# Run-Time Feasibility of Hard Real-Time Systems Containing Coupled Tasks*

A.W. Krings
Computer Science Dept.
University of Idaho
krings@cs.uidaho.edu

M.H. Azadmanesh
Computer Science Dept.
University of Nebraska at Omaha
azad@cmit.unomaha.edu

**Abstract**

*This paper investigates the problem of guaranteeing stability and run-time feasibility in real-time systems containing coupled tasks, in the context of non-preemptive priority scheduling. Instability is the result of so-called multiprocessor timing anomalies, where deadlines can be missed due to the reduction in task durations. Such reductions can also result in run-time infeasibility of coupled task pairs due to the inherent inter-task timing constraints. A scheduling environment, feasibility conditions and a general algorithm are presented that avoid both phenomena at run-time.*

## 1 Introduction

Many real-time control applications are operating in multiprocessor environments to take advantage of parallelism of the workload, or as a redundancy issue for fault-tolerant reasons. In hard real-time systems, computations of individual tasks are marked by task deadlines and inter-task timing constraints. In safety critical environments, violation of deadlines or timing constraints could have catastrophic results, i.e. loss of human lives, environmental damage or unacceptable cost. The algorithms scheduling the workload, typically represented by a task graph, must be provably correct and free of side effects.

Most real-time applications only include a small number of tasks with critical inter-task timing dependencies [3]. In the context of this paper such tasks are referred to as *coupled tasks*, since the execution of one task is coupled to the execution of another task by a fixed coupling delay. Other frequently used terminologies to describe coupled tasks are end-to-end and temporal distance constraint tasks. Examples of coupled tasks are: a delay of an actuator movement to compensate for mechanical movement of target objects, two messages that must be send within fixed intervals from each other, or navigation coordinates that must be updated at a fixed time after a course correction.

Different aspects of coupled events have been studied. Some research focused on scheduling based on the pinwheel problem [2, 6, 7]. Most work, including [3, 4, 23], address end-to-end constraints in the context of periodic processes. One way of dealing with coupled events has been to adopt automated design methods using reconstructing tools [4], or letting the scheduler adapt itself to varying execution times [21]. Methods of validating timing constraints for different scheduling environments are discussed in [5, 15, 16]. Real-world applications considering inter-task timing constraints are described in the context of projects such as the Spring Kernel [19] or the GMD-Snake robot [20].

This paper considers dispatching in systems containing coupled task-pairs which are embedded in a normal workload. Section 2 describes the scheduling environment and the problem of instability and infeasibility. The basics of run-time stabilization to avoid instability is discussed in Section 3. Section 4 specifies feasibility conditions and a general stable run-time dispatching algorithm. Finally, Section 5 concludes the paper with a summary.

## 2 The Scheduling Environment

### 2.1 Task Model

In general, a task $T$ is the basic unit of computation, consisting of a set of sequentially executed instructions. Associated with each task $T_i$ is a maximum and minimum computation time $c_i^{max}$ and $c_i^{min}$, release time $r_i$ at which the task becomes *ready* for execution, starting time $s_i$, and finishing time $f_i$. Dependencies among tasks are defined by a partial order, resulting in a directed acyclic precedence graph. Tasks are assumed to be executed on $M$ homogeneous processors.

*This work has been supported in part by the UI Microelectronic Research Center (MRC).

Coupled tasks are considered in pairs of tasks as shown in Figure 1. The first task, $T_i^p$, is the parent, coupled by a coupling delay $d_{ij}$ to the child task, $T_j^c$. Thus, the coupling delay constitutes an implicit precedence constraint between $T_i^p$ and $T_j^c$. The coupling is considered to be "simple" in that the child task $T_j^c$ has an in-degree of 1. Thus, $T_j^c$ has only one predecessor, namely the mechanism (called "enforcer phantom") that constitutes the coupling to parent $T_i^p$, as will be described in Section 4. This constraint reflects considerations of applicability of the concept of coupled events. Furthermore, each parent is assumed to have only one coupled child. This second constraint serves only as a simplification of the material presented here. However, the solutions presented can be modified to overcome this constraint. Given a coupled task pair $(T_i^p, T_j^c)$, several types of couplings can be defined, based on whether task starting times, task finishing times, or combinations thereof are considered. This research focuses on coupling of task starting times, i.e. $d_{ij} = s_j - s_i$, since task durations may vary at run-time. However, the approaches described here can be modified to reflect other couplings, i.e. starting-to-finishing times or finishing to finishing times. Coupled tasks are differentiated from *regular tasks*, i.e. task in the regular sense, because they have special properties inherent to their coupling as will be described later.

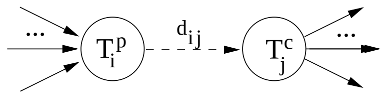


Figure 1: Coupled Task Pair

Special tasks called *phantom tasks* have been used to model events external to a processor, such as delayed task release, non-transparent overhead or task synchronization [9, 12, 14]. These tasks are fully incorporated into the precedence graph. Although they consume time, unlike regular tasks, they consume no resources. As a result, phantom tasks may *always* be started upon becoming released. Phantom tasks will be the basic mechanism for controlling coupled tasks at run-time, enforcing the coupling delay.

## 2.2 Definitions

The algorithms described in this paper are based on a variation of *priority list scheduling*, where whenever a processor becomes available, the run-time *dispatcher* scans the task list from left to right, and the first unexecuted ready task encountered in the scan is assigned to the processor. The dispatcher is distinct from the scheduling algorithm. Whereas the scheduler is executed only once at design time, the dispatcher arbitrates tasks during run-time.

A *Standard Scenario* describes the schedule obtained by using a particular set of task durations. It denotes a schedule in which each $T_i$ uses the maximum computation time $c_i^{max}$ [17]. The Gantt chart depicting the standard scenario is called the *Standard Gantt Chart* (SGC).

In a *Non-Standard Scenario*, tasks $T_i$ execute with $c_i^{min} \leq c_i \leq c_i^{max}$. However, at least one $T_j$ has duration $c_j$ less than its maximum computation time $c_j^{max}$, i.e. $c_j < c_j^{max}$. The resulting Gantt chart is called *Non-Standard Gantt Chart* (NGC).

The dispatcher selects tasks from a list called *projective list*. This list is in one-to-one correspondence with the SGC, i.e. its tasks are ordered according to the time each task is picked up on the SGC [17].

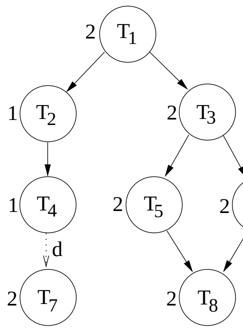
A scenario is *stable* if there exists no scenario in which the finishing time of any $T_i$ in the NGC exceeds its completion time on the SGC. With non-standard computation times not known apriori, i.e. $c_i^{min} \leq c_i \leq c_i^{max}$, given any task $T_i$, the "deadline" for $s_i$ is $s_i^{std}$, the starting time on the SGC as denoted by superscript *std*. Thus, if $s_i \leq s_i^{std}$, then $f_i \leq f_i^{std}$.

Several task sets will be used throughout the paper. Let $\mathbf{T}_{<i}$ denote the set of all tasks which started before $T_i$ on the SGC, i.e. $\mathbf{T}_{<i}$ is the set of tasks with indices less than $i$. $\mathbf{T}_{\leq i}$ is defined as $\mathbf{T}_{<i} \cup \{T_i\}$. Sets $\mathbf{T}_{>i}$ and $\mathbf{T}_{\geq i}$ are symmetric to $\mathbf{T}_{<i}$ and $\mathbf{T}_{\leq i}$ respectively. Let $\mathbf{T}^p$ and $\mathbf{T}^c$ denote the set of all coupled parent and child tasks $T_i^p$ and $T_j^c$ respectively. Then $\mathbf{T}^{pc} = \mathbf{T}^p \cup \mathbf{T}^c$ represents the set of all coupled task pairs.

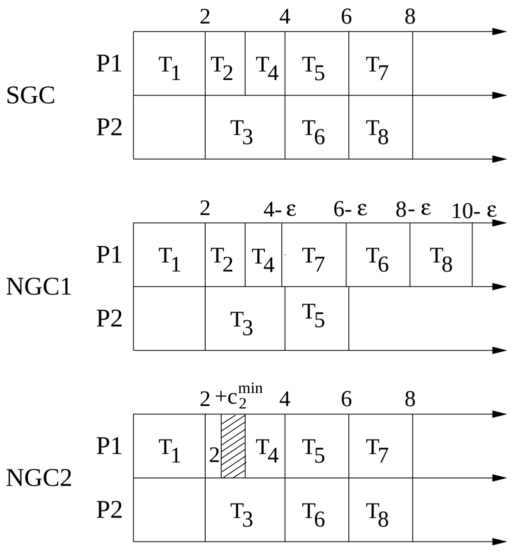
In a given scenario, a task $T_v$ is *unstable* if and only if it is the lowest numbered task to start late, i.e. $s_v > s_v^{std}$, and $s_i \leq s_i^{std} \; \forall \; T_i \in \mathbf{T}_{<v}$. Task $T_v$ is *vulnerable* to instability if there exists *any* scenario in which $T_v$ is unstable.

## 2.3 Instability and Infeasibility

Instability and infeasibility will be demonstrated using the example in Figure 2. The precedence graph contains eight tasks with maximum durations listed next to each vertex. To show instability, the edge between $T_4$ and $T_7$ is to be interpreted as a precedence constraint. Task priorities are defined in order of increasing starting times on the dual-processor SGC in Figure 2b. During execution, the dispatcher scans the projective list and selects the first ready task for execution. Scheduling instability can be observed on NGC1, where $T_4$ is shortened by an arbitrarily small value $\epsilon$. The shortened $T_4$ finished before $T_3$, and $T_7$ was then able to "usurp" processor P1. The results are missed deadlines and an increase in total makespan, i.e. both $T_6$ and $T_8$ started later on NGC1 than they did on the SGC and the makespan increased by $2 - \epsilon$.



a) Precedence Graph



b) Gantt Charts

Figure 2: Example of Instability

In order to demonstrate infeasibility, assume the edge between $T_4$ and $T_7$ indicates a coupling delay with $d_{4,7} = s_7^{std} - s_4^{std} = 3$. Now, assume that $T_2$ finished at $f_2 = s_2^{std} + c_2^{min}$, as shown in NGC2 of Figure 2b. At $f_2$ task $T_4$ becomes ready, but dispatching $T_4$ implies that $T_7$ has to be shifted as well due to the coupling delay. However, such shift of $T_7$ is infeasible, since both processors are occupied by $T_5$ and $T_6$. As a consequence, the coupling delay would be violated. Thus, although $T_4$ is ready, it cannot be dispatched in order to avoid infeasibility of $T_7$. Infeasibility of course results in instability. For instance, dispatching $T_4$ in NGC2 would have caused $T_7$ to start late.

A dispatching algorithm must avoid both run-time instabilities and infeasibilities. To avoid instabilities, two stabilization methods have been proposed that can be partitioned into *apriori* and *run-time* stabilization.

1. In *Apriori Stabilization* methods, stabilization is achieved by (1) restricting the dispatcher, i.e. fixing the task starting sequence or task starting times, or by (2) modifying the task graph by introducing additional precedence constraints [8, 17, 18, 22].

2. *Run-Time Stabilization* is a less restrictive stabilization method, where the dispatcher limits the depth of its scan into the task list in order to avoid instabilities. This approach takes advantage of information available at run-time [12, 13, 14].

Apriori stabilization is not equipped to deal with infeasibilities efficiently. However, it will be shown that a new variation of run-time stabilization can prevent both instability and infeasibility.

## 3 Basic Run-Time Stabilization

This section addresses issues of run-time stabilization in the absence of coupled task pairs, i.e. $\mathbf{T}^{pc} = \phi$. Thus, the task systems consists of non-coupled real tasks and phantom tasks. Stable solutions for such task model have been presented in [10, 11, 12, 13, 14]. The main concept will be repeated here, as it builds the basis for avoidance of infeasibility, as described in Section 4. The first step in the chain of events possibly leading to instability is a *priority inversion* by some task $T_x$, such that $s_x < s_i$ for some $T_i$ with $i < x$ [10]. Task $T_x$ is said to *usurp*. It is the responsibility of the run-time stabilization algorithm to only allow dispatching of such usurper tasks that cannot induce instability.

### 3.1 The Scan Window

When a processor finishes its current task, the traditional priority list dispatcher starts at the head of the list, scanning the list until it finds a ready task, if one exists. The scan window approach restricts the scan by limiting the scan depth to the size of a window. The *scan window* $\mathbf{\Sigma}$ is this subset of *unstarted* real tasks scanned by the dispatcher, i.e. $\mathbf{\Sigma} = \{T_u, ..., T_l\}$ where $T_u$ and $T_l$ are the first and last real tasks visible to the dispatcher. If the number of tasks scanned is limited such that no usurper task is ever started before a vulnerable task, stability can be enforced at run-time.

## 3.2 Fan-out

A *fan-out* occurs when a phantom task releases a real task, or when a real task causes the release of a second real task executing in parallel. The initiating release is called a *logical fork*, since either case causes the occupancy of one more processor, i.e. from 0 to 1 and from 1 to 2 processors for phantom and real forks respectively. Fan-out, caused by logical forks, is a necessary condition for instability to occur. It will be shown later that this is not generally true if $\mathbf{T}^{pc} \neq \phi$ as can be seen in NGC2 of Figure 2b.

A *fan-out task* is defined as a real task with *at least* one of the following properties: (1) it is descendent from an unfinished forking task, and it started executing on the SGC while another descendent of the same forking task was executing on another processor, (2) it is a descendent of an unfinished phantom task.

The second property deserves some explanation. To show the effect of a fan-out caused by a phantom task consider the scenario in Figure 3a, where the phantom task $T_p$ has real descendent $T_\alpha$. Figure 3b shows that a phantom task can be thought of as a phantom $T_p$ running on a phantom processor $P^p$, i.e. an imaginary processor. Upon finishing, $T_p$ releases a phantom child $T_{p'}$ and real task $T_\alpha$, which are dispatched on phantom processor $P^p$ and real processor $P1$ respectively. Thus, property (2) of the fan-out definition above is essentially the same as property (1), except that the existence of the phantom child $T_{p'}$ causing the fan-out of $T_\alpha$ might not be obvious.

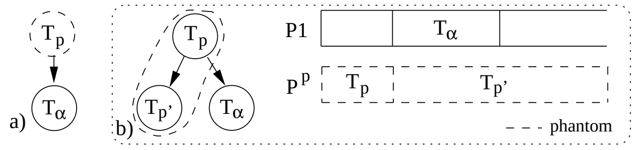


Figure 3: Phantom Task Model

Let $T_w$ be the first fan-out task and $T_u$ the first task in the scan-window. On the SGC, $T_w$ was the task that occupied an additional processor, i.e. $T_w$ caused a fan-out of 1. It can be shown that one can safely scan up to $T_w$ [11]. This defines a scan window $\mathbf{\Sigma} = \{T_u, ..., T_{w-1}\}$. Assume that $T_w$ is the only fan-out task in the workload. If one wants to scan past $T_w$, stability is guaranteed[1] only provided there is at least one idle processor that can be reserved to *absorb* the possible fan-out of $T_w$. In general, to scan past $T_w$, one needs to identify additional fan-out tasks in $\mathbf{T}_{>w}$. Let $T_{w1} = T_w$, and define $T_{wi}$ as the $i^{th}$ fan-out task. All $T_{wi}$ are called *basic fan-out tasks*, in that each $T_{wi}$ can cause a fan-out of 1.

[1]At this point we ignore issues of slack-time reclaiming and fan-in.

### 3.2.1 Effective Fan-out

Let $\mathcal{F}(T_w)$ be a function that indicates how many basic fan-out tasks with indices less than or equal $w$ are overlapping on the SGC at standard starting time $s_w^{std}$. Thus $\mathcal{F}(T_w)$ is the cardinality of set $\{T_i : T_i \in \mathbf{T}_{\leq w}, T_i$ *is a fan-out task, and* $s_i^{std} \leq s_w^{std} < f_i^{std}\}$. A task $T_w$ is said to have an *effective fan-out* of $\mathcal{F}(T_w)$.

Not every basic fan-out task contributes to an increase in the effective fan-out. Assume that several basic fan-out tasks exist such that their executions do not overlap on the SGC, i.e. $f_{wi}^{std} < s_{wj}^{std}$ for any $T_{wi}$ and $T_{wj}$, with $i < j$. It can be shown [12, 13] that these *non-overlapping* basic fan-out tasks can collectively contribute only to an effective fan-out of 1. Let $T_{ei}$ denote the *lowest numbered* basic fan-out task with $\mathcal{F}(T_w) = i$. Then $T_{ei}$ is called an *effective fan-out task*. $T_{ei}$ is thus the first task executing *in parallel* with $i-1$ other fan-out tasks from $\mathbf{T}_{<i}$. $T_{e1}$ is the first effective fan-out task ($\mathcal{F}(T_{e1}) = 1$), $T_{e2}$ is the second ($\mathcal{F}(T_{e2}) = 2$), and so forth. Every effective fan-out task is also a basic fan-out task, but the reverse is not necessarily true. It should be noted that $T_{ei}$ is not necessarily the only fan-out task with a fan-out of $i$, but it is the *first*. As a convention, task subscripts starting with letter $e$ will be reserved for effective fan-out tasks.

### 3.2.2 Scan Frames

The priority list can be partitioned starting with the first unstarted task. The general priority list at the time of the scan is $(T_{e0}, ..., T_{e1}, ..., T_{e2}, ..., T_{ek}, ...)$. Task $T_{e0} = T_u$ if[2] $\mathcal{F}(T_u) = 0$, otherwise $T_{e0}$ does not exist and the list starts with $T_{e1}$. $T_{ek}$ is the last effective fan-out task. Positioned between $T_{ei}$ and $T_{e(i+1)}$ are any number of tasks $T_j$ with effective fan-outs $0 \leq \mathcal{F}(T_j) \leq i$. These tasks, including $T_{ei}$, are called the *Scan-Frame* of $T_{ei}$ and are denoted by $\mathbf{\Delta}_{ei}$. Thus frame $\mathbf{\Delta}_{ei}$ is the set $\{T_{ei}, ..., T_{e(i+1)-1}\}$. The definition of scan-frames is with respect to the current scan. In general, scan-frames have to be newly defined whenever a fan-out task is released that causes a decrease in the effective fan-out of some $T_{ei}$, the task defining $\mathbf{\Delta}_{ei}$.

### 3.2.3 Dispatching Philosophy

With the knowledge of idle processors at the time of the scan, the safe scan window can now be expressed as a

[2]$\mathcal{F}(T_u) \neq 0$ if and only if $T_u$ is descended from a phantom task.

sequence of frames. Let $I_r \leq I - 1$ be the number of idle processors reserved at the time of the scan, leaving one processor to start the ready task being searched for. Then it can be shown [12, 14] that the safe scan window is $\mathbf{\Sigma} = \mathbf{\Delta}_{e0} \cup \mathbf{\Delta}_{e1} \cup ... \cup \mathbf{\Delta}_{eI_r}$. If $\mathbf{\Delta}_{eI_r}$ does not exist, then $\mathbf{\Sigma}$ extends over the entire task list.

## 4 Dispatching with Coupled Tasks

Task couplings imply temporal bindings of tasks $T_i^p$ and $T_j^c$ that are defined according to the starting times of the SGC. Returning to NGC2 of Figure 2b, one can observe that, if the dispatcher does not prevent early starting of parent $T_4^p$, infeasibility results, i.e. child task $T_7^c$ misses its deadline. Early starting of coupled tasks in the absence of stable dispatching algorithms may result in the following problems: (1) parent task $T_i^p$ may induce instability, and (2) corresponding child $T_j^c$ may be subject to infeasibility, and may cause instability.

### 4.1 Trivial Solution

One way of preventing infeasibility is to simply prevent any tasks in $\mathbf{T}^{pc}$ from starting early. This can be modeled by defining an *enforcer phantom* task $T_{pi}$ as a predecessor for each $T_i \in \mathbf{T}^{pc}$ as can be seen in Figure 4, setting $s_{pi} = 0$ and $c_{pi}^{max} = c_{pi}^{min} = s_i^{std}$. Any scan window algorithm, e.g. those presented in [12, 13, 14], can now be applied to this modified task system. However, this approach is very inefficient because the actual task durations of many real-time applications are much smaller than their maximum, standard durations, e.g. up to an order of magnitude shorter [1]. As a result, the response time of coupled tasks is always worst case, i.e. fixed as defined in the SGC, whereas response to non-coupled tasks improves drastically as the schedule compacts.

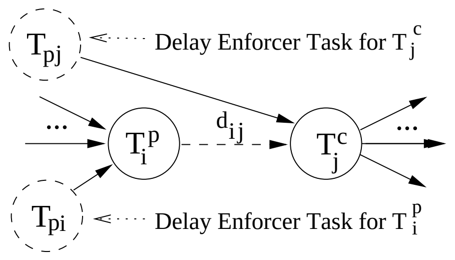


Figure 4: Delay Enforcer Phantom Tasks

### 4.2 General Solution

In the general approach only the starting of tasks in $\mathbf{T}^c$ is enforced by phantom tasks. However, a task $T_i^p \in \mathbf{T}^p$ can be started early *only* if its corresponding $T_j^c$ can be guaranteed the same shift in the future, without causing instability. This actually constitutes a "promotion", i.e. a left shift of $T_j^c$ on the SGC, together with the appropriate adjustment of the corresponding enforcer phantom's duration. This is fundamentally different from earlier run-time stabilization methods, as now task priorities generally will not be static anymore, i.e. the priority list order may change. In the following, appropriate adjustment of enforcer phantom task durations for tasks in $\mathbf{T}^c$ to reflect a promotion is implied and will not be explicitly mentioned. In the outline of the general run-time stabilization algorithm below, the term *standard algorithm* denotes any stable run-time stabilization algorithm used for workloads without coupled tasks. Scheduling workloads containing coupled tasks involves the following steps:

1. Non-coupled tasks and tasks from $\mathbf{T}^c$ are scheduled using standard stabilization algorithms.

2. Tasks $T_i^p \in \mathbf{T}^p$ are scheduled using standard run-time stabilization algorithms if the following is true for the corresponding $T_j^c$:

   C1: $T_j^c$ can be promoted into a vacant slot on the SGC.

   C2: The promotion of $T_j^c$ does not take over a processor that was reserved to compensate for fan-out in scan frames overlapping with the execution of currently executing usurper tasks.

Some explanations are needed for the two conditions. With respect to C1, as tasks finish, their corresponding SGC slots become vacant. Promotion into a vacant slot provides the basis for feasibility with respect to guaranteeing the coupling delay, but does not guarantee stability. Condition C2 indicates that just because the time slot on the SGC was vacant does not eliminate the possibility that this processor is reserved. The processor could be mortgaged to compensate for fan-outs due to previous usurpion.

#### 4.2.1 Standard Frame Based Dispatching Algorithm

Recall that "standard" implies the absence of coupled tasks. As indicated before, the scan window $\mathbf{\Sigma}$ can be expressed in terms of scan frames. Scan frame $\mathbf{\Delta}_{ei}$ is the set $\{T_{ei}, ..., T_{e(i+1)-1}\}$, and $\mathbf{\Sigma} = \mathbf{\Delta}_{e0} \cup \mathbf{\Delta}_{e1} \cup ... \cup$

$\mathbf{\Delta}_{eI_r}$, where $I_r$ indicates the number of reserved processors. When scanning the priority list, assume that standard task $T_x$ is the next ready task checked for safe starting. Checking for tasks vulnerable to instability, it can be shown that the only scan frames that need to be investigated are those which contain tasks $T_v$ whose SGC starting time $s_v^{std}$ overlap time-wise with the execution of usurper task $T_x$ on the NGC, assuming $T_x$ were started [12]. This means that scan frames $\mathbf{\Delta}_{ej}$ with $s_{ej}^{std}$ greater than the maximal finishing time of $T_x$ cannot be vulnerable to instability caused by starting $T_x$. Thus, a scan window algorithm has to reserve processors only for the frames whose associated $T_{ei}$ starts at or before the maximum finishing time of $T_x$, since safety of the succeeding frames follows.

**E-Algorithm** The following standard algorithm called *E-Algorithm*, restated from [12, 13], will be the basis for general frame based dispatching with coupled tasks.

1. Find the first ready task $T_x$ and determine its scan-frame $k'$.
2. Find the last task $T_v$ with index $v < x$ whose standard starting time overlaps the hypothetical execution of $T_x$ and find its scan frame $k$.
3. Then, $T_x$ can be safely started if $k$ idle processors can be reserved for tasks from $\{\Delta_{e0} \cup \ldots \cup \Delta_{ek}\}$.

### 4.2.2 General Frame Based Dispatching Algorithm

To include coupled tasks, the E-Algorithm needs to be modified in order to identify tasks from $\mathbf{T}^p$ and to account for conditions C1 and C2. Let $T_j^c$ be the coupled child corresponding to parent $T_x^p$. Furthermore, let $t_n$ denote the time of the scan, and let $\tilde{s}_j^{std}$ denote the standard time $T_j^c$ would have to be promoted to in order to satisfy the coupling, i.e. $\tilde{s}_j^{std} = s_j^{std} - (s_x^{std} - t_n)$. Next, assume that all tasks that have been started are marked on the SGC. This includes tasks already finished. Let $\mathcal{U}(t)$ indicate the number of unmarked tasks on the SGC at time $t$. Furthermore, let $\mathcal{E}(t)$ be the number of tasks $T_i$ that are currently executing on a processor for which $f_i^{max} > t$. Now, the following *Feasibility Conditions* can be formulated:

FC1: $T_j^c$ can be promoted into a vacant slot on the SGC for the entire Feasibility Interval $\mathbf{FI}_j = [\tilde{s}_j^{std}, \tilde{s}_j^{std} + c_j^{max}]$.

FC2: For each fan-out task $T_w$ whose standard starting time is in $\mathbf{FI}_j$, the number of processors assigned to unmarked tasks plus the number of processors occupied by currently executing tasks $T_i$ with $f_i^{max}$ in $\mathbf{FI}_j$ is less than or equal to $M-1$ at $s_w^{std}$. Formally, for every fan-out task $T_w$ with $s_w^{std}$ in $\mathbf{FI}_j$:

$$\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - 1. \qquad (1)$$

**GE-Algorithm** Now the *General E-Algorithm*, as an extension of the E-Algorithm utilizing the Feasibility Conditions, can be stated:

1. Find the first ready task $T_x$ and determine its scan-frame $k'$.
2. Find the last task $T_v$ with index $v < x$ whose SGC starting time overlaps the hypothetical execution of $T_x$ and find its scan frame $k$.
3. If $T_x \in \mathbf{T}^p$ then $T_x$ can be safely started if $k$ idle processors can be reserved for tasks from $\{\Delta_{e0} \cup \ldots \cup \Delta_{ek}\}$ and feasibility conditions FC1 and FC2 are met.
4. Else, $T_x$ can be safely started if $k$ idle processors can be reserved for tasks from $\{\Delta_{e0} \cup \ldots \cup \Delta_{ek}\}$.

It should be noted that in order to allow multiple child tasks to be coupled to a single parent task one only has to change the algorithm to account for the additional child's promotion. This will require a modification of the Feasibility Conditions to reflect the additional promotions and the number of processors in inequality (1).

## 4.3 Proof of Stability

The E-Algorithms has been proven stable for standard workloads in [12]. In order to prove stability of the GE-Algorithms in the presence of coupled tasks, it needs to be shown that the inclusion of the Feasibility Conditions avoids instability. First we restate a lemma from [14] that shows that only tasks whose standard starting times overlap with the execution of usurper task $T_x$ on the NGC, need to be considered as potentially vulnerable tasks in order to guarantee stability.

**Lemma 1** *Assume that a usurper task $T_x$ has started at time $s_x$, and define $f_x^{max} = s_x + c_x^{std}$. No task $T_v$ with $s_v^{std} > f_x^{max}$ can become unstable as a result of starting $T_x$.*

**Proof:** See [14, Lemma 3]. $\square$

**Theorem 1** *A task $T_x$ can be safely promoted on the SGC from $s_x^{std}$ to some $\tilde{s}_x^{std}$ with $\tilde{s}_x^{std} < s_x^{std}$, if Feasibility Conditions FC1 and FC2 hold.*

**Proof:** If FC1 does not hold, then at some time in the Feasibility Interval $\mathbf{FI}_x = [\tilde{s}_x^{std}, \tilde{s}_x^{std} + c_x^{max}]$ processor contention can occur. This will lead to instability, unless fan-in can be guaranteed. However, the cost of guaranteeing fan-in is exponential in the number of tasks [12] and thus not real-time feasible. Therefore assume that FC1 holds.

In order to prove the necessity of FC2 we first show that only tasks in $\mathbf{FI}_x$ need to be considered. For a given real task $T_v$, three General Instability Conditions (GIC1 - GIC3) have been derived that are both necessary and sufficient for $T_v$ to be unstable [9]. GIC1 indicates that priority inversion is a necessary condition for instability to occur (see also [17]). However, for $T_v$ with $s_v^{std} < \tilde{s}_x^{std}$ the promotion of $T_x$ does not constitute a priority inversion. This implies invulnerability of $T_v$ from GIC1.

For tasks in $\mathbf{T}_{<x}$ with starting times after $\mathbf{FI}_x$, i.e. for $T_v$ with $s_v^{std} > \tilde{s}_x^{std} + c_x^{max}$, promotion of $T_x$ does constitute a priority inversion. However, the effect of a usurper task is limited to those tasks overlapping on the SGC with the execution of the usurper. Invulnerability of tasks $T_v$ with standard starting times beyond $\mathbf{FI}_x$ follows from Lemma 1.

Next it will be shown that tasks $T_v$ with starting times in $\mathbf{FI}_x$ are invulnerable if FC2 holds. Let $\mathbf{T}^{FI_x}$ denote the set of tasks $T_v$ with $s_v^{std}$ in $\mathbf{FI}_x$. Now assume that FC2 is true. Let $f_\phi$ be the time of the last fork into $\mathbf{T}_{\leq v}$ to occur at or before ready time $r_v$. General Instability Condition GIC3 states that for $T_v$ to be unstable, there must exist *no* time in the interval $[f_\phi, s_v^{std}]$ at which all ready real tasks in $\mathbf{T}_{\leq v}$ are running. Finishing of $T_{\phi_i}$ causes the fan-out for fan-out tasks $T_{wi}$. However, according to inequality (1) in FC2, at each $s_{wi}^{std}$ there is a processor available for each unstarted task, independent of currently executing tasks. Thus at time $s_w^{std}$ latest, a processor is available for each task in $\mathbf{T}_{<w}$ and GIC3 cannot hold. Tasks $T_{v'}$ with $s_{v'}^{std} \neq s_{wi}^{std}$, for some $T_{wi}$ with $s_{wi}^{std}$ in $\mathbf{FI}_x$, need not be considered, since no new fan-out is introduced, and tasks in $\mathbf{T}_{<v'}$ are safe by assumption. $\square$

**Theorem 2** *The GE-Algorithm is stable.*

**Proof:** In the absence of coupled tasks, the GE-Algorithm degenerates into the E-Algorithm which has been proven stable [12, Theorem 5]. Including coupled tasks adds the child task promotion issue. However, from Theorem 1 instability can not result from a promotion if the Feasibility Conditions FC1 and FC2 hold. $\square$

## 5 Summary

This paper addressed the problem of instability and infeasibility of coupled tasks in non-preemptive priority list scheduling. Task couplings are assumed to consist of task pairs, where parent tasks are coupled to child tasks by fixed coupling delays. The task system allows for regular tasks, coupled tasks and phantom tasks. Task coupling is implemented using mechanisms of the latter type, so-called delay enforcement phantom tasks.

When task durations are specified with minimum and maximum run-times, early starting of parent tasks in a coupled pair can result in scheduling infeasibility and thus instability at run-time. A trivial method is presented that prevents run-time infeasibility. However, this method makes the response time to coupled tasks always maximal, whereas the rest of the workload compresses as typical actual task durations are much smaller than their standard durations. In order to allow the early starting of coupled tasks, child tasks in the coupled pair have to be promoted, i.e. shifted left on the Standard Gantt Chart (SGC). General Feasibility Conditions have been defined that are sufficient for promoting a task on the SGC.

A general run-time stabilization algorithm is presented that implements scan-window dispatching in the presence of coupled tasks. The algorithm is based on scan-frames and uses the General Feasibility Conditions to allow stable early dispatching of coupled tasks.

## References

[1] Carpenter, K., et al., "ARINC 659 Scheduling: Problem Definition", *Proc. IEEE Real-Time Systems Symposium*, pp. 165-169, 1994.

[2] Chan, Mee Yee, and Francis Y.L. Chin, "General Schedulers for the Pinwheel Problem Based on Double-Integer Reduction", *IEEE Transactions on Computers.*, Vol. 41, No. 6, pp. 755-768, June 1992.

[3] Gerber, Richard, S. Hong, and M. Saksena, "Guaranteeing Real-Time Requirements With Resource-Based Calibration of Periodic Processes", *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, pp. 579-592, July 1995.

[4] Gerber, Richard, D. Kang, S. Hong, and M. Saksena, "End-to-End Design of Real-Time Systems", *UMD Technical Report CS-TR-3476, UMIACS TR 95-61*, May 1995.

[5] Ha, Rhan, and Jane .W.S. Liu, "Validating Timing Constraints in Multiprocessor and Distributed Real-Time Systems", *Proc. IEEE* 14th *International Conference on Distributed Computing Systems*, 1994.

[6] Han Ching-Chih, and K.J., Lin, "Scheduling Distance-Constrained Real-Time Tasks", *IEEE Real-Time Systems Symposium*, pp. 300-308, 1992.

[7] Hsueh, Chih-wen, Kwei-Jay Lin, and Nong Fan, "Distributed Pinwheel Scheduling with End-to-End Timing Constraints", *Proc.* 16th *IEEE Real-Time Systems Symposium*, pp. 172-181, 1995.

[8] Kieckhafer, R.M., et al, "The MAFT Architecture for Distributed Fault-Tolerance", *IEEE Trans. Computers*, V. C-37, No. 4, pp. 398-405, April, 1988.

[9] Kieckhafer, R.M., and J.S. Deogun, " On the Stability of List Scheduling in Real-Time Multiprocessor Systems", UNL, Dept. of Comp. Sci., Report #99, Feb 1990.

[10] Kieckhafer R.M, J.S. Deogun and A.W. Krings, "The Performance of Inherently Stable Multiprocessor List Schedulers", UNL, Dept. of Comp. Sci., Report Series UNL–CSE–92–009, May 1992.

[11] Krings, A.W., and R.M. Kieckhafer, "Inherently Stable Priority List Scheduling in Systems with External Delays ", *Proc. Twenty-Sixth Annual Hawaii International Conference on System Sciences*, Vol. 2, pp. 622-631, 1993.

[12] Krings, A.W., "Inherently Stable Priority List Scheduling in an Extended Scheduling Environment", *PhD Thesis*, Dept. of Comp. Sci., Univ. of Nebraska, Lincoln, 1993.

[13] Krings, A.W., R. Kieckhafer, and J. Deogun, "Inherently Stable Real-Time Priority List Dispatchers", *IEEE Parallel & Distributed Technology*, pp. 49-59, Winter 1994.

[14] Krings, A.W., and M.H. Azadmanesh, *Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads*, Proc. 30th Hawaii International Conference on System Science, IEEE Computer Society Press, Vol I, pp. 616-625, 1997.

[15] Liu, Jane W.S., and Rhan Ha, "Efficient Methods for Validating Timing Constraints in Multiprocessor and Distributed Systems", *Proc. Proc.* 4th *Systems Reengineering Technology Workshop*, 1994.

[16] Liu, J.W.S., and Rhan Ha, "Efficient Methods for Validating Timing Constraints", *Advances in Real-Time Systems*, Prentice Hall, 1995.

[17] Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *JACM*, Vol. 14, No. 3, July 1967.

[18] McElvaney, M.C., et. al., "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *Journal of Real-Time Systems*, Vol. 3, No. 3, Sep 1991.

[19] Natale, Marco Di, and J.A. Stankovic, "Dynamic End-to-end Guarantees in Distributed Real Time Systems", *Proc. IEEE Real-Time Systems Symposium*, pp. 216-227, 1994.

[20] K.L. Paap, M. Dehlwisch, B. Klaassen, "GMD-Snake: A Semi-Autonomous Snake-like Robot", Distributed Autonomous Robotic Systems 2, Springer-Verlag, Tokio, 1996.

[21] Saksena, Manas Chandra, "Parametric Scheduling for Hard Real-Time Systems", *PhD Thesis*, Department of Computer Science, University of Maryland, 1993.

[22] Shen, C., et al, "Resource Reclaiming in Real-Time", *Proc. Sixth Real-Time Systems Symposium*, pp. 41-50, Dec 1990.

[23] Sun, Jun, and Jane 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 UIUCDCS-R-96-1949*, University of Illinois at Urbana-Champaign, 1996.