SCHEDULING STRATEGIES ADOPTED IN SPRING: AN OVERVIEW

K. Ramamritham and J.A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 91-45

Scheduling Strategies Adopted in Spring: An Overview *

Krithi Ramamritham and John A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003, USA

Abstract

In managing the tasks and resources in a real-time system, there are three phases to consider:

- *allocation* – the assignment of the *tasks* and *resources* to the appropriate nodes or processors in the system,
- *scheduling* – ordering the execution of tasks such that their timing constraints are met and the consistency of resources is maintained, and
- *dispatching* – executing the tasks in conformance with the scheduler's decisions.

Whereas a large part of our work deals with scheduling, more recently we have begun to examine allocation and dispatching issues in the context of our scheduling approach. To date, the main results of our work include (i) the adoption and development of *guarantee*, a notion fundamental to predictable scheduling, (ii) algorithms for the guarantee of dynamically arriving tasks, (iii) an analysis of the quality of the schedules produced by the basic version of our algorithm, (iv) the schemes for reclaiming unused time and resources when tasks complete early, (v) distributed scheduling and meta-level control techniques, and (vi) algorithms for the static allocation and scheduling of safety-critical tasks. These results form the basis for predictable, adaptable, and flexible scheduling support for complex real-time systems.

---

*This work was supported by ONR under contract NOOO14-85-K-0389, by NSF under grant DCR-8500332, and by a grant from Texas Instruments.

**To appear: Chapter for Foundations of Real-Time Computing: Scheduling and Resource Allocation 1991.

INTRODUCTION

The complexities of next generation real-time systems arise from a number of factors including different types of time constraints, different types of time granularities, the presence of tasks with complex characteristics such as resource requirements, precedence constraints, importance factors, and fault tolerance requirements, the dynamic and distributed nature of the environment, and, finally, the long lifetimes of the complex systems being designed today [20], [19].

These complex systems call for resource and task management strategies that are not only *predictable*, i.e., provide assurances that time constraints will be met, but are also *adaptable* and *flexible*. Adaptability calls for approaches that can deal with the dynamics and uncertainties of the environment effectively while flexibility demands approaches that can accommodate changes to the requirements as well as modification to the hardware and software structures. In managing the tasks and resources in a real-time system, there are three phases to consider:

- *allocation* – the assignment of the *tasks* and *resources* to the appropriate nodes or processors in the system,
- *scheduling* – ordering the execution of tasks such that their timing constraints are met and the consistency of resources is maintained, and
- *dispatching* – executing the tasks in conformance with the scheduler's decisions.

Each of the above phases cannot be dealt with in total isolation, since the mechanism used in one phase may greatly affect the performance of others. For example, the scheduling algorithm used influences the design of the allocation scheme, how well the allocation is done affects the run-time performance of the scheduling algorithm, and whether and how time and resources allocated to a task are reclaimed (when it finishes early) affects the correctness as well as the performance of the scheduling algorithm.

In providing support for the three phases, it is important to take into consideration the nature and characteristics of tasks in a given system.

In a system interacting with an environment that is dynamic, large, complex, and evolving, many types of tasks exist. These can be categorized based on their interaction with and impact on the environment. *Critical* tasks are those real-time tasks which must make their deadline, otherwise a catastrophic result might occur. It must be shown *a priori* that these tasks will always meet their deadlines subject to some specified number of failures. Thus, resources must be allocated and scheduling decisions must be made such that these tasks will always meet their deadlines. The number of truly critical tasks (even in very large systems) is likely to be small in comparison to the total number of tasks in the system. *Essential* tasks are tasks that are important to the operation of the system, have specific timing constraints, and will degrade the performance of the system if their timing constraints are not met. However, essential tasks will not cause a catastrophe if they are not finished on time. There are a large number of such tasks and the importance levels of these essential tasks may differ. It is necessary to treat such tasks in a dynamic manner as it is impossible to reserve enough resources for all contingencies with respect to these tasks. *Non-essential* tasks, whether they have deadlines or not, execute when they do not impact critical or essential tasks. Many background tasks, long range planning tasks, and maintenance functions fall into this category.

Whereas a large part of our work deals with scheduling, more recently, we have begun to examine allocation and dispatching issues in the context of our scheduling approach. Specifically, to date, the main results of our work include:

- *The adoption and development of guarantee, a notion fundamental to predictable scheduling.* A task is guaranteed by constructing a plan for task execution whereby all guaranteed tasks meet their timing constraints. A task is guaranteed subject to a set of assumptions, for example, about its worst case execution time, and the nature of faults in the system. If these assumptions hold, once a task is guaranteed it *will* meet its timing requirements. The guarantee notion is elaborated and its benefits are outlined in "The Notion of Guarantee and its Benefits".

  When essential tasks with different levels of importance are considered, we will find a need for the notion of "conditional guarantee" where a task's guarantee is predicated upon the non-arrival of tasks

with higher importance.

- *Algorithms for the guarantee of dynamically arriving tasks.* The basic version of our guarantee algorithm deals with independent, non-preemptable, equally important tasks that have deadlines and resource requirements. It is capable of dealing with both uniprocessors and multiprocessors. Extensions of the basic algorithm deal with precedence relationships among tasks, tasks with varying importance levels, tasks with different fault tolerance requirements, and tasks that are preemptable. The section entitled "Scheduling Tasks on a Multiprocessor Node" is devoted to various aspects of dynamic task scheduling on a multiprocessor node.

- *Analysis of the quality of the schedules produced by the basic version of our algorithm.* Both the *ability* to generate feasible schedules and the *quality* of the generated feasible schedules, expressed in terms of the schedule length, are important metrics for scheduling algorithms. Our theoretical analysis identified several ways in which the basic algorithm can be improved even further. The results of the analysis are discused under "Analysis of the Basic Scheme and the Improvements Motivated by it".

- *Reclaiming unused time and resources when tasks complete early.* The guarantees are based on worst case computation times, but when a task finishes early, as would typically be the case, the unused CPU and resource time may be reclaimed and used. This reclamation is considered in the dispatching phase and the reclaimed time is taken into account by the scheduler for subsequent guarantees. The section entitled "Reclaiming Unused Time and Resources" provides an outline of the scheme.

- *Distributed scheduling and meta-level control.* A suite of distributed scheduling algorithms has been developed and evaluated. We have also hypothesized the usefulness of a *meta-level controller* that can select the heuristic appropriate for a given system (state). Details can be found under "Scheduling Tasks in a Distributed System" and "Meta-Level Control".

- *Static allocation and scheduling of safety-critical tasks.* We have developed an algorithm that is suitable for the static allocation

and scheduling of complex, safety-critical periodic tasks. Besides periodicity constraints, tasks handled by the algorithm can have resource requirements and can possess precedence, communication, as well as fault tolerance constraints. Details of the algorithm as well as the extensions necessary to accommodate dynamic non-periodic arrivals are presented in the penultimate section of this paper.

The notion of guarantee is aimed at achieving *predictability. Adaptability* and *flexibility* are supported by a collection of dynamic task management techniques. These include dynamic guarantees with task characteristics identified at invocation time, distributed scheduling, meta-level control, and resource reclaiming.

Many practical instances of scheduling algorithms have been found to be NP-complete [23], [24]. A majority of scheduling algorithms reported in the literature perform static scheduling, and hence, have limited applicability since not all task characteristics are known *a priori* and further, tasks arrive dynamically. For dynamic scheduling, in single processor systems with independent preemptable tasks, the earliest deadline first algorithm and the least laxity first algorithm are optimal. For dynamic systems with more than one processor, and/or tasks that have mutual exclusion constraints an optimal scheduling algorithm does not exist [6]. These negative results point out the need for heuristic approaches to solve scheduling problems in such systems. In the rest of this paper we provide the details of our approaches.

THE NOTION OF GUARANTEE AND ITS BENEFITS

The notion of guarantee underlies our approach to achieving predictability.

Since all safety-critical tasks must meet their timing constraints, they must be guaranteed statically. Thus, all critical tasks are guaranteed *a priori* and resources are reserved for them. In addition, scheduling decisions for these tasks are also made *a priori*. While *a priori* dedicating resources to critical tasks is, of course, not flexible, due to the importance of these tasks, we have no other choice! On the positive side, recall that the ratio of critical tasks to essential tasks is typically very small.

Given the large numbers of essential tasks and the extremely large number of their possible invocation orders, preallocation of resources to essential tasks is not possible due to cost, nor desirable due to its inflexibility. Hence, such tasks are guaranteed dynamically when they arrive, in the context of the current load. Specifically, if a set S of tasks has been previously guaranteed and a new task T arrives, T is guaranteed if and only if a feasible schedule can be found for tasks in the set $S \bigcup \{T\}$.

The basic notion and properties of guarantee for essential tasks have the following characteristics:

- A task is guaranteed by planning the task executions. When a task is being guaranteed, the scheduler attempts to plan a schedule for it and the previously guaranteed tasks so that all tasks can make their deadlines. This enables our system to understand the total load of the system and to make intelligent decisions when a guarantee cannot be made. This is in contrast with other real-time scheduling algorithms which have a myopic view of the set of tasks. That is, these algorithms only know *which task to run next* and have no understanding of the total load or current capabilities of the system. This planning is done on the system processor in parallel with the execution of previously guaranteed tasks and so it must account for those tasks which may be completed before it itself completes.

- At any point in time the system knows exactly which tasks have been guaranteed to make their deadlines, and what, where and when spare resources exist. The on-line guarantee for essential tasks is an *instantaneous* guarantee with respect to the current state. This presents a *macroscopic* view that *all* critical tasks will make their deadlines and *exactly* which essential tasks will make their deadlines given the current load[1].

- Planning during guarantee *avoids* conflicts over resources. This

[1]It is also possible to develop an overall quantitative, but probabilistic assessment of the performance of essential tasks. For example, given expected normal and overload workloads, we can compute the average percentage of essential tasks that are guaranteed to make their deadlines.

eliminates the random nature of waiting for resources found in timesharing operating systems (this same feature also tends to minimize context switches since tasks are not being context switched to wait for resources). Basically, resource conflicts are resolved at guarantee time by scheduling tasks to execute in different time intervals if they contend for a given resource.

Current real–time scheduling algorithms schedule the CPU independently of other resources. For example, consider a typical real–time scheduling algorithm, earliest deadline first. Scheduling a task which has the earliest deadline does no good if it subsequently blocks because a resource it requires is unavailable. Our approach integrates CPU scheduling and resource allocation so that this blocking never occurs.

By integrating CPU scheduling and resource allocation at run time, we are able to understand (at each point in time), the current resource contention and completely control it so that task performance with respect to deadlines is predictable, rather than letting resource contention occur in a random pattern resulting in an unpredictable system.

- Guarantee can be subject to computation time requirements, deadline or periodic time constraints, resource requirements where resources are segmented, fault tolerance requirements, importance levels for tasks, precedence constraints, and I/O requirements. Specific guarantee algorithms can be designed depending on the task and resource characteristics.

- Dynamic guarantee allows use of dynamic task information. Information about tasks is retained at run time and includes formulas describing worst case execution time, deadlines or other timing requirements, importance level, precedence constraints, resource requirements, fault tolerance requirements, etc. This is utilized to guarantee timing and other requirements of the system. In other words, significant amounts of semantic information about a task can be utilized at run time.

- Guarantees and dispatching can be done independently allowing these system functions to run in parallel. The dispatcher is always

working with a set of tasks which have been previously guaranteed to make their deadlines and the guarantee routine operates on the set of currently guaranteed tasks plus any newly invoked tasks.

- Attempt to guarantee provides early notification. By performing the guarantee calculation when a task arrives there may be time to reallocate the task to another node of the system. Early notification also has *fault tolerance* implications in that it is now possible to run alternative error handling tasks early, before a deadline is missed.

- Guarantee can be designed to support the co-existence of real–time and non-real–time tasks, and note that this is non-trivial when non-real–time tasks might use some of the same resources as real–time tasks.

- Even though a task is guaranteed with respect to its worst case computation time and resource requirements, it is possible to reclaim unused resources, including time, when it completes early.

Even if it is not guaranteed on a node in a distributed system, it may still meet its deadline. This could coour in several different ways. First, it could receive idle cycles at this node, and, in parallel, there can be an attempt to get the task guaranteed on another node of the system subject to location dependent constraints. Consequently, it might either complete at this node or be guaranteed elsewhere. Second, based on the fault tolerance and correctness semantics of the task, an alternative task with smaller computation time, reduced resource needs, or larger deadline could be invoked and hence might get guaranteed.

DYNAMIC SCHEDULING

Scheduling Tasks on a Multiprocessor Node. A guarantee algorithm must consider many issues including the presence of periodic tasks, preemptable tasks, precedence constraints (which is used to handle task groups), multiple importance levels for tasks, and fault tolerance requirements. We first describe a basic version of the algorithm that handles task deadlines and resource requirements. Subsequently, we discuss the extensions required to handle the other considerations.

The Basic Algorithm. The basic algorithm attempts to guarantee non-preemptable tasks given their arrival time T_A, deadline T_D or period T_P, worst case computation time T_C, and resource requirements $\{T_R\}$. A task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes. The guarantee algorithm computes the earliest start time, T_{est}, at which task T can begin execution. T_{est} accounts for resource contention among tasks. It is a key ingredient in our scheduling strategy.

The heuristic scheduling algorithms we use try to determine a full feasible schedule for a set of tasks in the following way. It starts at the root of the search tree which is an empty schedule and tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, we use a heuristic function, H, which synthesizes various characteristics of tasks affecting real-time scheduling decisions to actively direct the scheduling to a plausible path. The heuristic function, H, is applied to at most k tasks that remain to be scheduled at each level of search. The task with the smallest value of function H is selected to extend the current schedule.

While extending the partial schedule at each level of search, the algorithm determines if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending this current schedule with any one of the remaining tasks are also feasible. Thus, if a partial feasible schedule is found not to be *strongly-feasible* because, say, task T misses its deadline when the current schedule is extended by T, then it is appropriate to stop the search since none of the future extensions involving task T will meet its deadline. In this case, a set of tasks can not be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a feasible complete schedule.)

However, it is possible to backtrack to continue the search even after a non-strongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it using a different task. The task chosen is the one with the *second* smallest H value. Even though we allow backtracking, the overheads of backtracking can be restricted either by restricting

the maximum number of possible backtracks or by restricting the total number of evaluations of the H function. We use the latter scheme because we found it to be more effective.

The algorithm works as follows: It starts with an empty partial schedule. Each step of the algorithm involves (1) determining that the current partial schedule is strongly-feasible, and if so (2) extending the current partial schedule by one task. In addition to the data structure maintaining the partial schedule, tasks in the task set S are maintained in the order of increasing deadlines. This is realized in the following way: When a task arrives at a node, it is *inserted*, according to its deadline, into a (sorted) list of tasks that remain to be executed. This insertion takes at most $O(n)$ time where n is the task set size. Let $n_k = \min(k$, number of tasks yet to be scheduled). Then when attempting to extend the schedule by one task, three steps must be taken: (1) strong-feasibility is determined with respect to the first (still remaining to be scheduled) n_k tasks in the task set, (2) if the partial schedule is found to be strongly-feasible, then the H function is applied to the first n_k tasks in the task set (i.e., the n_k remaining tasks with the earliest deadlines), and (3) that task which has the smallest H value is chosen to extend the current schedule. Given that only n_k tasks are considered at each step, the complexity incurred is $O(n \times k)$ since only the first n_k tasks are considered each time. If the value of k is constant (and in practice, k will be small when compared to the task set size n), the complexity is linearly proportional to n, the size of the task set [9]. While the complexity is proportional to n, the algorithm is programmed so that it incurs a fixed worst case cost by limiting the number of H function evaluations permitted in any one invocation of the algorithm. Also, see [11] for a discussion on how to choose k.

Given a partial schedule, for each resource, the earliest time the resource is available can be determined. This is denoted by EAT. Then the earliest time that a task that is yet to be scheduled can begin execution is given by

$$T_{est} = Max(\text{T's start time}, EAT_i^u)$$

where u = s or e if T needs resource R_i in shared or exclusive mode, respectively.

The heuristic function H can be constructed by simple or integrated heuristics. The following is a list of simple and integrated heuristics that

we have tested:

- Minimum deadline first (Min_D): $H(T) = T_D$
- Minimum processing time first (Min_C): $H(T) = T_C$
- Minimum earliest start time first (Min_S): $H(T) = T_{est}$
- Minimum laxity first (Min_L): $H(T) = T_D$ - $(T_{est}+T_C)$
- Min_D + Min_C: $H(T) = T_D + W * T_C$
- Min_D + Min_S: $H(T) = T_D + W * T_{est}$

The first four heuristics are considered simple heuristics because they treat only one dimension at a time, e.g., only deadlines, or only resource requirements. The last two are considered to be integrated heuristics. W is a weight used to combine two simple heuristics. Min_L and Min_S need not be combined because the heuristic Min_L contains the information in Min_D and Min_S.

Extensive simulation studies of the algorithm for uniprocessor and multiprocessors show that the simple heuristics do not work well and that the integrated heuristic (Min_D + Min_S) works very well and has the best performance among all the above possibilities as well as over many other heuristics we tested [26] [28] [29], [9], [11]. For example, combinations of three heuristics were shown not to improve performance over the (Min_D + Min_S) heuristic.

Analysis of the Basic Scheme and the Improvements Motivated by it. Both the *ability* to generate feasible schedules and the *quality* of the generated feasible schedules, expressed in terms of the schedule length bound, are important metrics for scheduling algorithms. Our analysis [25] showed that in the worst case, on a multiprocessor with m processors, the length of the schedules produced by the basic algorithm (called H_1 here) will be m times of the length of an optimal schedule. List scheduling, which is not as successful in finding feasible schedules, has a worst case schedule length bound, of $(m+1)/2$, almost half of the worst case schedule length bound of H_1. The two algorithms differ in the following way. In H_1, a task's H value consolidates the task's deadline and resource requirement information. Our algorithm

selects a task with the minimum H value among all unscheduled tasks, while list scheduling selects a task with the minimum H value among the unscheduled tasks which have the same earliest start time. Thus, in list scheduling the task's deadline is only a secondary factor. Hence, we developed algorithm H_2 that uses a task selection procedure that has elements from both H_1 and list scheduling. It selects a task to keep at least two processors busy whenever possible, otherwise its behavior is similar to H_1. This change was motivated by the fact that in general, list scheduling keeps more processors busy at a given time than H_1. H_2 has a better worst-case schedule length bound, namely $(m+1)/2$. At the same time, it has almost the same success as H_1 in finding feasible schedules. Further, H_2 can be implemented so that its scheduling overhead is not higher than that of H_1.

Can the bound of H_2 be reduced further by increasing the (polynomial time) complexity of the scheduling algorithm? Partly motivated by this question, we developed H_s, a generalized version of H_2, which keeps at least s processors busy whenever possible. It turns out that the added time complexity of H_s does not help to reduce the bound $((m+1)/2)$. The reason is that, in the worst case, there may be some tasks with very small computation times which contribute significantly to the number of tasks running in parallel at scheduling decision points, but they do not contribute much to the schedule length. If the ratio between the shortest task and the longest task in a task set is close to 1, the bound of H_s is likely to be smaller than $(m+1)/2$. In the extreme case, the ratio is one and in this case, in systems where the number of resources is larger than the number of processors, the bound of H_s reduces from $(m+1)/2$ to $m/s+\sum_{j=2}^{s} 1/j$. In practice, it is also very important to choose a proper s to balance the time overhead and the achieved bound.

One of the next steps we envisage is to better characterize the worst-case task characteristics that produce the above bounds. This will help avoid the pessimistic bounds.

Extensions to the Basic Algorithm. Recall that the basic algorithm takes tasks' computation time, timing constraints, and resource requirements into account. In this section, we present the extensions necessary to deal with periodic tasks, tasks that have fault tolerance requirements, tasks with different importance levels, tasks that have precedence constraints, and preemptable tasks.

Periodic Tasks: In what follows, we discuss several ways of guaranteeing periodic tasks in the presence of nonperiodic tasks. Depending on whether periodic and nonperiodic tasks need access to a common set of resources, one or more options will be applicable to a given system. We have yet to test the effectiveness of the various options.

We assume that when a periodic task is guaranteed, every instance of the task is guaranteed.

Consider a system with only periodic tasks. A schedule can be constructed using the basic algorithm described in the previous section. Specifically, we can assign start time and deadline constraints to the periodic task instances based on the periods and guarantee a periodic task only if all instances of the periodic tasks can be guaranteed. In fact, we need only construct a schedule whose length is equal to the least common multiple of all the periods. This can serve as a template according to which all periodic task instances are executed.

When a periodic task arrives dynamically, an attempt can be made to construct a new template. If the attempt succeeds, the new task is guaranteed, otherwise not.

Suppose periodic tasks *and* nonperiodic tasks occur in a system. If the resources needed by periodic tasks and nonperiodic tasks are disjoint then a subset of the processors in a node can be earmarked for periodic tasks. A template can be constructed to execute the periodic task instances on just these processors. The remaining processors are used for nonperiodic tasks guaranteed using our dynamic guarantee scheme. If however, periodic and nonperiodic tasks have common resources, more complicated schemes are needed.

If a periodic task arrives in a system consisting of previously guaranteed periodic and nonperiodic tasks, an attempt is made to construct a new template. If this fails, the new task is not guaranteed. If the attempt succeeds, we can immediately guarantee the task provided this will not jeopardize other guaranteed nonperiodic tasks. Otherwise, introduction of the periodic task into the system has to be delayed until the guaranteed nonperiodic tasks complete or at least until its introduction will not affect the remaining guaranteed tasks.

Suppose a new nonperiodic task arrives. In the presence of the template for periodic tasks, the task can be guaranteed in the idle slots

identified by the template. More flexibility can be gained by using the following scheme. Given that we can assign start time and deadline constraints to the periodic task instances based on the periods, applying our dynamic guarantee scheme, we can guarantee a nonperiodic task if all instances of the periodic tasks and all previously guaranteed nonperiodic tasks can also be guaranteed. In this scheme, the number of tasks that have to be considered in guaranteeing a new nonperiodic task (in the presence of periodic tasks) is higher and hence can involve high scheduling overheads.

Tasks with Fault Tolerance Requirements: When a task is not guaranteed a *timing fault* is forecast. In this situation, an alternative task which has a shorter computation time or less resource needs can be invoked. If the latter is guaranteed, the timing fault is masked from the next higher level. If not, the next higher level can decide whether some other activity should be performed instead of the original nonguaranteed task.

If a node with a guaranteed tasks fails, the guarantees do not hold. The problem of node failures and the reallocation of tasks guaranteed on that node is studied in [21]. Here reallocation is done so as to maximize the number of tasks that can still meet their deadlines.

If guarantees are required in spite of node failures, guarantees have to be provided on multiple nodes. Specifically, if a task is nonperiodic and does not share resources with other tasks, or if a task is an instance of a periodic task and shares resources only with other instances of the same task, then guaranteed execution with respect to t fail-stop node failures can be achieved by guaranteeing the execution of the task at $t+1$ nodes. When a task does not share resources, the following scheme reduces the overheads of executing $t+1$ copies of a task at all times: Start times of a task's replicates are staggered so that the i^{th} replicate is guaranteed with respect to a start time of $(s + (i-1)m)$ and a deadline of $(d - (t+1-i)m)$ where s and d are the start time and deadline of the task, respectively, and m is the communication delay. The idea is to avoid executing as many task replicates as possible. The first replicate to complete successfully, informs all others, and as a result, the following replicates stop further processing of the task. The assumption here is that all interactions with the environment take place when a replicate completes successfully. Obviously, this scheme is applicable only when

communication delays and task computation times are small compared to task deadlines.

These techniques also apply when guaranteeing in spite of processor failures. To guarantee a task on a node in spite of t processor failures, its execution is guaranteed on $t + 1$ processors. $t + 1$ staggered executions of a task can be scheduled on different processors within a node. To handle transient processor failures when nodes are not fail-stop, the latter scheme can be applied to a single processor. Once a task completes successfully, as determined by specified acceptance tests, subsequent replicates of the task are deleted and their time reclaimed.

Tasks with Different Levels of Importance: The two main task parameters, deadline and importance, may sometimes be in conflict with each other. That is, tasks with very short deadlines might not be very important, and vice versa. This causes a dilemma in choosing the next task to execute. Also, the semantics of guarantee needs to be refined when tasks with differing importance values are present. Suppose a task has been guaranteed and a task with a higher importance arrives. Also suppose that the new task can be guaranteed only if the lower importance task is removed from the guaranteed list, i.e., the guarantee is withdrawn. In this case the initial guarantee is not absolute but conditional upon the non-arrival of higher importance tasks which conflict with it. In most applications it is important to meet the deadlines of higher importance tasks even if that implies the withdrawal of guarantees to other (lower importance) tasks. The concept of 'guarantee' as *conditional* rather than *absolute* is found to be superior in terms of adaptiveness and responsiveness to changing system conditions.

We have proposed and evaluated two dynamic scheduling algorithms [2]. They integrate importance and deadline such that, not only do the more important tasks meet their deadlines, but many other less important tasks also meet their deadlines. Overall, their goal is to maximize the net worth of the executed tasks to the system.

Both the algorithms first attempt to guarantee an incoming task according to its deadline, ignoring its importance. If the task is guaranteed then the scheduling is successful.

However, if this first attempt at scheduling fails, then there is an attempt to guarantee the new task at the expense of previously guaranteed, but less critical tasks. If enough less critical tasks can be found

then the new task is guaranteed at this node and the removed tasks are transferred to alternative sites. If there are not enough less critical tasks, or the deadline of the new task is such that the removal of any such tasks does not allow the new task to meet its deadline, then the *new task* is not guaranteed.

The two algorithms differ only in how they remove previously guaranteed low importance tasks. In the first algorithm lower importance tasks are removed one at a time and in strict order from low to high importance. The second algorithm also only removes tasks of lower importance, but does not follow the strict order found in the first algorithm. Rather, it removes any task with lower importance, starting from tasks with the largest deadline.

Surprisingly, both the algorithms have the same performance for almost all system conditions and task parameters. We have also observed that at low loads pure deadline based algorithms tend to perform better than pure importance based algorithms. At high loads, the situation is reversed and pure importance based algorithms outperform pure deadline based algorithms. Also, we show that our algorithms outperform the pure deadline and importance based baselines since they combine the advantages of both the deadline and importance based algorithms.

We are beginning to evaluate an alternative scheme. This scheme adds another factor into $H(T)$. The goal is to produce a feasible schedule biased with the most important tasks first. Thus important tasks are likely to be scheduled earlier than unimportant ones.

Tasks with Precedence Constraints: Precedence constraints between tasks are used to model end-to-end timing constraints both for a single node and across nodes.

A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single deadline. Each task acquires resources before it begins and can release the resources upon its completion. For task groups, it is assumed that when the task group is invoked the worst case computation time and resource requirements of each task can be determined. In [3], an algorithm for guaranteeing tasks groups in distributed systems was developed and evaluated. This algorithm first clusters tasks in a group; each cluster is expected to be scheduled on a single node. From the deadline for the task group, deadline for individual clusters are identified. If a set of nodes, each

of which commits to executing tasks within a cluster by the deadline, is found, then the task group is guaranteed. Given the communication involved in such a scheme, this is applicable to task groups with large laxities.

Considering a task group that executes within a multiprocessor node, it is possible to extend our basic scheme to deal with precedence constraints. We have identified, but are yet to test, a number of strategies. When tasks with precedence constraints are considered, a task becomes eligible to execute only when all of its ancestors are scheduled. Hence, at a given point in the search for a feasible schedule, the H function is applied only to tasks whose ancestors have already been scheduled and whose scheduled finish times have occurred. Also, another factor is added to the heuristic function in order to provide a positive bias to those eligible tasks that lie along the critical path.

Currently, we are also studying precedence related tasks where tasks may have different importance levels. In this case, it is necessary to schedule the tasks such that not only are task deadlines satisfied, but the value of executed tasks is also maximized.

Preemptable Tasks: In scheduling a set of preemptable tasks on a multiprocessor, a *schedule* S for a set of preemptable tasks can be viewed as consisting of a sequence of *slices* $S_k, k = 1, ...,$ number of slices in the schedule S. A slice S_k is associated with a *slice start time*, a *slice length*, and a subset of tasks which can run in parallel in that slice. A task is said to be *preempted* between slices S_k and S_{k+1} if it is in slice S_k, does not complete in that slice, and is not in slice S_{k+1}.

With this view, scheduling decisions have to be made about the length of slices and about the tasks that belong to a slice. The slice length is chosen to be no more than the minimum remaining processing time of a task in the slice. Also, it should be no more than the minimum of the laxities of the remaining tasks. Thus, it is the smaller of the above two quantities. In the context of our dynamic search for a feasible schedule, the core of the algorithm lies in effectively and correctly identifying the subset of tasks associated with a new slice. Let us first understand the basis for including a task in a slice. Considering just the timing constraint, it is well known that both earliest-deadline-first and smallest-laxity-first scheduling schemes produce optimal preemptive schedules. Given that in addition to timing constraints there

are resource constraints, we need to include resource considerations in scheduling. Hence, we construct a slice with one primary and one or more secondary tasks. The primary task is selected based purely on timing considerations: one with the smallest laxity among tasks that have not completed is selected. Then we use a heuristic function similar to the one used for nonpreemptive scheduling to select the secondary tasks. Ideally, this function should consider both timing and resource constraints and select as many secondary tasks as possible to execute in parallel with the primary task. Thus, the heuristic function is applied to all the candidate tasks. The task with the minimum value returned by the function is selected for the slice. This process is repeated until no more tasks can be included in the slice. In [27] we identify and evaluate different heuristic functions. In addition to the considerations involved in the basic algorithm, here the heuristic function should also consider preemption costs as well as resource usage and their effect of preempting a task at the end of a slice.

Reclaiming Unused Time and Resources. In order to guarantee that hard real-time tasks will meet their deadlines in the worst case, most real-time scheduling algorithms schedule tasks with respect to their worst case computation times. In reality, this worst case computation time is only an upper bound and the actual execution time may vary between some minimum value and this upper bound, depending on various factors, such as the system state, the amount and value of input data, the amount of resource contention, and the types of tasks. *Resource reclaiming* refers to the problem of *correctly* and *effectively* utilizing the resources unused by a task when a task executes less than its worst case computation time, or when a task is deleted from the current schedule. Task deletion occurs either during an operation mode change [14], or when one of the copies of a task completes successfully in a fault-tolerant system.

We have designed two versions of a resource reclaiming algorithm that (1) avoid any run time anomalies, (2) effectively reclaim the unused portions of resource time and processor time, and (3) have time complexities independent of the number of tasks in the given schedule. The last requirement arises due to the fact that resource reclaiming will have to be done on every task completion and, hence, reclamation costs are added to task dispatching costs.

A feasible multiprocessor schedule provides task ordering information that is *sufficient* to guarantee the timing and resource requirements of tasks in the schedule. If two tasks overlap in time on different processors in a schedule, then we can conclude that no matter which one of them will be dispatched first at run time, they will never jeopardize each other's deadlines. On the other hand, if two tasks do not overlap in time, we cannot make the same conclusion without re-examining resource constraints or without total re-scheduling. Assume each task is assigned a scheduled start time sst and a scheduled finish time sft in the given feasible schedule, our resource reclaiming algorithms utilize this information to do *local* optimization at run time, while preserving the correct relative ordering among the scheduled tasks, thus ensuring the original guarantee. This local optimization is accomplished by reasoning only about the first m tasks in the schedule, m being the number of processors. By doing so, we avoid explicitly examining the availability of each of the resources needed by a task in order to dispatch a task when reclaiming occurs. Thus, the complexity of the algorithms is independent of the number of tasks in the schedule.

Each of our resource reclaiming algorithms consists of two steps. A task is removed from the schedule only upon its completion. In the first step, upon completion of a task, we try to identify idle periods on all processors and resources by computing a function $reclaim\_\delta = min(sst_i) - current\_time$; where sst_i is the scheduled start time of the current first task for processor i in the schedule, $1 \leq i \leq m$. The computational complexity of this function is $O(m)$. Any positive value of $reclaim\_\delta$ indicates the length of the idle period. The cumulative value of these idle periods is stored in θ. This first step is identical for the two versions of our algorithm. In the second step of the algorithm, the basic version of our resource reclaiming algorithm computes the *actual start time* ($= sst - \theta$) for the next task scheduled on the processor in question. The task will be dispatched if the *actual start time* equals the current time. In the extended version of our algorithm, we first compute a Boolean function $can\_start\_early = sst_i < sft_j, \forall j$ such that $j \neq i$ and $1 \leq j \leq m$, where sst_i is the scheduled start time of the first task on processor i and sft_j is the scheduled finish time of the first task on processor j. This function identifies parallelism between the first task on processor i and the first tasks on all other processors and has a complexity $O(m)$. The task will be dispatched if the value of

can_start_early is true. If it is false, *actual start time* is computed as in the basic version. Whenever a positive value of *reclaim_δ* is obtained in the first step of the algorithm, the second step of the algorithm must be executed for all currently idle processors. Thus the complexity of the basic version is: $O(m) + m * O(1) = O(m)$, while the extended version has a complexity of $O(m) + m * O(m) = O(m^2)$.

We have demonstrated the effectiveness of the two versions of the resource reclaiming algorithms through simulation [15]. Even though the extended version incurs a higher run time cost, it still performs much better than the basic version in most parameter settings of the simulation. Only when the resource conflict probability is very high, or when the system is either extremely overloaded or lightly loaded, the basic version demonstrates the same effectiveness as the extended version.

One of the positive fallouts of reclaiming is that now we can afford to be pessimistic about the computation times of tasks. This is because even if the dynamic guarantees are done with respect to tasks' worst case computation times, since any unused time is reclaimed, the negative effects of pessimism are considerably eliminated.

Scheduling Tasks in a Distributed System

As alluded to earlier, when a task arrives at a node, the scheduler at that node attempts to guarantee that the task will complete execution before its deadline, on that node. If the attempt fails, the scheduling components on individual nodes cooperate to determine which other node in the system has sufficient resource surplus to guarantee the task. In [10], four algorithms for cooperation are evaluated. They differ in the way a node treats a task that cannot be guaranteed locally:

- The *random scheduling algorithm*: The task is sent to a randomly selected node.
- The *focussed addressing algorithm*: The task is sent to a node that is estimated to have sufficient surplus to complete the task before its deadline.
- The *bidding algorithm*: The task is sent to a node based on the bids received for the task from nodes in the system.

- The *flexible algorithm*: The task is sent to a node based on a technique that combines *bidding* and *focussed addressing.*

Simulation studies were performed to compare the performance of these algorithms relative to each other as well as with respect to two baselines. The first baseline is the non-cooperative algorithm where a task that cannot be guaranteed locally is not sent to any other node. The second is an (ideal) algorithm that behaves exactly like the bidding algorithm, but incurs no communication overheads. We examined the relative performance of the various algorithms with respect to the system state, specifically, the system load, the load distribution among nodes, and the communication delays. They showed that:

- Distributed scheduling improves the performance of a hard real-time system. This is attested by the better performance of the flexible algorithm compared to the non-cooperative baseline under all load distributions.
- The performance of the flexible algorithm is better than both the focussed addressing and bidding algorithms. However, the performance difference between the bidding algorithm and the flexible algorithm under small communication delays is negligible. The same can be said about the performance difference between the focussed addressing algorithm and the flexible algorithm at large communication delays.
- The random algorithm performs quite well compared to the flexible algorithm, especially when system load is low as well as when system load is high and the load is unevenly distributed. Under moderate loads, its performance falls short by a few percentage points which may be significant in a hard real-time system.

Our studies showed that no algorithm outperforms all others in all system states. Though the flexible algorithm performs better than the rest in most cases, it is more complex than the other algorithms.

Details of the distributed scheduling algorithms can be found in [7], [10], and [18]. The stability of these algorithms is discussed in [17].

Meta Level Control

Any algorithm used for scheduling in complex real-time systems must be *adaptive*. That is, decisions made by the scheduling components must adapt to changes in the state of the system. Changes include availability of nodes, variations in task arrival rate, characteristics of tasks, load on individual nodes, and delays in the communication network.

As our discussion of distributed scheduling algorithm illustrates, no single scheduling algorithm can be effective under all situations, even if the parameters that control the behavior of the algorithm are fine-tuned.

Reexamining the flexible algorithm, one notices that, depending on the current state of the nodes and the communication network as well as the task characteristics, it elects to resort to focussed addressing alone, bidding alone, or attempts both in parallel. In one sense, what it does, albeit in a limited manner, is *control the scheduling*. Since scheduling is the control of task executions, we term this control of scheduling as *meta-level control*. The prime motivation for such control is *adaptability*.

We envisage each node having a component of the *Meta-Level Controller* (MLC) where the components cooperate in making meta-level control decisions. We propose to use the meta-level controller primarily for the following [8]:

- Controlling the choice of algorithm(s) used for the various scheduling components in a given situation. The primary components are the *local scheduler* and the *distributed scheduler*.

- Controlling the values of scheduling parameters used in the chosen algorithm(s). For example, one of the crucial set of parameters is the set of weights that govern the behavior of the (heuristic) *guarantee algorithm* invoked by the local scheduler; these weights determine the importance to be given to the deadline and resource requirements of a set of tasks in determining a schedule for these tasks. Important parameters used in global scheduling include the minimum surplus a node should have to be a candidate for focussed addressing and the minimum surplus a node should have for it to respond to a request-for-bids.

- Controlling the reallocation of periodic tasks to nodes. Through such a reallocation, loads on nodes can be adjusted such that the

number of nonperiodic tasks that are sent to other nodes can be reduced. Periodic tasks exist for reasonably long intervals of time. Instances of a periodic task execute on the same node until the periodic task is reassigned. Thus periodic task assignment determines the *basic* load on a node. If we assume that the distribution of dynamic task arrivals on each node is determined by the external world and thus we are not able to control it, then one solution to obtain a new "balance" of node loads is to dynamically reallocate the periodic tasks subject to constraints such as the resource needs of these tasks.

The potential benefits of meta-level control are clear. But we have not yet investigated are the cost/benefits tradeoffs.

STATIC ALLOCATION AND SCHEDULING OF COMPLEX PERIODIC TASKS

Resources needed to meet the deadlines of safety-critical tasks are typically preallocated. Also, these tasks are usually statically scheduled such that their deadlines will be met even under worst-case conditions. We have developed an algorithm that is suitable for the static allocation and scheduling of complex, safety-critical periodic tasks. Besides *periodicity constraints*, tasks handled by the algorithm can have *resource requirements* and can possess *precedence*, *communication*, as well as *fault tolerance constraints* [12].

Given a set of periodic tasks, the algorithm attempts to assign subtasks of the tasks to nodes in a distributed system and to construct a schedule of length L where L is the least common multiple of the task periods. A real-time system with the given set of tasks then repeatedly executes its tasks according to this schedule every L units of time.

The algorithm consists of four steps. Step-I constructs the *comprehensive graph* containing all instances of the tasks that will execute in an interval of length L. In Step-II, the comprehensive graph is embellished with replicates of the subtasks that have fault-tolerance requirements. This results in the addition of some subtasks and some arcs to the graph. Step-III involves clustering subtasks in the comprehensive graph. Specifically, based on the amount of communication involved between a pair of communicating subtasks and the computation time of

the subtasks, a decision is made as to whether the two subtasks should be assigned to the same node, thereby eliminating the communication costs involved. The algorithm makes its decision based on whether the fraction $\frac{sum\ of\ the\ computation\ time\ of\ the\ two\ subtasks}{cost\ of\ communication}$ is lower than a tunable parameter called the *communication factor*, CF. Applying the above scheme to *every pair* of communicating subtasks in the comprehensive graph derived in Step-II, a *communication graph* is generated with the current value of CF. Step-IV allocates the subtasks to nodes in the system, allocates the communication (between subtasks) to the time slots in the communication channel, and if possible, determines a feasible schedule. This is done using a heuristic search technique that takes into account the various task characteristics, in particular, subtask computation times, communication costs, deadlines, and precedence constraints. It allocates a subtask to a node, determines the order in which each node processes its subtasks, and schedules communication. The allocation and scheduling decisions are made in a coordinated fashion. Specifically, allocation and scheduling decisions about a subtask are made only after all its predecessors have been allocated and scheduled. Of course, these decisions take into account the communication and computational needs of the subtasks that follow. If at the end of Step-IV, a feasible allocation and schedule is not possible, the value of CF is altered, and Steps III and IV are repeated. In [12], we provide details of each of these steps.

Our experiments indicate that the algorithm conducts the search path so effectively that if the initial path does not lead to a feasible schedule for a given set of tasks, chances are high that we have an infeasible task set. This is attested by our test results which show that even a large number of additional backtracks produces only a small marginal improvement in performance.

Integrating Dynamic Task Management with Static Task Scheduling

The primary criterion in the static scheduling of periodic tasks is *feasibility*, i.e., determining a feasible schedule wherein all tasks meet their timing requirements, precedence constraints, etc. Under static allocation and scheduling, exactly when and where instances of a task will execute are fixed. But, if both periodic tasks as well as nonperiodic tasks exist in a system, provision should be made during static scheduling to cater to

the needs of dynamic arrivals. Some leeway should be provided such that the static schedules can be dynamically modified for better nonperiodic task schedulability while retaining the feasibility of the critical task set.

Suppose we consider a distributed system of multiprocessor nodes. When static schedules for distributed or multiprocessor systems are examined, two types of phenomena can be observed. The first relates to the computational loads imposed on the individual nodes or processors. Typically, there exists a load imbalance within the processors on a node as well as across the nodes. The second relates to the idle times on individual nodes or processors. Often, these occur in a clustered fashion towards the end of the schedule, that is, idle times are not evenly spread across the schedule. Both of these are to be avoided if the idle time and resources are to be used to better accommodate dynamic arrivals. In particular, we would like to balance the loads not only across individual nodes and processors within a node but also along the time axis.

We have investigated two approaches to address this issue. One involves defining a limit to the load imposed on a node during the allocation of tasks to processors and nodes. The other forces an idle time interval following the scheduled execution of a task. Effect of different limits on the loads and fixed length idle times as well as task execution time dependent idle times have been studied [13]. The experiments indicate that these very simple techniques can be used to enhance the responsiveness to dynamic arrivals without jeopardizing the schedulability of the static periodic tasks. In addition, more flexibility can be gained by modifying the start times of the components of a critical tasks allocated to a node. If this modification is done such that the ancestors of these components on other nodes are not delayed or that task deadlines are not missed, more dynamic arrivals can be guaranteed. [13] discusses such modifications as well.

SUMMARIZING REMARKS AND OUTSTANDING ISSUES

In this paper we have surveyed some of the significant aspects of scheduling research done within the Spring project. We have endeavored to examine task and system characteristics that are likely to be encountered in next generation real-time systems and provide a flexible and adaptable, and yet predictable scheduling support for these systems.

We believe that our solutions are well integrated. Some examples of this integration are the ability to deal with dynamic arrivals of periodic as well as nonperiodic tasks, to schedule tasks under timing, resource, precedence, as well as fault tolerance constraints, to accommodate the orthogonal characteristics of deadlines and importance levels, and to cater to dynamic arrivals within static schedules.

Given that most scheduling problems are computationally intensive, the Spring scheduling algorithms are driven by heuristics. Our simulation studies have demonstrated the efficacy of the heuristics and in many cases have suggested possible enhancements. Certain possibilities for improvements to the dynamic guarantee algorithm also came to light when mathematical analysis of the algorithm was done. The basic dynamic guarantee algorithm and the reclaiming algorithms have been implemented as part of the Spring kernel [5] [22]. Many race conditions that occur in a multiprocessor environment manifested themselves when this implementation was carried out. In particular, this necessitated schemes that allowed the dispatchers on individual processors to access system data structures while the scheduler was guaranteeing new arrivals.

Some of the extensions to the guarantee algorithm are yet to be implemented. Also we are currently addressing the problem of initial allocation of tasks and resources. The goal of the allocation algorithm, in the context of the integrated treatment of CPU and resources, is to partition the set of *tasks* and *resources* among processors to maximize the inherent concurrency. How well the allocation of tasks and resources is done directly influences the performance of the dynamic on-line scheduling of the real-time tasks. Hence we have developed algorithms using various heuristics to solve this allocation problem and are in the process of evaluating them in context of our scheduling algorithms.

Acknowledgements

Many people have worked on scheduling as part of the Spring Project. W. Zhao contributed significantly to the basic scheduling concepts. We also wish to thank K. Arvind, S. Biyabani, S. Cheng, E. Gene, J. Huang, L. Molesky, E. Nahum, M. Kuan, D. Niehaus, C. Shen, P. Shiah, F. Wang, and G. Zlokapa for their work in this area.

REFERENCES

[1] S. Biyabani, "The Integration of Deadline and Criticalness in Hard Real–Time Scheduling," Masters Thesis, Univ. of Mass., August 1987.

[2] S. Biyabani, J.A. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real–Time Scheduling," *Proc. Real-Time Systems Symposium*, December 1988.

[3] S. Cheng, J.A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Real-Time Systems Symposium*, December 1986.

[4] S. Cheng, "Dynamic Scheduling Algorithms for Distributed Hard Real-Time Systems," *Ph.D. Thesis*, University of Massachusetts, May 1987.

[5] L.D.Molesky, K. Ramamritham, C. Shen, J.A.Stankovic, and G. Zlokapa. "Implementing a Predictable Real-time Multiprocessor Kernel – the Spring Kernel", *7th IEEE Workshop on Real-time Operating Systems and Software*, May 1990.

[6] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", *Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass.*, May 1983.

[7] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, 1(3), 1984.

[8] K. Ramamritham, J. Stankovic, W. Zhao, "Meta-Level Control in Distributed Real-Time Systems," *Intl. Conference on Distributed Computing Systems*, September 1987.

[9] K. Ramamritham, J. Stankovic, P. Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems," *International Conference on Parallel Processing Systems*, August 1989.

[10] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[11] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[12] K. Ramamritham, "Scheduling Complex Periodic Tasks", *Intl. Conference on Distributed Computing Systems*, June 1990.

[13] K. Ramamritham and J. M. Adan, "Load Balancing During the Static Allocation and Scheduling of Complex Periodic Tasks", COINS Technical Report, October 1990.

[14] L. Sha, R. Rajkumar, J. Lehoczky and K. Ramamritham, "Mode Change Protocols for Priority-Driven Preemptive Scheduling", *Real-Time Systems*, pp. 243-264, Vol. 1, No. 3, December 1989.

[15] C. Shen, K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Proc Real-Time System Symposium*, December 1990.

[16] P. Shiah, "Real-Time Multiprocessor Scheduling Algorithms," MS Thesis, Univ. of Mass., January 1989.

[17] J.A. Stankovic, "Stability and Distributed Scheduling Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 10, October 1985.

[18] J.A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, December 1985, pp. 1130-1143.

[19] J. A. Stankovic, "Misconceptions About Real-Time Computing," *IEEE Computer*, October 1988.

[20] J. A. Stankovic and K. Ramamritham, *Hard Real–Time Systems*, Tutorial Text, IEEE Press, 1988.

[21] J.A. Stankovic, "Decentralized Decision Making for Task Allocation in a Hard Real-Time System" *IEEE Transactions on Computers*, March 1989.

[22] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real–Time Systems," *IEEE Software*, May 1992. pp. 54-71.

[23] J. D. Ullman, "Polynomial Complete Scheduling Problems", *Operating System Review*, Vol. 7, No. 4, October. 1973.

[24] J. D. Ullman, "NP-Complete Scheduling Problems", *Journal of Computer and System Science*, October. 1975.

[25] F. Wang, K. Ramamritham, and J.A. Stankovic, "Bounds on the Schedule Length for Some Heuristic Scheduling Algorithms for Hard Real-time Tasks", submitted for publication, September 1990.

[26] W. Zhao, "A Heuristic Approach to Scheduling with Resource Requirements in Distributed Systems," *Ph.D. Thesis*, February 1986.

[27] W. Zhao, K. Ramamritham and J. A. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *IEEE Trans. on Computers*, August 1987, pp. 949-960.

[28] W. Zhao, K. Ramamritham and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, SE-12(5), May 1987.

[29] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," *Journal of Systems and Software*, Vol 7, 1987, pp. 195-205.