Task Scheduling with RT Constraints

M. Di Natale
Dip. Ing. dell'Informazione
Università degli Studi di Pisa
Pisa, Italy
marco@sssup.it

A. Sangiovanni-Vincentelli
Department of EECS
Univ. of California at Berkeley
Berkeley, CA
alberto@eecs.berkeley.edu

F. Balarin
Cadence Berkeley Labs.
Addisson St. Berkeley, CA
felice@cadence.com

ABSTRACT

This paper addresses the problem of scheduling reactive real-time transactions (task groups) implementing a network of extended Finite State Machines communicating asynchronously. Task instances are activated in response to internal and/or external events. The objective is avoiding the loss of events exchanged by the tasks. This scheduling problem has many similarities with the conventional formulation of real-time problems and yet it differs enough to justify a rethinking of the assumptions and techniques used to solve the problem. Our iterative solution targets fixed priority systems and offers a priority assignment scheme together with a sufficiently tight worst-case analysis.

1. INTRODUCTION

This paper addresses a scheduling problem of great importance in real-time embedded systems, the scheduling of reactive (or event-based) systems where all tasks are executed on a single processor. Traditional real-time research has developed models for schedulability analysis together with tools and methodologies for the formal specification of real-time constraints, the design and the production of code. While many different definitions have been proposed for the scheduling problem, only a few have been adopted to analyze the code produced by commercial CASE tools such as the Objectime toolset ([7]). In other cases, such as the one presented here, the problem reveals a number of issues that are still new to the real-time research community and do not perfectly match any of the existing scheduling algorithms.

The importance of this problem arises from the study of embedded systems and from the availability of a number of tools for the automatic production of embedded code starting from a asynchronously communicating finite-state machine (Codesign Finite State Machines) description. In this model, tasks represent finite state objects that are executed asynchronously in response to external or internal events and communicate through the production and consumption of events. Furthermore, they do not have explicit deadlines (at least not in the traditional form). Rate constraints (a minimum inter-arrival time) apply to external events. The scheduling problem consists of making sure that no critical event (internal or external) is ever dropped by the system, that is, every critical event is always consumed before another event of the same kind is produced again. The optimal solution to this scheduling problem can be found in exponential time [4]. In this paper, we are interested in some worst-case (pessimistic) analysis that can be run quickly and still overcome some of the problems found in earlier solutions.

2. THE CO-DESIGN FINITE STATE MACHINE (CFSM) MODEL

We assume the system is modeled as a network of communicating CFSMs (Codesign Finite State Machines). Each CFSM is an extended FSM, supporting data handling and asynchronous communication. In particular, a CFSM has a finite state machine part, a data computation part, a locally synchronous behavior and a globally asynchronous behavior.

CFSM communicate through signals. Each signal can be a control signal, a data signal or both and can be associated with a Boolean control variable or an enumerated or integer sub-range variable. The event represents the presence of the signal and may be produced by a sender CFSM. Setting the event buffer to 1 sends an event. It may be detected and consumed by a receiving CFSM. Setting the buffer to 0 consumes the event.

A set of values for the inputs of a CFSM is termed an *input assignment.* The input assignment is consumed as a result of the execution of the CFSM. During operation, the scheduler schedules CFSMs whenever they have input events and according to the appropriate priorities. During its execution, a CFSM reads its inputs, performs a computation, possibly changes state and writes its outputs. For a formal definition of CFSM and their behavior please refer to [4].

3. THE PROBLEM

In a terminology more familiar to real-time scheduling researchers, each CFSM corresponds to a task and the network of CFSMs corresponds to a set of *transactions*, a transaction being a set of related tasks. Each task in a transaction can be activated by external or internal events. Tasks produce events at the completion of their execution and consume all the incoming events that are active at the time they start running. There exists a minimum inter-arrival time between two occurrences of the same external event. Each internal event is sent on a single position buffer. If an event is overwritten by a new one, it is said to be "dropped".

Formally, the system is a set of transactions $TR_1, TR_2, \dots TR_m$. Each transaction TR_k is a graph $TR_k = (T_k, E_k)$ where $T_k = \{\tau_{ik}\}$ is the set of nodes representing the tasks. Each task is identified by its index, τ_i being the task of index i. When necessary, the task index will be followed by the index of transaction it belongs to (τ_{ik} is the $i-th$ task belonging to transaction TR_k).
Some tasks, called *external tasks*, represent the environment and need not be scheduled. These tasks belong to a subset $U \in T$ of all tasks. $E_k = \{e_{k(i,j)}\}$ is the set of events. Events can be external or internal: external events are produced by external tasks. Event $e_{(i,j)}$ is produced by task τ_i and activates task τ_j. Once more, the transaction index can appear together with the event indices, $e_{k(i,j)}$ meaning the

event belongs to transaction TR_k. Each internal task has an associated worst-case computation time c_i. By definition $c_i = 0$ for all external tasks. External events have associated a minimum inter-arrival time P_i. All events are *critical*, i.e., they should never be dropped.

Although this is not a requirement, we assume a fixed-priority-scheduling scheme with preemption. Therefore each task and event will have an associated priority. By p_i we mean the priority of task τ_i and by p_{ij} the priority of event e_{ij}. An event is made *safe* by a priority assignment if the priority assignment itself is sufficient to make sure the event will never be dropped. An event is *guaranteed* if, given a priority assignment and considering the timing attributes of tasks, the timing analysis can guarantee that the event could never possibly be dropped.

4. STATE OF THE ART

While conventional real-time research focuses on problems where the scheduling algorithm is obtained by reasoning on the timing constraints and the buffer requirements are only considered (not very often) at a later stage, the research carried out in [2], where this problem was discussed first, had a different perspective: priorities were assigned trying to prevent the loss of internal events. Only in a second time the possible loss of external events (the ones carrying the timing constraints) was considered. Priorities were assigned on the basis of the task relationships and not on timing constraints.

There is an advantage and a possible disadvantage to this technique. The advantage is that internal events need not be guaranteed, therefore only external events need be analyzed. The disadvantage is the execution of multiple instances of the same task in response to a single external event. This additional load can prove extremely undesirable when multiple transactions are active in the system.

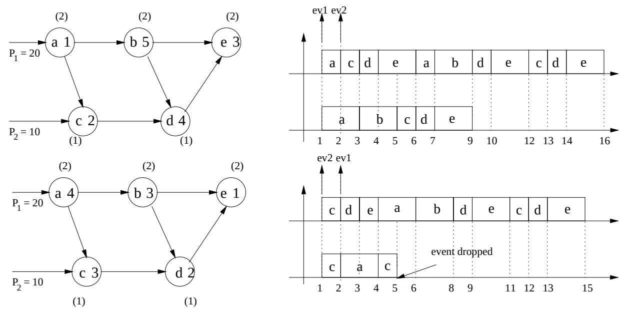


Figure 1: Completion times with different priority assignments

The effect of multiple activations is very clear if we consider the example taken from [2] and shown in figure 1 which is a simplified version of a shock absorber controller. Tasks are shown as circles and identified by letters. The numbers inside the circles are the priorities. The numbers between parenthesis are the worst case execution times. If priorities are assigned as proposed in [2] (top graph, the priorities prevent the loss of internal events), the scheduling sequence is 15 time units long, including three instances of task d. If priorities are assigned as shown in the lower half (tasks producing events always have priorities higher than tasks consuming them), the schedule length reduces to only 8 time units and each task is executed just once. On the other hand, if the arrival pattern were the one on the lower right half, the second priority assignment would not be feasible (event $e_{(c,d)}$ is dropped). A more recent work ([3]) from the same author removes most limitations of the previous one. The author proposes a new priority assignment scheme that avoids most of the undesirable (multiple) task executions. The algorithm produces an optimal solution, although only in a selected subspace of all possible priority assignments. Of course there is no guarantee that a better solution does not exist outside the selected subspace. On the other hand, this limitation could be an acceptable compromise in order to obtain a simpler (although pessimistic) procedure for predicting the schedulability of all events.

Solving the scheduling problem requires the assignment of priorities to tasks and/or events and a fast and not too pessimistic analysis procedure. The priority assignment must satisfy two contrasting requirement.

- Priorities should minimize the effect of multiple executions of the same task for one single event (to let tasks consume at once all the events that could possibly trigger their execution.)
- The priority assignment should minimize the possible loss of internal events making safe all events that could not be guaranteed otherwise. [2]

5. PRIORITY ASSIGNMENT AND ANALYSIS

We need the following additional definitions before presenting a priority assignment rule.

DEFINITION 1. *The level of a path in the transaction graph is the lowest priority of all tasks encountered in the path.*

DEFINITION 2. *n paths are $j - level$ disjoint if their level is lower than or equal to j and each contains at least a task with a priority lower than or equal to j which is not shared with the others.*

Then, we define the set of *elementary transactions*. For each external task τ_i belonging to transaction TR_k we build a new transaction TR_{ki} containing the graph of tasks and events originating from τ_i. In practice, the new transaction can be built adding all tasks and events encountered in a search of the graph starting from $\tau_i \in U$. At this point we have a set of elementary transactions activated by a single event, each with its specified rate. Tasks and events can appear in more than one elementary transaction. From now on, by transaction we mean a regular transaction, having possibly more than one external task. When elementary transactions will be used, it will be explicitly stated.

We first give a very simple priority assignment rule that minimizes the multiple activation of internal tasks. This rule is borrowed from a number of papers ([5] probably the first) although here it serves a completely different purpose. Assign the lowest priority level (1) to all tasks that do not generate events. For all other tasks τ_i set their priority p_i to

$$p_i = 1 + \max\{p_j | e_{(i,j)} \in E\}.$$

If priorities are assigned according to this rule (we call it *flow descendent*) each task is activated only once for each external event. The proof (trivial) will be skipped.

THEOREM 1. *An event $e_{(i,j)}$ is* safe *if (1) all paths containing it and going from an external event to task τ_j have level lower than p_j and (2) there exists at most one path containing $e_{(i,j)}$ from any internal task to τ_j with a level higher than or equal to p_j.*

Please note, as pointed out in [2] for an event $e_{(i,j)}$ to be safe, it suffices that $p_j > p_i$, (in contrast with the previous priority assignment rule).

THEOREM 2. *If event $e_{(i,j)}$ belongs to more than one elementary transaction, condition (1) of Theorem 1 is not only sufficient but necessary for the event to be* guaranteed.

DEFINITION 3. *(from [3]) A task τ_i is a* merge point *if it has more than one external predecessor, while some of its immediate predecessors has exactly one.*

We can generalize this definition as follows:

DEFINITION 4. *A task τ_i is a* merge point *for task τ_j if there is more than one path from τ_j to τ_i while there is exactly one from some of τ_i's immediate predecessors to τ_j.*

THEOREM 3. *In case task τ_i is a* merge point *for some task τ_l, necessary condition for event $e_{(k,j)}$ (where $k = i$ or task τ_k can be reached from τ_i) to be* guaranteed *is either all paths from τ_l to τ_i have a priority higher than p_i (not considering τ_i itself) or p_j is greater than the priority level of all paths from τ_l to τ_k having a priority lower than or equal to p_i.*

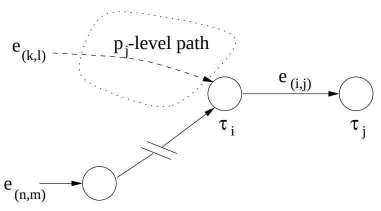


Figure 2: Necessary and sufficient condition for guaranteeing events with multiple external predecessors

The subspace Π of all possible priority assignments searched in [3] for an optimal solution is defined as follows (Proposition 1):

for every $(\tau_i, \tau_j) \in T \times T$:

- $p_i > p_j$ if $e_{(i,j)} \in E$ and τ_i is not a merge point,
- $p_i < p_j$ if τ_i is a merge point, $i \neq j$,
and τ_i is a predecessor of τ_j ($\tau_i \in \mathrm{pred}(\tau_j)$.)

It is possible to show that any priority assignment $P \in \Pi$ satisfies the conditions of Theorem 2 and 3.

Although the set of priority assignments Π satisfies all the necessary requirements, it could be too restrictive in some cases. For example, figure 3 shows a very special case where by assigning priorities according to proposition 1 (left side) the merge points (tasks τ_5 to τ_{10}) have a priority lower than task τ_{11}. As a consequence (see the lower side of the figure and the following Theorem 4) tasks τ_{11} to τ_{13} are executed six times for each activation of the external tasks. On the contrary, if priorities are assigned as shown in the right side tasks τ_{11} to τ_{13} are executed only once for each activation of an external node.

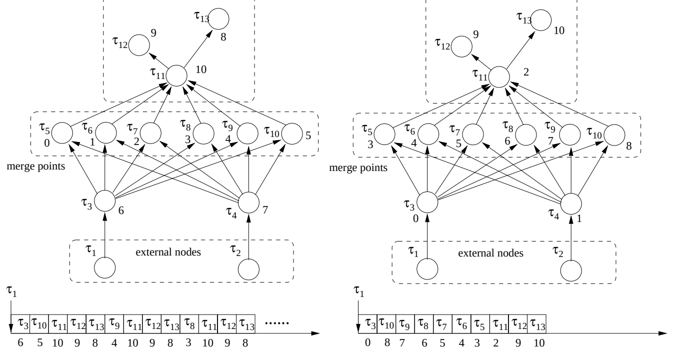


Figure 3: An optimal solution found outside the priority assignments of proposition 1

THEOREM 4. *The number of times task τ_j can be possibly activated by the execution of (internal or external) task τ_i is given by $\lambda + n_d$ where λ is 1 if there exists a path of level higher than p_j from τ_i to τ_j, 0 otherwise and n_d is given by $n_d = \mu_{i,j1} + \mu_{i,j2} + \ldots + \mu_{i,jn}$ where $\tau_{j1}, \tau_{j2}, \ldots \tau_{jn}$ are the tasks with a priority lower than or equal to p_j encountered first when traversing backwards the paths between τ_i and τ_j and $\mu_{i,jk}$ is the number of times task τ_{jk} is executed because of the activation of task τ_i.*

It remains to find an algorithm to decide if an event (internal or external) can be guaranteed on the basis of tasks' priorities and the rates of external events. Assume we have a priority assignment that does not prevent events from being guaranteed (See Theorems 2 and 3). Some of the internal events will be *safe* as a result of the priority assignment. In order to guarantee all the other (internal and external) events we use a standard (pessimistic) timing analysis (similar to the one proposed in e.g., [2] and [8]). We define $\delta(i)$ as the additional load caused by the execution of task τ_i and $\delta(i,k)$ the fraction of $\delta(i)$ when we limit the execution to those tasks having a priority greater than or equal to k. The first necessary condition for guaranteeing internal and external events is the following:

LEMMA 1. *Any event $e_{(i,j)}$ that is not* safe *is dropped if task τ_j starts executing more than P_k time units after the arrival of the external task τ_k that triggered the transaction.*

In fact, being $e_{(i,j)}$ not safe, it belongs to a path with a level higher than p_j originating from some external task τ_k. P_k time units after the arrival of the event that activated the transaction, another external event arrives and triggers the execution of all tasks in the path, including τ_i.

Let us focus on the elementary transaction TR_{kq} triggered by the external task τ_{kq}. We need to analyze all the events that can be guaranteed and are not safe. By Theorems 3 and 4, these events cannot belong to more than one elementary transaction. Objective of the analysis is calculating the worst-case time interval Δ occurring between the arrival of the external event triggering the transaction and the activation of task τ_{kj}. If the external event arrives in t=0, this interval corresponds to a p_j-level busy period, starting in t=0. The p_j-level busy period contains instances of tasks belonging to transaction TR_k as well as tasks belonging to other transactions.

The contribution of (elementary) transaction TR_k can be computed as follows. At time $t = 0$ some internal task having a priority lower than p_j can terminate its execution and enable tasks at priority p_j or higher. In the worst case such event can cause the execution of the following load

$$\max\{\delta(k, p_j) | k \in TR_k, p_k < p_j\}.$$

In the time interval Δ the external event triggering transaction T_k arrives and causes an additional workload at priority p_j. Since we are evaluating the p_j-level busy period delaying the execution of τ_j we should not consider the contribution of τ_j and those tasks activated solely by it.

$$\left\lceil \frac{\Delta}{P_k} \right\rceil \delta'(q, j) | \tau_q \in U_k.$$

External (elementary) transactions contribute to the busy period with an additional load

$$\max\{\delta(k, p_j) | k \in TR_l, l \neq k, p_k < p_j\} + \left\lceil \frac{\Delta}{P_r} \right\rceil \delta(r, p_j) | \tau_r \in U; \tau_r \in TR_l.$$

where $\delta(r, j)$ is the load at priority j or higher generated by the arrival of the external event τ_r in transaction TR_l. The $p_j - level$ load generated by all transaction can be computed adding up all the contributions from external events. Since only one task having a priority lower than p_j can be active at $t = 0$, the final (recursive) formula is

$$\Delta = \max\{\delta(k, j) | p_k < j\} + \sum_{TR_l} \left\lceil \frac{\Delta}{P_r, d} \right\rceil \delta(r, p_j) \mid \tau_r \in U; \tau_r \in TR_l. \qquad (1)$$

If $\Delta < P_k$ we can be sure that the first event instance in a p_j level busy period starting at $t = 0$ will not be dropped. Unfortunately the test gives no guarantees on the following instances. A sufficient test that allows to guarantee all event instances can be easily obtained from the previous one by making sure that any p_j level busy period starting at $t = 0$ (including task τ_j and those activated by it) terminates before the next instance of the external event that triggers $e_{(i,j)}$.

In order to perform timing analysis, we need to determine the amount of computation at any priority level (or higher) generated by the execution of any task or external event. Such analysis resembles the one in [2], although a few differences are worth pointing out. The algorithm is derived from Theorem 4, which states how many times a task is activated because of the completion of another task.

We use a matrix $\mathbf{A}$ to keep track of the number of times each task activates the others. Each term of the matrix $\mathbf{A}$ is a binary term $A_{i,j} = (a, d)$ where a is the number of times (0 or 1) task τ_j is executed as a result of all possible $(j + 1)$ (or higher) level paths from τ_i and d is the number of times τ_j is executed as a result of all disjoint j (or lower) level paths from τ_i.

We define the following sum operator between two terms of the matrix $\mathbf{A}$:

if $A_{i,j} = (a, b)$ and $A_{k,l} = (c, d)$ then $A_{i,j} + A_{k,l} = (e, f)$ where

$$e = \begin{cases} 1 & \text{if } a = 1 \text{ or } c = 1 \\ 0 & \text{elsewhere} \end{cases}$$

$$f = b + d$$

We also define the following unary operator DP on the terms of $\mathbf{A}$:

$$\text{if } A_{i,j} = (a, b) \text{ then } DP(A_{i,j}) = (0, a + b).$$

The procedure to compute matrix $\mathbf{A}$ is the following. Start with

$$A(i, j) = \begin{cases} (1, 0) & \text{if } i = j, \\ A(i, j) = (0, 0) & \text{elsewhere.} \end{cases}$$

Visit all tasks in topological order. When visiting task τ_i do a backwards traversal of the graph stopping when a task with a priority lower than or equal to p_i is encountered.
When task τ_j is encountered during the traversal set $A(i, j)$ to

$$A(i, j) = \begin{cases} (1, 0) + A(i, j) & \text{if } p_j > p_i, \\ \forall l A(i, l) = A(i, l) + DP(A_{j,l}) & \text{if } p_j \leq p_i. \end{cases}$$

This procedure allows to compute not only the load inferred by the execution of τ_i but also the load $\delta(i, k)$ for any priority level k. It suffices to consider only those links that originate or lead to a task having priority higher than or equal to k when doing the backwards traversal of the graph. Note that the j-th column of A shows the tasks activated by τ_j with their multiplicity. If we multiply the i-th row of A by c_i, the worst-case computation time of τ_i, then the sum of all elements in the j-th column of A shows the worst-case execution time of all tasks activated in response to the execution of τ_j.

Figure 5 shows one possible matrix computed for the sample graph shown in [2] when priorities are assigned as shown inside the circles. The elements are weighted by the computation times. *From now on, the matrix A will be represented with its rows weighted by the computation times.* In assigning priorities to tasks, our first objective will be to find a set of tasks that, if their priority is lowered, will make safe all the events that cannot be guaranteed because of Theorem 2 and Theorem 3. Let E_{ng} be this event set. Since there can be many of these task sets we must use some metrics in order to choose one of them. First, let us see how these task sets can be found by working on our sample graph. If priorities are set as in the bottom half of figure 1, the internal events that cannot be guaranteed are $e_{(2,4)}$ (because of Theorem 3) and $e_{(4,3)}$ (because of Theorem 3 and Theorem 4). To guarantee events $e_{(2,4)}, e_{(4,3)}$ it is necessary that the level of all paths containing them and going from the external tasks τ_6 and τ_7

to the consumer tasks τ_4 and τ_3 is lower than, respectively, p_4 and p_3 (Theorem 1).
This means we need to take an event on each of these paths and change the priority of the task producing it to, respectively, $p_4 - 1$ or $p_3 - 1$. In the general case, if we consider a set of internal events that cannot be guaranteed and the graph consisting of all the paths from the external tasks to the tasks consuming them (with a level higher than or equal to the priority of the consumer tasks), we need to find a cut separating the external tasks from the others (4).

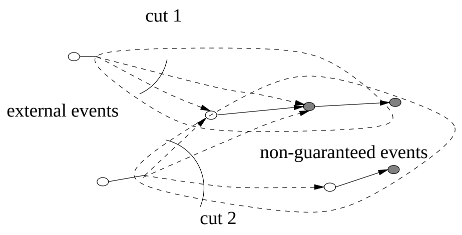


Figure 4: Paths from the external tasks to the internal events that are not guaranteed

The tasks generating the events on the cut are the ones that need to be modified. The schedulability of the events in E_{ng} (the tasks consuming the events) *depends* on these tasks. Now we have all the tools for constructing two iterative methods that try to assign priorities to task in order to guarantee all internal and external events. The purpose of the algorithms is to strike a balance between a policy where all priorities are assigned according to the *flow-descendent* rule (minimizing the number of activations) and a policy where priorities are assigned in order to make *safe* all internal events [2].

6. PRIORITY ASSIGNMENT WITH RESTRICTED PRIORITY LEVELS

First, we propose an iterative algorithm that moves from a priority assignment of the first kind towards a priority assignment of the second kind, making *safe* (by modifying priorities) at each step those tasks that cannot be guaranteed on the basis of their timing attributes. The algorithm begins by assigning priorities according to the *flow-descendent* rule, then enters a cycle. At each iteration the following steps are performed: considering the transactions one at a time, for each transaction

a) find the events that cannot be guaranteed on the basis of Theorem 3 and Theorem 4 or because of current task priorities (analysing the worst case start time of the consumer tasks). This set is empty in the first iteration.

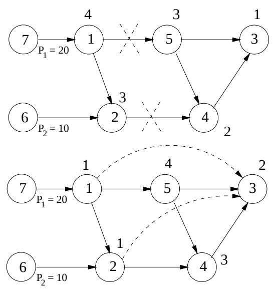
b) find a set of tasks and change their priorities in order to make *safe* all the events that could not be guaranteed in the previous step. The set should be chosen in order to ease the schedulability of external events (or reducing the load generated by their arrival).

c) assign a new priority to all other tasks according to the *flow-descendent* rule.

d) Repeat the analysis; exit if all events can be guaranteed (success) or an external event cannot be guaranteed (failure).

We know how to perform step **a**, **c** and **d** for each iteration. Step **b**, which is crucial for the good functioning of the algorithm, has been described in the previous section. Its objective is to find a set of tasks generating the events on a suitable cut. Since there are many possible cuts we need some way to choose one of them.

One possibility is to perform the changes for every possible cut and pick the one that results in the lowest margin left for guaranteeing the external events. When it is impossible to perform an exhaustive search (because of the large number of possible cuts) an heuristic should be chosen. For example, it could be possible to select the cut spanning the lowest number of events (which is an instance of a *bipartite graph matching* problem.) In our example, if we choose the cut $(2,4)(1,5)$ the priorities that are chaged are $p_2 = \min\{p_3 - 1, p_4 - 1\}$ and $p_1 = \min\{p_3 - 1, p_4 - 1\}$. If we consider these additional constraints and we assign the other priorities according to the flow descendent rule, we get the priority assignment and the activation matrix of figure 5.



1	2 0						2 0
2	0 1	1 0				1 0	0 1
3	0 4	0 2	2 0	2 0	2 0	0 2	0 4
4	0 2	0 1		1 0	1 0	0 1	0 2
5	0 2				2 0		0 2
6							
7							
						6	11

Figure 5: Evaluation of the cut

Since there is no task with a priority lower than 2, external events $e_{(7,1)}$ and $e_{(6,2)}$ will have a waiting time of 19 and 15 time units respectively (by evaluating formula 1). If the cut $(2,4)(5,4)(5,3)$ was chosen, the external event $e_{(7,1)}$ could not be guaranteed (waiting time > 20.)

It is easy to show how the procedure outlined in the algorithm is guaranteed to terminate after a finite number of steps.
Although the algorithm can be implemented as described, it performs quite well after a single iteration (see Section 9). In this case, one of its advantages is the need for a limited number of priority levels and, therefore, a simpler run-time support mechanism. A second algorithm will prove to be much better (optimal when all cuts are considered) in case a wider range of priorities can be used.

7. PRIORITY ASSIGNMENT WITH WIDER PRIORITY RANGE

The other algorithm we present in this paper is a straightforward extension of the one presented in [3]. The first step consists in finding the events E_{ng} that cannot be guaranteed (Theorems 2 and 3) and making them safe by finding a suitable cut in the event graph and lowering the priority of the tasks generating events on the cut.

If $T^c = \{\tau_1^c, \tau_2^c, \ldots, \tau_n^c\}$ is the set of tasks generating the events on the cut, then each task τ_i^c must have a priority lower than the priority of those tasks $\{\tau_1^{ci}, \tau_2^{ci}, \ldots, \tau_n^{ci}\}$ consuming the events in E_{ng} generated because of its execution. We say tasks $\{\tau_1^{ci}, \tau_2^{ci}, \ldots, \tau_n^{ci}\}$ *depend* on task τ_i^c.

Given a cut, we are interested in all priority assignments that satisfy the following proposition:
For every $(\tau_i, \tau_j) \in T \times T$:

- $p_i > p_j$ if $e_{(i,j)} \in E$ and task τ_j does not depend on τ_i
- $p_i < p_j$ if τ_j depends on τ_i $i \neq j$, $\tau_i \in \text{pred}(\tau_j)$.

An adaptation of Audsley's algorithm can be used to choose an assignment consisting of a (unique) priority level for all tasks. This assignment is optimal with respect to the given cut [1]. First, we build a graph G containing an arc from task τ_i to task τ_j if τ_i must have a lower priority than τ_j. Then, we assign priorities sequentially, starting from the lowest. The algorithm works iteratively: on each step only those tasks that have no incoming links are eligible for scheduling. We select one of those tasks and tentatively assign it the lowest available priority level. If all events received by the selected task are found to be schedulable (by using the procedure described in Section 7), the task is assigned its priority and removed from G together with its outgoing arcs. If it is not found schedulable another eligible task is tried. If all tasks can be successfully assigned a priority the set is found to be schedulable.

The algorithm is identical to the one in [3], except that it can be tried on many different cuts instead of just one, removing some limitations. The most important problem with this solution is probably the need for a (possibly) large number of different priority levels, since each task must be assigned a unique priority.

8. EXPERIMENTAL RESULTS

We implemented the two algorithms shown in the paper and we tried them on a number of task sets with the following parameters (All uniform distributions except for the rate that is fixed.)

parameter	values
min-max num. of tasks per transaction	8 to 20
min-max num. of external tasks per trans.	1 to 2
min-max num. of events per task	1 to 4
min-max num. of events per external task	1 to 2
rate of external events	400 units
minimum utilization of processor	0.1 to 1.0

The minimum utilization has been computed as the ratio of the sum of the computation times of all tasks divided by the period of activation of the external tasks. Since tasks can be activated by more than one external task and more than once in response to an external event (Theorem 4) the actual utilization is mostly an underestimate of the actual one.

On our test computer (a 350MHz Pentium II class PC) the number of tasks was practically limited to be less than 30 if an exhaustive search on all possible cuts had to be performed. Beyond that number a limitation on the number of cuts to be evaluated is necessary. The outcome of the experiments was the percentage of schedulable transactions and the average minimum laxity on the schedulable sets. For 12 total tasks, 2 external tasks per transaction, and 1 to 2 events per task (internal or external) the results (on 10000 sample sets) are represented in figure 6. Figure 6 shows a performance comparison between the two algorithms. The behavior of the two algorithms is almost identical. For 10 total tasks, 2 external tasks per transaction, 1 event per external task and 1 to 3 events per internal task the results obtained with the two algorithms are represented in figure 7.

9. CONCLUSIONS

In this paper, we presented an iterative solution to the scheduling problem in reactive real-time systems represented by a network of asynchronously communicating FSMs. We developed a priority assignment scheme and a tight worst-case analysis. All results are rigorously derived. Experimental results are offered to exemplify the procedure and its quality.

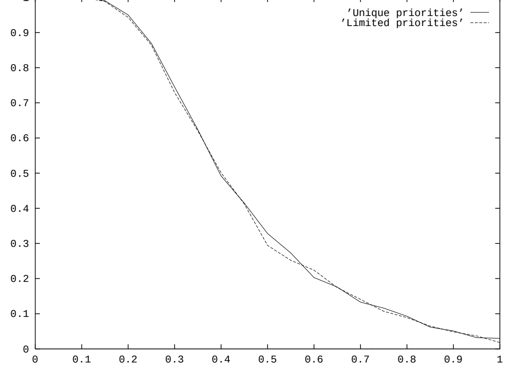


Figure 6: Schedulable ratio vs. utilization. 12 tasks per transaction, comparison of the two algorithms

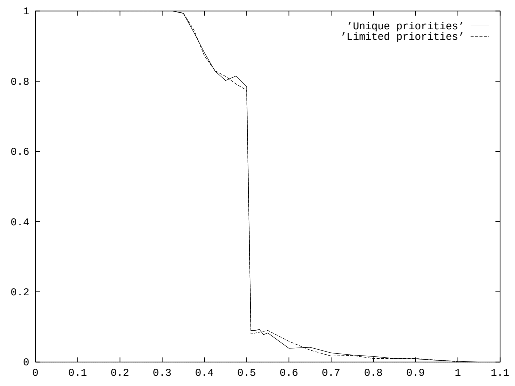


Figure 7: Schedulable ratio vs. utilization. 10 tasks per transaction, 1 to 3 links per task, 1 link per external task. Comparison of the two algorithms

10. REFERENCES

[1] N. Audsley et al. "Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling", *Software Engineering Journal, pp. 284-292*, Sept. 1993.

[2] Balarin F. and Sangiovanni-Vincentelli A. "Schedule Validation for Embedded Reactive Real-Time Systems," *DAC Conference*, Anaheim (CA) 1997.

[3] Balarin F. "Priority Assignment for Embedded Reactive Real-Time Systems," *LCTES Workshop 1998*, June 17-19 Montreal 1998.

[4] Balarin F., H. Hsieh, A. Jurecska, L. Lavagno and Sangiovanni-Vincentelli A. "Formal Verification of Embedded Systems based on SCFM networks," *Proceedings of the 33rd ACM/IEEE DAC Conference*, June 1996.

[5] Blazewitcz J., "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines" *in Modeling and Performace Evaluation of Computer Systems*, Amsterdam, 1976.

[6] Gerber R., Saksena M. and Hong H. "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes", *Proc. of the The IEEE Real-Time Systems Symposium*, Puerto Rico, Dec. 1994.

[7] Saksena M. et al. "Schedulability Analysis for Automated Implementation of Real-Time Object-Oriented Models" *Proc. of the The IEEE Real-Time Systems Symposium*, Madrid, Dec. 1998.

[8] Tindell, K. "Adding Time-Offsets to Schedulability Analysis", *Technical Report YCS 221*, Dept. of Comp. Science, univ. of York, Jan 1994.