Journal of Systems Architecture (The EUROMICRO Journal), vol. 46, no. 11, pp. 991-1004, september/2000

Fixed Priority Scheduling of Tasks with Arbitrary Precedence Constraints in Distributed Hard Real-Time Systems

Romulo Silva de Oliveira

I I - Univ. Fed. do Rio Grande do Sul
Caixa Postal 15064
Porto Alegre-RS, 91501-970, Brazil
romulo@inf.ufrgs.br

Joni da Silva Fraga

LCMI/DAS - Univ. Fed. de Santa Catarina
Caixa Postal 476
Florianopolis-SC, 88040-900, Brazil
fraga@lcmi.ufsc.br

Abstract

This paper considers the schedulability analysis of real-time distributed applications where tasks may present arbitrary precedence relations. It is assumed that tasks are periodic or sporadic and dynamically released. They have fixed priorities and hard end-to-end deadlines that are equal to or less than the respective period. We develop a method to transform arbitrary precedence relations into release jitter. By eliminating all precedence relations in the task set one can apply any available schedulability test that is valid for independent task sets.

1 Introduction

Deadlines are critical in a hard real-time system. In this case it is necessary to have a pre-runtime guarantee that every task will always meet its deadline. Such pre-runtime guarantee can be attained with fixed priority scheduling [11]. Tasks receive priorities according to some policy and a schedulability test is used off-line to guarantee that all tasks will meet their deadlines in a worst-case scenario. At runtime a preemptive scheduler selects the next task to run based on theirs fixed priorities.

Real-time scheduling in distributed systems is usually divided into two phases: allocation and local scheduling. Initially each task is assigned to a specific processor. The original allocation is permanent since it is usually not possible to migrate tasks at runtime. The second phase analyzes the schedulability of each processor.

Precedence relations are common in distributed systems. They are created by the necessity of synchronization and/or transfer of data between two tasks. It is possible to use offsets [2] to implement precedence relations. By defining an offset between the releases of two tasks it is possible to guarantee that the successor task will start its execution only after the predecessor is finished. This technique is sometimes called "static release of tasks" [14].

It is also possible to implement precedence relations by having the predecessor task to send a message to the successor task. It also informs the scheduler that the predecessor task has finished and the successor task can be released. This technique is sometimes called "dynamic release of tasks" [14]. The uncertainty about the release time of the successor task can be modeled as a release jitter [18].

Most studies on precedence relations deal only with linear precedences ("pipelines"), where each task has at most a single predecessor and a single successor. Task models that include arbitrary precedence relations do not impose such restriction. Studies that consider arbitrary precedence relations usually assume statically released tasks instead of the dynamically released tasks. Also, only a few papers consider this problem in a distributed environment.

The schedulability of statically released tasks with arbitrary precedence relations executing on a single processor is analyzed in [3] and [8]. In [5] arbitrary precedence relations are considered in a distributed environment, also for statically released tasks.

Linear precedence relations are analyzed in [7], [9], [12], [13], [14], [15] and [16].

The work described in [2] implements arbitrary precedence relations by dynamically releasing tasks. It shows that release jitter can be used to model precedence relations caused by communication between tasks in different processors. That paper does not develop the approach in a detailed way.

The work presented in [6] develops an appropriate schedulability analysis for distributed systems built on a point-to-point network. The task model includes arbitrary precedence relations, fixed priority and dynamically released tasks. Precedence relations are transformed into release jitter, so all tasks can be analyzed as they were independent.

Similarly, [18] presents a schedulability analysis for distributed applications on a bus-based network. It assumes fixed priority and dynamically released tasks. Linear precedence relations are transformed into release jitter, so all tasks are analyzed as they were independent. Arbitrary precedence relations are possible through statically released tasks.

In this paper we consider the scheduling of hard real-time tasks with arbitrary precedence constraints in a distributed environment. We assume a task model that includes fixed priority and dynamic release of tasks. The objective is to present a technique for schedulability analysis that is valid for systems where tasks are released as soon as the necessary messages have arrived. We develop transformation rules that, starting from the original task set, create equivalent sets of independent tasks with release jitter. These new sets of independent tasks can be used to evaluate the schedulability of the original task set.

The method presented in this paper to transform arbitrary precedence relations into release jitter is less pessimistic than a simple direct transformation as presented in [6]. By eliminating all precedence relations present in the original task set one arrive at a set of independent tasks. This new task set can be analized by any available schedulability test that is valid for independent task sets.

The remainder of the paper is organized as follows: section 2 describes the approach adopted in this work; section 3 formulates the problem; in section 4 we develop a set of transformation rules; section 5 describes a schedulability test algorithm; simulation results are presented in section 6; finally, concluding remarks are presented in section 7.

2 Description of our approach

There are in the literature several tests to analyze the schedulability of independent task sets when fixed priorities are used. Some of those tests allow tasks with release jitter, such as [2] and [17]. Those two works in particular analyze the schedulability of each task by computing its maximum response time and then comparing this value with the deadline.

It is possible to make a simple and direct transformation of precedence relations to release jitter [6]. When a task T_i has many predecessor tasks it is enough to identify the predecessor task that will cause the maximum delay to the start of T_i. This delay includes the maximum response time of the predecessor task plus any communication delay due to inter-processor message passing. This maximum delay to the start of T_i is transformed in the release jitter of task T_i. Its precedence relations can now be ignored during the schedulability analysis [6]. The schedulability analysis is done by transforming the system with precedence relations into an equivalent system composed of tasks with release jitter but without explicit precedence relations. Schedulability tests are applied to this equivalent system.

Actually, the transformation described above is pessimistic in the sense that the maximum response time of a task can be increased in the process. Figure 1 shows a very simple application that illustrates that pessimism for the monoprocessor case. Tasks T_0, T_1 and T_2 are characterized by their respective period P_0, P_1 and P_2 and their respective worst-case execution time C_0, C_1 and C_2. Task T_0 has the highest priority and task T_2 has the lowest priority.

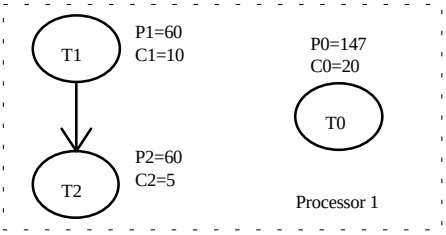

Figure 1 - Hypothetical application.

The maximum response time of task T_1 is 30. So, we can assign a release jitter of 30 to task T_2 and

eliminate its precedence relations. In the transformed system task T_2 presents a maximum response time given by adding its release jitter 30 and its maximum execution time 5 to the maximum interference it receives from other tasks, which is 20. The result 55 is too pessimistic since task T_0 was assumed to interfere twice with task T_2. Task T_0 interference was included in the computation of the maximum response time of T_1, which became the release jitter of T_2. Task T_0 interference was again considered when the maximum response time of T_2 was calculated. The period 147 of task T_0 makes it impossible for a second appearance of T_0 within a single activation of task pair (T_1,T_2). In this simple application it is possible to realize that, when one considers all precedence relations, the maximum response time of task T_2 is indeed 35.

The method presented in this paper creates an equivalent task set for each task T_i of the original set. The equivalent task set is such that: the maximum response time of task T_i in its equivalent set is greater than or equal to the maximum response time of this same task in the original task set.

This work follows the general approach described in [6]. We assume fixed priorities, dynamically released tasks and arbitrary precedence relations. As in [6], all deadlines are less than or equal to the respective period and release jitter is used to partially model the effects of a precedence relation. Differently from [6], this work takes into account the fact that a precedence relation limits the possible patterns of interference. That results in a less pessimistic analysis for the assumed task model.

2.1 Priority assignment

When tasks with precedence relations are distributed among many processors, there may be mutual dependence between the release jitter of a task and the maximum response time of another task. This problem was identified in [18] and solved by simultaneously computing the maximum response time of all tasks through successive iterations.

The approach adopted in this work is to limit the way priorities are assigned to tasks. The problem can be eliminated if the priority assignment policy satisfies two requirements:

• Local priorities are drawn from a global ordering that includes all tasks in the system. Although only the order of local tasks is important to the scheduler of each processor, we assume that this local ordering accords with the global ordering.

• The global ordering is such that priorities decrease along precedence relations. If T_i precedes T_j then T_i has a higher priority than T_j. This restriction is reasonable, since it gives higher priorities to the first tasks of an activity, reducing their response time.

Deadline Monotonic (DM) [1,10] is a simple policy used to assign priorities that can satisfy both requirements stated above. DM is a simple and fast way to assign priorities, although it is not optimal in systems with precedence relations [4]. Anyway, DM is optimal in the class of policies that generate decreasing priorities along precedence relations [9] and can be considered a good general purpose heuristic. In this work we assume that every task receives a globally unique priority according to DM. Deadlines are relative to the arrival of the respective activity. The task with the smallest relative deadline receives the highest priority. This globally unique priority results in a global ordering of tasks.

To satisfy the requirement of decreasing priorities along the precedence relations it is necessary to add two new rules to DM:

• If task T_i is a direct predecessor of task T_j, then it is necessary to have $D_i \leq D_j$, where D_i and D_j are the respective deadlines relative to the arrival of the activity. This requirement does not reduce system schedulability. It makes no sense task T_j to have a smaller deadline than T_i since T_j can only start its execution after T_i is finished.

• When two tasks have the same deadline, DM does not specify which one should receive the highest priority between them. We assume that when there is a precedence relation between two tasks, the predecessor task receives the highest priority.

By combining DM with those two rules, we make sure that task priorities produce a global ordering and that priorities are always decreasing along precedence relations. A side effect of our priority assignment policy is a simpler schedulability analysis. A similar effect is showed in [9] for tasks with precedence relations that are dynamically released and execute on a single processor.

3 Problem formulation

In this paper is assumed that a real-time application executes on a distributed system

composed of a set **H** of **h** processors. Remote communication is bounded and has a maximum delay of **Δ**. Communication protocols are executed on an auxiliary processor and do not compete with application tasks for processor time. The delay associated with local communication is supposed to be zero. Precedence relations are implemented by an explicit message sent from the predecessor task to the successor task.

An application is defined by a set **A** of **m** periodic or sporadic activities. Each activity A_x, $A_x \in A$, generates a possible infinite set of arrivals.

A periodic activity A_x is characterized by its period P_x, that is, the time interval between two successive arrivals. We assume that the first invocation of all periodic activities occurs at time zero. Periodic activities may have many initial tasks, that is, tasks without predecessors. A sporadic activity A_x is characterized by its minimum time interval P_x between arrivals. We also assume that sporadic activities have only one initial task that starts the activity. With a small abuse of notation we call A_x the set of all tasks that belong to activity A_x.

Every application is associated with a set of **n** tasks, that is, the set of all tasks that belong to some activity of set **A**. Any task can be preempted at any time and restarted later. Each task T_i is characterized by a global priority $\rho(T_i)$, a maximum execution time C_i and a deadline D_i relative to the arrival time of its activity. For all tasks T_i, $1 \le i \le n$, $T_i \in A_x$, we have $D_i \le P_x$.

If task T_i is the initial task of its activity, it may have a maximum release jitter J_i greater than zero. It is assumed that if tasks T_i, T_k are both initial tasks of activity A_x, then $J_i = J_k$.

Each task T_i is also characterized by the set $dPred(T_i)$. It includes all tasks that are direct predecessors of T_i. Task T_i is not released until it receives a message from each of its direct predecessors. By referencing set $dPred(T_i)$ of each task T_i it is possible to define five other sets:

- Set $iPred(T_i)$ of all indirect predecessors of T_i:
- Set $Pred(T_i)$ contains all predecessors of T_i:
- Set $dSuc(T_i)$ contains all direct successors of T_i:
- Set $iSuc(T_i)$ contains all indirect successors of T_i;
- Set $Suc(T_i)$ contains all successors T_i.

Each task receives a unique individual priority according to Deadline Monotonic. We assume that deadlines increase along precedence relations, that is, if $T_i \in Pred(T_j)$ then $D_i < D_j$ and $\rho(T_i) > \rho(T_j)$. Without loss of generality tasks are named in the decreasing order of their priorities. So, $i<j$ implies that $\rho(T_i) > \rho(T_j)$.

All tasks T_i were previously allocated to processors of set H. Allocation algorithms are beyond the scope of this paper. It is possible that, as the result of an allocation algorithm, tasks from the same activity will be spread over many processors. A task can not migrate during execution. Figure 2 in section 4 illustrates some of the concepts presented in the problem formulation.

We introduce R_i^A to denote the maximum response time of task T_i in application **A** and $H(T_i)$ to denote the processor where task T_i executes. Also, $\rho(A_x)$ will denote the highest priority among all tasks that belong to A_x.

Given an application defined as described above, it is necessary to determine if all releases of all tasks will always be completed before deadline.

4 Schedulability analysis

In this section we develop transformation rules that create a set of independent tasks with release jitter that is equivalent to the original one but does not include precedence relations. Each rule is presented as a theorem.

Theorems 1 to 5 describe rules to transform a system where a task T_i, the only task of activity A_x, receives interference from tasks of other activities. Initially we consider the interference with task T_i caused by a completely local activity A_y. Since priorities decrease along precedence relations, there are three cases: all tasks of A_y have a priority lower than $\rho(T_i)$; all tasks of A_y have a priority higher than $\rho(T_i)$; some tasks of A_y have a higher priority than $\rho(T_i)$ while others have a lower priority. These three cases originate theorems 1, 2 and 3. Their prove is straightforward and will not be presented.

Theorem 1

Assume **A** is an application where $A_x \in A$, $A_y \in A$, $A_x \neq A_y$ are activities. Task T_i is the only task of activity A_x. Assume also that

$\forall T_j \in A_y \,.\, H(T_j)=H(T_i) \wedge \rho(T_j)<\rho(T_i)$

and that **B** is an application such that $B = A - \{A_y\}$.

In the above conditions we have $R_i^A = R_i^B$. □

At this point we introduce $\#dPred(T_i)$ to denote the number of elements in the set $dPred(T_i)$ of direct predecessors of task T_i. Similarly, we use $\#dSuc(T_i)$, $\#iPred(T_i)$ and $\#iSuc(T_i)$ to respectively denote the number of direct successors, indirect predecessors and indirect successors of task T_i.

Theorem 2

Assume **A** is an application where $A_x \in A$, $A_y \in A$, $A_x \neq A_y$ are activities. Activity A_x has a single task T_i and

$\forall T_j \in A_y \,.\, H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$.

Also, there is exactly only one task T_k such that:

$T_k \in A_y \wedge \#dPred(T_k)=0$.

Finally, $B = (A-\{A_y\}) \cup \{A_e\}$ is an application where A_e is an activity with the same period of activity A_y, $P_e=P_y$, composed of a single task T_e such that: $J_e=J_k$, $H(T_e)=H(T_i)$, $\rho(T_e)=\rho(A_y)$, and

$$Ce = \sum_{\forall Tj.Tj \in Ay} Cj \;.$$

In the above conditions we have $R_i^A = R_i^B$. □

Theorem 3

Assume **A** is an application where $A_x \in A$, $A_y \in A$, $A_x \neq A_y$ are activities. Activity A_x is composed of a single task T_i. Activity A_y is such that:

$\forall T_j \in A_y \,.\, H(T_j) = H(T_i)$,

$\exists T_j \in A_y \,.\, \rho(T_j)>\rho(T_i) \;\wedge\; \exists T_j \in A_y \,.\, \rho(T_j)<\rho(T_i)$.

Also, activity A_y has a single initial task T_k, that is, $T_k \in A_y \wedge \#dPred(T_k)=0$.

Finally, assume $B = (A - \{ A_y \}) \cup \{ A_e \}$ is an application where A_e is an activity composed of a single task T_e such that: $P_e >> P_i$, $J_e=J_k$, $H(T_e)=H(T_i)$, $\rho(T_e) = \rho(A_y)$, and

$$Ce = \sum_{\forall Tj.Tj \in Ay \wedge \rho(Tj)>\rho(Ti)} C_j \;.$$

In the above conditions we have $R_i^A = R_i^B$. □

At this point we define $\delta_{i,j}$ as the delay associated with sending a message from task T_i to task T_j. So we have: $\delta_{i,j} = \Delta$ when $H(T_i) \neq H(T_j)$, and $\delta_{i,j} = 0$ when $H(T_i)=H(T_j)$.

Theorem 4 assumes that task T_j is executed on the same processor of task T_i and that T_j has many direct predecessors. Among the predecessors of T_j there is task T_k, the one able to generate the biggest delay to the release of T_j. It is showed that we can transform the application in a way that T_j will have T_k as its single direct predecessor.

Theorem 4

Assume **A** is an application where $A_x \in A$, $A_y \in A$, $A_x \neq A_y$ are activities. Activity A_x has a single task T_i and activity A_y is such that

$\exists T_j \in A_y \,.\, H(T_i)=H(T_j) \wedge \#dPred(T_j) \geq 2$.

Also task $T_k \in A_y$ is such that $T_k \in dPred(T_j)$ and

$$R_k^A + \delta_{k,j} = \max_{\forall T_l \in dPred(T_j)} (R_l^A + \delta_{l,j}) \,.$$

Finally, $B = (A - \{A_y\}) \cup \{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_y, except for T_j that has $dPred(T_j)=\{ T_k \}$.

In the above conditions we have $R_i^A \leq R_i^B$.

Proof

Application **B** is what results after we eliminate all precedence relations in **A** where T_j appears as the successor task, except for the precedence relation where T_k is the predecessor.

We define Γ_i^A as the set of all possible schedules generated by application **A** on processor $H(T_i)$. Observe that R_i^A is defined by a schedule that belongs to Γ_i^A and where there is the biggest temporal distance between the arrival and the conclusion of T_i.

The maximum release jitter of task T_j in application **A** is given by $R_k^A+\delta_{k,j}$. Since the precedence relation between T_k and T_j is also present at application **B**, the same happens to its maximum release jitter in **B**. The single consequence of removing the other precedence relations is the possible increasing in size of the set of all possible schedules, that is, $\Gamma_i^A \subseteq \Gamma_i^B$.

Since $\Gamma_i^A \subseteq \Gamma_i^B$, any possible schedule of Γ_i^A is also a possible schedule of Γ_i^B, including the schedule that defines R_i^A. We can not say anything about the existence in Γ_i^B of schedules that would define a bigger response time for T_i. So, we have $R_i^A \leq R_i^B$. □

Theorem 5 assumes a task T_j that has a single direct predecessor and executes on the same processor task T_i does. It shows that the interference of T_j with T_i is not reduced when T_j precedence relation is transformed into a release jitter.

Theorem 5

Assume **A** is an application where $A_x \in A$, $A_y \in A$, $A_x \neq A_y$ are activities. Activity A_x is composed of a single task T_i and activity A_y is such that:
$\exists T_j \in A_y, \exists T_k \in A_y \,.\, H(T_i)=H(T_j) \wedge dPred(T_j)=\{T_k\}$.

Finally, $B = (A - \{ A_y \}) \cup \{ A_e \}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_y, except for task T_j that has $dPred(T_j)=\{ \}$ and $J_j = R_k^A + \delta_{k,j}$.

In the above conditions we have $R_i^A \leq R_i^B$.

Proof

Application **B** is what results after we eliminate the only precedence relation in **A** where T_j appears as the successor task. We again define Γ_i^A as the set of all possible schedules generated by application **A** on processor $H(T_i)$.

The maximum release jitter of T_j in application **A** is given by $R_k^A+\delta_{k,j}$. The same maximum release jitter is present at application **B**. The single consequence of removing that precedence relation is the possible enlargement of the set of all possible schedules, that is, $\Gamma_i^A \subseteq \Gamma_i^B$.

Since $\Gamma_i^A \subseteq \Gamma_i^B$, any possible schedule of Γ_i^A is also a possible schedule of Γ_i^B, including the schedule that defines R_i^A . We can not say anything about the existence in Γ_i^B of schedules that would define a bigger response time for T_i. So, we have $R_i^A \leq R_i^B$. □

Theorems 6 to 11 deal with a task T_i, subject to precedence relations within its activity A_x, that is interfered with by tasks from other activities. These theorems transform task T_i own activity to the point T_i becomes an independent task.

Theorem 6 shows that, when task T_i has no predecessors, it is possible to eliminate precedence relations that connect task T_i to activity A_x.

Theorem 6

Assume **A** is an application, $A_x \in A$ is an activity and $T_i \in A_x$ is a task such that $dPred(T_i)=\{ \}$. Also assume that $B = (A-\{A_x\}) \cup \{A_i, A_e\}$ is an application where A_i is an activity, $P_i=P_x$, composed exclusively of task T_i. A_e is an activity composed of the same tasks and precedence relations of activity A_x, except for task T_i that is eliminated and for its period $P_e >> P_x$.

In the above conditions we have $R_i^A = R_i^B$.

Proof

All other tasks of A_x can be released at most once within an execution of T_i in **A**. A second release would imply a second arrival of A_x and, as well, a second arrival of T_i. When we consider application **B**, the period of A_e guarantees that its tasks can be released at most once within a single execution of T_i. So, the amount of interference with T_i in **A** due to all other tasks of A_x will be equal to the amount of interference with T_i in **B** due to tasks of A_e.

Also, by eliminating precedence relations that include T_i as predecessor we do not affect the interference with T_i caused by tasks that belong to other activities and have a higher priority than T_i.

Since the differences between **A** and **B** do not affect the interference with T_i, we have $R_i^A=R_i^B$. □

Theorem 7 deals with a situation where task T_i has a single predecessor task T_k that executes on the same processor T_i does. In this case, it is possible to merge both tasks to create a single equivalent task.

Theorem 7

Assume **A** is an application, $A_x \in A$ is an activity and $T_i \in A_x$ is a task such that $dPred(T_i)=\{T_k\}$ and $H(T_i)=H(T_k)$.

Assume $B = (A-\{A_x\}) \cup \{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_x, except for tasks T_k and T_i that are replaced by task T_e. Task T_e is characterized by:

$H(T_e)=H(T_i)$, $\rho(T_e)=\rho(T_i)$,
$J_e= J_k$, $C_e=C_k+C_i$,
$dPred(T_e)=dPred(T_k)$, $dSuc(T_e)=dSuc(T_i)$.

In the above conditions we have $R_i^A = R_e^B$.

Proof

We transform application **A** into **B** step by step and show the maximum response time of T_i in **A** is equal to the maximum response time of task T_e in **B**.

It is possible for some tasks in **A** to succeed T_k but not succeed T_i. These tasks have a priority lower

than $\rho(T_k)$, but may have a priority that is higher than $\rho(T_i)$ and interfere with T_i. This fact does not depend on the precedence relations of T_k. So, those relations can be eliminated without affecting R_i^A.

Assume time interval [t,t') is defined by the release of T_k and by the conclusion of T_i in the worst case scenario for T_i. We now set T_k priority to $\rho(T_i)$. There is enough processor time within interval [t, t') to completely execute tasks T_k, T_i and all task T_j such that $\rho(T_j)>\rho(T_i) \wedge H(T_j)=H(T_i)$, that were released before **t'** but not finished before **t**. So, by setting T_k priority to $\rho(T_i)$, we have a new task ordering within that interval, such that T_k may be executed at the end of the interval, close to the execution of T_i. Any way, T_i again will be finished at **t'** in the worst case.

We now add to $dPred(T_i)$ all tasks that belong to set $dPred(T_k)$. This change does not alter the release time of T_i, since all tasks that belong to $dPred(T_k)$ were originally elements of set $iPred(T_i)$ and will be already finished by the time T_i is released.

We now eliminate the precedence relation between T_k and T_i so task T_i is released at the same time task T_k is released. After this transformation task T_i presents the same maximum release jitter of task T_k, since both have the same arrival and release times. Again, we have a reordering of the tasks executed within interval [t,t'), such that T_i may execute before task T_k is finished. Any way, tasks T_k and T_i will be finished by the end of interval [t,t') in the worst case, since the demand for processor time was not changed and equal priorities imply an arbitrary execution order.

Once that, after all transformations, T_k and T_i have the same release time, the same maximum release jitter, the same priority, they execute on the same processor, they have the same predecessor and successor tasks, we can consider them as a single task. The equivalent single task T_e has a maximum execution time equal to the sum of the maximum execution times of T_i and T_k and it will be finished at time **t'**, in the worst case. So, $R_i^A = R_e^B$. □

Theorem 8 assumes task T_i has a single task T_k as predecessor, and T_k executes on another processor. It is possible to replace the precedence relation by a release jitter associated with task T_i.

Theorem 8

Assume **A** is an application, $A_x \in A$ is an activity and $T_i \in A_x$ is a task such that $dPred(T_i)=\{T_k\}$, where $H(T_i) \neq H(T_k)$. Also $B=(A-\{A_x\}) \cup \{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_x, except for task T_i that is replaced by a task T_e, and for all tasks of set $Pred(T_i)$ in **A** that, by definition, do not interfere with T_e in **B**. Task T_e is defined by: $H(T_e)=H(T_i)$, $\rho(T_e)=\rho(T_i)$, $C_e=C_i$, $J_e=R_k^A+\Delta$, $dPred(T_e)=\{\ \}$.

In the above conditions we have $R_i^A = R_e^B$.

Proof

A precedence relation creates release jitter for the successor task at the same time it reduces the set of possible application schedules. Predecessors may delay the release of a successor task but they do not interfere with it. We will compare task T_i in **A** with task T_e in **B** about their maximum release jitter and the interference they receive.

The release jitter of T_i caused by T_k in **A** is maximum when T_k has its maximum response time and the communication between T_k and T_i exhibits its maximum delay, that is, $R_k^A+\Delta$. By definition, task T_e has a maximum release jitter in **B** that is equal to the maximum release jitter of T_i in **A**.

At the moment task T_i is released in **A**, all its indirect predecessors are, by necessity, already finished. They will not be released again before the conclusion of task T_i. So, they do not interfere with the execution of T_i beyond their contribution to the release jitter of T_i. By the way T_e was defined, it does not receive any interference in **B** from tasks that indirectly precede T_i in **A**.

Once the effects of the precedence relation between T_k and T_i in **A** are compensated by the characteristics of T_e in **B**, we can say $R_i^A = R_e^B$. □

Theorems 9, 10 and 11 consider a task that has many direct predecessors. They show that all but one precedence relation can be eliminated. As defined before, Δ represents a bound to remote communication delay.

Theorem 9

Assume **A** is an application, $A_x \in A$ is an activity, $T_i \in A_x$ is a task such that $\#dPred(T_i) \geq 2$ and all direct predecessors of T_i are local, that is,

$\forall T_j \in dPred(T_i)$, $H(T_j) = H(T_i)$.

We define task T_k, critical predecessor of T_i in **A**, by the following:

$$R_k^A = \max_{\forall T_l . T_l \in dPred(T_i)} R_l^A .$$

Finally, assume B = (A - $\{A_x\}$) $\cup$ $\{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_x, except for task T_i that in A_e has $dPred(T_i)=\{ T_k \}$.

In the above conditions we have $R_i^A = R_i^B$.

Proof

In the worst case, task T_k is the last one among the direct predecessors of T_i to finish its execution and to send a message to T_i. Since all direct predecessors of T_i are in the same processor, it is guaranteed that, when T_k finishes in the worst case, all other tasks that are direct predecessors of T_i will also be finished. By keeping the precedence relation between T_k and T_i we also keep the maximum response time of T_i. □

Theorem 10

Assume **A** is an application, $A_x \in A$ is an activity, $T_i \in A_x$ is a task such that $\#dPred(T_i) \geq 2$ and all direct predecessors of Ti are remote, that is,

$\forall T_j \in dPred(T_i)$, $H(T_j) \neq H(T_i)$.

We define task T_k, critical predecessor of T_i in **A**, by the following:

$$R_k^A + \Delta = \max_{\forall T_l . T_l \in dPred(T_i)} R_l^A + \Delta .$$

Finally, assume B = (A - $\{A_x\}$) $\cup$ $\{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_x, except for task T_i that in A_e has $dPred(T_i)=\{ T_k \}$ and that all tasks T_j in **A** so that

$T_j \in Pred(T_i) \wedge H(T_j)=H(T_i)$,

by definition, do not interfere with T_i in **B**.

In the above conditions we have $R_i^A = R_i^B$.

Proof

In the worst case, task T_k is the last one among the direct predecessors of T_i to finish its execution and to send a message to T_i. It is guaranteed that, by the time T_k finishes in the worst case, all other tasks that are direct or indirect predecessors of T_i will also have finished. By keeping the precedence relation between T_k and T_i we also keep the maximum response time of T_i.

By the moment task T_i is ready to run in **A**, all local predecessors are already finished. They will not be released again before the conclusion of T_i. So, they do not interfere with T_i in **A**. The same happens in **B** by definition. □

Theorem 11

Assume **A** is an application, $A_x \in A$ is an activity and $T_i \in A_x$ is a task such that $\#dPred(T_i) \geq 2$. Also assume that:

$\exists T_j \in dPred(T_i)$, $H(T_j)=H(T_i)$ and that

$\exists T_j \in dPred(T_i)$, $H(T_j) \neq H(T_i)$.

Task $T_{loc} \in A_x$ is a task such that $T_{loc} \in dPred(T_i)$ and $H(T_{loc})=H(T_i)$, where

$$R_{loc}^A = \max_{\forall T_l . T_l \in dPred(T_i) \wedge H(T_l)=H(T_i)} R_l^A .$$

Similarly, assume $T_{rem} \in A_x$ is a task such that $T_{rem} \in dPred(T_i)$ and $H(T_{rem}) \neq H(T_i)$, where

$$R_{rem}^A + \Delta = \max_{\forall T_l . T_l \in dPred(T_i) \wedge H(T_l) \neq H(T_i)} R_l^A + \Delta .$$

We define task T_k, critical predecessor of T_i in **A**, by the following rules:

[a] Case $R^A_{rem}+\Delta \geq R^A_{loc}$ then $T_k = T_{rem}$.

[b] Case $R^A_{rem}+\Delta < R^A_{loc} - I^A_{loc}$ then $T_k=T_{loc}$, where I^A_{loc} is the maximum interference received by T_{loc} in **A**.

Finally, assume B = (A - $\{A_x\}$) $\cup$ $\{A_e\}$ is an application where A_e is an activity composed of the same tasks, precedence relations and period of activity A_x, except for task T_i that in A_e has $dPred(T_i)=\{ T_k \}$ and that, if $H(T_k) \neq H(T_i)$ then all tasks T_j such that $T_j \in Pred(T_i) \wedge H(T_j)=H(T_i)$ in **A**, by definition, do not interfere with T_i in **B**.

In the above conditions we have $R_i^A = R_i^B$.

Proof

Case [a]

By the time the message from task T_{rem} is available for task T_i, all direct and indirect

predecessors of T_i will be finished and their messages will already be available for T_i. So, the maximum response time of T_i is defined by the precedence relation that includes T_{rem}.

When task T_i is ready to run in **A**, all local predecessors are already finished. They will not be released again before the conclusion of T_i. They do not interfere with T_i in **A** and the same happens in **B** by definition.

Case [b]

The worst scenario for T_i may be different from the worst scenario for T_{loc}. For example, assume another task in the same processor of T_i with a higher priority and a period much greater than P_x. In the worst case for T_i this other task is released exactly at the same moment T_i is released, after T_{loc} is finished. This is certainly not the worst case for T_{loc}. So, I^A_{loc} is computed by assuming the worst possible case for T_{loc}, but that is not necessarily the worst possible case for T_i. Since $R^A_{rem}+\Delta<R^A_{loc}$ the message from T_{rem} will arrive before T_{loc} finishes in the worst case for T_{loc}. But that does not guarantee that the message from T_{rem} will arrive before T_{loc} finishes in the worst case for T_i. So, it is not enough to have $R^A_{rem}+\Delta < R^A_{loc}$ to say T_{loc} is the critical precedence of T_i.

By assuming that $R^A_{rem} + \Delta < R^A_{loc} - I^A_{loc}$ we guarantee that, independently from the patterns of interference with T_{loc} and T_i, the message from T_{rem} will be available for T_i before T_{loc} finishes. So, the worst case for T_i will be associated with the conclusion of T_{loc} and not with the message sent by T_{rem}. By preserving the precedence relation between T_{loc} and T_i we also preserve the worst case for T_i and its maximum response time. □

Theorem 11 does not define T_k when:

$R^A_{loc} - I^A_{loc} \le R^A_{rem}+\Delta < R^A_{loc}$.

In this case, it is possible to determine an upper bound for the maximum response time of T_i by artificially increasing Δ between T_{rem} and T_i. Figure 2 shows an application composed of two activities. One activity has three tasks while the other have a single task. Two processors are used. The figure also shows the periods (P0, P1, P2), the maximum release jitters (J0, J1, J2), the maximum execution times (C0, C1, C2, C3) and the precedence relations.

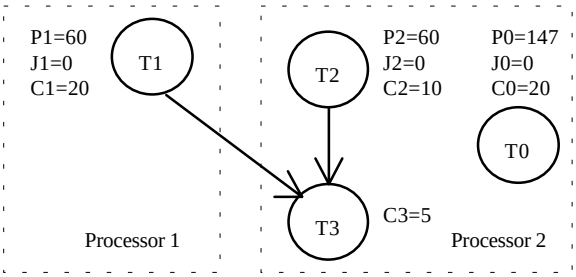

Figure 2 - Hypothetical application, Δ=7.

Since task T_3 has two direct predecessors we should apply theorem 11 to determine the critical precedence. There is a single remote direct predecessor, so $T_{rem}=T_1$. There is also a single local direct predecessor, $T_{loc}=T_2$. We also have that

$R^A_{rem}+\Delta=R^A_1+\Delta=20+7=27$,

and that

$I^A_{loc}=I^A_2=20$ and $R^A_{loc}=R^A_2=10+20=30$.

Since

$R^A_{rem}+\Delta < R^A_{loc}$, and

$R^A_{rem}+\Delta \ge R^A_{loc} - I^A_{loc}$,

theorem 11 does not define T_k. Since this is a small application, it is possible to test all possibilities. Case $T_k=T_{rem}=T_1$, then the maximum response time of T_3 would be given by 27+5+20=52. If we had $T_k=T_{loc}=T_2$, the maximum response time of T_3 would be given by 10+5+20=35. We choose the worst case, that is, $T_k=T_{rem}$.

In general this analysis has an exponential complexity and is not feasible in practice for systems much larger than that of figure 2. In those cases, we can artificially increase the value of Δ for the specific communication between T_1 e T_3. We do it such that condition [a] of theorem 11 is satisfied. In the case of the example, by assuming Δ=10 we have:

$R^A_{rem}+\Delta =30 = R^A_{loc}$ and then task T_1 will be the critical precedence of T_3.

This artificial increase in the value of Δ introduces a certain amount of pessimism in the analysis. It happens when the critical precedence is originally the remote precedence and the increasing of Δ was actually unnecessary.

It is important to note that just one delay is increased, that is, the delay associated with the communication between T_{rem} e T_i. Also, the increased value is used only to compute the

maximum response time of task T_i. The maximum response time of all remaining tasks will be computed by using the original value of Δ.

In order to recognize this situation as a too pessimistic one, it would be necessary to compute the response time of T_i considering first $T_k=T_{loc}$ and then $T_k=T_{rem}$. The complexity of this analysis depends on the precedence graph being considered. For certain precedence graphs it can present a complexity of 2^n, where **n** is the number of tasks in the application.

We now take a closer look at some particular cases that may occur when one applies theorems 3, 6, 8, 9, 10 and 11. These theorems implicitly assume that the maximum response time of every task is less than or equal to its activity period. This assumption allows one to ignore the second arrival of an activity while the final tasks of its previous arrival have not finished yet. It can be proved that if a task T_i in application **A** has a maximum response time greater than its period, then the maximum response time of a task T_k in **B** may be smaller than that in **A**, but still greater than its own period. Since all deadlines are less than or equal to the respective periods, the transformation described in those theorems is such that: if T_k is schedulable in **A**, it is also schedulable in **B**; if T_k is not schedulable in **A**, it is not schedulable in **B** neither, but it may have a maximum response time that is smaller than what it is in **A**. In the latter case the transformation is optimistic, but not so to make an unschedulable task in **A** to become a schedulable task in **B**.

If the implicit assumption holds true, then all calculated maximum execution times are exact. If any calculated maximum response time is greater than its respective activity period, the application is declared to be not schedulable and the calculated maximum response times must be considered an approximation. It is possible to summarize the effect of theorems 3, 6, 8, 9, 10 and 11 in the way showed bellow, where R_i represents the calculated maximum response time of task T_i using the transformation rules and R_i^A represents the true maximum response time of T_i in the original system:

- If $\exists T_i \in A_x . R_i > P_x$ then the application is not considered schedulable and the calculated maximum response times of all tasks are approximations;
- If $\forall T_i \in A_x . R_i \le P_x \wedge \exists T_j \in A_y . R_j > D_j$ then the application is not considered schedulable and the calculated maximum response times of all tasks are pessimistic approximations, that is, $\forall T_k . R_k \ge R_k^A$;
- If $\forall T_i \in A_x . R_i \le P_x \wedge \forall T_i \in A_x . R_i \le D_i$ then the application is schedulable and the calculated maximum response times of all tasks are pessimistic approximations, that is, $\forall T_k . R_k \ge R_k^A$.

5 Algorithm

In this section we present an algorithm that calculates the maximum response time of each task. Once this value is calculated, it can be compared with the task deadline.

Basically, we make successive transformations of the application until all precedence relations are eliminated. These transformations are done in such a way that the maximum response time of task T_i in the transformed application is greater than or equal to its maximum response time in the original application. Transformations take into account how task T_i is affected by the other tasks. So, for each task T_i it is necessary to apply transformations starting from the original application.

The algorithm is composed of three parts. The first part controls its execution. It is a loop which body is repeated for each task T_i. Initially, the precedence relations within the activity that includes T_i are eliminated. Then, the maximum response time of this independent task is computed. These two steps are done by the second and third parts.

Table 1 presents the main part of the algorithm. Higher priority tasks are analyzed first. So, by the time R_i^A is being calculated, it is already known the value R_j^A for all tasks T_j such that $\rho(T_j)>\rho(T_i)$. Lines 2 and 3 are just calls to the second and third parts, respectively.

[1]	For each T_i, **i** from **1** to **n**, do
[2]	Compute an equivalent application where T_i is an independent task
[3]	Compute the maximum response time of T_i in the equivalent application

Table 1 - First part of the algorithm: main part.

Table 2 presents the second part of the algorithm. The activity that includes T_i goes through successive transformations until we arrive at an independent

task. When T_i has a single direct predecessor that executes in the same processor, both tasks are merged (lines 5 and 6). In case T_i has a single direct predecessor that executes in another processor, the precedence relation is transformed into release jitter (lines 7 and 8). Finally, when T_i has many direct predecessors, all but one precedence relations are eliminated (lines 9 and 10).

It is important to note that the loop that includes lines 5 to 10 is executed until T_i no longer has any predecessor (line 4). Then, line 11 eliminates all precedence relations that still connect T_i to its successor tasks in the activity. By the end of the second part we have an independent task.

[4]	While $\#dPred(T_i) > 0$
[5]	Case $\#dPred(T_i)=1 \wedge H(T_i)=H(T_k), T_k \in dPred(T_i)$
[6]	Apply theorem 7, merge T_k and T_i
[7]	Case $\#dPred(T_i)=1 \wedge H(T_i)\neq H(T_k), T_k \in dPred(T_i)$
[8]	Apply theorem 8, reduce to $\#dPred(T_i)=0$
[9]	Case $\#dPred(T_i) > 1$
[10]	Apply theor.9,10,11, reduce to $\#dPred(T_i)=1$
[11]	Apply theorem 6, disjoint T_i from A_x

Table 2- Second part: makes T_i an independent task.

Table 3 shows the third part of the algorithm. It is a loop that eliminates all precedence relations in all activities that have at least one task on the same processor T_i executes (lines 12 and 13). Among these activities we include what is left from the original activity of task T_i, after we removed T_i.

Each activity is decomposed in fragments. Each fragment is defined by a single initial task. This decomposition is achieved through the elimination of multiple precedence relations (lines 14 to 16) and the elimination of precedence relations that includes another processor (lines 17 to 19).

Finally, each fragment (line 20) is reduced to an independent task that generates an equivalent interference with T_i. If the priority of every task of the fragment is lower than the priority of T_i, the fragment is eliminated (lines 21 and 22). If the priority of every task of the fragment is higher than the priority of T_i, the fragment becomes a single task that generates the same interference (lines 23 and 24). In case some tasks have a lower priority while other tasks have a higher priority than T_i, the fragment again becomes a single equivalent task (lines 25 and 26). By the end (line 27), it is possible to apply an algorithm that calculates the maximum response time of a task within a system made of independent tasks. For instance, the algorithm described in [2] can be used.

[12]	For each $A_y \in A$, $T_i \notin A_y$, do
[13]	If $\exists T_j . T_j \in A_y \wedge H(T_j)=H(T_i)$ then
[14]	For each T_j. $T_j \in A_y \wedge H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$
[15]	If $\#dPred(T_j) \geq 2$ then
[16]	Apply theor.4, reduce to $\#dPred(T_j)=1$
[17]	For each T_j. $Tj \in A_y \wedge H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$
[18]	If $T_k \in dPred(T_j) \wedge H(T_k)\neq H(T_i)$ then
[19]	Apply theor.5, reduce to $\#dPred(T_j)=0$
[20]	For each independent fragment A_z from A_y in $H(T_i)$
[21]	Case $\forall T_j \in A_z$, $\rho(T_j)<\rho(T_i)$
[22]	Apply theor.1, eliminate fragment A_z
[23]	Case $\forall T_j \in A_z$, $\rho(T_j)>\rho(T_i)$
[24]	Apply theor.2, reduce A_z to single task
[25]	Other cases
[26]	Apply theor.3, reduce A_z to single task
[27]	Compute the maximum response time of T_i

Table 3 - Third part: calculate the response time.

By a simple inspection of the algorithm it is possible to note that any acyclic task graph can be analyzed. We consider now the complexity of the algorithm presented in this section. The first part is a loop that is executed **n** times. The second part can be executed at most **n** times, when T_i is the last task of a linear activity composed of **n** tasks. The third part main loop (line 12) can be executed at most **n-1** times, when there are no precedence relations in the application. In this case, the internal loops (lines 14, 17 and 20) are limited to a single iteration. In another extreme situation, T_i receives interference from an activity that includes all remaining tasks of the application. In this case the main loop executes only once and the number of iterations of the internal loops is limited by **n** (no activity can have more than **n** tasks). We assume that line 27 has complexity **E**. This value depends on the schedulability test used for independent tasks. The algorithm as a whole has a complexity given by: $n(n+(n+E))$, that is, $\mathbf{O(n^2+n.E)}$.

6 Simulation

The method described in the previous section to transform arbitrary precedence relations into release jitter is evaluated by simulation. Its performance is compared with the performance of a simple and direct transformation, where every precedence relation is transformed into release jitter. Both methods transform the original task set into a set of independent tasks that can be analyzed by, for example, the algorithm presented in [2].

Workload generation was based on the method used in [15]. Every application is composed of 5 activities with T tasks per activity, plus 5×T independent tasks. In this context, an independent task can be viewed as an activity composed of a single task. We simulate activities with T equal to 3, 5 and 7. All activities are periodic, with period between 100 and 10000, according to an exponential distribution. All task deadlines are made equal to its activity period. Task deadlines are relative to the arrival of the respective activity.

The simulated system has Δ=20. Tasks are randomly assigned to four processors. The maximum execution time of each task is based on a number randomly chosen from a uniform distribution in the range 0.01 to 1. The maximum execution time of each task is equal to its period times processor utilization times its random number divided by the sum of the random numbers of all tasks on the same processor. In each particular application all processors have the same utilization. Priorities are assigned to tasks based on its deadline, its position within the activity and its precedence relations.

Table 4 summarizes the simulation results. The main concern of the simulations is the number of applications declared schedulable by our algorithm in comparison with the number of applications declared schedulable by the simple method mentioned earlier. For example, a figure of 45% means that the simple method was able to declare schedulable a number of applications that corresponds to 45% of the number of applications our test declared schedulable.

Table 4 shows the results when processor utilization varies from 10% to 90% and the number of tasks per activity varies from 3 to 7. The number of task sets generated was such that our test was able to declare as schedulable at least 1000 applications. The same set of applications was given as input to both algorithms. Since task's characteristics are mostly random, many generated applications are actually not schedulable.

Tasks/activity	**3**	**5**	**7**
Utilization			
10%	100%	100%	100%
20%	100%	100%	100%
30%	100%	100%	100%
40%	100%	100%	100%
50%	100%	99%	97%
60%	99%	94%	92%
70%	94%	85%	80%
80%	81%	65%	54%
90%	53%	40%	34%

Table 4 - Simulation results.

Both tests perform equally when processor utilization is 40% or less. When processor utilization is 50% or higher our test is less pessimistic. For example, when there are 3 tasks per activity and processor utilization is 90%, a simple method is able to declare schedulable only 53% of the number of applications our method finds schedulable.

The difference between the methods increases when we increase the number of tasks per activity. For example, the figure for 90% utilization goes from 53% to 34% when we increase the number of tasks per activity from 3 to 7. Table 4 shows that the algorithm presented here is less pessimistic than a simple direct transformation of precedence relations into release jitter.

7 Conclusion

This paper presented a new method to transform arbitrary precedence relations into release jitter in a distributed real-time application. It is supposed that tasks are dynamically released and are dispatched according to theirs fixed priorities. We considered periodic and sporadic tasks that have a deadline less than or equal to their period. By eliminating all precedence relations in the task set one can apply any available schedulability test that is valid for independent task sets.

The correctness of our method was showed through the demonstration of several theorems. These theorems are the building blocks of the

algorithm presented in section 5. Simulations were done to highlight the advantages of this method when compared with a simple direct transformation of precedence relations to release jitter.

Although our method results in a sufficient but not necessary schedulability test, it is less pessimistic than a simple direct transformation of each precedence relation in a correspondent release jitter. As far as we known, this is the only alternative method to analyze the schedulability of task sets with arbitrary precedence relations and dynamic release of tasks in a distributed environment.

Acknowledgements

This work was supported in part by the brazilian agencies FAPERGS and CNPq.

References

[1] N. C. Audsley, A. Burns, A. J. Wellings. Deadline Monotonic Scheduling Theory and Application. Control Engineering Practice, Vol. 1, No. 1, pp. 71-78, feb. 1993.

[2] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal, Vol. 8, No. 5, pp.284-292, 1993.

[3] N. Audsley, K. Tindell, A. Burns. The End of the Line for Static Cyclic Scheduling? Proc. of the Fifth Euromicro Workshop on Real-Time Syst., pp.36-41, 1993.

[4] N. C. Audsley. Flexible Scheduling of Hard Real-Time Systems. Department of Computer Science Thesis, University of York, 1994.

[5] R. Bettati, J. W.-S. Liu. End-to-End Scheduling to Meet Deadlines in Distributed Systems. Proc. of the 12th Intern. Conf. on Distributed Computing Systems, pp. 452-459, june 1992.

[6] A. Burns, M. Nicholson, K. Tindell, N. Zhang. Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System. Proceedings of the Workshop on Parallel and Distributed Real-Time Systems,pp.11-20, 1993.

[7] S. Chatterjee, J. Strosnider. Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems. Proceedings of the 15th International Conf. on Dist. Computing Systems, may 1995.

[8] R. Gerber, S. Hong, M. Saksena. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. Proceedings of the IEEE Real-Time Systems Symp., dec. 1994.

[9] M. G. Harbour, M. H. Klein, J. P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. IEEE Trans. on Soft. Eng., Vol.20, No.1, pp.13-28, jan. 1994.

[10] J. Y. T. Leung, J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. Performance Evaluation, 2 (4), pp. 237-250, december 1982.

[11] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, Vol. 20, No. 1, pp. 46-61, january 1973.

[12] L. Sha, R. Rajkumar, S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. Proceedings of the IEEE, Vol. 82, No. 1, pp. 68-82, january 1994.

[13] J. Sun, R. Bettati, J. W.-S. Liu. An End-to-End Approach to Scheduling Periodic Tasks with Shared Resources in Multiprocessor Systems. Proc. of the IEEE Workshop on Real-Time Operating Syst. and Software, pp. 18-22, 1994.

[14] J. Sun, J. W.-S. Liu. Bounding the End-to-End Response Time in Multiprocessor Real-Time Systems. Proc. Workshop on Par. Dist. Real-Time Syst., pp91-98, Santa Barbara, april 1995.

[15] J. Sun, J. W.-S. Liu. Synchronization Protocols in Distributed Real-Time Systems. Proc. of 16th Intern. Conf. on Dist. Comp. Syst., may 1996.

[16] J. Sun, J. W.-S. Liu. Bounding the End-to-End Response Times of Tasks in a Distributed Real-Time System Using the Direct Synchronization Protocol. Tech. Report UIUC-DCS-R-96-1949, Univ. of Ill. at Urbana-Champaign, june 1996.

[17] K. W. Tindell, A. Burns, A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Real-Time Systems, pp. 133-151, 1994.

[18] K. Tindell, J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. Microproc. and Microprog., 40, 1994.