external events. Depending on the event type, different parts of such code blocks get executed. Recently, a task model for such a setup was proposed in [2] and was called the *recurring real-time task model*. It is particularly suitable for representing embedded code as it allows the modeling of conditional branches and fine-grain deadline constraints. However, it turns out that the schedulability analysis problem for this task model is computationally intractable and at best can be solved in pseudo-polynomial time [9]. Clearly, this restricts its applicability to real-life setups. To get around this problem, a scheme for *approximate* schedulability analysis was proposed in [11]. It runs in polynomial time and has the following property. For some task sets which are *not* schedulable, it incorrectly returns *schedulable*. However, in such cases it is guaranteed that at run-time no task would miss its deadline by more than a prespecified error value. The smaller the value of this specified error parameter, higher is the running time of the algorithm (albeit always polynomial in the problem size). The scheme can also be modified to err in the opposite direction, i.e. task sets which are not schedulable always result in the correct answer but some schedulable task sets are incorrectly labeled as *not* schedulable. Such task sets are those that lead to a very high processor utilization, and here the error parameter can be formulated in terms of this utilization.

The schedulability analysis algorithm proposed in [2], as well as the algorithms for approximate schedulability analysis in [11] only handle the fully *preemptive* version of the problem. However, unrestricted task preemptions always result in significant overheads in terms of context saving and switching and may not be feasible in many applications, particularly in an embedded systems setup. It would be far more realistic if the schedulability analysis can take into account the constraint that preemptions are only allowed at task boundaries. For most realistic task models, deriving the conditions for *non-preemptive schedulability analysis* are significantly more complex than their preemptive counterparts. Further, the resulting algorithms are also computationally more complex [8]. As a result, although there exists a large volume of work on preemptive schedulability analysis for various task models [1, 7, 6, 14, 15], relatively little is known about their corresponding non-preemptive versions. Among previous studies on non-preemptive schedulability analysis, we are aware of [13] and [12], which considered the sporadic task model with implicit and explicit deadlines. Very recently, the non-preemptive scheduling of periodic tasks was studied in [4, 3].

Our contributions: In this paper we study the non-preemptive schedulability analysis problem for the recurring real-time task model. Our main result is fairly counter-intuitive. We show that the schedulability analysis of any collection of concurrently executing recurring real-time tasks can be reduced to at most a polynomial number of preemptive schedulability analysis problems. This implies that *for the recurring real-time task model, the non-preemptive schedulability analysis problem is no more difficult than its preemptive counterpart, at least from an asymptotic-complexity standpoint*. A less surprising result would have been that the non-preemptive constraint results in an exponentially large number of additional schedules that might need to be considered in order to verify the schedulability

of a task set. These additional schedules would typically arise from different orderings of blocking tasks.

In addition, we show that it is possible to derive polynomial-time approximation algorithms for the non-preemptive version as well. Unfortunately, the running times of the approximation algorithms (both for the preemptive, as well as for the non-preemptive cases) turn out to be high-degree polynomials. However, we believe that with more aggressive notions of approximation, it might be possible to substantially reduce the running times of these algorithms (which is an important direction for future work).

Apart from being an interesting intellectual exercise in its own right, our results in this paper have important implications on the applicability of the recurring real-time task model. Further, these results are especially interesting because the recurring real-time task model generalizes a number of well-known task models such as the sporadic [14], multiframe [15], generalized multiframe [6] and recurring branching [1]. Recently, [10] studied the non-preemptive schedulability analysis of a task model whose syntax is similar to the recurring real-time task model. However, the tasks in this model do not execute in an infinite loop (i.e. the recurring behavior was not modeled).

Organization of the paper: The rest of the paper is organized as follows. We briefly introduce the recurring real-time task model in Section 2 and discuss some of its properties. The main result of this paper (which we outlined above) is derived in Section 3. Using this result, we obtain an algorithm for the exact non-preemptive schedulability analysis of recurring real-time tasks in Section 4. Finally, we discuss our approximation algorithms in Sections 5 and 6. Due to space constraints, some of the proofs and the statement of the algorithm for exact non-preemptive schedulability analysis have been omitted here; they may be found in [5].

2 Recurring Real-Time Task Model

A *recurring real-time task* T is represented by a *task graph* $G(T)$ and a *period* $P(T)$. The task graph $G(T)$ is a directed acyclic graph (DAG) with a unique *source* vertex and a unique *sink* vertex. Vertices in this DAG represent subtasks, and each edge represents a possible flow of control. Each vertex u is labeled by two integer parameters $e(u)$ and $d(u)$ with the following interpretation: each time subtask u is triggered, a *job* is generated with ready-time equal to the triggering-time of the subtask, an execution requirement of $e(u)$ time units and a deadline $d(u)$ time-units after the triggering-time. Each edge (u, v) is labeled by an integer parameter $p(u, v)$ denoting the minimum amount of time that must elapse after vertex u is triggered, before vertex v can be triggered. To simplify our presentation, we will add the restriction that $p(u, v) \geq d(u)$ for each edge (u, v) in the DAG — this restriction is related to the *frame separation property* [16], which is a requirement that a task may generate a job only after the deadline of the previous job generated by it has elapsed.

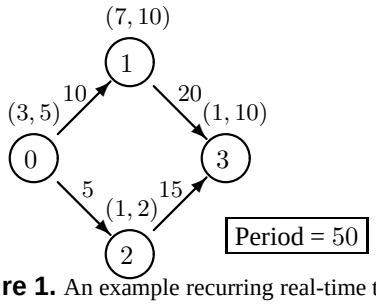

Figure 1. An example recurring real-time task.

The execution semantics of a recurring real-time task may be described as follows. When a subtask u is triggered, it generates a job which needs to be executed on a shared processor for $e(u)$ units of the next $d(u)$ units of time. Initially the source vertex of the DAG may be triggered at any time. Suppose that vertex u is triggered at time t, then: (i) if u is *not* the sink vertex of $G(T)$, then the next vertex of $G(T)$ to be triggered is some vertex v such that (u, v) is an edge in $G(T)$; vertex v is triggered at or after time $t + p(u, v)$, (ii) if u is the sink vertex of $G(T)$, then the next vertex of $G(T)$ to be triggered is the source vertex; it can be triggered at any time after t subject to the constraint that at least $P(T)$ time units should have elapsed since its last triggering.

We can use a 3-tuple (T, t, u) to denote the fact that subtask (or vertex) u of the recurring real-time task T is triggered at time-instant t; such a 3-tuple will be called an *event*. Further, we will use $E(T)$ to denote the maximum possible cumulative execution requirement on any path in a task graph $G(T)$, from its source node to its sink node. $\rho_{\text{ave}}(T)$ will be used to denote the quantity $E(T)/P(T)$ and will be referred to as the *utilization* of T. The *system utilization* of a set of tasks τ is defined as $\rho_{\text{ave}}(\tau) \stackrel{\text{def}}{=} \sum_{T \in \tau} \rho_{\text{ave}}(T)$.

Finally, an example recurring real-time task is shown in Figure 1. The ordered pair above each vertex u represents its execution requirement and the relative deadline – $(e(u), d(u))$. The single integer on each edge (u, v) represents the associated inter-triggering separation $p(u, v)$. The period of the task — the minimum time that must elapse between successive triggerings of the source node — is 50. For this task T, $E(T)$ is equal to $7 + 3 + 1 = 11$, and $\rho_{\text{ave}}(T) = 11/50 = 0.22$.

The non-preemptive constraint that we consider in this paper allows for preemption only at task boundaries. In this example, it implies that jobs generated as a result of triggering any of the vertices $0, \ldots, 3$, once scheduled on the processor, will have to be run till completion.

The demand bound function: Our schedulability analysis algorithms rely on the computation of the maximum *de-*

mand that can be generated by a task within time intervals of different lengths. The resulting *demand bound function* of a recurring real-time task quantifies the maximum amount of execution that can be required by jobs of the task in any interval of a given size.

Definition 1 (Demand Bound Function) *Let T be a task, and t a non-negative integer. The* demand bound function $\text{DBF}(T,t)$ *denotes the maximum cumulative execution requirement by jobs of T that have both ready times and deadlines within any time interval of duration t.*

■

Further, for any vertex $v \in T$, we denote by $\text{DBF}^v(T,t)$, the maximum cumulative execution requirement demanded by T within any time interval of length t, due to any triggering sequence ending at the vertex v. We will see in Section 3 that efficient computation of the demand bound functions is critical for the schedulability analysis of systems of non-preemptive recurring real-time tasks. Demand bound functions were also used in *preemptive* schedulability analysis, and have previously been studied in that context.

We close this section with a lemma that provides a (not necessarily tight) upper bound on the value of the demand bound function.

Lemma 1 *For any task T and any $t \geq 0$,*

$$dbf(T,t) \quad \leq \quad 2 \cdot E(T) + t \cdot \rho_{ave}(T) \qquad (1)$$

Proof Sketch: Any legal event sequence σ of task T can be considered to be the concatenation of at most three subsequences, as follows.

- The first subsequence ends with the first occurrence of the job corresponding to the triggering of the source vertex. If a job corresponding to the source vertex is not present in σ, then the second and third subsequences are empty.
- The third subsequence begins with the job immediately following the last occurrence of the job corresponding to the triggering of the source vertex. If there is exactly one job corresponding to the source vertex in σ, then the third subsequence is empty.
- The second subsequence begins with the job immediately following the first occurrence of the job corresponding to the triggering of the source vertex, and ends with the last occurrence of the job corresponding to the triggering of the source vertex.

The lemma follows from the observation that the maximum cumulative execution requirement of jobs in the first and third subsequences is $E(T)$ each, and that the maximum cumulative execution requirement of jobs in the second subsequence is at most $t \cdot \rho_{\text{ave}}(T)$. ■

3 Properties of Non-Schedulable Systems

In what follows, we will only be concerned with (uniprocessor) schedulability analysis under the earliest deadline first (EDF) scheduling policy. This is because EDF is known to be optimal for the preemptive case and it is also optimal for the non-preemptive case if ready jobs are not kept waiting when the processor is free (which is a policy that is used in any practical implementation).

Since a system of recurring real-time tasks may generate infinitely many different legal collections of jobs, it is not possible to explicitly check each such legal collection for EDF-schedulability. In the remainder of this section, we will identify characteristics that are shared by all systems that are not EDF-schedulable; in the following sections, we will design algorithms for efficiently recognizing task systems that possess these properties.

Let τ denote a system of recurring real-time tasks comprised of n tasks T_1, T_2, ..., T_n. Let us suppose that this system is *not* schedulable under EDF; we will now derive certain properties that τ must satisfy as a consequence. Since τ is not schedulable using EDF, there are legal sequences of jobs generated by τ for which EDF would miss some deadline. Let $\sigma(\tau)$ denote one such legal collection of jobs, satisfying the following additional properties:

- It is the smallest such collection, in the sense of having the fewest number of jobs in it.
- Let σ' denote any legal collection of jobs from τ, let t_o denote the earliest arrival time of any job in σ', and let us define the *relative arrival time* of any job in σ' to be the difference between its (actual) arrival time and t_o. $\sigma(\tau)$ is then a smallest legal collection of jobs from τ for which EDF misses one or more deadlines and for which the cumulative sum of the relative arrival times of the jobs is the minimum.

Let t_f denote the instant at which a deadline is missed in the EDF schedule for $\sigma(\tau)$. Let $t_a < t_f$ denote the earliest arrival time of any job in $\sigma(\tau)$.

Claim 1 *The processor is never idle over $[t_a, t_f)$ in the* EDF *schedule for $\sigma(\tau)$.*

Proof Sketch: Suppose that the processor were to be idled at time-instant t', $t_a < t' < t_f$. By definition of EDF, it must be the case that no jobs are active — i.e., have arrived but not yet been executed – at time-instant t'. Observe that the legal collection of jobs comprised of all jobs in $\sigma(\tau)$ that arrived after t' will also miss a deadline at time-instant t_f if executed by EDF. But this contradicts the claimed minimality of $\sigma(\tau)$. ■

Corollary 1 follows immediately:

Corollary 1 *The sum of the execution requirements of jobs in $\sigma(\tau)$ exceeds $t_f - t_a$.*

Claim 2 *There is at most one job in $\sigma(\tau)$ with deadline greater than t_f. If there is such a job, then it arrives at the time-instant t_a and is the first job to be executed in the* EDF *schedule for $\sigma(\tau)$.*

Proof Sketch: Suppose that there is more than one job in $\sigma(\tau)$ with deadline $> t_f$, and consider the time interval $[t_1, t_2]$ during which the *last* such job is scheduled in the EDF schedule of $\sigma(\tau)$. By definition of EDF, it must be the case that no jobs with deadline $\leq t_f$ are active — i.e., has arrived but not yet been executed – at time-instant t_a. It is not hard to see that the legal collection of jobs comprised of this job that executes over $[t_1, t_2]$ arriving at time-instant t_1, plus all jobs in $\sigma(\tau)$ that arrived after t_1, will also miss a deadline at time-instant t_f if executed by EDF. But this collection of jobs contradicts the claimed minimality of $\sigma(\tau)$. ■

We thus see that there are zero or one jobs in $\sigma(\tau)$ with deadline greater than t_f. Claims 3 and 4 below consider these two cases separately.

Claim 3 *Suppose that there are no jobs in $\sigma(\tau)$ with deadline $> t_f$. If task T_i ($1 \leq i \leq n$) has one or more jobs in $\sigma(\tau)$ then the first job generated by T_i in $\sigma(\tau)$ arrives at the time-instant t_a, and subsequent jobs arrive as soon as it is legal.*

Proof Sketch: We prove this claim by contradiction. Suppose then that T_i has one or more jobs in $\sigma(\tau)$, but these jobs do not each arrive as soon as possible. Consider the collection $\sigma'(\tau)$ obtained from $\sigma(\tau)$ by moving the arrival-times of all the jobs of T_i to occur as soon as possible, within the interval $[t_a, t_f)$. Notice that the largest deadline of any job in $\sigma'(\tau)$ is no more than t_f. Observe, too, that EDF must successfully schedule $\sigma'(\tau)$, since the cardinality of $\sigma'(\tau)$ is equal to the cardinality of $\sigma(\tau)$, and the sum of the relative arrival times of jobs in $\sigma'(\tau)$ is smaller than the sum of the relative arrival times of jobs in $\sigma(\tau)$. But this contradicts Corollary 1, which asserts that the sum of the execution requirements of the jobs in $\sigma(\tau)$ (and hence, $\sigma'(\tau)$ as well) exceeds $t_f - t_a$. ■

Claim 4 *Suppose that there is one job in $\sigma(\tau)$ with deadline greater than t_f. Let this job be generated by task T_j. If task T_i ($1 \leq i \leq n, i \neq j$) has one or more jobs in $\sigma(\tau)$ then the first job generated by T_i in $\sigma(\tau)$ arrives at time-instant $t_a + \epsilon$, where ϵ is a positive real number arbitrarily close to zero, and subsequent jobs arrive as soon as legal.*

Since the cumulative execution requirement of all the jobs of task T_i in $\sigma(\tau)$ with arrival times $\geq t_a$ and deadlines $\leq t_f$ is—by very definition of the demand bound function—no larger than $\text{DBF}(T_i, t_f - t_a)$, the following two claims follow directly from Claims 3 and 4.

Claim 5 *Suppose that there are no jobs in $\sigma(\tau)$ with deadline $> t_f$. Then*

$$\sum_{i=1}^{n} \text{DBF}(T_i, t_f - t_a) > t_f - t_a \tag{2}$$

Claim 6 *Suppose that there is one job in $\sigma(\tau)$ with deadline $> t_f$. Let this job be generated by task T_j, and have an execution requirement e_j and relative deadline d_j. Then*

$$e_j + \sum_{\substack{i=1 \\ i \neq j}}^{n} \text{DBF}(T_i, t_f - t_a) > t_f - t_a \ . \tag{3}$$

Furthermore, $t_f - t_a < d_j$.

Recall that we had constructed $\sigma(\tau)$ under the assumption that τ is not schedulable under EDF. In essence, Claims 5 and 6 assert that if τ is not schedulable under EDF, then *either* there are t_f and t_a, $t_f - t_a > 0$, such that inequality 2 holds, *or* there is a recurring real-time task T_j with a node labeled (e_j, d_j), and t_f and t_a, $0 < t_f - t_a < d_j$, such that inequality 3 holds.

The converse of the above statement also holds: if inequality 2 or 3 evaluates to true, then τ is not schedulable — the sequences of jobs that define the corresponding DBF's can be combined to come up with a legal collection of job arrivals that is not schedulable under EDF.

Based upon the above discussion, we can now formulate an exact — i.e., necessary and sufficient — condition for determining if a given system of recurring real-time tasks is not schedulable under EDF. This condition is formally stated in Theorem 1 below.

Theorem 1 *System of recurring real-time tasks $\tau = \{T_1, \ldots, T_n\}$ is not schedulable under* EDF *if and only if*

$$\exists\, t_\phi \ \geq 0 : \sum_{i=1}^{n} \text{DBF}(T_i, t_\phi) > t_\phi \tag{4}$$

or there is a T_j, $1 \leq j \leq n$, which can generate a job with execution-requirement e_j and relative deadline d_j, and

$$\exists\, t_\phi \ : 0 \leq t_f < d_j : e_j + \sum_{\substack{i=1 \\ i \neq j}}^{n} \text{DBF}(T_i, t_\phi) > t_\phi \tag{5}$$

■

4 Non-Preemptive Schedulability Analysis

Suppose that a system τ of recurring real-time tasks is not schedulable. It follows from Theorem 1 that either Condition 4, or one of the n conditions from Condition 5 must

hold. Suppose that Condition 4 holds. It follows that

$$\sum_{i=1}^{n} \text{DBF}(T, t_\phi) > t_\phi$$
$$\Rightarrow \quad \text{(From Inequality 1 in Lemma 1)}$$
$$\sum_{i=1}^{n} (2 \cdot E(T) + t_\phi \cdot \rho_{\text{ave}}(T)) > t_\phi$$
$$\Rightarrow \quad t_\phi \left(1 - \sum_{i=1}^{n} \rho_{\text{ave}}(T)\right) < \sum_{i=1}^{n} 2 \cdot E(T)$$
$$\equiv \quad t_\phi < \frac{2}{1 - \rho_{\text{ave}}(\tau)} \times \sum_{i=1}^{n} E(T) \qquad (6)$$

Furthermore, if Condition 5 holds for a specific j ($1 \leq j \leq n$), then it must be the case that the t_ϕ which causes Condition 5 to hold has a value less than d_j. This bound on d_j is clearly pseudo-polynomial in the parameters of τ. The bound given by inequality 6 may in general be exponential in the parameters of τ. However, it is pseudo-polynomial if the system utilization $\rho_{\text{ave}}(\tau)$ is *a priori* bounded from above by a constant less than one. Any system τ with $\rho_{\text{ave}}(\tau)$ greater than one is clearly not schedulable. Hence, requiring that $\rho_{\text{ave}}(\tau)$ be at most c, for some constant $c < 1$, is in effect "wasting" at most a fraction $(1-c)$ of the processor's computation bandwidth. We therefore obtain the following theorem.

Theorem 2 *If a system of recurring real-time tasks $\tau = \{T_1, \ldots, T_n\}$, with system utilization $\rho_{ave}(\tau)$ a priori bounded from above by a constant strictly less than one is not schedulable under* EDF*, then there is a t_ϕ of value pseudo-polynomial in the parameters of τ which will cause one of the $(n+1)$ conditions in Theorem 1 to hold.*

■

On the other hand, a task set is not schedulable in a *preemptive* setting if and only if a condition of the form of inequality 4 holds (i.e. inequality 5 need not be tested), which reduces the number of tests required by a polynomial factor.

An algorithm for schedulability analysis: Using the necessary and sufficient conditions for schedulability that we derived above, it is now possible to design an algorithm that takes as an input a set of recurring real-time tasks and returns if the set is schedulable or not. Details of this algorithm and its proof of correctness may be found in [5]. This algorithm uses the two functions $\text{DBF}(T, t)$ and $\text{DBF}^v(T, t)$ that we introduced in Section 2.

5 Approximate Schedulability Analysis

The exact schedulability test derived above relies on the functions DBF and DBF^v. For any task graph T, computing the value of $\text{DBF}(T, t)$ for some values of t might involve multiple traversals (loops) through the task graph. It was shown in [2] that if for a task graph T, $\text{DBF}(T, t)$ is known for all *small values* of t then it is possible to calculate from these, the value of $\text{DBF}(T, t)$ for any t. *Small* values of t for a task graph T are those for which the sequence of vertices that contribute towards computing $\text{DBF}(T, t)$ contain the source vertex at most once. The value of $\text{DBF}(T, t)$ for larger values of t is made up of some multiple of $E(T)$ plus $\text{DBF}(T, t')$ where t' is *small* in the sense described above. It follows that $\text{DBF}(T, t)$ for any t can be computed as follows (for a more detailed description, see [2])

$$\text{DBF}(T, t) = \max\{\lfloor t/P(T) \rfloor E(T) + \text{DBF}(T, t \bmod P(T)),$$
$$(\lfloor t/P(T) \rfloor - 1)E(T) + \text{DBF}(T, P(T) + t \bmod P(T))\} \qquad (7)$$

To compute $\text{DBF}(T, t)$ for small values of t, [2] constructs a new task graph by taking two copies of the task graph of T and adding an edge from the sink vertex of the first graph to the source vertex of the second and finally replacing the source vertex of the first with a *dummy* vertex with execution requirement and deadline equal to zero. The intertriggering separations on all edges outgoing from this source vertex is also made equal to zero. $\text{DBF}(T, t)$ for all values of t are then calculated by enumerating all possible paths in this new graph. In [9] it was shown that the problem of computing $\text{DBF}(T, t)$ for a task T is NP-hard and a fully polynomial-time approximation scheme (FPTAS) for computing it was given. In the next section we describe this algorithm and show how it can be used for approximate schedulability analysis for non-preemptive recurring real-time tasks. Note that the algorithm presented in [9] considered only the preemptive case, and additionally did not consider the recurring behavior of the tasks. The derivation of the FPTAS first requires a pseudo-polynomial time algorithm.

5.1 Approximating the Demand Bound Function

We first give an algorithm for computing the demand-bound function of a task graph for small values of t. Using this, we can compute the demand-bound function for any value of t as described above.

Given a task graph T, let T' denote the graph formed by joining two copies of T by adding an edge from the sink vertex of the first graph to the source vertex of the second, and replacing the source vertex of the first copy by a dummy vertex as described above. If the frame separation property is followed then the newly added edge is labeled with an intertriggering separation of $p = d(v_{sink})$, where v_{sink} denotes the sink vertex of T. Now we give a pseudo-polynomial time algorithm based on dynamic programming, for computing $\text{DBF}(T', t)$ for values of t that do not involve any looping through T', i.e. we consider only *one-shot* executions of T'.

Let there be n vertices in T' denoted by $v_1, \ldots, v_n$, and without any loss of generality we assume that there can be a directed edge from v_i to v_j only if $i < j$. Following our notation described in Section 2, associated with each

Algorithm 1 Computing $\text{DBF}(T', t)$

Input: Task graph T', and a real number $t \geq 0$

for $e \leftarrow 1$ to nE **do**

$t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$

$t^1_{1,e} \leftarrow t_{1,e}$

end for

for $i \leftarrow 1$ to $n-1$ **do**

for $e \leftarrow 1$ to nE **do**

Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}

$t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \mid j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$

$t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$

end for

end for

$\text{DBF}(T', t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$

vertex v_i is its execution requirement $e(v_i)$ which here is assumed to be integral (a pseudo-polynomial algorithm is meaningful only under this assumption), and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$.

Let $t_{i,e}$ be the minimum time interval within which the task T' can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$, if all the triggered vertices are to meet their respective deadlines. Let $t^i_{i,e}$ be the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i, can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. Lastly, let $E = \max_{i=1,\ldots,n} e(v_i)$. Clearly, nE is an upper bound on $\text{DBF}(T', t)$ for any $t \geq 0$ for one-shot executions of T'. It can be shown by induction that Algorithm 1 correctly computes $\text{DBF}(T', t)$, and has a running time of $O(n^3 E)$.

Given this algorithm, any $t \geq 0$, and an $0 < \varepsilon \leq 1$, let T'_t be the subgraph of T' consisting only of those vertices v_i for which $d(v_i) \leq t$, and let E_t denote the maximum execution requirement of a vertex from among all vertices of T'_t. Now we scale all the execution requirements associated with the vertices of T'_t by $K = \varepsilon E_t / n$ i.e. $e'(v_i) = \lfloor e(v_i)/K \rfloor$ and run the algorithm with the new $e'(v_i)$s and the graph T'_t. Let V be the set of vertices (with the scaled execution requirements) that result in the computation of $\text{DBF}(T', t)$ in this algorithm. We claim that the summation of the original (unscaled) execution requirements of these vertices is greater than or equal to $(1 - \varepsilon)$ times the actual demand-bound function for the task graph for this value of t. Further, this algorithm now runs in time $O(n^4/\varepsilon)$ (with the scaled execution requirements) and hence is an FPTAS for computing $\text{DBF}(T', t)$. We denote this approximate value of $\text{DBF}(T', t)$ computed by this algorithm by $\text{DBF}'(T', t)$.

If a task graph T has $O(n)$ vertices, then $E(T)$ can be computed in $O(n^2)$ time. Using the FPTAS given above, $\text{DBF}(T, t)$ for *any* value of t can therefore be approximately computed using Eqn. 7 in $O(n^4/\varepsilon)$ time. If we denote this approximate demand-bound function by $\text{DBF}'(T, t)$ then for any t, $\text{DBF}(T, t) \geq \text{DBF}'(T, t) \geq (1 - \varepsilon)\text{DBF}(T, t)$.

5.2 Bounding the Number of Tests

Recall from Section 4 that the number of tests (on the sum of the demand-bound functions for different values of t) required by our algorithm is equal to $\frac{2}{1-\rho_{\text{ave}}(\tau)} \times \sum_{i=1}^{n} E(T)$ times a polynomial function of the input/problem size (follows from inequality 6). Let us denote this as $t_{\max}$.

Since $t_{\max}$ is pseudo-polynomial in the input size, approximating the functions DBF and DBF^v alone does not give us a polynomial time schedulability test. In this subsection we show that combined with the approximate demand-bound function computed in the last subsection, a polynomial number of checks result in a bounded error. This gives us a polynomial-time algorithm for the schedulability analysis of a system of non-preemptive recurring real-time tasks. We assume that the number of tasks in τ is m.

Let us first consider the case where we use the exact functions DBF and DBF^v, but test the schedulability condition only for a polynomial number of time intervals $\hat{\Delta}$. More precisely, we iterate only over the values $\hat{\Delta} = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$, where $K = \frac{\delta t_{\max}}{poly(m)}$. Here δ is an input error parameter to the algorithm and $poly(m)$ is any polynomial function of m. Hence, the total number of values of $\hat{\Delta}$ over which we iterate is $O(\frac{poly(m)}{\delta})$, which is polynomial in the size of our input specification.

It may be shown that if τ is schedulable, then such an algorithm always returns the correct answer. However, if τ is not schedulable, then this algorithm might sometimes incorrectly return YES as well. However, in such cases it can be guaranteed that any vertex might miss its deadline by at most K time units. Since this algorithm always returns the correct answer if a system of recurring real-time tasks τ is schedulable, and it might err only if τ is not schedulable, we refer to this algorithm as *optimistic*. It is also possible to design a corresponding *pessimistic* algorithm (as mentioned in Section 1). If a system of non-preemptive recurring real-time tasks τ is not schedulable, then such an algorithm always returns the correct answer. For task systems τ which are schedulable, this algorithm can return an incorrect answer and the error in these cases is bounded by K, i.e. for such τ there exist time intervals of length $\hat{\Delta}$ over which the processor can be occupied for at least $\hat{\Delta} - K$ time units.

Since $K = \frac{\delta t_{\max}}{poly(m)}$, a smaller value of δ reduces the maximum error that can be incurred by both the optimistic and the pessimistic algorithms, at the cost of increasing the number of checks to be performed and hence also increasing the running times of the algorithms.

5.3 Putting Everything Together

Given the FPTAS described in Section 5.1 for approximating the value of $\text{DBF}(T,t)$, it is possible to obtain the following bound on the approximate value of the demand-bound function: $\text{DBF}'(T,t)$: $\text{DBF}'(T,t) + \varepsilon E_T \geq \text{DBF}(T,t)$. Here E_T is the maximum execution requirement of any vertex in the task graph T. Further, since $\frac{1}{1-\varepsilon}\text{DBF}'(T,t) \geq \text{DBF}(T,t)$, we have

$$\text{DBF}(T,t) \leq \min\{\frac{1}{1-\varepsilon}\text{DBF}'(T,t), \text{DBF}'(T,t) + \varepsilon E_T\} \quad (8)$$

which therefore gives the better of the two bounds for any value of t.

For any task set τ, Algorithm 2 always returns the correct answer if τ is schedulable but might err if τ is not schedulable. Hence, whenever this algorithm returns NO, the decision is guaranteed to be correct. But YES answers might be wrong. The maximum error incurred by this algorithm stems from: (i) Our approximation of the functions DBF and DBF^v by DBF' and $\text{DBF}^{v\prime}$, and (ii) The fact that we do not check the condition for schedulability (given in line 26 of Algorithm 2) for all values of $0 \leq \hat{\Delta} \leq t_{\max}$. But instead we check this condition only for $\hat{\Delta} = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$. Following the discussions in Sections 5.1 and 5.2, it is possible to show that for any vertex v belonging to a task $T_i \in \tau$, this error is bounded by: $K + \frac{\varepsilon}{1-\varepsilon}(\text{DBF}^{v\prime}(T_i, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K) + \sum_{T \in \hat{\tau}} \text{DBF}'(T, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K))$ (v' and $\hat{\tau}$ are as defined in Algorithm 2). It is possible to further tighten this bound using Eqn. 8. Hence, in the case where τ is not schedulable, but Algorithm 2 falsely returns YES, the vertex v can miss its deadline at most by the above error bound.

It is also possible to design the corresponding *pessimistic* algorithm, by using an upper bound on the value of $\text{DBF}(T,t)$, which is given by Eqn. 8. Combined with this, we again check the condition for schedulability at $\hat{\Delta} = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$. If τ is not schedulable, then this algorithm is guaranteed to return NO. However, this algorithm might err in case τ is schedulable, and sometimes falsely return NO. Using the same techniques as in the case of our optimistic algorithm, it is also possible to bound the error made in this case.

Algorithm 2 Optimistic Algorithm ($t_{\max}, \delta$)

Input: System of recurring real-time tasks $\tau, t_{\max}, \delta, m$ = number of task graphs in τ

1: $K \leftarrow \frac{\delta t_{\max}}{poly(m)}$
2: $decision \leftarrow YES$
3: **for all** tasks $T_i \in \tau$ and **for all** vertices $v \in T_i$ and **for all** $\hat{\Delta} \leftarrow 1$ **to** $\lfloor \frac{t_{\max}}{K} \rfloor + 1$ **do**
4: Let $\tilde{\tau} \leftarrow \tau \backslash \{T_i\}$
5: $\tau_{DBF=0} \leftarrow \{T \in \tilde{\tau} \mid \text{DBF}'(T, \hat{\Delta}K + d(v)) = 0\}$
6: $e_{max} \leftarrow \max_{v'}\{e(v') \mid v' \text{ is a vertex of a task } T \in \tau_{DBF=0}\}$
7: Let $\tau_{DBF>0} \leftarrow \{T \in \tilde{\tau} \mid \text{DBF}'(T, \hat{\Delta}K + d(v)) > 0\}$ and $q \leftarrow |\tau_{DBF>0}|$
8: $index \leftarrow 0$
9: **for** $p \leftarrow 1$ to q **do**
10: Let $e'_{max} \leftarrow \max\{e(v') \mid v' \in T_p,\ d(v') > \hat{\Delta}K + d(v)\}$
11: **if** $index = 0$ **then**
12: **if** $e'_{max} > (\text{DBF}'(T_p, \hat{\Delta}K + d(v)) + e_{max})$ **then**
13: $e_{max} \leftarrow e'_{max}$
14: $index \leftarrow p$
15: **end if**
16: **else**
17: **if** $e'_{max} + T_{index}(\hat{\Delta}K + d(v)) > (\text{DBF}'(T_p, \hat{\Delta}K + d(v)) + e_{max})$ **then**
18: $e_{max} \leftarrow e'_{max}$
19: $index \leftarrow p$
20: **end if**
21: **end if**
22: **end for**
23: **if** $index \neq 0$ **then**
24: $\hat{\tau} \leftarrow \tau_{DBF>0} \backslash \{T_{index}\}$
25: **end if**
26: **if** $\hat{\Delta}K + d(v) < (\text{DBF}^{v\prime}(T_i, \hat{\Delta}K + d(v)) + \sum_{T \in \hat{\tau}} \text{DBF}'(T, \hat{\Delta}K + d(v)) + e_{max})$ **then**
27: $decision \leftarrow NO$
28: **end if**
29: **end for**
30: return $decision$

6 Running Time Analysis

As mentioned in Section 5, the problem of computing the function DBF for a general task graph is NP-hard. Moreover, the number of checks of the schedulability condition is pseudo-polynomial in the size of the problem specification. Recall from our discussion in Sections 5.1 and 5.2 that our algorithms for approximate schedulability analysis require as an input two error parameters ε and δ. The smaller the values of these parameters, the smaller is the error in the decisions made by our algorithms, however, at the cost of higher running times. The parameter ε is used to approximate the function DBF and δ determines the number of checks on the sum of the demand bound functions to determine if for any time interval length, the sum of the demand bound functions within this interval exceeds the length of the interval.

It may be shown that computing all possible values of $\text{DBF}'(T,t)$ for *small values* of t and for all tasks $T \in \tau$, takes $O(n^5 m/\varepsilon)$ time, where each task graph in τ contains $O(n)$ vertices and τ contains m task graphs. All these values are first computed and stored in a table. It is easy to see from Algorithm 1 that all $\text{DBF}^{v\prime}(T,t)$ can also be computed in the same time.

During the second phase in our algorithms (where the sum of the demand bound functions are checked to determine if their value exceeds the time interval length), for any t, computing $\sum_{T \in \tau} \text{DBF}'(T,t)$ needs a table lookup which takes $O(n^2 m \varepsilon^{-1} \log n)$ time. Computing $\text{DBF}^{v\prime}(T,t)$ requires $O(n^2 \varepsilon^{-1} \log n)$. The number of checks performed, i.e. the number of times the outer loop (lines 3 to 29) is executed in Algorithm 2 is $O(m \cdot n \cdot \frac{poly(m)}{\delta})$. Hence, the total running time of Algorithm 2 is $O(n^5 m \varepsilon^{-1} + n^3 m^2 \varepsilon^{-1} \delta^{-1} poly(m) \log n)$. The running time of the pessimistic version of Algorithm 2 is exactly the same, but has a different bound on the error. Hence, our algorithms for

approximate schedulability analysis for the non-preemptive version of the recurring real-time task model run in polynomial time. Since the run time is also polynomial in ε^{-1} and δ^{-1} our algorithms are fully polynomial time approximation schemes (FPTAS), which is the best one could hope for, given the intractability of the problem.

7 Concluding Remarks

In this paper we studied the non-preemptive schedulability analysis of recurring real-time tasks upon uniprocessor platforms. The preemption model we considered allows a task graph to be preempted only at node boundaries. Our main result is that non-preemptive schedulability analysis is reducible to a polynomial number of preemptive schedulability analysis problems. From an asymptotic complexity perspective, the two problems are therefore essentially equivalent, which is surprising given the general consensus that non-preemptive versions of schedulability analysis problems tend to be significantly more complex compared to their preemptive counterparts. From a theoretical standpoint, this result provides new insight into the *space* of task models from the complexity-of-schedulability-analysis perspective. From an application standpoint, this result is important since the preemption model we consider in this paper is more realistic in most setups compared to the fully preemptive model.

Since the recurring real-time task model is one of the more general models studied in the real-time systems area, our results are applicable across a variety of task models. As mentioned in Section 1, very recently the non-preemptive version of schedulability analysis was studied for more restricted, periodic and sporadic task models [4, 3]. Our result in this paper can therefore be considered as a generalization of some of the results presented in [4, 3]. In fact, we can be even more general: although we have focused our attention on the recurring real-time task model in this paper, our results can be extended in a straightforward manner to any task model for which the *task independence assumptions* [6, page 8] hold:

- The runtime behavior of a task does not depend upon the behavior of other tasks in the system.
- The workload constraints can be specified without making any references to *absolute* time. That is, specifications such as *Task T generates a job at time-instant* 3 are forbidden.

As argued in [6], these assumptions are extremely general and hold for a very large class of real-time applications. Hence, the results and algorithms presented in this paper should find applicability in a wide variety of real-time systems.

There are two possible directions for future work that stem from this paper. First, it may be noted that the computational complexity of our approximation schemes are high-degree polynomials. An interesting work would be to consider more aggressive notions of approximation and see if they can further help in reducing the running time of the analysis. Second, non-preemptive schedulability analysis for multiprocessors was recently considered in [4], albeit for the much simpler periodic task model. It would be worthwhile extend the work in [4] to more general task models, such as the one we studied in this paper and also see if some of the results proposed here extend to the multiprocessor case.

References

[1] S. K. Baruah. Feasibility analysis of recurring branching tasks. In *Euromicro Workshop on Real-time Systems (ECRTS)*, 1998.

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

[3] S. K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[4] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-time Systems, to appear*.

[5] S. K. Baruah and S. Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks, 2006. www.comp.nus.edu.sg/~samarjit/psfiles/BC06-TR.ps.

[6] S. K. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[7] S. K. Baruah, A. Mok, and L. Rosier. The preemptive scheduling of sporadic, real-time tasks on one processor. In *IEEE Real-Time Systems Symposium (RTSS)*, 1990.

[8] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 1995.

[9] S. Chakraborty, Erlebach, and Thiele. On the complexity of scheduling conditional real-time code. In *Workshop on Algorithms and Data Structures (WADS), LNCS 2125*, 2001.

[10] S. Chakraborty, T. Erlebach, S. Künzli, and L. Thiele. Schedulability of event-driven code blocks in real-time embedded systems. In *Design Automation Conference (DAC)*, 2002.

[11] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[12] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report RR-2966, INRIA, 1996.

[13] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium (RTSS)*, 1991.

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

[15] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[16] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Workshop on Real-Time Computing Systems and Applications (RTCSA)*, 1997.