On the Complexity of Scheduling Conditional Real-Time Code

Samarjit Chakraborty, Thomas Erlebach, and Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland
E-mail: {samarjit,erlebach,thiele}@tik.ee.ethz.ch

Abstract. Many real-time embedded systems involve a collection of independently executing event-driven code blocks, having hard real-time constraints. Portions of such codes when triggered by external events require to be executed within a given deadline from the triggering time. The feasibility analysis problem for such a real-time system asks whether it is possible to schedule all such blocks of code so that all the associated deadlines are met even in the worst case triggering sequence. Each such conditional real-time code block can be naturally represented by a directed acyclic graph whose vertices correspond to portions of code having a straight-line flow of control and are associated with execution requirements and deadlines relative to their triggering times, and the edges represent conditional branches. Till now, no complexity results were known for the feasibility analysis problem in this model, and all existing algorithms in the real-time systems literature have an exponential complexity. In this paper we show that this problem is NP-hard under both dynamic and static priorities in the preemptive uniprocessor case, even for a set of only two task graphs. For dynamic-priority feasibility analysis we give a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for approximate feasibility testing. For the special case where all the execution requirements of the vertices are identical, we present a polynomial time exact algorithm. For static-priority feasibility analysis, we introduce a new sufficient condition and give a pseudo-polynomial time algorithm for checking it. This algorithm gives tighter results for feasibility analysis compared to those known so far.

1 Introduction

Over the years there have been several efforts to correctly model real-time systems and answer scheduling-theoretic questions arising in these models. All of the resulting models are based on an abstract framework in which a real-time system is modelled as a collection of independent *tasks*. Each task generates a sequence of *jobs*, each of which is characterized by a *ready-time*, an *execution requirement*, and a *deadline*. *Hard-real-time systems* require that for each job generated by a task, an amount of processor time equal to the job's execution requirement be assigned to it between its ready-time and its deadline. The feasibility analysis of such a hard-real-time task set is concerned with determining whether it is possible to schedule all the jobs generated by the tasks, such that they meet their deadlines under all possible circumstances.

In the context of most real-time embedded systems, each such real-time task is required to model an event-driven block of code, parts of which are triggered by external events and require to be executed within a given deadline from the triggering time. A natural representation of such a task is a directed acyclic graph whose vertices represent portions of code having a straight-line flow of control, and the edges represent possible conditional branches. The vertices are triggered by external events and have to be executed within their associated deadlines. The feasibility analysis of such a set of task graphs answers whether it is possible to schedule all the graphs so that all the associated deadlines are met even in the worst case triggering sequence. The difficulty of such an analysis lies in the fact that what constitutes a worst case triggering sequence for an individual graph can not be determined in isolation, due to the presence of the conditional branches. To illustrate this, consider the following example taken from [2].

while (*external event*) **do**
  execute code block B_0 {having execution time e_0 and deadline d_0}
  if (C) **then**
    execute code block B_1 {execution time e_1, deadline d_1}
  else
    execute code block B_2 {execution time e_2, deadline d_2}
  end if
end while

In the above block of code, if the condition C depends on some external event, or on the value of a variable which can not be determined at compile time, then the worst case branch here would depend on the other blocks of code executing concurrently with this one. Let $e_1 = 2$, $d_1 = 2$, $e_2 = 4$ and $d_2 = 5$. If another code block is simultaneously executing with $e = 1$ and $d = 1$ then the (e_1, d_1) branch corresponds to the worst case, whereas if $e = 2$ and $d = 5$ then the (e_2, d_2) branch corresponds to the worst case. Hence the usual method followed for the feasibility analysis of hard-real-time systems, of approximating a piece of code by its worst case behaviour does not work in the presence of conditional branches.

In this paper we consider the feasibility analysis of a collection of such code blocks with conditional branches and real-time constraints, and present a series of results on the complexity of various versions of this problem.

1.1 The model

A task modelling a block of code is represented by a directed acyclic graph with a unique source and a unique sink vertex. Associated with each vertex v is its execution requirement $e(v)$ (which can be determined at compile time), and deadline $d(v)$. Whenever the vertex v is *triggered*, the code corresponding to it has to be executed (which takes $e(v)$ amount of time) within the next $d(v)$

time units. Each directed edge (u, v) in the graph is associated with a minimum intertriggering separation $p(u, v)$, denoting the minimum amount of time that must elapse before the vertex v can be triggered after the triggering of the vertex u. This can be used to model a possible communication delay between u and v.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and once a vertex u is triggered then the next vertex v can be triggered only if there exists a directed edge (u, v) and at least $p(u, v)$ amount of time has elapsed since the triggering of u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u then only one among v_1 and v_2 can be triggered, after the triggering of u. Therefore, a sequence of vertices $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$ is legal if and only if there are directed edges (v_i, v_{i+1}) and $t_{i+1} - t_i \geq p(v_i, v_{i+1})$ for $i = 1, \ldots, k-1$. The real-time constraints require that the code corresponding to vertex v_i be executed within the time interval $(t_i, t_i + d(v_i)]$. Note that in general the condition $t_{i+1} \geq t_i + d(v_i)$ may not hold, i.e. a vertex can be triggered before the deadline of the last triggered vertex has elapsed. A consequence of this might be that the code corresponding to a vertex v is executed before that corresponding to a vertex u, although there exists a directed edge (u, v). Since this might not be allowable in most applications, throughout this paper we assume that $t_{i+1} \geq t_i + d(v_i)$ which is equivalent to requiring that $p(u, v) \geq d(u)$. Most of the previous work is based on this assumption and in the real-time systems literature this is referred to as the *frame separation property*.

Task sets and feasibility analysis. A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ consists of a collection of task graphs, the vertices of which can get triggered independently of each other. A triggering sequence for such a task set $\mathcal{T}$ is legal if and only if for every task graph T_i, the subset of vertices of the sequence belonging to T_i constitutes a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks.

The feasibility analysis of a task set $\mathcal{T}$ is concerned with determining whether for all possible legal triggering sequences of $\mathcal{T}$, the codes corresponding to the vertices of the task graphs can be scheduled such that all their associated deadlines are met. In this paper we consider the preemptive uniprocessor version of this problem.

Many scheduling algorithms are implemented by assigning priorities at each time instant (according to some criteria), to all jobs that are ready to execute and then allocating the processor to the highest priority job. Based on this, scheduling algorithms can be broadly classified into either *dynamic-priority* or *static-priority* (also known as fixed-priority) algorithms. Dynamic-priority algorithms allow the switching of priorities between tasks. This means that for two tasks, both having ready jobs at two time instants, at one instant the first task's job might have a higher priority than the second task's job, while at the other instant the priorities might switch. Static-priority algorithms, in contrast to this, do not allow such priority switching. Here we will be concerned with both dynamic- and static-priority algorithms.

1.2 Our results

The task model considered in this paper, apart from being of independent interest, forms the core of the *recurring real-time task model* very recently proposed by Baruah in [2, 4]. This model is especially suited for accurately modelling conditional real-time code with recurring behaviour, i.e. where code blocks run in an infinite loop, and generalizes many of the previous well known models like the sporadic [7], multiframe [8], generalized multiframe [5], and recurring branching [3]. All of these previous models can be shown [2] to be special cases of the recurring real-time task model. However, the algorithms presented in [2] for the feasibility analysis problem in this model for the preemptive uniprocessor case, both with dynamic and static priorities, have a running time which is exponential in the number of vertices of the task graphs. It was also remarked that the feasibility analysis problem for this model is 'likely to be intractable', and in contrast to the previous (less general) models, no longer runs in pseudo-polynomial time.

The main contribution of this paper is that it answers all the questions raised in [2] and thereby settles the complexity of the feasibility analysis problem for scheduling conditional real-time code. For the ease of presentation, the model we consider here is slightly simpler than that of [2] in the sense that we do not consider the recurring behaviour of the task graphs. We postpone the details of how the results derived here for this simpler model can be extended to the recurring real-time task model, to a full version of this paper. Firstly, we show that the feasibility analysis problem, both for dynamic and static priorities, is NP-hard. For the dynamic-priority feasibility analysis we give a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for approximate feasibility testing. We also show that for the special case where all the vertices of a task graph have equal execution requirements, this problem can be solved in polynomial time.

For static-priority feasibility analysis Baruah had introduced a sufficient condition in [2]. We give a tighter condition for sufficiency and show that this can be checked in pseudo-polynomial time. Further, our condition is simpler than that of [2]. Our results imply that for all practical purposes the feasibility analysis problem in the recurring real-time task model is efficiently solvable.

We present the hardness results in Section 2. In Section 3 we present the algorithms for dynamic-priority feasibility analysis, followed by those for static-priority feasibility analysis in Section 4. Proofs of all the theorems presented here are deferred to the appendix.

2 NP-hardness of feasibility analysis

In this section we obtain that both the dynamic- and static-priority feasibility analysis problems for our task model, and therefore for the recurring real-time task model as well, are NP-hard for the preemptive uniprocessor case. Our proofs rely on a reduction from the knapsack problem which is known to be NP-hard.

Theorem 1. *The dynamic-priority feasibility analysis problem for the task model described in Section 1.1 in a preemptive uniprocessor environment is NP-hard.*

The next result shows that the static-priority feasibility analysis problem is NP-hard. The pseudo-polynomial time algorithm that we present later for this problem, and also the algorithm presented in [2], are based on testing whether a given task from a task set is *lowest-priority feasible.* A task $T \in \mathcal{T}$ is lowest-priority feasible if and only if all the vertices of T can always meet their deadlines with T assigned the lowest priority and all the remaining tasks of $\mathcal{T}$ having any arbitrary priority assignment. The existence of a lowest-priority feasible task in any static-priority feasible task set is given later by Theorem 6.

The next theorem says that the lowest-priority feasibility testing problem is NP-hard, and as a corollary of this it follows that static-priority feasibility analysis is also NP-hard.

Theorem 2. *The problem of determining whether a given task is lowest-priority feasible is NP-hard.*

Corollary 1. *The static-priority feasibility analysis problem is NP-hard.*

3 Dynamic-priority feasibility analysis

A necessary and sufficient condition for the dynamic-priority feasibility of the recurring real-time task model was stated in [2]. It was stated without proof that the condition follows from the *processor demand criterion* introduced in [6]. It is possible to give a simple independent proof (given in the appendix) showing that the same condition works for our model. It is based on an abstraction of a task, represented by a function called the *demand-bound function*. The demand-bound function of a task T, denoted by $T.dbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered by a legal triggering sequence and have both their ready times and deadlines within a time interval of length t. Intuitively, $T.dbf(t)$ denotes the maximum possible execution requirement that can possibly be demanded by T within any time interval of length t, if all its vertices are to meet their deadlines.

Theorem 3. *A task set $\mathcal{T}$ is dynamic-priority feasible if and only if for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$.*

We next show that the problem of computing $T.dbf(t)$ for a task T is NP-hard and then give a FPTAS for approximating it, which immediately leads to an approximate decision algorithm for the feasibility analysis problem.

Theorem 4. *The problem of computing $T.dbf(t)$ is NP-hard.*

Given a task graph T we first give a pseudo-polynomial time algorithm for computing $T.dbf(t)$ for any $t \geq 0$, based on dynamic programming. 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 1.1, associated with each 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 $T.dbf(t)$ for any $t \geq 0$. It can be trivially shown by induction that Algorithm 1 correctly computes $T.dbf(t)$, and has a running time of $O(n^3 E)$.

Algorithm 1 Computing $T.dbf(t)$

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}

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

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

    end for

  end for

  $T.dbf(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$

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. By using the same arguments as in the FPTAS for the knapsack problem, it is possible to show that for any $t \geq 0$, the algorithm outputs a value $\geq (1-\varepsilon) T.dbf(t)$ and runs in time $O(n^4/\varepsilon)$, and is therefore an FPTAS for computing $T.dbf(t)$. We denote the result computed by this algorithm by $T.dbf'(t)$.

For our approximate decision algorithm, note that for all $t \geq 0$, there can be at most n distinct values of E_t for any task graph. For each such E_t, we consider the corresponding subgraph that gives rise to this E_t as described above, and scale the execution requirements of the vertices of this subgraph by $K = \varepsilon E_t / n$. In each such subgraph T_t, the number of values of time intervals t' at which the value of $T_t.dbf'(t')$ changes is bounded by $O(n^2/\varepsilon)$, and hence the number of values of time intervals t at which the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ changes is bounded by $O(|\mathcal{T}|n^3/\varepsilon)$. Our fully polynomial time approximate decision algorithm for dynamic-priority feasibility analysis is now given as Algorithm 2.

Algorithm 2 Approximate decision algorithm for feasibility analysis

```
Input: Task set 𝒯 and a real 0 < ε ≤ 1
  decision ← YES
  for all values of t at which T.dbf'(t) changes for any T ∈ 𝒯 do
    if 1/(1−ε) Σ_{T∈𝒯} T.dbf'(t) > t then {Condition (*)}
      decision ← NO
    end if
  end for
  return decision
```

Theorem 5. *If a task set $\mathcal{T}$ is infeasible then Algorithm 2 always returns the correct answer. If $\mathcal{T}$ is feasible and $t \geq \frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(t)$ for all values of t, then the algorithm always returns the correct answer YES, otherwise it might return a NO. YES answers are always correct. The running time of the algorithm is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$, if all task graphs have $O(n)$ vertices.*

For each task T, computing the $t_{n,e}$ values for each of its subgraphs T_t, using Algorithm 1 and the scaled execution requirements requires $O(n^4/\varepsilon)$ time, and these values are stored in a table. Hence computing all such values for all the task graphs in $\mathcal{T}$ takes $O(n^5|\mathcal{T}|/\varepsilon)$ time. For each value of t for which $\sum_{T \in \mathcal{T}} T.dbf'(t)$ changes, computing $T.dbf'(t)$ for any $T \in \mathcal{T}$ requires a binary search to identify the appropriate table corresponding to a subgraph T_t, and then a linear search through the table. Therefore, computing the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ for any value of t takes $O(|\mathcal{T}|n^2\varepsilon^{-1} \log n)$ time. Hence the total running time of Algorithm 2 is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$. The algorithm is overly pessimistic in the sense that for certain feasible task sets it might return a NO. However, for task sets which can be in some sense comfortably scheduled even in the worst case, leaving some idle processor time (which can be parameterized by ε), the algorithm always returns a YES. Therefore, any ε characterizes a class of task sets for which the algorithm errs. Decreasing ε reduces this class of such task sets for which the algorithm errs, at the cost of increasing the running time quadratically in $1/\varepsilon$, thereby giving a fully polynomial-time approximate decision scheme for approximate feasibility testing.

It may be noted that changing Condition ($*$) in Algorithm 2 to

if $\sum_{T \in \mathcal{T}} T.dbf'(t) > t$ **then**
  $decision \leftarrow NO$
end if

will result in an overly optimistic algorithm which might incorrectly return a YES for certain classes of infeasible task sets. For all feasible task sets it always returns YES, and NO answers are always correct. The task sets for which the algorithm might err are those in which the cumulative execution requirement by tasks of $\mathcal{T}$ within any time interval of length t exceeds the maximum execution requirement that can be feasibly scheduled, by an amount of less than $\varepsilon \sum_{T \in \mathcal{T}} T.dbf(t)$ time units. Again, decreasing ε reduces the class of such task sets, at the cost of the running time increasing linearly in $1/\varepsilon$.

Lastly, it might be noted that Theorem 3 along with Algorithm 1 also imply a pseudo-polynomial time algorithm for dynamic-priority feasibility analysis. To see this, let for any task $T \in \mathcal{T}$, t^T_{max} denote the maximum amount of time elapsed among all execution sequences starting from the source vertex of T and ending at the sink vertex, if every vertex is triggered at the earliest possible time (respecting the minimum intertriggering separations). Let $t_{max} = \max_{T \in \mathcal{T}} t^T_{max}$. It follows from Theorem 3 that $\mathcal{T}$ is dynamic-priority feasible if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t = 1, \ldots, t_{max}$. $T.dbf(t)$ for any t can be determined in pseudo-polynomial time by Algorithm 1 and clearly, t_{max} is pseudo-polynomially bounded, implying a pseudo-polynomial algorithm for dynamic-priority feasibility analysis.

3.1 Vertices with equal execution requirements

We now show that for the special case where for every task T belonging to a task set $\mathcal{T}$, all the vertices of T have equal execution requirements, the feasibility analysis problem for $\mathcal{T}$ can be solved in polynomial time. This result holds even when all execution requirements and deadlines take values over the reals.

We denote the vertices of a task graph T by $v_1, \ldots, v_n$ and assume that there can be a directed edge from v_i to v_j only if $i < j$. Let $t_{i,k}$ denote the minimum time interval within which exactly k vertices of T from the set $\{v_1, \ldots, v_i\}$ (obviously $k \leq i$) need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines. Let $t^i_{i,k}$ denote the minimum time interval within which exactly k vertices of T consisting of v_i and any other $k-1$ vertices from $\{v_1, \ldots, v_{i-1}\}$ need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines.

Given any vertex v_i of T, let there be directed edges from the vertices $v_{i_1}, \ldots, v_{i_l}$ to v_i. Then for any $k \leq i$,

$$t^i_{i,k} = \min\{t^{i_j}_{i_j,k-1} - d(v_{i_j}) + p(v_{i_j}, v_i) + d(v_i) \,|\, j = 1, \ldots, l\} \quad \text{(and } d(v_i) \text{ if } k = 1)$$

$$t_{i,k} = \min\{t_{i-1,k}, t^i_{i,k}\}$$

Using the fact that $t_{1,1} = t^1_{1,1} = d(v_1)$, it is now possible to compute any $t_{i,k}$ within at most $O(n^3)$ time, where n is the number of vertices in the task graph.

Now, if each task graph $T \in \mathcal{T}$ has n_T vertices then let us consider the set $S = \bigcup_{T \in \mathcal{T}} \bigcup_{i=1,\ldots,n_T} \{t_{n_T,i} \text{ for task graph } T\}$. If each vertex of task graph T has an execution requirement of e, then for ant $t \geq 0$, $T.dbf(t) = \max\{ie \mid t_{n_T,i} \leq t\}$. Clearly, the task set $\mathcal{T}$ is feasible if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t \in S$. Computing all the necessary dbf values for each task graph and storing them in a table takes $O(n^3)$ time if the number of vertices in any task graph is $O(n)$. Since there are $|\mathcal{T}|$ task graphs, this whole process takes $O(|\mathcal{T}|n^3)$ time. For each value of t, verifying whether the sum of the dbfs exceeds t requires a search through the previously computed tables and takes $O(|\mathcal{T}| \log n)$ time. Since there are $O(|\mathcal{T}|n)$ values of t for which this has to be verified, this takes $O(|\mathcal{T}|^2 n \log n)$ time. Hence the total run time is bounded by $O(|\mathcal{T}|n^3 + |\mathcal{T}|^2 n \log n)$.

4 Static-priority feasibility analysis

The static-priority feasibility analysis of a task set $\mathcal{T}$ is concerned with determining whether there exists an assignment of priorities to the tasks of $\mathcal{T}$ under which they can be scheduled by a static-priority run time scheduler so that all deadlines are met even in the worst case triggering sequence. Any such priority assignment is defined to be a *good* static-priority assignment for $\mathcal{T}$. As mentioned in Section 2 solving this feasibility analysis problem is based on testing whether a given task $T \in \mathcal{T}$ is lowest-priority feasible. Clearly, if there is a procedure for testing lowest-priority feasibility, and the task set $\mathcal{T}$ is static-priority feasible, then $|\mathcal{T}|$ calls to this procedure will be sufficient to identify a lowest-priority feasible task of $\mathcal{T}$. Therefore, if $|\mathcal{T}| = n$ then with $O(n^2)$ calls to this procedure a good static-priority assignment for $\mathcal{T}$ can be determined based on the following theorem.

Theorem 6 (Audsley, Tindell, Burns [1]). *Suppose a task $T \in \mathcal{T}$ is lowest-priority feasible. Then there is a good static-priority assignment for $\mathcal{T}$ if and only if there is a good static-priority assignment for $\mathcal{T} \setminus \{T\}$.*

An algorithm for static-priority feasibility analysis therefore reduces to devising an algorithm for lowest-priority feasibility testing. An algorithm implementing a sufficient condition for lowest-priority feasibility was given by Baruah in [2] for the recurring real-time task model. It is also based on an abstraction of a task, similar to the demand-bound function presented in Section 3, and uses a function called the *request-bound function.* The request-bound function of a task T, denoted by $T.rbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered according to some legal triggering sequence and have their ready times within any time interval of length t. Intuitively, $T.rbf(t)$ is an upper bound on the maximum amount of time, within any time interval of length t, for which T can deny the processor to all lower-priority tasks. Based on this function, the following sufficiency condition was given for lowest-priority feasibility testing in [2].

Theorem 7 (Baruah [2]). *A task $T \in \mathcal{T}$ is lowest-priority feasible if $\forall t : \exists t' \leq t$ such that $t' - \sum_{T' \in \mathcal{T} \setminus \{T\}} T'.rbf(t') \geq T.dbf(t)$.*

For any task $T \in \mathcal{T}$, in our task model described in Section 1.1, let t^T_{max} be as described in Section 3. Clearly, T is lowest-priority feasible if the condition given by Theorem 7 is satisfied for all values of $t = 1, \ldots, t^T_{max}$. Although t_{max} is pseudo-polynomially bounded by the representation of $\mathcal{T}$, the algorithm for computing $T.rbf(t)$ for any t and $T \in \mathcal{T}$ as given in [2] runs in time which is exponential in the number of vertices of T.

We first obtain that the problem of computing $T.rbf(t)$ is NP-hard and then give a modified request-bound function, which we denote by $T.rbf'(t)$, and give a pseudo-polynomial time algorithm for computing it for any value of $t \geq 0$ based on dynamic programming. Using $rbf'(t)$ we then give a new sufficiency condition for testing lowest-priority feasibility. This gives a tighter test compared to that of Theorem 7 in the following sense: for any task set $\mathcal{T}$, if a task $T \in \mathcal{T}$ is returned as lowest-priority feasible by the test in Theorem 7 then it is also returned as lowest-priority feasible by our test, and there exist task sets $\mathcal{T}$ and tasks $T \in \mathcal{T}$ which although being lowest-priority feasible, fail the test in Theorem 7 but are returned as lowest-priority feasible by our test. Lastly, we show that for any task set consisting of exactly two tasks, our test is both a necessary and sufficient condition.

Theorem 8. *The problem of computing $T.rbf(t)$ is NP-hard.*

Our new $T.rbf'(t)$ is similar to $T.rbf(t)$ and returns the maximum possible cumulative execution requirement by vertices of T within any time interval of length t, that have been triggered by a legal triggering sequence. To illustrate the difference between the two functions, consider a task graph T consisting of a single vertex having an execution requirement of 5 and any arbitrary deadline. Whereas $T.rbf(t) = 5$ for any $t \geq 0$ (since the ready time of T is at time 0), $T.rbf'(t) = t$ for $t \leq 5$ and is equal to 5 for any $t > 5$.

Following the notation used in Section 3, given a task graph T, let $t_{i,e}$ denote the minimum time interval within which 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 $\{v_1, \ldots, v_i\}$. Let $t^i_{i,e}$ be the minimum time interval within which any execution sequence consisting of vertices from the set $\{v_1, \ldots, v_{i-1}\}$ and ending with the vertex v_i can have an execution requirement of exactly e time units. Now recall the definition of $t^i_{i,e}$ as used in Section 3 for computing $T.dbf(t)$, which is 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. This we denote here by $dbf^i_i(e)$. We assume, as in Section 3, that T consists of n vertices $v_1, \ldots, v_n$ and that there can be a directed edge from v_i to v_j only if $i < j$, and that all the execution requirements are integral. If $E = \max_{i=1,\ldots,n} e(v_i)$, then Algorithm 3 correctly computes $T.rbf'(t)$ and has a running time of $O(n^3E^2)$. Our new sufficiency condition for lowest-priority feasibility is based on the following lemma.

Algorithm 3 Computing $T.rbf'(t)$

Input: Task graph T, and a real number $t \geq 0$
for $e \leftarrow 1$ to nE **do**
  $t_{1,e} \leftarrow \begin{cases} e & \text{if } e \leq e(v_1) \\ \infty & \text{if } e > e(v_1) \end{cases}$
  $t^1_{1,e} \leftarrow t_{1,e}$
end for
Computing $t_{i+1,e}$**:**
  Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
  Let $t^{i_j,i+1}_{i+1,e}(l) \leftarrow dbf^{i_j}_{i_j}(e - e(v_{i+1}) + l) - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + e(v_{i+1}) - l$
  Let $t^{i_j,i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e}(l) \mid l = 0, \ldots, e(v_{i+1}) - 1\}$
  $t^{i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e} \mid j = 1, \ldots, k\}$
  $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
$T.rbf'(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$

Lemma 1. *Let $T \in \mathcal{T}$ and the task graph corresponding to T have n vertices $v_1, \ldots, v_n$. If each of these vertices v_i is lowest-priority feasible in the task set $\mathcal{T} \backslash \{T\} \bigcup \{v_i\}$, then T is also lowest-priority feasible.*

Theorem 9. *A task $T \in \mathcal{T}$ is lowest-priority feasible if for all vertices v belonging to the task graph of T, $\exists 0 \leq t \leq d(v)$ for which $t - \sum_{T' \in \mathcal{T} \backslash \{T\}} T'.rbf'(t) \geq e(v)$.*

It is easy to see that if a task $T \in \mathcal{T}$ is returned as lowest-priority feasible by the test given by Theorem 7 then it also passes the test of Theorem 9. Additionally, if T is returned as lowest-priority feasible, then it is really so. To show that this represents a tigher test, consider a task set consisting of two task graphs T_1 and T_2. T_1 is a simple chain of three vertices with the first two vertices having their execution requirements equal to 1 and deadlines equal to 2, and the third vertex having an execution requirement of 3 and deadline equal to 6. The intertriggering separation on any directed edge (u, v) is equal to the deadline of u. T_2 consists of a single vertex having an execution requirement of 1 and deadline equal to 4. It can be seen that T_2 is indeed lowest-priority feasible and passes the test of Theorem 9, but fails the test given by Theorem 7. Lastly, we show that for any set of exactly two task graphs, the test given by Theorem 9 is both a necessary and sufficient condition.

Theorem 10. *For any task set $\mathcal{T}$ consisting of exactly two task graphs, a task $T \in \mathcal{T}$ is lowest-priority feasible if and only if it satisfies the test given by Theorem 9.*

It now follows from Theorem 6, Algorithm 3, and Theorem 9 that there exists a pseudo-polynomial algorithm for static-priority feasibility analysis that implements the sufficiency condition stated by Theorem 9. Further, the same approach of scaling the execution requirements associated with the vertices as described in Section 3 for the dynamic-priority feasibility analysis, and then using Algorithm 3 with the scaled values will give a polynomial time approximate decision algorithm for static-priority feasibility analysis. Lastly, in the case where for each task graph all the vertices have equal execution times, this problem can also be solved in polynomial time. We skip the details of any of these in this paper.

5 Concluding remarks

This paper settles the complexity of the feasibility analysis problem involved in scheduling a collection of code blocks with conditional branches and real-time constraints. In particular it shows that although the feasibility analysis of the recently introduced recurring real-time task model is NP-hard, there exists a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for solving it. All the results presented here pertain to the preemptive uniprocessor version of this problem. It would be natural to extend these results to the non-preemptive and different multiprocessor cases. Following [2], our algorithms were based on an abstraction of a task represented by the demand-bound and the request-bound functions, which in some sense captured the worst case behaviour of a task. It seems unlikely that this same approach might work for any non-preemptive or multiprocessor case, except for very restricted classes of tasks such as those where all vertices have unit execution requirements and time is integral. In more general cases, the worst case triggering sequence of the vertices of a task graph as identified by the demand- or request-bound functions need not be the worst case when the issue of feasibly packing these jobs (on multiple processors, for example) is taken into account.

References

1. N.C. Audsley, K.W. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proc. Euromicro Conf. on Real-Time Systems*, Finland, 1993. IEEE Computer Society Press.
2. S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. To appear in Real-Time Systems.
3. S. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, pages 138–145, 1998.
4. S. Baruah. A general model for recurring real-time tasks. In *Proc. Real-Time Systems Symposium*, pages 114–122. IEEE Computer Society Press, 1998.
5. S. Baruah, D. Chen, S. Gorinsky, and A.K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
6. S. Baruah, R.R. Howell, and L.E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.
7. A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, Laboratory for Computer Science, MIT, 1983. Available as Technical Report No. MIT/LCS/TR-297.
8. A.K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

Appendix

A Proof of Theorem 1

Given a knapsack problem instance, we transform it in polynomial time into a task set consisting of two tasks with the property that there is a solution to the knapsack problem achieving the specified profit goal if and only if the task set is not dynamic-priority feasible. This shows that the dynamic-priority feasibility analysis problem is NP-hard.

The knapsack problem instance specifies n items with integral sizes s_i and profits p_i, $i = 1, \ldots, n$, and an integral size constraint C and profit goal P, and asks if there exists a subset of items, the sum of whose profits is $> P$ and the sum of their sizes $\leq C$. To transform this into our problem we first scale all the profits p_i, $i = 1, \ldots, n$, and the profit goal P such that the new p_is and P satisfy the following (this could be done, for example, by dividing all the p_is and P by $\max\{\sum_{i=1}^{n} p_i, P\} + 1$). (It might be noted here that instead of scaling down the profits p_i and the profit goal P, it is also possible to construct a proof where the sizes s_i and the size constraint C are scaled up so that the constructed instance has integral e and d values.)

(i) $p_i < 1$, $i = 1, \ldots, n$
(ii) $\sum_{i=1}^{n} p_i < 1$
(iii) $P < 1$

We then construct two task graphs T_1 and T_2, where T_1 consists of a single vertex with an execution requirement $e = C - P$ and a deadline $d = C$, and T_2 is as shown in Figure 1.

For each item in the knapsack problem instance, having a size s_i and profit p_i, there is a vertex in T_2 having an execution requirement $e = p_i$ and deadline $d = s_i$, and the outgoing edge from each such vertex is labelled with s_i, denoting the minimum intertriggering separation between the two vertices connected by the edge. All the unlabelled vertices and edges have their execution requirements, deadlines and intertriggering separations equal to zero.

The claim is that the task set $\mathcal{T} = \{T_1, T_2\}$ is not dynamic-priority feasible if and only if there exists a subset of items from the knapsack instance, the sum of whose sizes does not exceed C and the sum of whose profits is greater than P. To see this, let us first consider the if-part. This implies that there is a valid execution of a set of vertices of T_2 having an execution requirement greater than P within a time interval of length C, if all the deadlines associated with the vertices are to be met. Within this same time interval the task T_1 can have an execution requirement equal to $C - P$. Therefore, the total execution requirement of $\mathcal{T}$ within a time interval of length C exceeds C, implying that $\mathcal{T}$ is not dynamic-priority feasible.

For the proof in the other direction, it is to be noted that $\mathcal{T}$ will not be dynamic-priority feasible only in the following two cases.

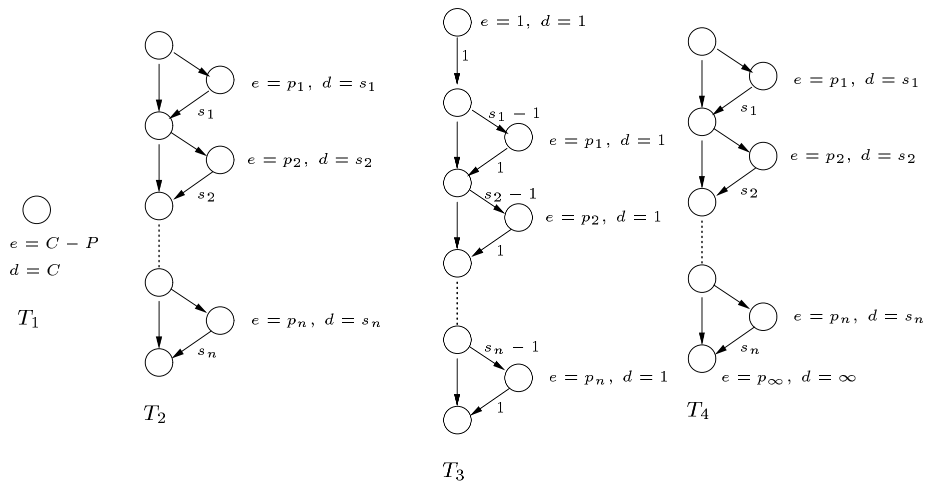

Fig. 1. Task graphs used for proving different hardness results

(i) Within a time interval of length C, T_2 has an execution requirement greater than P.

(ii) Within any time interval of length t such that

$$C + 1 \leq t = s_{i_1} + s_{i_2} + \ldots + s_{i_k}$$

where all pairwise different $i_1, \ldots, i_k \in \{1, \ldots, n\}$, T_1 has an execution requirement exceeding $t - (p_{i_1} + \ldots + p_{i_k})$.

Case (ii) can not occur because within any such time interval of length t, T_1 has an execution requirement of less than C and $p_{i_1} + p_{i_2} + \ldots + p_{i_k} < 1$. Case (i) implies that there is a solution to the knapsack problem satisfying the size constraint C and having a profit greater than P.

B Proof of Theorem 2

Given a knapsack problem instance as described in the proof of Theorem 1, we scale the profits and the profit goal to meet the three conditions also mentioned there. Then we construct two task graphs T and T_3, where T consists of a single node with an execution requirement $e = C - P$ and a deadline $d = C + 1$, and T_3 is as shown in Figure 1. Here, for each item in the knapsack problem instance, having size s_i and profit p_i, there is a vertex in T_3 having an execution requirement $e = p_i$ and deadline $d = 1$, and there is one incoming edge to such a vertex labelled with a minimum intertriggering separation of $s_i - 1$ and an outgoing edge labelled with 1. The source vertex of T_3 has $e = 1$, $d = 1$, and an outgoing edge labelled with 1. As in the previous proof, all the unlabelled vertices and edges have their execution requirements, deadlines and intertriggering separations equal to zero.

We claim that the task T is not lowest-priority feasible if and only if there is a solution to the knapsack instance satisfying the size constraint C and having a profit greater than P. If there is such a solution to the knapsack problem, then the vertices in T_3 corresponding to the items constituting the solution define an execution sequence in which T_3 within a time interval of length C can occupy the processor for an amount of time greater than P. Therefore, an execution sequence defined by the source vertex of T_3 along with these vertices, can occupy the processor for an amount of time greater than $P+1$ within an interval of length $C+1$, implying that T is not lowest-priority feasible.

For the other direction, it should be noted that for T_3 to occupy the processor for an amount of time greater than $P+1$, the source vertex must always be triggered because the sum of the execution requirements of all the other vertices is less than 1. Any such execution sequence defines a solution to the knapsack instance satisfying the size constraint C and having a profit greater than P. To see this, let the sequence of vertices having non-zero execution requirements and deadlines, in such an execution sequence be the source vertex, followed by $(p_{i_1}, 1)$[1], $(p_{i_2}, 1), \ldots, (p_{i_k}, 1)$. Then this clearly corresponds to a knapsack solution in which items having (*profit*, *size*) equal to $(p_{i_1}, s_{i_1}), (p_{i_2}, s_{i_2}), \ldots, (p_{i_k}, s_{i_k} - 1 + p_{i_k})$ satisfy the size constraint C and the profit goal P. However, since C and all the item sizes are integers, $C \geq s_{i_1} + s_{i_2} + \ldots + s_{i_k} - 1 + p_{i_k}$ implies that $C \geq s_{i_1} + s_{i_2} + \ldots + s_{i_k}$. Hence the execution sequence mentioned implies the desired solution to the knapsack problem instance.

C Proof of Corollary 1

In the proof of Theorem 2, clearly T_3 is not lowest-priority feasible, implying that the task set $\{T, T_3\}$ is static-priority feasible if and only if T is lowest-priority feasible. By Theorem 2, the problem of determining whether T is lowest-priority feasible is NP-hard.

D Proof of Theorem 3

First we prove that the condition is necessary (i.e. the 'only if' part). This states that the task set $\mathcal{T}$ being dynamic-priority feasible implies that for all $t \geq 0, \sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, which is equivalent to the proposition that if there exists some $t \geq 0$ for which $\sum_{T \in \mathcal{T}} T.dbf(t) > t$ then the task set $\mathcal{T}$ is not dynamic-priority feasible. We will prove the second proposition.

Let for some $t > 0$, $T_1, T_2, \ldots, T_k$ be all tasks of $\mathcal{T}$ for which $T_i.dbf(t) > 0$ for all $i = 1, 2, \ldots, k$. At time zero, let the triggering sequence of the vertices of the task graphs corresponding to each of these tasks, which result in the

[1] This tuple denotes the vertex having execution requirement equal to p_{i_1} and deadline equal to 1

computation of $dbf(t)$, be started. Then for all the vertices occurring in this triggering sequence, to meet their associated deadlines, an amount of processor time equal to $\sum_{i=1}^{k} T_i.dbf(t)$ has to be allocated within time t. This will not be possible if $\sum_{i=1}^{k} T_i.dbf(t) > t$.

Now we give the proof for sufficiency i.e. for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ implies that $\mathcal{T}$ is dynamic-priority feasible. To prove this let us assume that for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ and still $\mathcal{T}$ is not dynamic-priority feasible. If this is the case then let in some execution sequence, a vertex of $T \in \mathcal{T}$ be the first vertex that misses its deadline, when the triggered vertices of the tasks of $\mathcal{T}$ are scheduled according to the Earliest Deadline First (EDF) rule. Note that EDF is known to be an optimal scheduling policy in this case, in the sense that if there is any feasible schedule then EDF will also schedule the vertices such that all deadlines are met. Assume that the vertex which missed its deadline was generated at time t_r and its deadline was at time t_d. Clearly, during the entire time interval $[t_r, t_d]$ the processor was always occupied, because otherwise this vertex could have been executed during such a free time. Now let us look back in time before t_r and stop immediately before the instant at which either the processor was idle or we encounter some vertex being executed whose deadline is beyond t_d. Let us call the time instant at which we stopped as t'_r.

Clearly t'_r is a time at which some vertex of a task graph was just triggered, and all the vertices that were executed during the time interval $[t'_r, t_d]$ were triggered on or after t'_r and had their deadline before or at t_d (by our definition of t'_r). Let these vertices belong to the tasks $T_1, T_2, \ldots, T_k$ and T of $\mathcal{T}$. The processor was always occupied during the time interval $[t'_r, t_d]$, which implies that the summation of the execution requirements of $T_1, T_2, \ldots, T_k$ and T during this interval exceed $(t_d - t'_r)$. Hence, $\sum_{i=1}^{k} T_i.dbf(t_d - t'_r) + T.dbf(t_d - t_r) > (t_d - t'_r)$ contradicting our assumption that for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$.

E Proof of Theorem 4

Given a knapsack problem instance consisting of n items, a profit goal P and a size constraint C, as described in the proof of Theorem 1, construct the task graph T_2 shown in Figure 1 and described in the same proof. Clearly, $T_2.dbf(C) > P$ if and only if there is a solution to the knapsack problem satisfying the size constraint C and having a profit greater than P.

F Proof of Theorem 8

Consider a knapsack problem instance as described in the proof of Theorem 1 with an additional item having a very large profit (for example greater than the sum of the profits of all the other items), which we denote by p_∞, and zero size. Now construct the task graph T_4 as shown in Figure 1. This is similar to the task graph T_1 in the same figure with the only difference being that the last vertex is labelled with an execution requirement of $e = p_\infty$ and deadline $d = \infty$. It may be noted that any solution to the knapsack problem will include this item with profit p_∞. The rest of the proof is based on similar arguments as given in the proof of Theorem 4.

G Proof of Lemma 1

If the vertex v_i has an execution requirement of $e(v_i)$ and deadline equal to $d(v_i)$ and is lowest-priority feasible in the task set $\mathcal{T}\backslash\{T\}\bigcup\{v_i\}$ then within any time interval of length $d(v_i)$, tasks of $\mathcal{T}\backslash\{T\}$ never occupy the processor for an amount of time exceeding $d(v_i) - e(v_i)$. Clearly, if this is true for all vertices of T then no matter how they are triggered, they can always be executed to meet their deadlines, implying lowest-priority feasibility of T. It is trivial to see that the lemma is true even for the other direction.

H Proof of Theorem 9

To prove the sufficiency of this test, we are required to prove that if $\exists 0 \leq t \leq d(v)$ for which $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$ for all vertices v of T then T is lowest-priority feasible. Assume that $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$ and still the vertex v is not lowest-priority feasible. Then let v miss its deadline at time t_d when being scheduled by a static-priority scheduler and let it be the case that v was triggered at time t_r. Therefore, within the time duration $t_d - t_r = d(v)$, the processor was occupied for more than $d(v) - e(v)$ amount of time by the higher priority tasks, contradicting the assumption that $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$. The proof follows from the above argument along with Lemma 1.

I Proof of Theorem 10

Let $\mathcal{T}$ consist of two task graphs T_1 and T_2. By Lemma 1, we assume that T_1 consists of a single vertex with an execution requirement of e and deadline equal to d. T_2 is an arbitrary task graph. We want to test whether T_1 is lowest-priority feasible. The claim is that T_1 is lowest-priority feasible if and only if $\exists t \leq d$ for which $t - T_2.rbf'(t) \geq e$.

To prove that the condition is necessary, it is sufficient to prove that if $\forall t \leq d$, $t - T_2.rbf'(t) < e$ then T_1 is not lowest-priority feasible. The proof of this is straightforward and hence we omit the details.