Interactive Schedulability Analysis

UNMESH D. BORDOLOI and SAMARJIT CHAKRABORTY
Department of Computer Science
National University of Singapore

A typical design process for real-time embedded systems involves choosing the values of certain system parameters and performing a schedulability analysis to determine whether all deadline constraints can be satisfied. If such an analysis returns a negative answer, then some of the parameters are modified and the analysis is invoked once again. This iteration is repeated till a schedulable design is obtained. However, the schedulability analysis problem for most task models is intractable (usually co-NP hard) and hence such an iterative design process is often very expensive. To get around this problem, we introduce the concept of "interactive" schedulability analysis. It is based on the observation that if only a small number of system parameters are changed, then it is not necessary to rerun the full schedulability analysis algorithm, thereby making the iterative design process considerably faster. We refer to this analysis as being "interactive" because it is supposed to be run in an interactive mode. This concept is fairly general and can be applied to a wide variety of task models. In this paper we have chosen the recurring real-time task model because it can be used to represent realistic applications from the embedded systems domain (containing conditional branches and fine-grained deadline constraints). Our experimental results show that using our scheme can lead to more than 20× speedup for each invocation of the schedulability analysis algorithm, compared to the case where the full algorithm is run.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; J.7 [**Computer Applications**]: Computers in other systems—*Real Time*

General Terms: Algorithms, Design, Performance, Verification

Additional Key Words and Phrases: Schedulability analysis, Recurring real-time task model, Interactive design, Performance debugging, Non-functional constraints

1. INTRODUCTION

Schedulability analysis plays an integral role in the system-level design of real-time embedded systems. Once a designer chooses the values of certain system parameters, schedulability analysis is used to determine whether it is possible to assign to each job a processor time equal to its worst-case execution requirement, between its ready time and its deadline. If such an analysis returns a negative result

A preliminary version of this paper appeared in the Proceedings of 12th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2006.

Authors' addresses: U. D. Bordoloi and S. Chakraborty, Department of Computer Science, National University of Singapore, Singapore 117543; email: {unmeshdu, samarjit}@comp.nus.edu.sg.

(i.e. there exist legal scenarios where certain jobs might miss their deadlines), then some of the system parameters are modified and the analysis is invoked once again. In a typical system design process, this iteration is repeated a number of times, until a schedulable system is obtained.

Unfortunately, the schedulability analysis problem for most task models is intractable (usually co-NP hard). Therefore, known algorithms for these models have an exponential complexity and at best run in pseudo-polynomial time. As a result, the above-mentioned iterative design process can become overly tedious for even reasonably-sized problems. To get around this, recent research in the real-time systems area has focused on either obtaining efficient pseudo-polynomial time algorithms or on *approximately* solving the schedulability analysis problem [Albers and Slomka 2004; Chakraborty et al. 2002; Fisher and Baruah 2005].

In this paper we propose another possible approach to beat the high running times associated with schedulability analysis algorithms, especially in the context of an iterative design process. It is based on the observation that if only a small number of design parameters are changed, then it is not required to invoke the full schedulability analysis machinery. Rather, certain data structures can be created when the algorithm is run for the first time, and on subsequent invocations of the algorithm it is possible to exploit these data structures and run only a small subset of the regular schedulability analysis algorithm. We refer to this as *interactive* schedulability analysis because it would typically be used in an interactive mode—a designer would keep on modifying the values of a small number of system parameters and use this algorithm to test whether the system becomes schedulable.

This concept of interactive schedulability analysis is fairly general and can be applied to a number of well-known task models. In this paper, we have chosen the recently proposed recurring real-time task model [Baruah 2003] to illustrate this scheme. It has been shown in [Baruah 2003] that this model generalizes a number of task models. Further, it can be used to model realistic applications with conditional branches and fine-grained deadline constraints.

Before proceeding further, we would like to clarify what we mean by "modifying the values of system parameters". In the context of scheduling a set of task graphs, the relevant system parameters are determined by the underlying task model. For example, in the recurring real-time task model, vertices of task graphs are annotated with worst-case *execution times* and *deadlines*. The edges are annotated with *minimum intertriggering separation times* and each task graph is associated with a *period*, which specifies the minimum time interval between two consecutive triggerings of the graph. When the schedulability analysis of a task set returns a negative answer (i.e. *not* schedulable), a designer would typically relax a few deadline constraints associated with some of the vertices of the task graphs and run the algorithm once again. Other possible modifications might consist of increasing the values of some intertriggering separations, or increasing the period associated with a task graph, or decreasing the execution times associated with some of the vertices (possibly by rewriting/optimizing the code corresponding to those vertices). It might even be possible to split a vertex into two or more vertices, i.e. change the structure of a task graph.

Note that once a task set becomes schedulable, it is possible that a designer might now want to *constrain* (or reduce) the values of some of the above-mentioned parameters like deadlines, intertriggering separations, or task periods. This is in order to test whether the task set still remains schedulable with a tighter deadline, intertriggering separation, or period constraint. Often such an iterative process is used to obtain the tightest set of constraints under which a task set remains schedulable.

In this paper, we will however only be concerned with two specific types of modifications – relaxing and constraining the deadline associated with a vertex of a task graph. First, these are the most likely changes for a designer to make in an iterative design process. Second, from the standpoint of our proposed interactive schedulability analysis scheme, incorporating other types of modifications would essentially follow the same principles as those used for handling deadline modifications. Our goal in this paper is to lay the groundwork for interactive schedulability analysis and demonstrate the potential speedups that can be obtained. As a part of our future work we plan to extend this scheme to handle other types of modifications as well, such as the ones we listed above.

1.1 Overview of the Proposed Scheme

In this paper, we discuss our proposed interactive scheme in the context of dynamic priority feasibility analysis in a preemptive uniprocessor environment. A standard methodology based on the *processor demand criteria* (see [Baruah et al. 1999] and [Buttazzo 1997]) has emerged for the feasibility analysis of such systems. Towards this, the worst-case workload that can possibly be generated by a task (graph) is 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 positive real number t and returns the maximum possible cumulative execution requirement of jobs that can be legally generated by T and which have their ready-times and deadlines both within a time interval of length t. A set of concurrently executing tasks $\mathcal{T}$ is then schedulable under a fully preemptive uniprocessor model if and only if for all $0 < t \leq t_{\max}$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, where $t_{\max}$ is a function of the execution requirements of the tasks in $\mathcal{T}$ and their periods. This scheme therefore involves two stages:

(i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and

(ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t,\ \forall\ 0 < t \leq t_{\max}$.

For the recurring real-time task model, it turns out that for an arbitrary task graph T, computing $T.dbf(t)$ for any t is NP-hard (see [Chakraborty et al. 2001]). Further, $t_{\max}$ is pseudo-polynomial in the size of problem. Hence, a pseudo-polynomial number of checks have to be performed in stage (ii).

While computing $T.dbf(t)$ for different values of t in stage (i), we construct a table for each task graph $T \in \mathcal{T}$ (the details of which are described later in this paper). In an iterative design cycle, once the deadline $d(v)$ of a vertex $v \in T$ is changed and the schedulability analysis algorithm is invoked, the table corresponding to T need not be recomputed from scratch. Rather, only parts of it are updated—which is significantly faster than recomputing the entire table. For any t, $T.dbf(t)$ (where T

is the task graph with the changed $d(v)$) can now be computed from this updated table.

Similarly, we also avoid checking the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for *all* $0 < t \leq t_{\max}$. When the deadline $d(v)$ of a vertex $v \in T$ is changed, we compute the values of t at which the condition for schedulability i.e. $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ can possibly change due to $d(v)$. We then check the schedulability condition only for these values of t, which again can be considerably faster than checking this condition for all $t \leq t_{\max}$.

1.2 Related Work

To the best of our knowledge, the concept of interactive schedulability analysis—in the form that we present in this paper—has not been investigated before. The need for appropriate tool sets for interactive timing analysis has been emphasized in [Tokuda and Kotera 1988] and several other papers. [Tokuda and Kotera 1988] introduced an interactive tool, which helps to debug timing errors in real time programs. However, no formal or algorithmic results were presented. Neither did [Tokuda and Kotera 1988] present any result on how to speedup *interactive* timing analysis.

Most of the previous research on obtaining efficient algorithms for schedulability analysis for different real-time task models focused on designing either efficient pseudo-polynomial algorithms, or polynomial time solutions for restricted versions of task models. More recently, the concept of *approximate schedulability analysis* has been investigated in a number of papers (see, for example, [Chakraborty et al. 2002], [Albers and Slomka 2004], and [Fisher and Baruah 2005]). Unlike exact schedulability analysis, approximate schedulability analysis might return false positives or false negatives. Here, the basic idea is that if the schedulability analysis algorithm is occasionally allowed to return a false answer, then such an algorithm can be designed to run in polynomial time. For example, if the algorithm is allowed to return false positives then in some cases although a task set is *not* schedulable, the algorithm incorrectly returns *schedulable*. However, it can be guaranteed that even in such cases no task will miss its deadline by more than a prespecified time interval. Further, for *most* task sets the algorithm will return the correct answer. A similar algorithm that only returns false negatives can also be designed.

None of the above research directions however exploit the fact that often the schedulability analysis algorithm is repeatedly invoked, with minor modifications in the task graphs. This is the scenario we address in this paper. Although not directly related to the problem we address in this paper, recently there has been some work on computing the *space* of task periods and worst-case execution times that lead to schedulable systems (this is often referred to as computing the *schedulable region*) [Bini and Natale 2005]. The problem we address here, on the other hand, is an online or an interactive debugging scenario, where the designer is concerned with identifying *one* set of system parameters that lead to a schedulable design.

1.3 Organization of this Paper

The rest of the paper is organized as follows. In the next section we describe the recurring real-time task model and its schedulability analysis. Towards this, we present a dynamic programming algorithm for computing the *demand-bound func-*

tion for this model in Sections 2.2 and 2.3. In Section 3 we then present our scheme for interactive schedulability analysis, which partly makes use of the dynamic programming algorithm. Our experimental results are described in Section 4. When a task set is not schedulable, it is often helpful if the system designer can be provided feedback on the potential system parameters that can be changed to obtain a schedulable system. In Section 5 we outline some techniques for providing such feedback. Finally, we conclude by discussing some directions for future work.

2. THE RECURRING REAL-TIME TASK MODEL AND ITS SCHEDULABILITY ANALYSIS

The recurring real-time task model was recently proposed by Baruah in [Baruah 1998b; 2003]. It is especially suited for accurately modeling conditional real-time code with recurring behavior, i.e. where code blocks have conditional branches and run in an infinite loop, as is the case in many embedded applications. Further, this model also generalizes a number of well-known task models such as the multiframe model [Mok and Chen 1997], the generalized multiframe model [Baruah et al. 1999] and the recurring branching task model [Baruah 1998a].

A recurring real-time task T is represented by a task graph which is a directed acyclic graph with a unique source (a vertex with no incoming edges) and a unique sink (a vertex with no outgoing edges) vertex. Associated with each vertex v of this graph is its execution requirement $e(v)$, and deadline $d(v)$. Whenever the vertex v is *triggered*, it generates a job which has to be executed for $e(v)$ amount of time within $d(v)$ time units from the triggering-time. 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.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and if some 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 passed since the triggering of the vertex u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u (representing a conditional branch) then only one among v_1 and v_2 can be triggered, after the triggering of u. The triggering of the sink vertex can be followed by the source vertex getting triggered again but any two consecutive triggerings of the source vertex should be separated by at least $P(T)$ units of time, called the *period* of the task graph.

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 only exception is that v_{i+1} can also be the source and v_i the sink vertex, and in that case if there exists some vertex $v_j, j < i$, in the sequence such that v_j is also the source vertex then $t_{i+1} - t_j \geq P(T)$ must be additionally satisfied. The real-time constraints require that the job generated by triggering vertex $v_i, i = 1, \ldots, k$, be assigned the processor for $e(v_i)$ amount of time within the time interval $(t_i, t_i + d(v_i)]$.

Once jobs are generated, they execute independently of each other (and therefore a restriction like first-come-first-served can not hold). Therefore, to ascertain that a job generated by a vertex u completes execution before a job generated by

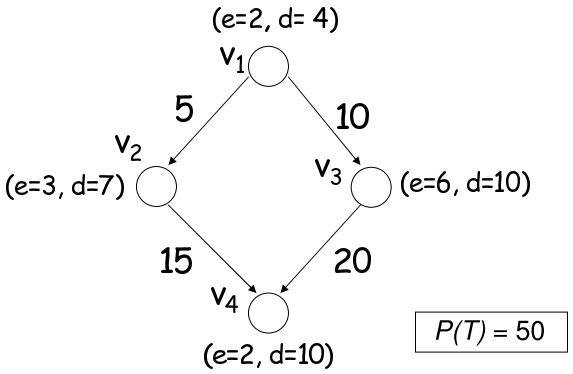


Fig. 1. An example recurring real time task.

a vertex v, when u and v belong to the same task graph and there is a directed edge from u to v, then either of the following conditions must hold: $p(u, v) \geq d(u)$, which guarantees that the vertex v can be triggered only after the job generated by vertex u has completed execution, or that $d(u) \leq p(u, v) + d(v)$, which guarantees that the absolute deadline of the job generated by vertex v is larger than or equal to the absolute deadline of the job generated by vertex u. In the real-time systems literature the first requirement is referred to as the *frame separation property* [Takada and Sakamura 1997] and the second as the *localized Monotonic Absolute Deadlines property* (*l-MAD*) [Baruah et al. 1999]. In this paper we assume either one of these two properties to hold.

Two points may be noted here. First, the original recurring real-time task model and its schedulability analysis, as proposed by Baruah in [Baruah 2003], is based on the *frame separation property* assumption. Second, our assumption that the *l-MAD* property leads to a job generated by a vertex u completing its execution before a job generated by a vertex v (when there is a directed edge from u to v) is based on the implicit assumption of the underlying scheduler uses the earliest deadline first (EDF) policy. We believe that this is a realistic assumption because EDF is known to be the optimal preemptive scheduling policy (i.e. if a task set is schedulable then EDF results in a feasible schedule) and it is widely used in real-life systems. Clearly, if the scheduling policy is not EDF then the *l-MAD* property along with the *processor demand criteria* for schedulability does not guarantee that a job generated by a vertex u will complete its execution before a job generated by v whenever there is a directed edge from u to v. Hence, we will from now on assume that the scheduling policy being used is EDF whenever the *l-MAD* property is assumed to hold true.

Figure 1 illustrates an example recurring real-time task. In this task, vertex v_3, for instance, has an execution requirement $e(v_3) = 6$, which must be met within 10 time units (its deadline) from its triggering time. The edge (v_1, v_3) has been labeled 10, which implies that the vertex v_3 can be triggered only after a minimum of 10 time units from the triggering of v_1 (i.e. the minimum intertriggering separation time). Edges (v_1, v_2) and (v_1, v_3) from vertex v_1 imply that either v_2 or v_3 can be triggered after v_1. The period of the task (the minimum time interval between two consecutive triggerings of the source vertex) is 50.

2.1 Task Sets and Schedulability Analysis

A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ 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 constitute 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 schedulability analysis of a task set $\mathcal{T}$ is concerned with determining whether the jobs generated by all possible legal triggering sequences of $\mathcal{T}$ can be scheduled such that their associated deadlines are met. Algorithms for the schedulability analysis of such task sets, in a preemptive uniprocessor setup, are based on certain *task independence assumptions*. These are: (i) The runtime behavior of a task is independent of any other tasks in the system. (ii) The constraints according to which legal job sequences are generated can be specified without any references to *absolute time*. Assumption (i) states that each task generates jobs independently of the jobs generated by other tasks in the system. Therefore, it is not permissible, for example, to require a task to generate a job in response to a job generated by another task. Assumption (ii) states that all temporal specifications defining the rules according to which jobs are generated by a task can only be relative to the time at which the task begins execution, or can be relative to the ready-time of another job of the same task. Therefore, a constraint like the ready-times of two consecutive jobs of a task must be separated by at least p time units, conforms to this requirement. Lastly, the time at which a task begins execution (i.e. the first job is generated) is not *a priori* known. For example, a task can begin execution in response to some external event.

Note that although the task independence assumptions restrict the job generation process of a task (for example, by specifying the minimum separation between the generation of two jobs), they make no assumptions about the interactions between the jobs once they are generated. Once a job is generated, it executes independently of any other job in the system, including those generated by the same task.

Given a sequence of jobs generated by a task set $[(T_i, a_i, e_i, d_i), (T_j, a_j, e_j, d_j), \ldots]$ (T_i refers to a task, a_i is the ready time of a job, e_i is its execution requirement, and d_i is its absolute deadline), the task independence assumptions imply that the sequence is legal if and only if all subsequences formed by jobs from the individual tasks are also legal (follows from Assumption (i)). Assumption (ii) implies that if $[(a_1, e_1, d_1), (a_2, e_2, d_2), \ldots]$ is a legal sequence of jobs generated by a task, then the sequence $[(a_1 - t, e_1, d_1 - t), (a_2 - t), e_2, d_2 - t), \ldots]$ is also legal, where t is any real number.

It directly follows from the description of the recurring real-time task model in Section 2 that the model indeed satisfies the above task independence assumptions (and so does a wide variety of other task models such as the sporadic, multi-frame, generalized multiframe, and the recurring branching models). The recurring real-time task model therefore lends itself to schedulability analysis based on the *processor demand criteria*, that we outlined in Section 1.1.

2.2 The *demand-bound function*

Recall from Section 1.1 that a task set $\mathcal{T}$ is schedulable if and only if $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ for all $0 < t \leq t_{\max}$. It can be proved that

$$t_{\max} = \frac{\sum_{T\in\mathcal{T}} 2E(T)}{1 - \sum_{T\in\mathcal{T}} \frac{E(T)}{P(T)}}$$

where $E(T)$ is the maximum cumulative execution requirement arising from a sequence of vertices on any path from the source to the sink vertex of the task graph T (see [Baruah 2003] for details).

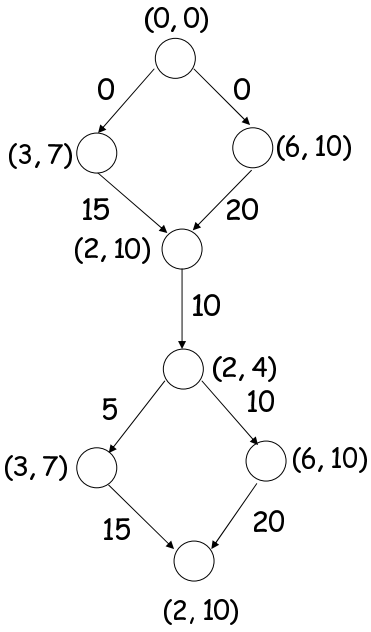
For any task graph T, computing the value of $T.dbf(t)$ for some (large) value of $t \leq t_{\max}$ might involve multiple traversals (loops) through the task graph. It was shown in [Baruah 2003] that if for a task graph T, $T.dbf(t)$ is known for all "small values" of t then it is possible to calculate from these, the value of $T.dbf(t)$ for *any* t. "Small values" of t for a task graph T are those for which the sequence of vertices that contribute towards computing $T.dbf(t)$ contain the source vertex at most once. The value of $T.dbf(t)$ for larger values of t is made up of some multiple of $E(T)$ plus $T.dbf(t')$ where t' is "small" in the sense described above. $T.dbf(t)$ for any t can hence be computed as follows (for a more detailed description, refer to [Baruah 2003]).

$$\begin{aligned} T.dbf(t) = \max\{&\lfloor t/P(T)\rfloor E(T) + T.dbf(t \bmod P(T)), \\ &(\lfloor t/P(T)\rfloor - 1)E(T) + T.dbf(P(T) + t \bmod P(T))\} \end{aligned} \tag{1}$$

To compute $T.dbf(t)$ for "small" values of t, [Baruah 2003] 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. (Two copies of the task graph in Figure 1 are joined in the fashion described above, and the resulting task graph is shown in Figure 2). $T.dbf(t)$ for all values of t are then calculated by enumerating all possible paths in this new graph. For arbitrary task graphs, this incurs a computation time which is exponential in the number of vertices in the task graph. The list alongside the task graph in Figure 2 gives us few values of $T.dbf(t)$ corresponding to some selected "small" values of t for this task graph. For instance, when $t = 4$, the $T.dbf(t)$ is 2, implying that within any time interval of 4 units the total execution requirement of jobs which have both their ready times and deadlines within this interval is 2. This means that there is no other permissible sequence of jobs which will have a demand greater than 2 within an time interval of 4. Similar explanation applies to other pairs of values listed in the table.

2.3 Computing the *demand-bound function*

In this section we present a dynamic programming algorithm for computing the demand-bound function $T.dbf(t)$ for any task graph T. It was shown in [Chakraborty et al. 2001] that computing $T.dbf(t)$ for any t is NP-hard for an arbitrary task graph T. The dynamic programming algorithm that we present here runs in pseudo-



t	$T.dbf(t)$
2	0
4	2
7	3
10	6
20	8
30	10

Fig. 2. Finding $T.dbf(t)$ for "small" values of t.

polynomial time and constructs a table, which is then used by our interactive schedulability analysis framework that we describe in Section 3.

The algorithm given below constitutes stage (i) of the two stages that we listed in Section 1.1. We first give an algorithm for computing the demand-bound function of a task graph for "small values" of t. Using this, we then compute the demand-bound function for any value of t as explained in Section 2.2.

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. If the frame separation property is followed then the newly added edge is labeled with an intertriggering separation of $p = d(v_{sink})$, and if the l-MAD property is followed then it is labeled with $p = \max\{0, d(v_{sink}) - d(v_{source})\}$, where v_{source} and v_{sink} denotes the source and the sink vertices of T. Now we give a pseudo-polynomial time algorithm based on dynamic programming, for computing $T'.dbf(t)$ 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 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 $p(v_i, v_j)$.

Algorithm 1 Computing $T'.dbf(t)$

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

1: **for** $e \leftarrow 1$ to nE **do**
2: $\quad t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$
3: $\quad flag_{1,e} \leftarrow \begin{cases} \texttt{SELF} & \text{if } e(v_1) = e \\ \texttt{PREVIOUS} & \text{otherwise} \end{cases}$
4: $\quad t^1_{1,e} \leftarrow t_{1,e}$
5: **end for**
6: **for** $i \leftarrow 1$ to $n-1$ **do**
7: $\quad$ **for** $e \leftarrow 1$ to nE **do**
8: $\quad\quad$ Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
9: $\quad\quad t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
10: $\quad\quad t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
11: $\quad\quad$ **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
12: $\quad\quad\quad flag_{i+1,e} \leftarrow$ `SELF`
13: $\quad\quad$ **else**
14: $\quad\quad\quad flag_{i+1,e} \leftarrow$ `PREVIOUS`
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end for**
18: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

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 $t \geq 0$ for one-shot executions of T'.

It can be shown by induction that Algorithm 1 correctly computes $T'.dbf(t)$, and has a running time of $O(n^3E)$. This algorithm, in addition, computes the values of a set of boolean variables which are referred to as $flag_{i,e}$. For any given value of i and e, $flag_{i,e}$ is set to $PREVIOUS$ if $t_{i-1,e} < t^i_{i,e}$ else it is set to $SELF$. The use of this variable will be explained in Section 3 when we describe our interactive schedulability analysis framework.

3. INTERACTIVE SCHEDULABILITY ANALYSIS FOR THE RECURRING REAL-TIME TASK MODEL

Having introduced all the necessary background, we are now in a position to describe our framework for interactive schedulability analysis. Recall from Section 1.1 that this framework is composed of two steps: (i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and (ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, $\forall\, 0 < t \leq t_{\max}$.

When the schedulability analysis algorithm is invoked for the first time, for each task graph $T \in \mathcal{T}$, Algorithm 1 is used to compute the values of $t^i_{i,e}$, $t_{i,e}$, and $flag_{i,e}$, which constitutes step (i). These are then stored in a table, which we will refer to as the *dbf-table*. For any task graph T, its *dbf-table* consists of rows which correspond to the vertices of T (ranging from 1 to n, assuming that T consists of n vertices) and columns which correspond to the different execution requirements that may be demanded by T due to a triggering of these vertices (ranging from 1 to nE). A cell (i, e) in this table contains three different values: $t_{i,e}$, $t^i_{i,e}$ and $flag_{i,e}$.

Now suppose that the schedulability analysis algorithm fails in step (ii), i.e. there exists some $\hat{t} \leq t_{\max}$ such that $\sum_{T \in \mathcal{T}} T.dbf(\hat{t}) > \hat{t}$. Then the system designer might choose to modify certain system parameters and run the schedulability analysis algorithm once again. Typically, this would involve rerunning steps (i) and (ii) from scratch. However, using our scheme for interactive schedulability analysis, we would instead only *update* the existing *dbf-tables* and recompute the appropriate $T.dbf(t)$ values from the updated tables. In most cases, this would be considerably faster than recomputing all the $T.dbf(t)$ values from scratch. Clearly, only the *dbf-tables* of task graphs that have been modified will have to be updated. Once the appropriate $T.dbf(t)$s have been recomputed, depending on the nature of the modifications made (e.g. deadlines have only been *relaxed*), the checking involved in step (ii) can be resumed from $\hat{t}$ onwards. There is no need to check the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for values of $t < \hat{t}$ since the task set already passed the schedulability test for these values of t.

The second possible scenario is when the task set $\mathcal{T}$ satisfies the schedulability test in step (ii) for all $t \leq t_{\max}$ (i.e. $\mathcal{T}$ is schedulable). In this case, the designer might still want to modify certain system parameters (e.g. constrain the deadlines associated with some of the vertices) and run the schedulability analysis algorithm once again. This might be to test if the task set remains schedulable under a tighter set of constraints. In this case, we would again update the *dbf-tables* and recompute the appropriate $T.dbf(t)$ values from the updated tables, as before. However, step (ii) will now become more involved—rather than checking the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for *all* $t \leq t_{\max}$, we check this condition only for those values of t at which the sum $\sum_{T \in \mathcal{T}} T.dbf(t)$ might have changed.

In the following two subsections we discuss the details of the two above-mentioned scenarios. Recall from Section 1 that in this paper we shall only be concerned with deadlines associated with vertices of task graphs being modified.

3.1 Relaxing the Deadline of a Vertex

Given a task graph T, let us assume that T' is obtained by joining two copies of T, followed by adding an edge from the sink vertex of the first copy to the source vertex of the second and replacing the source vertex of the first copy by a "dummy" vertex (as described in Section 2.3). We also assume that the *dbf-table* of T' has been computed. Now let us suppose that the deadline $d(v)$ associated with a vertex $v \in T$ has been relaxed. Unless v is the source vertex of T, this results in the deadlines of two vertices in T' (both of which correspond to the same vertex v in T) getting changed. Algorithm 2 then correctly updates *dbf-table* to reflect this change. Note that it has to be invoked either once or twice depending on whether v is a source vertex of T or not.

Algorithm 2 *dbf-table* update: Deadline relaxed case

Input: Task graph T', a real number $t \geq 0$, and a vertex number *node* such that deadline associated with vertex v_{node} in T' has been relaxed.

1: **for** $e \leftarrow 1$ to nE **do**
2: **for** $i \leftarrow node - 1$ to $n - 1$ **do**
3: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
4: $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
5: $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
6: **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
7: $flag_{i+1,e} \leftarrow$`SELF`
8: **else**
9: $flag_{i+1,e} \leftarrow$`PREVIOUS`
10: **end if**
11: **if** $i + 1 = node$ **then**
12: **if** $flag_{i+1,e} =$ `PREVIOUS then`
13: `break;`
14: `else if` $flag_{i+2,e} =$ `SELF then`
15: `break;`
16: `end if`
17: **else if** $flag_{i+2,e} =$ `SELF then`
18: `break;`
19: `end if`
20: **end for**
21: **end for**
22: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

To understand how Algorithm 2 works, let us assume that the deadline associated with the vertex v_{node} in T' has been relaxed, where the vertices of T' are $v_1, \ldots, v_n$, with a directed edge from v_i to v_j only if $i < j$. The algorithm traverses the rows of the *dbf-table* starting from the row *node* and recomputes the values of all the cells in these rows. Note that the lines 3 to 10 of this algorithm are the same as lines 8 to 15 of Algorithm 1; they compute the value of the $(i + 1, e)$th cell of the *dbf-table*. From lines 11 to 19 of Algorithm 2 it may be seen that cells corresponding to vertices numbered higher than *node* are recomputed depending on the values of the $flag$ variables.

Let us understand the principle behind the lines 11 to 19 first, and then we will work through an example. Let k be such that $node < k \leq n$. Note that the value $t^k_{k,e}$(for any k, and any e, where $1 \leq e \leq n$) is not changed if we relax the deadline of v_{node}(this follows from line 4.). Thus, a variable $t_{k,e}$ might change if and only if $t_{k-1,e}$ has changed(see line 5). Also, recall that the variable $flag_{node,e}$ had been assigned to either $PREVIOUS$ or $SELF$. If $flag_{node,e} = PREVIOUS$, it implies that $t_{node,e}$ has not modified. From the above two observations we conclude, that we need not update any cell on the column e, if $flag_{node,e} = PREVIOUS$.

On the other hand, if $flag_{node,e} = SELF$, $t_{node,e}$ would change with the relaxing of the deadline. This implies that $t_{node+1,e}$ would change too, if $flag_{node+1,e} =$

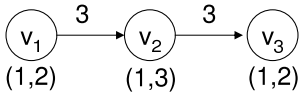


Fig. 3. The task graph T.

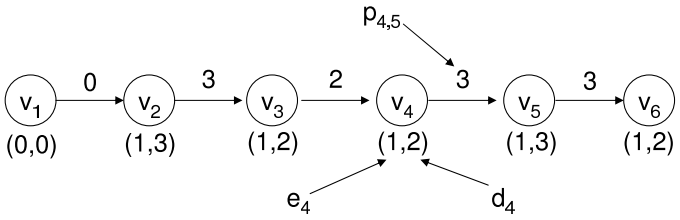


Fig. 4. The task graph T'.

$i\uparrow$	$e \rightarrow$					
	1	**2**	**3**	**4**	**5**	**6**
6	2, 2, S	4, 5, P	7, 8, P	10, 10, S	13, 13, S,	∞, ∞, S
5	2, 3, P	4, 6, P	7, 8, P	11, 11, S	∞, ∞, S,	∞, ∞, S
4	2, 2, S	4, 4, S	7, 7, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
3	2, 2, S	5, 5, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
2	3, 3, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
1	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S

Table I. *dbf-table* of T'.

PREVIOUS. The change might then propagate along the higher cells of the column e, depending on the value of their respective flags. This selectively computation is what exactly taken care of in the lines 11 to 19, and where in many cases, a large chunk of computation is avoided, compared to recomputing the entire *dbf-table* from scratch.

To appreciate why Algorithm 2 will often be computationally less expensive compared to recomputing the entire *dbf-table*, let us consider a small example. Let T be a task graph with 3 vertices, v_1, v_2, v_3, such that an edge from v_i to v_j exists if and only if $j = i + 1$. Let $e(v_i) = 1$ for all $1 \leq i \leq 3$ in T. The deadlines of the vertices are $d(v_1) = 2$, $d(v_2) = 3$, and $d(v_3) = 2$. The minimum intertriggering separation times associated with the edges are $p(v_1, v_2) = 3$, and $p(v_2, v_3) = 3$. (see Figure 3) Let T' be the graph that is formed by joining two copies of this task graph T in the fashion described in Section 2.3. T' is shown explicitly in Figure 4.

The *dbf-table* of T' is shown Table I. For any $1 \leq i \leq 6$ and $1 \leq e \leq 6$, the (i, e)th cell of this table contains the values of $t_{i,e}$, $t^i_{i,e}$, and $flag_{i,e}$ (in this order), where P and S denotes the $PREVIOUS$ and $SELF$ values of $flag_{i,e}$ respectively.

Assume the deadline of source vertex of T has been changed from 2 to 3. This implies the deadline of v_4 in T' is relaxed from 2 to 3. The task graph with its new deadlines is illustrated in Figure 5. We then update the *dbf-table* using Algorithm 2. The new *dbf-table* is shown in Table II. Only the cells of Table I which were updated using Algorithm 2 are shown using a bold-italic font in Table II.

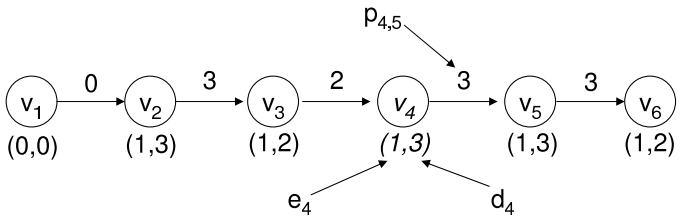


Fig. 5. Graph T' after relaxing the deadline associated with the vertex v_4 from 2 to 3.

$i\uparrow$	$e \rightarrow$					
	1	**2**	**3**	**4**	**5**	**6**
6	2, 2, S	***5, 5, S***	***8, 8, S***	10, 10, S	13, 13, S,	∞, ∞, S
5	2, 3, P	***5, 6, P***	***8, 8, S***	11, 11, S	∞, ∞, S,	∞, ∞, S
4	***2, 3, P***	***5, 5, S***	***8, 8 S***	∞, ∞, ***S***	∞, ∞, ***S***	∞, ∞, ***S***
3	2, 2, S	5, 5, S	∞, ∞, S	∞, ∞, S	∞, ∞,S,	∞, ∞, S
2	3, 3, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S
1	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S	∞, ∞, S,	∞, ∞, S

Table II. The updated *dbf-table* after relaxing the deadline associated with the vertex v_4 from 2 to 3.

Since only the deadline of v_4 was relaxed, the execution demand arising from any vertex numbered less than 4 remains unchanged. Hence, the only potential cells of Table I which might be effected are on or above row 4. Algorithm 2 first traverses row 4 of this table and recomputes the values of its cells. However, it does not "propagate" a change upwards, along the column of a cell, if the $flag$ in the cell is equal to $PREVIOUS$. If the value of the $flag$ equals to $PREVIOUS$, then it implies that the value of $t_{4,e}$ is equal to $t_{3,e}$. This means that $t^4_{4,e} > t_{3,e}$. Since the deadline of v_4 is being relaxed, the new value of $t^4_{4,e}$ will definitely be greater than its previous value, and hence also greater than $t_{3,e}$. So any $t_{j,e}$, where $j > 4$, need not be changed as a result of relaxing $d(v_4)$. For example, we can verify from Table II that when $e = 1$, this is the scenario that occurs. This clearly saves a significant amount computation, compared to the case where the full *dbf-table* is recomputed.

The second scenario is when one of the cells has its $flag$ set to $flag = SELF$. In our example, cells $(4, 2)$, and $(4, 3)$ illustrate this scenario. Let us consider cell $(4, 3)$, where $flag_{4,3} = SELF$ implies that $t^4_{4,3} < t_{3,3}$. Hence, the value of $t_{4,3}$ will change and might in turn lead to changes in the higher cells along this column. Therefore, we need to check whether any higher numbered vertices might also be effected. The cell $(5, 3)$ has $flag = PREVIOUS$ and hence $t_{5,3}$ needs to be recomputed. Similarly cell $(6, 3)$ is also recomputed. Note that cell $(4, 4)$, has its $flags$ set to $SELF$, however since $flag_{5,4} = SELF$ we need not propagate the change along the higher numbered columns. This follows from line 17 of Algorithm 2.

If the schedulability test for a task set $\mathcal{T}$ fails at $t = \hat{t}$ then in this case (i.e. when deadlines associated with vertices are only being relaxed), after the deadlines associated with one or more vertices are relaxed, the check in step (ii) of our scheme can be resumed at $t = \hat{t}$.

Algorithm 3 *dbf-table* update: Deadline constrained case

Input: Task graph T', a real number $t \geq 0$, and a vertex number *node* such that deadline associated with vertex v_{node} in T' has been constrained.

1: **for** $e \leftarrow 1$ to nE **do**
2: **for** $i \leftarrow node - 1$ to $n - 1$ **do**
3: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
4: $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
5: $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
6: **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
7: $flag_{i+1,e} \leftarrow$`SELF`
8: **else**
9: $flag_{i+1,e} \leftarrow$`PREVIOUS`
10: **end if**
11: **if** $i + 1 = node$ **then**
12: **if** $flag_{i+1,e} =$ `PREVIOUS then`
13: `break;`
14: `else if` $flag_{i+2,e} =$ `SELF then`
15: `if` $t_{i+1,e} \geq t^{i+2}_{i+2,e}$ `then`
16: `break;`
17: `end if`
18: `end if`
19: **else if** $flag_{i+2,e} =$ `SELF then`
20: `if` $t_{i+1,e} \geq t^{i+2}_{i+2,e}$ `then`
21: `break;`
22: `end if`
23: `end if`
24: **end for**
25: **end for**
26: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$

3.2 Constraining the Deadline of a Vertex

Let us now consider the case where the deadline of a vertex $v \in T$ is *constrained*. As in the previous case, depending on whether v is a source vertex in T or not, this would result in two vertices in T' getting affected (where T' is obtained by joining two copies of T). Again, let v_{node} be a vertex in T' whose deadline is constrained. Then Algorithm 3 updates the *dbf-table* corresponding to T'.

Algorithm 3 is similar to Algorithm 2, except for a pair of extra conditions in lines 15 and 20. The reason behind these will be clarified in the following discussion.

Let us assume that the deadline of vertex v_{node} is constrained. Let k be such that $node < k \leq n$. Note that the value $t^k_{k,e}$(for any k, and any e, where $1 \leq e \leq n$) is not changed if we constrain the deadline of v_{node}. A variable $t_{k,e}$ might change if and only if $t_{k-1,e}$ has changed. If flag $flag_{node,e} = PREVIOUS$, the case is exactly similar to the case of deadlines being relaxed.

On the other hand, if $flag_{node,e} = SELF$, then we know that $t_{node,e} = t^{node}_{node,e}$ and consequently, constraining the deadline of v_{node} implies $t_{node,e}$ decreases. In

such a case, if $flag_{node+1,e} = PREVIOUS$, then the scenario is again similar to the case when the deadline of v_{node} was relaxed and the value $t_{node+1,e}$ will have to be updated. The change might then "propagate" along the higher cells of the column e, depending on the value of their flags.

However, if $flag_{node+1,e} = SELF$ (which implies that $t_{node+1,e} = t^{node+1}_{node+1,e}$), the scenario is different from when the deadline of v_{node} was relaxed. The reason for this being, after the deadline was constrained, it might now be that $t_{node,e} < t^{node+1}_{node+1,e}$. Hence, despite $flag_{node+1,e} = SELF$ being true, $t_{node+1,e}$ will change and we need to update the cell $(node + 1, e)$. Similar reasoning also holds true when we select any cell (i, e) for updating, where $i > node$. This explains the need of the extra pair of conditions.

3.2.1 *Efficiently Performing Step (ii).* As we discussed before, here we would like to avoid performing the check $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ for all values of $t \leq t_{\max}$. Let us assume that the deadline associated with a certain vertex of T has been constrained. We also assume that T belongs to a task set $\mathcal{T}$, which was originally schedulable. Algorithm 3 is then used to update the *dbf-table* associated with T. Now our goal is to identify those values of t at which the sum $\sum_{T\in\mathcal{T}} T.dbf(t)$ was modified; we would like to check the condition $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ only at these values of t. Towards this, we first scan the updated *dbf-table* and identify those values of t for which $t < P(T)$ and either $T.dbf(t)$ or $T.dbf(t + P(T))$ have been updated. Let t_{change} be the first such value of t in this table. Let t_{check} be a possible value of t that we are interested in identifying. It then follows from Eq. 1 of Section 2.2 that for each value of t_{change}, there will be multiple t_{check}s. These t_{check}s are given by:

$$t_{check} = t_{change} + kP(T)$$

where $k = 0, \ldots, N$ and N is the largest integer satisfying the inequality $t_{change} + NP(t) \leq t_{max}$.

The above procedure has to be repeated for all possible values of t_{change} in the updated *dbf-table* and the corresponding t_{check}s are identified. The schedulability test $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ is then performed at these t_{check}s.

3.3 Running Times

Note that both the algorithms for updating the *dbf-table* (i.e. Algorithms 2 and 3), have a worst-case running time of $O(n^3E)$. Hence, in the worst-case, updating the *dbf-table* involves the same computational cost as that involved in computing this table from scratch. Clearly, at least from a theoretical standpoint, our scheme would have been more attractive had this been otherwise. However, as we have pointed out in Section 3.1, for most problems the actual running time incurred by our algorithms would be significantly less than what would be involved in recomputing the entire *dbf-table*. As an example, let us consider Algorithm 2. We saw that when the deadline of a vertex v_{node} was relaxed, then the cells $1, 2, \ldots, nE$ of row *node* were unconditionally recomputed. However, any cell on a row numbered higher than *node* will have to be updated depending on the conditions in lines 11 to 19 of the algorithm. Hence, updating a single column of the *dbf-table* will incur the worst-case cost only when the value of $t_{node,e}$ is less than $t_{i,e}$ for all $i > node$. Further,

for the worst-case (in terms of updating the *dbf-table*) to occur, the worst-case update scenario of a column must happen for *all* columns $1, 2, \ldots, nE$. For most problem instances, such corner cases are unlikely to happen and as our experimental results show in Section 4, our scheme results in a significant speedup compared to recomputing the *dbf-table* for each change.

Similarly, in the worst-case, stage (ii) might also require that the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ to be checked for all $t \leq t_{\max}$. But once again, for most problem instances, this is unlikely to happen.

Finally, note that the space complexity of storing a *dbf-table* with n vertices is $O(n^2E)$. For each vertex i we store $t_{i,e}$, $t^i_{i,e}$, and $flag_{i,e}$, where e ranges from 1 to nE.

4. EXPERIMENTAL RESULTS

We conducted two broad categories of experiments. In Section 4.1 we report some experimental results that were obtained by running the dynamic programming algorithm (Algorithm 1) and our proposed algorithms for interactive schedulability analysis (Algorithms 2 and 3) on a set of synthetic task graphs. In Section 4.2 we illustrate the benefits of efficiently performing $Step(ii)$ of the schedulability analysis (which we described in Section 3.2.1).

4.1 Experiments with Step (i)

For our experiments we randomly generated synthetic task graphs using two parameters. The first is the maximum execution requirement, E, associated with any vertex of a graph. The second parameter is called the *connectivity factor*. If $v_1, \ldots, v_n$ are the vertices of a task graph such that there is an edge from v_i to v_j only if $j > i$, then while generating the graph, for each vertex v_j we construct an edge from v_i to v_j with a probability equal to the connectivity factor of the graph, for all $i = 1, \ldots, j-1$.

The parameters (i.e. E and the connectivity factor) used to generate our synthetic graphs were chosen such that the graphs represent realistic network packet processing applications. The details of this application may be found in [Chakraborty et al. 2002]. A connectivity factor equal to 0.4 was used to generate all the task graphs since this results in graphs which are similar to those arising in practice. It may be noted here that a higher connectivity factor would clearly result in more paths in any graph. Hence, this would lead to higher savings from our scheme compared to when all the paths in a graph are exhaustively enumerated to compute the *demand-bound function*. E was set equal to either 200 or 600, representing two possible cases in the above-mentioned application.

Figure 6 shows the running times involved in computing the *dbf-table* of a single task graph. Once the deadline associated with a vertex of this task graph was relaxed, we have (i) recomputed the entire *dbf-table* using Algorithm 1, and (ii) updated the *dbf-table* using Algorithm 2. Figures 6(a) and 6(b) show the running times incurred for task graphs with number of vertices ranging from 50 to 200, which were generated by setting $E = 200$ and $E = 600$ respectively. The task graphs formed by joining together two copies of our original task graphs had 100 to 400 vertices (as explained in Section 2.3), and the computation of the *dbf-table* used these graphs.

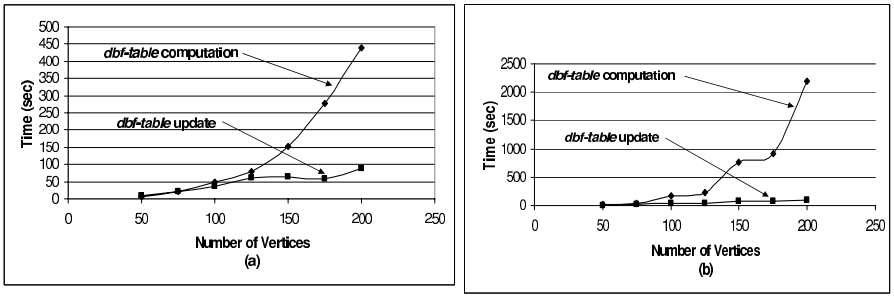


Fig. 6. Running times for updating the *dbf-table* when the deadline of a vertex was relaxed (a) $E = 200$ and (b) $E = 600$.

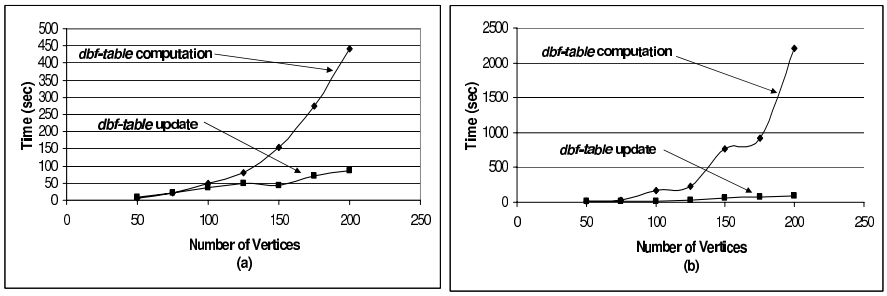


Fig. 7. Running times for updating the *dbf-table* when the deadline of a vertex was constrained (a) $E = 200$ and (b) $E = 600$.

For each randomly generated task graph, we randomly selected a vertex of this graph and relaxed its deadline by a certain amount. The *dbf-table* associated with this task was then (i) entirely *recomputed*, and (ii) *updated* using our proposed scheme. For each task graph, this process was repeated for five randomly selected vertices. The results in Figures 6(a) and 6(b) report the maximum *dbf-table* update time incurred among these five vertices, along with the time required to recompute the entire *dbf-table*. These results illustrate the savings achieved by our proposed scheme. With $E = 600$, we obtain a speedup of more than 20×, which translates into the schedulability analysis running in approximately 2 minutes instead of 40 minutes. In an interactive design environment, the former waiting time is clearly more tolerable than the latter. It should also be noted that with larger values of E, even higher speedups will be obtained. Figures 7(a) and 7(b) show similar results for the case where the deadline of a vertex was constrained.

We also conducted another set of experiments with relatively smaller task graphs (containing 50 vertices), while varying the value of E from 1000 to 10000. Here, it may be noted that the execution requirement associated with any vertex of a graph is expressed in terms of *time units*. Such time units depend on the application at hand and might denote milliseconds, microseconds, or even the number of clock cycles of the processor on which the task graphs are required to execute. Hence,

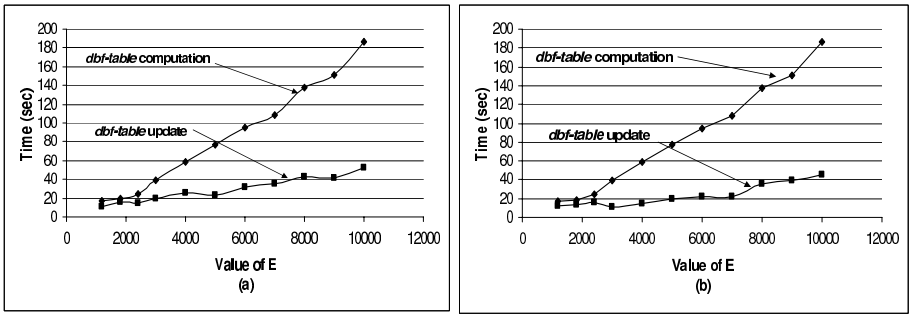


Fig. 8. Running times for updating the *dbf-table* for a task graph with 50 vertices, as the maximum execution requirement associated with a vertex (E) is increased. (a) Deadline of a randomly chosen vertex is relaxed, and (b) Deadline of a randomly chosen vertex is constrained.

experiments with large values of E are completely realistic. Our motivation behind experimenting with small task graphs is that most realistic applications are likely to be represented by task graphs containing relatively few vertices. The steps involved in this set of experiments are exactly similar to those of the earlier experiments. Figure 8(a) shows how the *dbf-table* update time and computation time changes with increasing E (the maximum execution time associated with a vertex), when the deadline associated with a randomly chosen vertex of a task graph is relaxed. Figure 8(b) shows the corresponding results when the deadline associated with a vertex is constrained. Note that in both the cases we obtain speedups of around 5×, which are significant if a design tool is to be used in an interactive fashion.

All the CPU times reported above were measured on a Linux machine with Fedora Core 3, running on a 3.0 GHz CPU with a 2 GB RAM.

It may be noted that all our implementations were done in C++, did not make use of any graphical interfaces for specifying the task graphs, and the code was specifically optimized for running the schedulability analysis. In practice, a design tool supporting schedulability analysis would be more involved. More specifically, the task graphs might be integrated with other application-specific data structures that are not be optimized for the schedulability analysis algorithm. In such cases, the speedups obtained by our interactive schedulability analysis might be considerably higher compared to the results reported here. This is because it involves fewer traversals through these task graphs in subsequent invocations of the analysis, thereby saving the overheads associated with these traversals due to the potentially complicated data structures. This observation stems from our attempt to integrate this schedulability analysis algorithm inside a tool-suite [Esser and Janneck 2001] where the task graphs were specified using a graphical user interface and were embedded inside other data structures that were a part of this tool-suite. In this implementation we observed 20× speedups using our algorithm for task graphs with less than 40 vertices. However, with the optimized C++ implementation of our algorithm, such speedups could only be seen for task graphs with around 200 vertices.

Task Sets	Task Graphs	
Set 1 #vertices/task graph = 10 max. exec. req. of a vertex (E) = 200 $t_{\max} = 54.6 \times 10^3$	T_1 T_2 T_3	1.647×10^3 1.799×10^3 4.474×10^3
Set 2 #vertices/task graph = 20 max. exec. req. of a vertex (E) = 200 $t_{\max} = 368.353 \times 10^3$	T_1 T_2 T_3	3.759×10^3 2.662×10^3 84.634×10^3
Set 3 #vertices/task graph = 30 max. exec. req. of a vertex (E) = 200 $t_{\max} = 823.834 \times 10^3$	T_1 T_2 T_3	8.657×10^3 4.975×10^3 104.517×10^3
Set 4 #vertices/task graph = 40 max. exec. req. of a vertex (E) = 200 $t_{\max} = 806.714 \times 10^3$	T_1 T_2 T_3	7.017×10^3 13.906×10^3 55.96×10^3
Set 5 #vertices/task graph = 50 max. exec. req. of a vertex (E) = 200 $t_{\max} = 1431 \times 10^3$	T_1 T_2 T_3	6.861×10^3 13.005×10^3 8.945×10^3

Table III. Number of checks required in *Step (ii)* of the proposed interactive schedulability analysis, versus $t_{\max}$, which is equal to the number of checks that a regular schedulability analysis algorithm would perform. This table shows the results for five task sets, with each set containing three task graphs. The numbers in the rightmost column are the number of checks in *Step (ii)* when the deadline associated with a randomly chosen vertex of the task graph in the same row is constrained.

4.2 Experiments with Step (ii)

In Section 3.2.1, we had outlined an efficient method to perform *Step*(ii) of our proposed interactive schedulability analysis. This section illustrates the savings obtained by using that method. For our experiments, we generated five task sets with each set consisting of three task graphs. The number of vertices in these task graphs ranged over 10 to 50, with the first task set consisting of task graphs with 10 vertices, the second task set consisting of task graphs with 20 vertices, and so on. The value of E for all the task graphs was set to 200.

We randomly chose a vertex of a task graph and constrained its deadline. We then computed the number of checks that were needed to perform *Step*(ii), following the description in Section 3.2.1. The results obtained are shown in Table III. This experiment was repeated for each task graph in the five task sets. Note from Table III that there are cases where the number of checks of the schedulability condition reduce to almost 0.5% of the total number of checks that would be performed by a regular schedulability analysis algorithm. This again illustrates the potential savings that our interactive schedulability analysis can achieve.

5. PROVIDING FEEDBACK TO THE SYSTEM DESIGNER

In what we have seen so far, if a task set fails the schedulability test for a certain $\hat{t}$, a system designer is allowed to randomly select some of the vertices of certain task graphs, relax their deadlines and rerun the analysis. However, relaxing the deadline of some randomly selected vertex might not make the task set schedulable. Hence, it would be meaningful to provide some feedback to the designer about potential vertices, whose deadlines might be changed to make the task set schedulable. Other types of feedback like changing the periods of certain task graphs or increasing the intertriggering separation times associated with some of the edges of a task graph might also be meaningful. Such feedback can be provided using the scheme we have presented in this paper.

Towards this, the algorithm used for computing the *dbf-table* (i.e. Algorithm 1) needs to be changed, so that some additional data structures are computed. These data structures, $Q_{i,e}$ and $Q^e_{i,e}$, are computed by Algorithm 4.

Recall that each cell in our *dbf-table* contains three different values: $t^i_{i,e}$, $t_{i,e}$, and $flag_{i,e}$. In addition to these, we now store two lists $Q_{i,e}$ and $Q^e_{i,e}$ in each cell. $Q_{i,e}$ records the subset of vertices from the set $\{v_1, \ldots, v_i\}$, whose triggering demands an execution time of e, within any time interval of length $t_{i,e}$. Similarly, $Q^e_{i,e}$ lists the subset of vertices from $\{v_1, \ldots, v_i\}$, which ends with the vertex v_i and has an execution requirement of e within any time interval of length $t^i_{i,e}$. Algorithm 4 not only returns $T'.dbf(t)$, but also the list of vertices $Q(t)$ whose triggering results in the execution demand of $T'dbf(t)$.

We now explain how $Q(t)$ can be used to provide useful feedback to a system designer. Recall from Section 2.2 that we create a list of $T.dbf(t)$ for all "small" values of t. To this list, we now add the data structure $Q(t)$ containing the vertices that contribute to $T.dbf(t)$. During the schedulability test in step (ii), suppose the test fails at $\hat{t}$. If $\hat{t}$ is "small", then we can find the desired list of vertices $Q(\hat{t})$ directly from the table. If $\hat{t}$ is "large", we check whether $T.dbf(\hat{t})$ is equal to $\lfloor \hat{t}/P(T) \rfloor E(T) + T.dbf(\hat{t} \bmod P(T))$ or $(\lfloor \hat{t}/P(T) \rfloor - 1)E(T) + T.dbf(P(T) + \hat{t} \bmod P(T))$(see Eqn. 1) ($T.dbf(\hat{t})$ has to be equal to either of these two values). If $T.dbf(\hat{t})$ is equal to the former expression then we select the vertices listed as $Q(\hat{t} \bmod P(T))$ from our table, otherwise we select the vertices corresponding to $Q(P(T) + \hat{t} \bmod P(T))$.

Hence, given any $\hat{t}$ for which the schedulability test failed, for any task graph T we can identify the legal sequence of vertices whose triggering contributed to $T.dbf(\hat{t})$. This sequence of vertices can now be used by the system designer to modify their associated deadlines or the intertriggering separations associated with their edges. In what follows, we refer to this sequence of vertices as the *critical path* of a task graph that is responsible for its (non-) schedulability.

5.1 Illustration of the Feedback Provided for an Example Task Set

Consider a task set τ, consisting of two task graphs T_1 and T_2, shown in Figure 9. Now assume that we would like to verify whether τ is schedulable, and in case it is not, we would like to change the deadlines of the *appropriate* vertices in order to make it schedulable. Here we illustrate how the scheme that we presented above can be used to effectively identify such appropriate vertices.

Algorithm 4 Computing of $T'.dbf(t)$ with data structures for providing feedback

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

1: **for** $e \leftarrow 1$ to nE **do**
2: **if** $e(v_1) = e$ **then**
3: $t_{1,e} \leftarrow d(v_1)$
4: $flag_{1,e} \leftarrow$ `SELF`
5: $enqeue(Q_{1,e}, v_1)$
6: **else**
7: $t_{1,e} \leftarrow \infty$
8: $flag_{1,e} \leftarrow$ `PREVIOUS`
9: **end if**
10: $t^1_{1,e} \leftarrow t_{1,e}$
11: **end for**
12: **for** $i \leftarrow 1$ to $n-1$ **do**
13: **for** $e \leftarrow 1$ to nE **do**
14: Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}
15: $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \,|\, j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
16: Let v_{min} be the vertex from amongst the set of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$, which gave us the minimum value for the expression evaluated in line number 15
17: **if** $e(v_{i+1}) < e$ **then**
18: $Q^{i+1}_{i+1,e} \leftarrow Q_{min, e-e(v_{i+1})}$
19: $enqueue(Q^{i+1}_{i+1,e}, v_{i+1})$
20: **else if** $e(v_{i+1}) = e$ **then**
21: $enqueue(Q^{i+1}_{i+1,e}, v_{i+1})$
22: **end if**
23: $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
24: **if** $t_{i+1,e} = t^{i+1}_{i+1,e}$ **then**
25: $Q_{i+1,e} \leftarrow Q^{i+1}_{i+1,e}$
26: $flag_{i+1,e} \leftarrow$`SELF`
27: **else**
28: $Q_{i+1,e} \leftarrow Q_{i,e}$
29: $flag_{i+1,e} \leftarrow$`PREVIOUS`
30: **end if**
31: **end for**
32: **end for**
33: $T'.dbf(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$
34: $Q(t) \leftarrow Q_{n,e}$

T'_1 and T'_2 (shown in Figure 10) were obtained by joining two copies of T_1 and T_2 respectively, and will be used to compute $dbf(t)$ for "small" values of t.

Clearly, the schedulability analysis returns a negative answer for the task set τ. Further, Algorithm 4 provides the following feedback concerning the potential vertices whose deadlines may be relaxed:

—Critical Path for Task Graph T'_1: v_6

—Critical Path for Task Graph T'_2: v_7

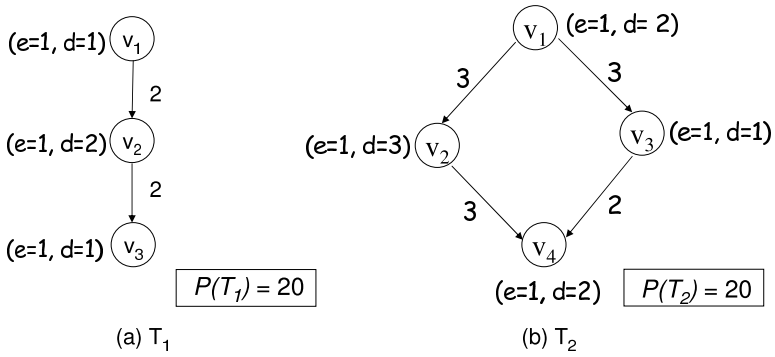


Fig. 9. Task graphs (a) T_1 and (b) T_2 of our example task set τ.

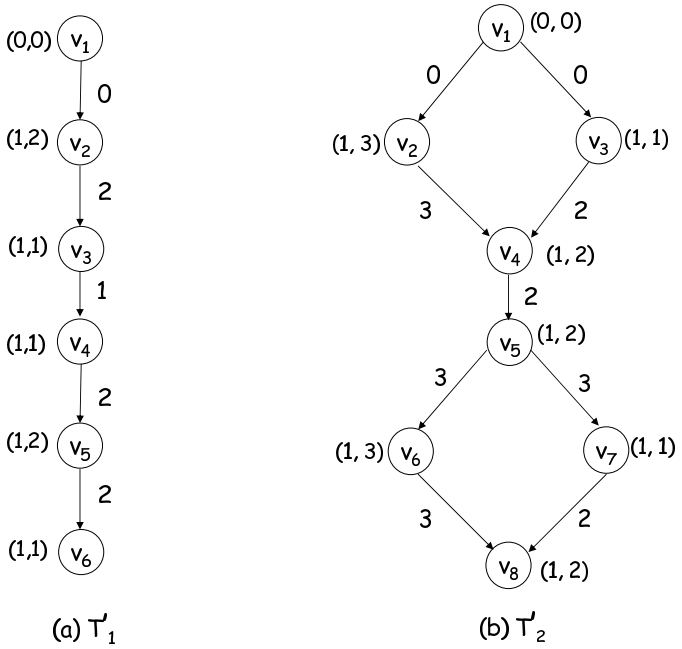


Fig. 10. Task graphs (a) T_1' and (b) T_2' obtained from T_1 and T_2 respectively.

Indeed from Figure 10, we see that v_6 of T_1' and v_7 of T_2', both demand 1 unit of execution time within a time interval of 1 unit. Thus, $\sum_{T \in \mathcal{T}} T.dbf(1) = 2$, implying that the condition $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ is not satisfied at $t = 1$. Now, one might choose to relax the deadlines associated with v_3 and v_7 of T_2' from 1 to 2. It may be noted here that in practice, the task graphs T_1' and T_2' will not be visible to a designer and he or she will only work with the original graphs T_1 and T_2. Any changes made in these two task graphs can easily be translated to appropriate changes in T_1' and T_2'.

Now we re-run the analysis and find that the task set is still not schedulable, along with the following feedback:

—Critical Path for Task Graph T_1': v_3, v_4

—Critical Path for Task Graph T_2': v_8

To see that these paths are indeed critical to schedulability, note that from the path v_3, v_4 we get $T_1'.dbf(2) = 2$. Similarly, in task graph T_2', v_8 leads to $T_2'.dbf(2) = 1$. Thus, $\sum_{T \in \mathcal{T}} T.dbf(t) > t$, at $t = 2$. Again, to move towards a schedulable system, we now relax the deadline of v_4 of T_1' from 1 to 2, and rerun the analysis.

However, the task set is still not schedulable, and the feedback provided is as follows:

—Critical Path for Task Graph T_1': v_3, v_4, v_5, v_6

—Critical Path for Task Graph T_2': v_3, v_4, v_5

One can verify that the above sequence of paths lead to $\sum_{T \in \mathcal{T}} T.dbf(6) = 7$, thereby failing the schedulability test. This time we select v_5 of T_2', and relax its deadline from 2 to 3, thereby obtaining a schedulable system.

In the above example, we have seen the benefits of the feedback mechanism on a small task set. In larger systems where many more task graphs and more vertices would be involved, this mechanism would certainly be of immense benefit.

6. CONCLUDING REMARKS

In this paper we presented a scheme for efficient schedulability analysis of recurring real-time task sets, where the schedulability analysis is repeatedly invoked with small modifications in the task set. Since this scheme is used in an interactive fashion, we referred to it as *interactive schedulability analysis*.

Although in this paper we have focused on the specific problem of schedulability analysis, we believe that such a scheme can be used for a variety of timing analysis problems e.g. worst-case execution time analysis of programs using program path analysis techniques. To the best of our knowledge, the idea of such *interactive timing analysis* has not been studied before.

There are a number of other directions in which our work can be extended, the most notable among which is handling modifications other than extending or relaxing deadlines associated with the vertices of a task graph. As we have mentioned before, such modifications can include changing intertriggering separations associated with the edges of a task graph. We also believe that it would be interesting to identify specific classes of changes for which updating the *dbf-table* can be done in polynomial time. Further work should also be done towards providing more directed feedback to a system designer, compared to what we have presented in this paper. Lastly, there are a number of recently developed tools for timing/schedulability analysis of embedded systems (see for example, [Amnell et al. 2003; Hamann et al. 2004]). It would certainly be meaningful to explore if our analysis can be incorporated inside these tools in a smooth way.

ACKNOWLEDGMENTS

This research was funded by the NUS URC grant R-252-000-190-112. Thanks are also due to the anonymous RTAS 2006 reviewers and the ACM TECS reviewers for their suggestions and helpful comments, which have improved this paper.

REFERENCES

Albers, K. and Slomka, F. 2004. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE Computer Society, Catania, Italy.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. 2003. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *International Workshop on Formal Modeling and Analysis of Timed Systems*. Lecture Notes in Computer Science 2791. Springer-Verlag, Marseille, France.

Baruah, S. 1998a. Feasibility analysis of recurring branching tasks. In *Proceedings of the Euromicro Workshop on Real-Time Systems*. IEEE Computer Society, Berlin, Germany.

Baruah, S. 1998b. A general model for recurring real-time tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Madrid, Spain.

Baruah, S. 2003. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems 24,* 1, 93–128.

Baruah, S., Chen, D., Gorinsky, S., and Mok, A. 1999. Generalized multiframe tasks. *Real-Time Systems 17,* 1, 5–22.

Bini, E. and Natale, M. D. 2005. Optimal task rate selection in fixed priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Miami, USA.

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

Chakraborty, S., Erlebach, T., Künzli, S., and Thiele, L. 2002. Schedulability of event-driven code blocks in real-time embedded systems. In *Proceedings of the Design Automation Conference*. ACM, New Orleans, USA.

Chakraborty, S., Erlebach, T., and Thiele, L. 2001. On the complexity of scheduling conditional real-time code. In *Proceedings of the International Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, vol. 2125. Springer, Rhode Island, USA.

Chakraborty, S., Künzli, S., and Thiele, L. 2002. Approximate schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Austin, USA.

Esser, R. and Janneck, J. W. 2001. MOSES - a tool suite for visual modeling of discrete-event systems. In *IEEE International Symposium on Human-Centric Computing Languages and Environments*. IEEE Computer Society, Stresa, Italy. http://www.tik.ee.ethz.ch/∼moses/.

Fisher, N. and Baruah, S. 2005. A polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE Computer Society, Porto, Portugal.

Hamann, A., Jersak, M., Richter, K., and Ernst, R. 2004. Design space exploration and system optimization with SymTA/S – symbolic timing analysis for systems. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Lisbon, Portugal.

Mok, A. and Chen, D. 1997. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering 23,* 10, 635–645.

Takada, H. and Sakamura, K. 1997. Schedulability of generalized multiframe task sets under static priority assignment. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*. IEEE Computer Society, Taipei, Taiwan.

Tokuda, H. and Kotera, M. 1988. Scheduler 1-2-3: an interactive schedulability analyzer for real-time systems. In *Proceedings the IEEE International Computer Software and Applications Conference*. IEEE Computer Society, Chicago.