Schedulability of Event-Driven Code Blocks in Real-Time Embedded Systems

Samarjit Chakraborty Thomas Erlebach Simon Künzli Lothar Thiele
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zürich
CH-8092 Zürich, Switzerland
{samarjit,erlebach,kuenzli,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. Tasks in many such systems, like network processors, are either not preemptable or have restrictions on the number of preemptions allowed. All the previous work on the schedulability analysis of such systems either have exponential complexity, or allow unbounded number of preemptions and are usually based on heuristics. In this paper we present the exact necessary and sufficient conditions under EDF, for the schedulability of such a collection of code blocks in a non-preemptive environment, and give efficient algorithms for testing them. We validate our analytical results with experiments and show that the schedulability analysis problem in such systems can be exactly and efficiently solved in practice.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*

General Terms

Algorithms, Performance, Verification

1. INTRODUCTION

Real-Time systems are generally modeled as a collection of independent *tasks*, where each task generates a sequence of *jobs*, each of which is characterized by a *ready-time*, an *execution requirement*, and a *deadline*. The schedulability analysis of such a system is concerned with determining whether it is possible to assign to each job a processor time equal to its execution requirement, between its ready-time and its deadline. In the context of most real-time embedded systems, each such 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. Such blocks of code are usually represented by their control flow graphs on some appropriate level of abstraction, where the vertices have associated deadlines and represent portions of code implementing some functionality, and the edges represent the flow of control. The vertices when triggered by external (or internal) events have to be executed within their associated deadlines. The schedulability analysis of such a collection of control flow graphs answers whether it is possible to execute all the vertices of all graphs within their deadlines, under all possible event triggering sequences.

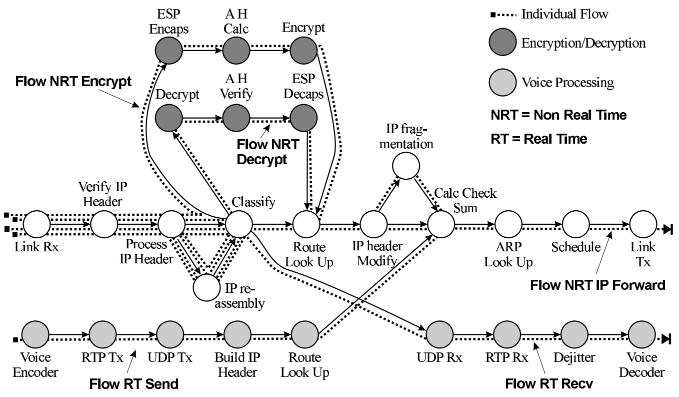

Figure 1: Control flow graph of a code implementing parts of a network packet processor

As an example, Figure 1 shows on a very high level of abstraction, the control flow graph of a code, implementing parts of an embedded network packet processor. The vertices in this graph represent different packet processing functions and get triggered by incoming packets (external events) or by preceding vertices when they complete execution and are ready to forward the packet for further processing. Such a packet processor may have several input ports through which packets flow in, and after being processed they are put out on the appropriate output ports. The code (whose control flow graph is shown in the figure) corresponding to each input port is responsible for handling packets flowing through that port, and all such blocks of code execute concurrently. The schedulability analysis of a collection of such graphs is necessary to determine if all packets belonging to real-time flows (such as voice) can be processed within their respective deadlines.

In this example and also in many other embedded systems scenarios, due to constraints on memory and also due to efficiency reasons the number of preemptions allowed is restricted. This is because of the usually large overhead involved in preemptions. Hence, once a vertex of the control flow graph in the above example starts executing on a processor, it is required to continue till completion before another vertex (probably from a different graph) can be scheduled for execution. The advantages of preemption in such cases are usually offset by the large overhead involved in preempting a process which has only partially processed a packet.

Conditional real-time code. The main difficulty in the schedulability analysis (both for preemptive and non-preemptive cases) of a collection of such control flow graphs lies in the fact that for general directed acyclic graphs, what constitutes a worst case event triggering sequence for an individual graph can not be determined in isolation, due to the presence of conditional branches. To illustrate this, consider our next example.

```
while ($external\ event$) do
    execute code block $B_0$        /* $(e_0, d_0)$ */
    if ($C$) then
        execute code block $B_1$        /* $(e_1, d_1)$ */
    else
        execute code block $B_2$        /* $(e_2, d_2)$ */
    end if
end while
```

In the above code, for each code block B, the tuples (e, d) enclosed within the comments indicate the execution requirement and the deadline of B. Now, 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 behavior does not work in the presence of conditional branches. The alternative, which involves enumerating all possible execution paths in the control flow graph, leads to exponential complexity.

Previous results. There is a large body of work on modeling real-time embedded systems and on answering scheduling theoretic questions arising in these models (see [1] for an overview). Whereas most of the previous work considered independently executing tasks, of late there has been a considerable amount of work on trying to model data and control dependencies between tasks and addressing scheduling issues in such models (see [11] and the references therein). Very recently, a new model called the *recurring real-time task model* was proposed in [2, 4] for modeling codes with conditional branches as shown in the examples above. It generalizes many of the previous models like the sporadic [9], multiframe [10], generalized multiframe [5], and recurring branching [3]. However, the algorithms presented in [2] for the feasibility analysis problem in this model for the preemptive uniprocessor case, have a running time which is exponential in the number of vertices of the task graphs, and the complexity of this problem was undecided. Recently it was proved that this problem is NP-hard [7].

Our results. Relatively little is known about the non-preemptive version of the problem, where tasks are triggered by external events and there exist control dependencies between them as shown with our network packet processor example. All the work in this area (see [8] and the references therein) is based on heuristics and no exact test for schedulability is known till now. Moreover, in view of the recent NP-hardness result for the preemptive recurring real-time task model, the non-preemptive version is very likely to be NP-hard as well.

In this paper we study the non-preemptive version of the recurring real-time task model, for modeling a set of concurrently executing event-driven code blocks with real-time constraints. Our main contributions are that we give an exact necessary and sufficient condition for schedulability under Earliest-Deadline-First (EDF), and show that this condition can be efficiently tested. Towards this, we give a pseudo-polynomial time algorithm and efficient approximation schemes which involve a trade-off between the running time of the algorithms and the quality of the results produced. We validate our analytical results using experiments and show that for all practical purposes the schedulability analysis of a collection of such code blocks can be accurately and efficiently done. 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 behavior of the code blocks. Using techniques described in [2] it is possible to extend our results to incorporate this recurring behavior and we postpone the details of this to a full version of this paper.

Apart from the results that we present here for EDF, it is also possible to derive a sufficient condition for fixed priority scheduling and efficiently test this condition. Due to space restrictions, this result and some other proofs are omitted here and can be found in [6]. In the next section we formally describe the model. Section 3 presents the schedulability test for EDF, following which we describe our approximation schemes in Section 4. Finally, Section 5 describes our experimental results.

2. THE TASK MODEL

A task modeling a block of code is represented by a directed acyclic graph with a unique source and a sink vertex. Associated with each vertex v is its execution requirement $e(v)$ (which can be previously determined, for example 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. Since we consider a non-preemptive environment, once a vertex has started execution it can not be preempted, and continues executing till completion. After it completes, another vertex which has already been triggered, possibly belonging to a different task graph, can be scheduled for execution.

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, and $p(u, v) \geq d(u)$.

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 (representing a conditional branch) 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 $[t_i, t_i + d(v_i)]$.

Task sets and schedulability 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 schedulability 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. As already mentioned before, here we are interested in the non-preemptive uniprocessor version of this problem.

3. SCHEDULABILITY ANALYSIS FOR EDF

An EDF scheduler always selects a ready job with the shortest deadline for execution and is known to be optimal under preemption. In the non-preemptive case EDF is known to be optimal for independently executing jobs if the scheduler is work conserving or non-idle (i.e. if a job is ready then it has to be scheduled if the processor is empty). In this section we derive an exact necessary and sufficient condition for the schedulability of a set of task graphs under EDF. Not surprisingly, we show that for our task model EDF is also an optimal non-preemptive work conserving scheduler.

3.1 Demand-Bound Function ($T.dbf(t)$)

Our schedulability analysis 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. As

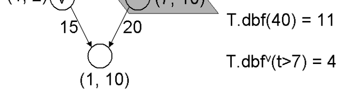

Figure 2: Demand-bound function for task graph T

an example, consider the task graph T shown in Figure 2. For this graph, $T.dbf(2) = 1$ because the vertex having an execution requirement of 1 and deadline 2, can trigger at the beginning of any time interval of length 2 and has an execution requirement of 1 if it has to meet its deadline. Similarly, $T.dbf(20) = 10$ because of a possible triggering of the two shaded vertices in the graph within any interval of length 20.

In addition to $T.dbf(t)$, we denote by $T.dbf^v(t)$, the maximum execution requirement demanded by T within any time interval of length t, due to any triggering sequence ending at the vertex v.

3.2 Conditions for Schedulability

In this section we give a necessary and sufficient condition for the schedulability of a set of task graphs under EDF scheduling. This condition is specified by Algorithm 1, which uses the two functions $T.dbf(t)$ and $T.dbf^v(t)$ introduced in the last subsection.

Algorithm 1 Algorithm for schedulability analysis under EDF

```
Input: Task set $\mathcal{T}$
 1: $decision \leftarrow YES$
 2: for all tasks $T_i \in \mathcal{T}$ and for all vertices $v \in T_i$ and for all $\hat{\tau} \geq 0$ do
 3:     Let $\tilde{\mathcal{T}} \leftarrow \mathcal{T} \backslash \{T_i\}$
 4:     $\mathcal{T}_{dbf=0} \leftarrow \{T \in \tilde{\mathcal{T}} \mid T.dbf(\hat{\tau} + d(v)) = 0\}$
 5:     $e_{max} \leftarrow \max_{v'}\{e(v') \mid v'$ is a vertex of a task $T \in \mathcal{T}_{dbf=0}\}$
 6:     Let $\mathcal{T}_{dbf>0} \leftarrow \{T \in \tilde{\mathcal{T}} \mid T.dbf(\hat{\tau} + d(v)) > 0\}$ and $q \leftarrow |\mathcal{T}_{dbf>0}|$
 7:     $index \leftarrow 0$
 8:     for $p = 1$ to $q$ do
 9:         Let $e'_{max} \leftarrow \max\{e(v') \mid v' \in T_p,\ d(v') > \hat{\tau} + d(v)\}$
10:         if $index = 0$ then
11:             if $e'_{max} > (T_p.dbf(\hat{\tau} + d(v)) + e_{max})$ then
12:                 $e_{max} \leftarrow e'_{max}$
13:                 $index \leftarrow p$
14:             end if
15:         else
16:             if $e'_{max} + T_{index}.dbf(\hat{\tau} + d(v)) > (T_p.dbf(\hat{\tau} + d(v)) + e_{max})$ then
17:                 $e_{max} \leftarrow e'_{max}$
18:                 $index \leftarrow p$
19:             end if
20:         end if
21:     end for
22:     if $index \neq 0$ then
23:         $\hat{\mathcal{T}} \leftarrow \mathcal{T}_{dbf>0} \backslash \{T_{index}\}$
24:     end if
25:     if $\hat{\tau} + d(v) < (T_i.dbf^v(\hat{\tau} + d(v)) + \sum_{T \in \hat{\mathcal{T}}} T.dbf(\hat{\tau} + d(v)) + e_{max})$ then      /* Condition (†) */
26:         $decision \leftarrow NO$
27:     end if
28: end for
29: return $decision$
```

Theorem 1. *A task set $\mathcal{T}$ is schedulable under EDF if and only if Algorithm 1 returns YES.*

Proof. Let v be any vertex of a task graph $T_i \in \mathcal{T}$. The vertex v has an execution requirement of $e(v)$ and a deadline equal to $d(v)$. Let v be triggered at time t and it completes execution at time $t + \delta$.

Let $R^{\leq d}_{\mathcal{T}}[t, t+\tau]$ denote the sum of the execution requirements of the vertices of any task graph $T \in \mathcal{T}$ which have been triggered in the time interval $[t, t+\tau]$ and which have their deadlines less than or equal to d. Let $W^{v,t}(t+\tau)$ $(0 \leq \tau \leq \delta)$ denote the total execution requirement at time

$t+\tau$ that was generated by all the tasks in $\mathcal{T}$, and which must be met by the processor (under EDF scheduling) before the vertex v that was triggered at time t can complete its execution. $W^{v,t}(t+\tau)$ includes the execution requirement $e(v)$ of the vertex v as well. We assume that the processor was idle before time 0.

If we look back in time, let $t-\hat{\tau}$ be the first time before the time instant t when the processor does not have any vertex to execute with deadline less than or equal to $t+d(v)$ (i.e. the deadline of the vertex v). Hence, during the entire interval $[t-\hat{\tau}, t+\delta)$, the processor always has some vertex to execute with deadline less than or equal to $t+d(v)$. $W^{v,t}(t+\tau)$ for any $0 \leq \tau \leq \delta$ is therefore composed of the following: (1) The remaining execution requirement of the vertex that is in execution at time $t-\hat{\tau}$, denoted by $P(t-\hat{\tau})$. By our assumption of $\hat{\tau}$, the deadline of this vertex is greater than $t+d(v)$. (2) The execution requirement generated by the vertices of the task T_i during the time interval $[t-\hat{\tau}, t]$. This includes the vertex v. Clearly, all these vertices have a deadline less than or equal to $t+d(v)$. Therefore, this equals to $R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t]$. (3) The execution requirement generated by vertices with deadlines less than or equal to $t+d(v)$, from all tasks belonging to a set, say $\hat{\mathcal{T}}$, where $\hat{\mathcal{T}} \subseteq \mathcal{T} \backslash \{T_i\}$, during the time interval $[t-\hat{\tau}, t+\tau]$. Therefore, this is equal to $\sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+\tau]$. (4) The execution requirement served by the processor during the time interval $[t-\hat{\tau}, t+\tau]$.

Since we are considering a non-preemptive environment, the vertex which is in execution at the time $t-\hat{\tau}$ has to finish executing before any vertex having a deadline less or equal to $t+d(v)$ can be executed. Therefore, the processor always executes some vertex having a deadline less than or equal to $t+d(v)$ during the interval $[t-\hat{\tau}+P(t-\hat{\tau}), t+\tau]$. Hence,

$$\begin{aligned} W^{v,t}(t+\tau) &= P(t-\hat{\tau}) + R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t] + \\ &\sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+\tau] - (\hat{\tau}+\tau) \quad (1) \end{aligned}$$

Now, note that if there exists a τ $(0 \leq \tau \leq d(v))$ such that $W^{v,t}(t+\tau) = 0$, then the vertex v completes execution on or before its deadline. Substituting $\tau = d(v)$ in Equation (1), we obtain:

$$\begin{aligned} W^{v,t}(t+d(v)) &= P(t-\hat{\tau}) + R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t] + \\ &\sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+d(v)] - \hat{\tau} - d(v) \end{aligned}$$

Following our definition of the *demand-bound functions* (dbf and dbf^v), clearly,

$$\begin{aligned} W^{v,t}(t+d(v)) &\leq P(t-\hat{\tau}) + T_i.dbf^v(\hat{\tau}+d(v)) + \\ &\sum_{T\in\hat{\mathcal{T}}} T.dbf(\hat{\tau}+d(v)) - \hat{\tau} - d(v) \quad (2) \end{aligned}$$

To compute an upper bound on $W^{v,t}(t+d(v))$ we would like to maximize the right hand side of the above inequality (2). For this, note that if a vertex v' of a task T contributes to the term $P(t-\hat{\tau})$, then T can not belong to the set $\hat{\mathcal{T}}$. Following this constraint, for any task T_i and any vertex $v \in T_i$, Algorithm 1 computes $P(t-\hat{\tau}) = e_{max}$ and the task set $\hat{\mathcal{T}}$ which maximizes the right hand side of Inequality (2). Therefore, if the algorithm returns YES, then we have (from Condition (†) of the algorithm),

$$W^{v,t}(t+d(v)) \leq \hat{\tau} + d(v) - (\hat{\tau}+d(v)) = 0$$

Hence, there exists a $\tau \leq t+d(v)$ such that $W^{v,t}(t+d(v)) \leq 0$ and therefore the vertex v completes execution before its deadline.

The proof of necessity is based on similar arguments. The interested reader is referred to [6]. □

Note that Step 2 in Algorithm 1 involves a loop over all possible values of $\hat{\tau} \geq 0$. However, it suffices to consider only a finite set of $\hat{\tau}$s and this is explained at the end of Section 4.2. The optimality of EDF in this model follows from the fact that the proof of necessity makes no assumptions about the scheduling discipline.

4. APPROXIMATE SCHEDULABILITY ANALYSIS

The demand bound function of a task graph can clearly be computed by enumerating all possible paths in the graph and computing the execution requirement and deadline corresponding to each path. In the worst case, since the number of such paths can be exponential in the number of vertices in the graph, this procedure will incur an exponential running time. It can be shown by a reduction from the knapsack problem that computing $T.dbf(t)$ for a task graph T is NP-hard, implying that our algorithm for schedulability analysis can have a worst case running time which is exponential in the number of vertices in any task graph.

In this section we first show that $T.dbf(t)$ for any task graph can be efficiently approximated. Towards this we give a fully-polynomial time approximation scheme (FPTAS) for computing $T.dbf(t)$. Using this result we then give approximate decision algorithms for schedulability analysis.

4.1 Approximating the Demand-Bound Function

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 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 $p(v_i, v_j)$.

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 $t_{i,e}^i$ 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 2 correctly computes $T.dbf(t)$, and has a running time of $O(n^3E)$.

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

Algorithm 2 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 $v_{i+1}$
      $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + \\ d(v_{i+1}) \mid j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
      $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
    end for
  end for
  $T.dbf(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$
```

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 Algorithm 2 with the new $e'(v_i)$s and the graph T_t. Let V be the set of vertices (with the scaled execution requirements) that result in the computation of $T.dbf(t)$ in this algorithm. We claim that the summation of the original (unscaled) execution requirements of these vertices is greater than or equal to $(1 - \varepsilon)$ times the actual demand-bound function for the task graph for this value of t. Further, since this algorithm now runs in time $O(n^4/\varepsilon)$, (with the scaled execution requirements), it is an FPTAS for computing $T.dbf(t)$. We denote this approximate value of $T.dbf(t)$ computed by this algorithm by $T.dbf'(t)$.

4.2 Approximate Decision Algorithms

Our approximate decision algorithms use the approximate demand-bound function ($T.dbf'(t)$) introduced in the last subsection, instead of the exact values $T.dbf(t)$, in Algorithm 1. The decision algorithms are parameterized by $0 < \varepsilon \leq 1$, where the number of wrong answers and the running time depend on the value of ε chosen. For smaller values of ε the percentage of possible wrong answers decrease, but at the expense of the running time of the algorithm.

Using the same notation as in the last subsection, note that for all possible values of $t \geq 0$, 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 $O(|\mathcal{T}| n^3/\varepsilon)$.

It follows that in Step 2 of Algorithm 1 it is sufficient to run the loop only for $O(|\mathcal{T}| n^3/\varepsilon)$ values of $\hat{\tau}$, since the value of $\sum_{T \in \mathcal{T}} T.dbf'(\hat{\tau})$ can change at most this number of times. Therefore, the loop in Step 2 executes for a total of $O(|\mathcal{T}|^2 n^4/\varepsilon)$ times.

For each task $T \in \mathcal{T}$, computing the $t_{n,e}$ values for each of its subgraphs T_t, using Algorithm 2 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. The Step 25 in the algorithm dominates the running time among all the steps inside the loop (Steps 3-27), and requires a computation of $T_i.dbf'^v(t) + \sum_{T \in \hat{\mathcal{T}}} T.dbf'(t) + e_{max}$ where $t = \hat{\tau} + d(v)$. To compute this, note that 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 this table. Therefore, this requires $O(n^2 \varepsilon^{-1} \log n)$ time. The exactly same time is for computing $T.dbf'^v(t)$. Hence, computing the value of $T_i.dbf'^v(t) + \sum_{T \in \hat{\mathcal{T}}} T.dbf'(t) + e_{max}$ for $t = \hat{\tau} + d(v)$ in Step 25 requires a total of $O(|\mathcal{T}| n^2 \varepsilon^{-1} \log n)$ time. Therefore, the total run time of Algorithm 1 using the approximate demand-bound functions is $O(|\mathcal{T}|^3 n^6 \varepsilon^{-2} \log n)$.

Since $T.dbf'(t) \leq T.dbf(t)$ for any $t \geq 0$, this algorithm is overly *optimistic*, in the sense that for certain task sets which are not schedulable, the algorithm might still return a YES. However, for task sets where some vertices might miss their deadlines by a large time lengths, the algorithm always returns a NO. So the algorithm errs only for task sets where some vertices might miss their deadlines by "small" amounts of time and this can be parametrized by ε. Therefore, any $0 < \varepsilon \leq 1$ 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 $1/\varepsilon$, and therefore this gives a fully polynomial-time approximate decision scheme for approximate feasibility testing.

It may be shown that for any $t \geq 0$, $T.dbf'(t) + \varepsilon E_t \geq T.dbf(t)$. Hence, using $(T.dbf'(t) + \varepsilon E_t)$ instead of $T.dbf(t)$ in Algorithm 1 results in a *pessimistic* algorithm with analogous properties as in our optimistic algorithm.

A Pseudo-Polynomial Time Algorithm. Lastly, it may be noted that Algorithm 1 along with the pseudo-polynomial time algorithm for computing the demand-bound function of a task graph also implies a pseudo-polynomial time algorithm for schedulability 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 $t_{max} = \max_{T \in \mathcal{T}} t^T_{max}$. Clearly, it is sufficient to test the Condition (†) in Algorithm 1 only for $\hat{\tau} = 1, \ldots, t_{max}$. Both $T.dbf^v(\hat{\tau} + d(v))$ and $T.dbf^v(\hat{\tau} + d(v))$ in the Step 25 of the algorithm for any $\hat{\tau}$ can be determined in pseudo-polynomial time by Algorithm 2 and clearly, t_{max} is pseudo-polynomially bounded, implying a pseudo-polynomial algorithm for schedulability analysis.

5. EXPERIMENTAL RESULTS

In spite of the theoretical guarantees in our algorithms the experiments reported here are interesting because of two reasons. Firstly, many approximations schemes are exceedingly difficult to implement and in practice might have running times which are comparable or even worse than the equivalent simpler exponential time algorithms, for practical input instances. Secondly, the parameter ε in our algorithms represents a trade-off between the quality of the results obtained and the running time. Hence, it is interesting to identify a suitable value of ε for any realistic input instance.

We have implemented the pseudo-polynomial exact algo-

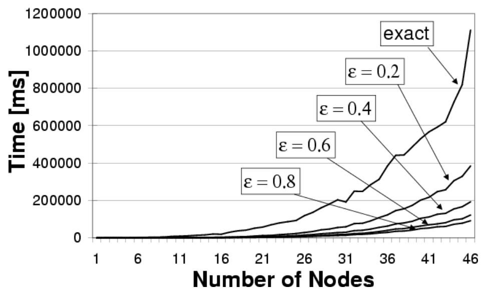

Figure 3: Running time versus the number of vertices in the task graphs when $E = 500$

rithm for feasibility analysis, and also the approximation scheme. For our experiments, we have randomly generated synthetic task graphs, using two parameters. The first is the maximum execution requirement associated with any vertex of the graph, E, which effects the running time of the pseudo-polynomial time algorithm and the quality of the results generated by the approximation scheme. We call the second parameter 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 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 $i = 1, \ldots, j-1$.

Figure 3 shows the running time of the exact pseudo-polynomial algorithm and the approximation scheme for four different values of ε on a task set consisting of three graphs, when the number of vertices in each of these graphs is gradually increased. The maximum execution requirement (E) associated with any vertex was set to 500. The connectivity factor in all the graphs was set to 0.4; this generates realistic control flow graphs [6]. The CPU time was measured on a moderately loaded Sunblade 1000 running SunOS 5.8 with 750 MHz CPU and 2 GB RAM. All the algorithms were implemented in Java.

Note that for any set of task graphs, the optimal choice of ε depends on the maximum execution requirement E, associated with any vertex. For instance, in the above example if $E = 100$ then the performance of the exact algorithm is better than the approximation scheme with $\varepsilon = 0.2$. To give an example of the values of E that might occur in practice, typical execution requirements associated with the vertices in Figure 1 (for the voice flow) are 300, 190, 640 and 500 μsecs for UDP Tx, Build IP Header, Route Look Up, and ARP Look Up respectively when these are implemented on a DSP like TMS320C620x. These values are 270, 130, 420 and 330 μsecs respectively when implemented on a RISC processor like ARM9TDMI.

Figure 4 shows the exact value of the demand-bound function $T.dbf(t)$ computed by the pseudo-polynomial algorithm, and its upper and lower bounds ($T.dbf'(t)+\varepsilon E$ and $T.dbf'(e)$ respectively) computed by the approximation scheme. It should be noted that the value of $T.dbf'(t)$ for all values of t is almost equal to $T.dbf(t)$, and this is better than the worst case theoretical bound. The values shown in this graph are for a task graph with $E = 1000$, $\varepsilon = 0.6$ and the number of vertices in the task graph and the connectivity being equal to 30 and 0.4.

Lastly, the percentage of wrong answers returned by the approximation scheme for different values of ε are given by $(0.2, 5\%)$, $(0.4, 6\%)$, $(0.6, 11\%)$ and $(0.8, 15\%)$. Each of the tuples indicate the value of ε and the percentage of wrong answers. For this we have used 100 task sets, each consist-

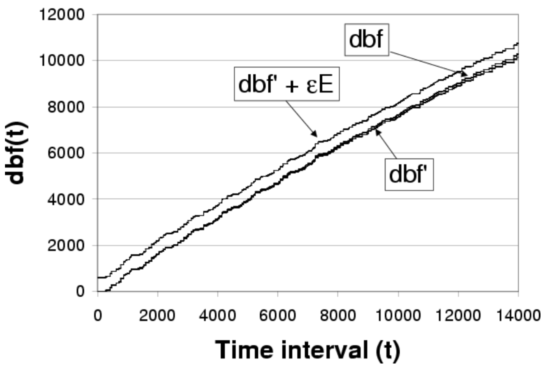

Figure 4: The demand-bound function ($T.dbf(t)$) and the upper and lower bounds on its approximation

ing of three task graphs with 30 vertices in each graph and having a connectivity factor of 0.4. The maximum execution requirement of any vertex was set to 100. All the task sets considered here either almost fully load the processor, or when not schedulable, the vertices miss their deadlines only by small amounts of time, and therefore these represent the difficult cases for our approximation algorithms.

6. CONCLUSION

In this paper we have considered the schedulability analysis of a collection of event-driven real-time code blocks. Although this problem is very likely to be computationally difficult (NP-hard) we have shown that for all practical purposes it can be efficiently solved. For fixed priority schedulers only a sufficient condition for schedulability is known [6]. Here it would be interesting to come up with a test which is both necessary and sufficient.

7. REFERENCES

[1] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling of embedded real-time systems. *IEEE Design and Test of Computers*, pages 71–82, January-March 1998.

[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. IEEE RTSS*, pages 114–122, 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. Chakraborty, T. Erlebach, S. Künzli, and L. Thiele. Schedulability of event-driven code blocks in real-time embedded systems. Technical Report TIK 130, ETH Zürich, 2002.

[7] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. Technical Report TIK 107, ETH Zürich, 2001.

[8] P. Eles *et al.* Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proc. Design, Automation and Test in Europe*, 1998.

[9] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, LCS, MIT, 1983.

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

[11] P. Pop, P. Eles, and Z. Peng. Schedulability analysis for systems with data and control dependencies. In *Proc. 12th Euromicro Conference on Real-Time Systems*, 2000.