DISPATCHING WITH MULTIPLE TASK COUPLINGS

A.W. KRINGS
Computer Science Dept.
University of Idaho
Moscow, ID 83844–1010, USA

M.H. AZADMANESH
Computer Science Dept.
University of Nebraska at Omaha
Omaha, NE 68182–0500, USA

Abstract – The problem of guaranteeing stability and run-time feasibility in real-time systems containing coupled tasks is addressed in the context of non-preemptive priority list scheduling. Task couplings represent strict timing constraints between a parent task and one or more child tasks with respective coupling delays. Whereas a reduction of task durations can cause instabilities called timing anomalies for non-coupled workloads, the introduction of task couplings can cause additional run-time infeasibility due to the inherent inter-task timing constraints. This paper describes a scheduling environment and presents feasibility conditions for task couplings as well as a general algorithm that avoid instability and infeasibility at run-time.

Index Terms – task scheduling, coupled tasks, hard real-time systems, non-preemptive scheduling

1 INTRODUCTION

The use of multiprocessor systems in real-time applications is motivated by increasing computational workloads or the need for higher reliability by means of redundancy. In hard real-time systems, deadlines are associated with individual tasks, and inter-task timing constraints may further restrict the workload. Failure to meet deadlines or timing constraints may render the applications useless. In the case of safety critical applications violations of timing constraints can lead to catastrophe, e.g. loss of live, environmental damage or unacceptable cost. As a result, the scheduling environment must be reliable and the algorithms scheduling the workload, which is typically represented by a task graph, must be predictable and free of side effects.

Most real-time applications only include a small number of tasks with critical inter-task timing dependencies [5]. 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 one or more tasks by fixed coupling delays. Examples of single task couplings are the delay of an actuator movement to compensate for mechanical movement of target objects, or two messages that must be sent within fixed intervals from each other. An example of multiple task couplings is the flow control of multiple chemicals once a reaction has been initiated.

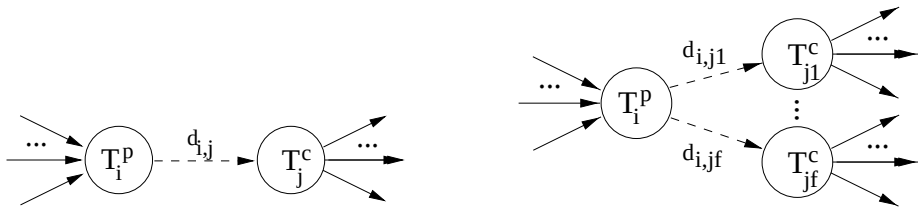
This research is based on the theoretical foundation of single task couplings presented in [12]. It generalizes task couplings by considering multiple couplings. Other research on coupled events has been mainly focused on the problem of scheduling or validating workload with coupled tasks. Scheduling of coupled tasks has been related to the pinwheel problem as well [2, 8]. Most work, including [5, 6, 21], 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 [6], or letting the scheduler adapt itself to varying execution times [19]. End-to-end constraints considering tasks with imprecise computation are presented in [4]. Methods of validating timing constraints for different scheduling environments are discussed in [7, 14]. More practical implications of real-world applications considering inter-task timing constraints are described in the context of projects such as the Spring Kernel [17] or the GMD-Snake robot [18].

We consider issues of dispatching, rather than the generation of efficient initial schedules. It is assumed that at system design time a feasible, not necessarily optimal, schedule is available. 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 addresses multiple task couplings and specifies feasibility conditions and a general stable run-time dispatching algorithm. Finally, Section 5 concludes the paper with a summary.

2 SCHEDULING ENVIRONMENT

2.1 Task Model

In general, a task T is the basic unit of computation, consisting of a set of sequentially executed in-



a) Single Coupling b) Multiple Couplings

Figure 1: Task Couplings

structions. 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.

We first consider the special case of a *single* task coupling as shown in Figure 1a. The first task, T_i^p, is the parent, coupled by a coupling delay $d_{i,j}$ to the child task, T_j^c. A child task T_j^c has an in-degree of 1, i.e. T_j^c has only one predecessor.

The general case, i.e. *multiple* task couplings, is shown in Figure 1b. Again T_i^p is the parent, coupled to multiple child tasks T_{j1}^c to T_{jf}^c by respective coupling delays $d_{i,j1}$ to $d_{i,jf}$. Furthermore, each child task T_{jl}^c, $1 \leq l \leq f$, has T_i^p as its sole predecessor. Multiple child tasks may execute on the same processor and there are no restrictions on f, e.g. f may be greater than M. It is assumed that the set of coupled parent and child tasks are disjoint. This restriction serves only as a simplification of the material presented here. 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. By definition, $\mathbf{T}^p \cap \mathbf{T}^c = \phi$.

Given a coupled task pair (T_i^p, T_j^c), we define coupling with respect to task starting times, rather than considering couplings related to finishing times. The reason for this is the fact that for each task T durations are only known to be between c^{max} and c^{min}. The actual durations are determined at run-time. Coupled tasks are differentiated from *regular tasks*, i.e. tasks in the regular sense, because they have special properties inherent to their coupling as will be described later. Regular and coupled tasks are collectively called *real tasks*.

The basic mechanisms for enforcing the coupling delay at run-time are special tasks called *phantom tasks* [3]. These are tasks which consume time, but unlike real tasks, they consume no resources. As a result, phantom tasks may *always* be started upon becoming released.

2.2 Definitions

Our scheduling model is 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. Traditionally, in list scheduling 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. However, in our paper dispatching may change the schedule. Thus, in general, dispatching and scheduling can not be treated as different operations. The following definitions are restated from [12].

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} [15]. The Gantt chart depicting the standard scenario is called a *Standard Gantt Chart* (SGC). Task deadlines of tasks are the respective finishing times in the 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 [15]. Furthermore, without loss of generality, tasks are assumed to be ordered by increasing indices.

A schedule 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 $f_i \leq f_i^{std}$. Similarly, task couplings are also defined with respect to their standard starting times, i.e. a task coupling from T_i^p to T_j^c is given by $d_{i,j} = s_j^{std} - s_i^{std}$.

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.

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 [12]. 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 ϵ. The shortened T_4 finishes before T_3, and T_7 is able to "usurp" processor P1. The results are missed deadlines and an increase in total makespan, i.e. both T_6 and T_8 start later on NGC1 than they did on the SGC and the makespan increases by $2 - \epsilon$.

In order to demonstrate infeasibility, assume the edge between T_4 and T_7 indicates a coupling delay with $d_{4,7} = s_7^{std} - s_4^{std} = 6 - 3 = 3$. Now, assume that T_2 finishes 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 is violated. Thus, although T_4 is ready, it should not be dispatched in order to avoid infeasibility of T_7. Infeasibility of course results in instability. For instance, dispatching T_4 early in NGC2 causes 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 [12]. 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 STABILIZATION ISSUES

Stable solutions for task models without coupled tasks, i.e. $\mathbf{T}^{pc} = \phi$, have been presented in [10, 11]. These concepts will be the basis for avoidance of infeasibility.

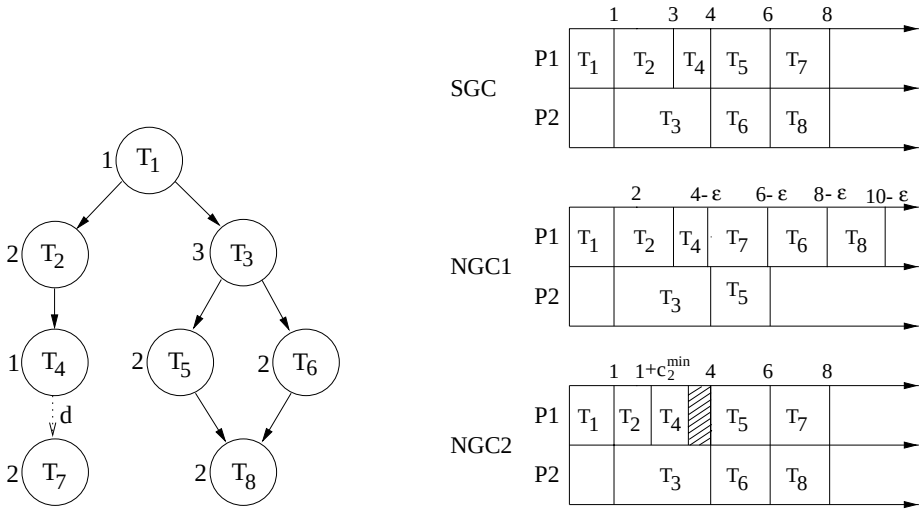
The first step in the chain of events possibly leading to instability is a *priority inversion* by some task T_x, such that $s_x < s_i$ for some T_i with $i < x$ [3]. Task T_x is said to *usurp*. It is the responsibility of the run-time stabilization algorithm to only allow the dispatching of those usurper tasks that cannot induce instability.

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 so-called *scan window*. The scan window $\mathbf{\Sigma} = \{T_u, ..., T_l\}$ is this subset of *unstarted* real tasks scanned by the dispatcher, where T_u and T_l are the first and last real tasks visible to the dispatcher respectively. 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 [10].

At the heart of possible processor contention is so-called 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 with the first real task [12]. The releasing task is called a *forking task* and 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. An example of a forking task is T_3 in Figure 2. Fan-out, caused by logical forks, is a necessary condition for instability to occur [15].

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 [12]. An example of a fan-out task is T_6 in Figure 2.

Let T_w be the first fan-out task in the scan-window. It can be shown that one can safely scan up to T_w [10]. If one wants to scan past T_w, a processor needs to be reserved to *absorb* the possible fan-out of



a) Precedence Graph b) Gantt Charts

Figure 2: Example of Instability and Infeasibility

T_w. In general, 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.

In [12] the concept of fan-out task is extended to *effective fan-out tasks*. 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 $\mathcal{F}(T_w)$. 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.

The priority list can now 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[1] $\mathcal{F}(T_u) = 0$, 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 $\mathbf{\Delta}_{ei}$.

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

4 DISPATCHING WITH MULTIPLE TASK COUPLINGS

Task couplings imply pair-wise temporal bindings of parent and child tasks that are defined according to the starting times of the SGC. Returning to NGC2 of Figure 2, recall that T_4^p and T_7^c are coupled by $d_{4,7} =$ 3 time units. One can observe in NGC2 that, if the dispatcher does not prevent early starting of parent T_4^p, infeasibility results. Here, child task T_7^c cannot satisfy its coupling delay and misses its deadline by $c_2^{max} - c_2^{min}$ time units. Early starting of coupled tasks in the absence of stable dispatching algorithms may result in the following problems: (1) the parent task may induce instability, (2) the corresponding child tasks may be subject to infeasibility, and thus may cause instability.

Recall that $\mathbf{T}^p$ and $\mathbf{T}^c$ are the sets of parent and child tasks respectively. Starting a task in $\mathbf{T}^c$ can be controlled by inserting a phantom task into the workload that enforces the coupling delay. This has been demonstrated for single couplings in [12], where a so-called *enforcer phantom* task T_{pq} was defined as a predecessor for each $T_q \in \mathbf{T}^c$, setting $s_{pq} = 0$ and

$c_{pq}^{max} = c_{pq}^{min} = s_q^{std}$.

A task $T_i^p \in \mathbf{T}^p$ can be started early *only* if *all* 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. Thus our method describes a scheduler rather than a pure dispatcher. 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. Also, it is assumed that the index of the promoted task is adjusted to reflect the task's new position in the projective list.

Let T_j^c be a coupled child corresponding to parent T_x^p. Furthermore, let t_n denote the time of the scan (the time "now"), 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.

Assume that parent task T_i^p is coupled to child tasks T_{jl}^c with $1 \leq l \leq f$. Two feasibility conditions can now be stated.

C1: Each T_{jl}^c, $1 \leq l \leq f$, can be promoted into a vacant slot on the SGC for the entire respective Feasibility Interval $\Phi_{jl} = [\tilde{s}_{jl}^{std}, \tilde{s}_{jl}^{std} + c_{jl}^{max}]$.

Φ_{jl} specifies the time interval needed for feasibility of child task T_{jl}^c. For the second condition some notation is needed. Let $\mathcal{U}(s_w^{std})$ be the number of unmarked tasks on the SGC at s_w^{std}, $\mathcal{E}(s_w^{std})$ the number of tasks T_i that are currently executing (on a processor) for which the maximal finishing times $f_i^{max} > s_w^{std}$, and $\mathcal{O}(s_w^{std})$ the number of feasibility intervals Φ_{jl}, $1 \leq l \leq f$, which are overlapping at standard time s_w^{std}.

C2: For every fan-out task T_w with s_w^{std} in any Φ_{jl}, $1 \leq l \leq f$, we have $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - \mathcal{O}(s_w^{std})$

Thus, for each fan-out task T_w whose standard starting time overlaps with any Φ_{jl}, at s_w^{std}, the number of processors assigned to unmarked tasks plus the number of processors occupied by currently executing tasks T_i with f_i^{max} beyond s_w^{std} is less than or equal to M minus the number of feasibility intervals Φ_{jl} overlapping at s_w^{std}.

Now the *Multiple-Coupling Algorithm*, a generalization of a single-coupling algorithm [12], can be stated:

1. Find the first ready task T_x.

2. Find the last task T_v with index $v < x$ whose SGC starting time overlaps with the hypothetical execution of T_x and find its scan frame $\mathbf{\Delta}_{ek}$.

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 C1 and C2 are met.

4. Else, T_x can be safely started if k idle processors can be reserved for tasks from $\{\Delta_{e0} \cup \ldots \cup \Delta_{ek}\}$.

Due to space restrictions the proof of correctness for the Multiple-Coupling Algorithm is not presented. The interested reader is referred to [13].

Next, we want to address the run-time complexity of the Multiple-Coupling Algorithm. Feasibility condition C1 has a cost of $O(f)$, where f is the maximum coupling degree. The highest complexity is contributed by condition C2. With f feasibility intervals and possibly long task durations C2 has time complexity of $O(fMN)$. This is also the complexity of the algorithm.

5 SUMMARY

This paper has presented a flexible dispatching approach that introduces general Feasibility Conditions to allow coupled tasks, i.e. parent tasks with one or more coupled child tasks, to start early. The conditions address the problem that, when a parent task is started early, all child tasks have to be guaranteed a processor by their individual deadlines induced by their respective coupling delays. This guarantee requires coupled child tasks to be promoted when a parent task starts early, i.e. the standard starting times of child tasks change to reflect the original coupling delay in the presence of the parent's shift on the Gantt chart. However, task promotions can modify a task's priority, which is fundamentally different from static priority list scheduling algorithms. Thus, the solutions described address not only dispatching but also scheduling.

The Feasibility Conditions can be used with many dispatching algorithms. They are presented in the context of a general frame-based scan window approach. The result is a multiple-coupling dispatching algorithm that uses run-time information in order to avoid instability and infeasibility, allowing for early dispatching of coupled tasks.

References

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

[2] Chan, M.Y., and F.Y.L. Chin, "General Schedulers for the Pinwheel Problem Based on Double-Integer Reduction", *IEEE Trans. Computers*, (41)6, June 1992, pp. 755-768.

[3] Deogun, J.S., et. al., "Stability and Performance of List Scheduling With External Process Delays," *Real-Time Systems*, Vol. 15, No. 1, July 1998, pp. 5-39.

[4] Feng, W., and J.W. Liu, "Algorithms for Scheduling Real-Time Tasks with Input Error and End-to-End Deadlines", *IEEE Trans. Software Engineering*, (23)2, 1997, pp. 93-106.

[5] 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, July 1995, pp. 579-592.

[6] 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.

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

[8] Hsueh, Chih-wen, et. al., "Distributed Pinwheel Scheduling with End-to-End Timing Constraints", *Proc.* 16^{th} *IEEE Real-Time Systems Symposium*, 1995, pp. 172-181.

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

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

[11] 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, 1997, pp. 616-625.

[12] Krings, A.W., and M.H. Azadmanesh, "Avoiding Run-Time Infeasibility in Systems Containing Coupled Tasks", *INFOR (Information Systems & Operational Research)*, Vol. 37, No. 1, 1999, pp. 77-88.

[13] Krings, A.W., and M.H. Azadmanesh, "Single and Multiple Task Couplings in Hard Real-Time Systems," Tech. Report UI-CS-TR-00-051, Computer Science Department, University of Idaho, May 2000.

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

[15] Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *JACM*, Vol. 14, No. 3, July 1967, pp. 439-465.

[16] McElvaney, M.C., et. al., "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *Journal of Real-Time Systems*, Vol. 3, No. 3, September 1991, pp. 275-305.

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

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

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

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

[21] 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.