The Design of the Spring Kernel*

John A. Stankovic
Krithi Ramamritham
Dept. of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

COINS Technical Report 87-53
10 April 1987

1 Introduction

Recently, there has been an increased interest in hard real-time systems and such systems are becoming more and more sophisticated. We define Hard Real-time systems as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Further, if these real-time constraints are not met there may potentially be catastrophic consequences. Examples of this type of real-time system are command and control systems, nuclear power plants [1], process control systems, flight control systems, and the space shuttle avionics system [6]. In the future, such systems are expected to become more and more complex, have long lifetimes, and exhibit very dynamic, adaptive and even intelligent behavior. The most critical part of supporting such new systems is the ability to guarantee that real-time constraints can be met. Because of the large number of combinations of tasks that might be active at the same time and because of the continually changing demands on the system, it will generally be impossible to pre-calculate all possible schedules *off-line* to statically guarantee real-time constraints. Our approach provides a method for on-line

*This work was supported by ONR under contracts NOOO14-84-K0734 and NSF under grant DCR-8500332.

dynamic guarantee of deadlines. The keys to our approach are the scheduling algorithm, the design of the operating system, and their synergism.

Real-time systems usually include a real-time kernel [7] [19]. However, most existing real-time kernels [23] are simply stripped down and optimized versions of timesharing operating systems. More specifically, the general characteristics of most *current* real-time kernels typically include:

- a fast context switch,
- a small size (with its associated minimal functionality),
- the ability to respond to external interrupts quickly,
- multi-tasking with task coordination being supported by features such as mailboxes, events, signals, and semaphores,
- fixed or variable sized partitions for memory management (no virtual memory),
- the presence of special sequential files that can accumulate data at a fast rate,
- priority scheduling,
- the minimization of intervals during which interrupts are disabled,
- support of a real-time clock,
- primitives to delay tasks for a fixed amount of time and to pause/resume tasks, and
- special alarms and timeouts.

These features provide a basis for a good set of primitives upon which to build real-time systems. These features are also designed to be fast which a laudable goal. However, fast is a relative term and not sufficient when dealing with real-time constraints. The main problems with these primitives are that they do not *explicitly* address real-time constraints, nor does their use (without extensive simulations) provide system designers with a high degree of confidence that the system will indeed meet its real-time constraints. Even though such kernels are successfully used in today's real-time embedded systems, it is only at extremely high cost and inflexibility. For example, when using the above primitives it is difficult to *predict* how tasks invoked dynamically interact with other active tasks, where blocking over resources will occur, and what the subsequent effect of this interaction and blocking is on the timing constraints of all the tasks. The current technology burdens the designer with the unenviable task of mapping a set of specified real-time constraints into a priority order in such a manner that all tasks will meet their deadlines. It is common practice to attempt to verify real-time constraints under such conditions by extensive and

costly simulations and testing on the actual system [8]. One round of changes is subject to another *extensive* round of testing. As the next generation hard real-time systems become more sophisticated, it will be necessary to develop cheaper ways to guarantee real-time constraints and to meet the flexibility requirements [25] [24]. The main characteristics of *next* generation hard real-time systems are:

- new operating system and task designs to support predictability,
- a high degree of adaptability (short term and long term),
- physical distribution (of multiprocessors) with a high degree of cooperation,
- incorporation of integrated solutions to deal with real-time, reliability, and large system requirements,
- an interface to AI programs, and
- the ability to handle complex applications.

The Spring project at the University of Massachusetts is conducting research into next generation hard real-time systems. The project has 4 major thrusts:

1. The development of dynamic, distributed, on-line real-time scheduling algorithms. This work is well underway with many such algorithms already developed and evaluated, each based on a different set of assumptions. The plan is that any of these algorithms can slip into the kernel depending on the requirements of the application.
2. The development and implementation of the Spring kernel which supports a network of multiprocessors. The design of the kernel is now complete and a plan for rapid prototyping the kernel has been established. The purpose of this paper is to describe the main ideas in the design of the Spring kernel.
3. The development of multiprocessor nodes in order to directly support the kernel and the scheduling algorithm. A preliminary design has been completed.
4. The development of real-time tools. A sophisticated simulator is under development in this area.

The remainder of this paper is organized as follows. Section 2 outlines the goals of the Spring kernel. Section 3 introduces our model of a hard real-time system. Section 4 describes the scheduling algorithm which is the crux of the kernel. This scheduling algorithm avoids unpredictable waiting and accounts for the use of exclusive and shared resources. Section 5 describes the main primitives found in the Spring kernel. This includes the task management primitives, the memory management primitives, the IPC primitives, and a discussion of the I/O subsystem and interrupt handling. The current status of the Spring project is presented in Section 6.

2 Goals

The Spring kernel has an ambitious goal to support highly dynamic, distributed, complex, hard real-time applications. The most essential ingredient of this support is the ability to guarantee real-time constraints. To achieve confidence that real-time constraints can be met requires a new scheduling approach and a kernel design that operates synergistically with the scheduling approach. The major innovations exhibited in the Spring kernel are the scheduling algorithm itself which can dynamically guarantee real-time constraints, and the way in which the rest of the kernel supports the scheduling algorithm. For example, one major feature is that the scheduling algorithm and other primitives of the kernel cooperate to *avoid* blocking, thereby making it possible to attain predictability. Of course, the Spring kernel contains other features which closely resemble functions found in current real-time kernels.

The main characteristics of the Spring kernel are:

- predictability – the kernel itself and the applications that it supports are predictable with respect to deadlines and other real-time constraints,
- flexibility – it is easy to alter application task's requirements including their real-time constraints, the kernel supports dynamic changes to the collection of tasks active at any point in time, and it is possible to add or delete new application tasks over time, and
- the kernel supports a distributed network of multi-processors.

The Spring kernel does not contain support for security, large address spaces, general timesharing (although it does support the concurrent existence of both hard and soft real-time tasks), nor is it intended for development into a production operating system with all of the incumbent overheads and size. In this paper we do not describe the reliability features.

3 System Model

This section presents the basic structure of a distributed real-time system that we are assuming. It is based on the notion of a flexible on-line scheduler that can guarantee that tasks make their deadlines [15]. While the details of our scheduling algorithm and the analysis of them have appeared elsewhere [15], [21], [26], we will repeat the basic ideas of the algorithm in this paper with the intent of showing how it interfaces to the rest of the Spring kernel, and why it is different than today's real-time kernels.

We assume that the Spring system is physically distributed and composed of a network of multiprocessors. Each multiprocessor contains at least one application processor, one or more system processors, and an I/O subsystem. System processors[1] offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and so that this overhead does not cause uncertainty in executing already guaranteed tasks. All system tasks are resident in the memory of the system processors. The I/O subsystem is a separate entity from the Spring kernel and it handles non-critical I/O, slow I/O devices, and fast sensors. The I/O subsystem can be controlled by some current real-time kernel such as VRTX. The I/O subsystem interface to the Spring kernel is best explained after the scheduling algorithm is described so we defer any more discussion of I/O until section 5.4.

It is important to note that although system tasks run on system processors, application tasks can run on both application processors and system processors by explicitly reserving time on the system processors. This only becomes necessary if the surplus processing power of the application processor(s) is (are) not sufficient at a given point in time. If both the application processors and a portion of the system processors are still not sufficient to handle the current load, then we invoke the distributed scheduling portion[2] of our algorithm.

To be more specific, the system processors run the operating system which includes programs such as the scheduling algorithm. The scheduling algorithm separates policy from mechanism and is composed of 4 modules. At the lowest level there exists a dispatcher. The dispatcher is the mechanism of the scheduler and it simply removes the next task from a system task table (STT) that contains all guaranteed tasks already arranged in the proper order for the multiple application processors. The second module is a local scheduler. The local scheduler is responsible for locally *guaranteeing* that a new task can make its deadline, and for ordering the tasks properly in the STT. The third module is the global (distributed) scheduler which attempts to find a site for execution for any task that cannot be locally guaranteed. The final module is a Meta Level Controller which has the responsibility of adapting various parameters by noticing significant changes in the environment and serving as the user interface. The distributed scheduling component and the Meta Level Controller are not discussed any further in this paper since they can be considered upper levels of the OS and are not part of the Spring kernel itself.

As stated above, each node in the Spring system is multiprocessor. The multiprocessor is currently being designed and is to be built from off the shelf components (68020s) and the VME bus. Preliminary designs have shown that it is possible to efficiently support the Spring kernel and applications running on the kernel. Care is taken to assign data structures to certain memories to reduce or even avoid conflict.

[1] Ultimately, system processors could be specifically designed to offer harware support to our system tasks such as the guarantee routine.

[2] See [15] [16] [17] for details on distributed scheduling.

3.1 Tasks

Tasks are execution traces through programs, and are the dispatchable and guaranteeable entities in the system. Tasks are periodic or non-periodic. Non-periodic tasks have deadlines by which they must finish. Periodic tasks are defined to have recurring initializations (reactivated) and deadlines until they are terminated. All tasks are characterized by a worst case computation time and by the resources required. Tasks may have precedence constraints with other tasks, or with I/O devices, or with the arrival or transmission of data, including transmission or reception of data on a bus. There might be contention over resources and this contention must be avoided or resolved so that no deadlines or periods are missed, and that integrity and consistency (based on the semantics of the data) are preserved. Not all tasks are handled by the Spring kernel; some are handled by the I/O subsystem. The scheduling algorithm presented below can handle all the above requirements at a local node, as well as the fact that we are dealing with multiprocessors. The scheduling algorithm presented in this paper does not discuss distributed scheduling, nor precedence constraints that exist across nodes. We have also developed other local scheduling algorithms which make other assumptions. For example, in another algorithm [27] we consider preemptible tasks. In this case, and for best performance, tasks should be preemptible whenever possible, but not in completely unpredictable ways, nor if it means missed deadlines. One goal of the Spring kernel is to allow various scheduling algorithms to be substituted for each other, depending on the requirements of a particular system. This, in a small way, adheres to the *open system* philosophy found in Smalltalk and Cedar.

3.2 Principle of Segmentation

The design of the kernel is based on the principle of segmentation as applied to hard real-time systems. Due to space limitations we cannot fully discuss the use of segmentation and its implications in this paper. Further details on segmentation for hard real-time systems can be found in [22]. We present a brief description of the segmentation principle here to provide some motivation for our kernel design decisions. The reader should be able to see the use of the principle of segmentation througout the description of the kernel in the rest of this paper.

Segmentation is the process of dividing resources of the system into units where the size of the unit is based on various criteria particular to the resource under consideration and to the application requirements. For example, dividing time on a bus into fixed slots with fixed start times is an example of segmentation. Dividing time on a bus into slots of two different sizes is also segmentation which might be more suitable for application environments where there is roughly a bimodal distribution on message sizes. Dividing time into bounded size packets, but allowing the start of the packet to begin anywhere is yet another "more relaxed" example of segmentation in that the packet size is segmented,

but the bus time is not. Allowing variable length messages which can begin at any time is not segmentation. The goals of using segmentation in hard real-time systems are to develop well defined units of each resource, to increase understandability, and to allow us to put these units together by an on-line algorithm in such a manner as to provide predictability with respect to timing constraints.

Segmentation is a powerful concept and is used in many circumstances. For example, timesharing systems relate the size of a page in memory with block sizes on disks. One can refer to this as spatial segmentation. Hardware designers strive to balance the data flow cycle time and data path widths with the timing of the control memory, local store, and main memory so that no one component is either overdesigned or is a bottleneck to performance. Thus hardware designers are integrating spatial (data path widths) and multiple timing segments with respect to the hardware. While segmentation is used in some form in many systems, we are advocating the need to elevate the notion of segmentation to a central *principle* of hard real-time systems. That is, we must extend the integration of spatial and timing segmentation to the system level, not just the hardware level.

For example, in a hard real-time system *time* is one of the most important resources to segment. However, a system is composed of many different time segments, e.g., time segments for the cpu, for the network bus, for the internal bus, for the disk controller, etc. The system is being driven by multiple, coordinated drum beats (i.e., time segments) with significant differences of time granularity being interfaced by various techniques such as latches[3]. Buffers and device virtualization are used when timing differences are greater. For example, memory might be used as a virtual disk, so that time to store data on this virtual disk is fast and predictable, and the actual disk write is done in background mode without a severe timing constraint. Again, one way of thinking of time segmentation is as if the timing and control circuits of a cpu are being extended to the entire system. But timing segmentation is not sufficient by itself. Other resources such as data and memory are segmented with respect to functionality and size, and programs are segmented with respect to functionality, size and time.

4 The Scheduling Algorithm

The scheduling algorithm has two primary contributions: one is the ability to perform a guarantee on-line, and two is the ability to utilize all the nodes of a distributed system for a hard real-time system. One main ingredient of the scheduling algorithm is the guarantee routine.

[3]Latches are flip-flops organized as a storage register used to hold signals for brief periods to overcome small differences in timing between the cpu and other system components.

The basic notion and properties of the guarantee routine have been developed elsewhere [15] and has the following characteristics,

- conflicts over resources are *avoided* thereby eliminating 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),
- there is a separation of dispatching and guarantee allowing these system functions to run in parallel; the dispatcher is always working with a set of tasks which have been validated to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,
- by performing the guarantee calculation when a task arrives there may be time to reallocate the task on another host of the system via the global module of the scheduling algorithm,
- the guarantee can employ different strategies for deciding if a task can meet its deadline as a function of the deadline, resource requirements, and precedence constraints of the incoming task,
- within this approach there is notion of still "possibly" making the deadline even if the task is not guaranteed, that is, if a task is not guaranteed it receives any idle cycles and in parallel there is an attempt to get the task guaranteed on another host of the system subject to location dependent constraints; an alternative is to run an alternative task and/or error handler early, rather than only after a deadline is missed,
- some real-time systems assign fixed size slots to tasks based on their worst case execution times, we guarantee based on worst case times but any unused cpu cycles are automatically reclaimed and not lost,
- the guarantee routine supports the co-existence of hard and soft real-time tasks, and
- the guarantee can be subject to computation time, deadline or period, resource requirements where resources are segmented, priority, precedence constraints, I/O requirements, etc. depending on the specific guarantee algorithm in use in a given system. This is a realistic set of requirements. Current real-time executives provide little support with respect to handling tasks with deadlines and general resource constraints.

The Segmentation principle applied to the kernel provides well defined units of allocation for the scheduling algorithm to work with, and a precedence/allocation graph for each task (known a priori) describe the needs of a task in terms of these units. Each graph

indicates which resources are needed by a task and the length of time a resource must be held (usually for the length of the task execution). The guarantee algorithm maps the precedence/allocation graphs plus timing constraints of tasks into resource segments. In general, this mapping is NP–hard, hence heuristics are required. We now describe the scheduling algorithm and give an example to provide the reader a full understanding of the problem.

The Spring kernel local scheduler considers the problem of scheduling a set of n tasks $\mathcal{T}$, in a system with r resources $\mathcal{R}$. To simplify the discussion, we first describe the algorithm for independent tasks on nodes with a single application processor and a single system processor. At the end of this section we then describe the extensions needed to handle precedence constraints, periodic tasks, and multiple application processors. We now begin the discussion by concentrating on the most difficult aspect, handling the resource requirements of tasks. It is this aspect of scheduling that provides resource conflict avoidance and thereby predictability.

A resource can be used in two different modes: When in *shared mode*, several tasks can use the resource simultaneously; when in *exclusive mode*, only one task can use it at a time. A file or data structure are examples of such resources: a file can be *read* by multiple users simultaneously but can be *written* by a single user only. A CPU, on the other hand, is a resource that can be used only in exclusive mode. Each task $T \in \mathcal{T}$, has

1. *Processing time,* $T_P > 0$,
2. *Deadline,* T_D,
3. *Resource requirements,* $\mathbf{T_R} = (T_R(1), T_R(2), ..., T_R(r))$, where

$$T_R(i) = \begin{cases} 0 & \text{T does not require resource } R_i; \\ 1 & \text{T requires } R_i \text{ in shared mode;} \\ 2 & \text{T requires } R_i \text{ in exclusive mode,} \end{cases}$$

and

4. *Scheduled start time,* T_{sst} (determined from the processing time, deadline, and resource requirements of all tasks).

A *partial schedule* is a subset of the tasks in $\mathcal{T}$ whose scheduled start times have been assigned. A partial schedule S is *feasible* if the scheduled start times are such that all the tasks in S will meet their deadlines, i.e., $\forall T \in S(T_{sst} + T_P \leq T_D)$. For the tasks in a feasible schedule, the resources required by each task are available in the mode required by the task at its scheduled start time. A set of tasks is *schedulable* if there exists a feasible schedule for it. Thus, the scheduler must determine if a feasible schedule for a set of tasks

exists. Also, it should be obvious from the above description that we are interested in non-preemptive scheduling. Thus, once a task begins execution, it will release its resources only after it has executed for T_P units of time.

Suppose tasks in set Γ have been previously scheduled and a new task arrives. We attempt to schedule the set of tasks $\Pi = \Gamma \cup \{\text{new task}\}$. If this set of tasks is found schedulable, the new task is scheduled, otherwise not. In either case, tasks in Γ remain scheduled. If the system we are dealing with is distributed and the task arriving at a node is not schedulable on the node where it arrives, then the scheduler on the node can send the task to some other node which can attempt to schedule it. A scheme for accomplishing such a *decentralized scheduling* is described in [16]. In this paper, we focus on the local scheduling algorithm for a set of tasks on a node having multiple resources.

For a given set of tasks, the problem of finding a feasible schedule is, in fact, a search problem. The structure of the search space is a *search tree*. The *root* of the search tree is the empty schedule. An *intermediate vertex* of the search tree is a partial schedule. A *leaf*, a terminal vertex, is a complete schedule. Note that not all leaves correspond to feasible schedules. The goal of the scheduling algorithm is to search for a leaf that corresponds to a feasible schedule.

An optimal algorithm, in the worst case, may make an exhaustive search which is computationally intractable. In order to make the algorithm computationally tractable even in the worst case, we take a heuristic approach for this search. We develop a heuristic function, H, which can synthesize the various factors affecting real-time scheduling decisions to *actively* direct the scheduling process to a plausible path. That is, on each level of the search, function H is applied to each of the tasks that remain to be scheduled. The task with the minimum value of function H is selected to extend the current (partial) schedule. As a result of the above directed search, even in the worst case, our scheduling algorithm is not exponential. Fortunately, our simulation studies, described in [26], [27], and [28] show that algorithms using linear combinations of simple heuristics perform very well — very close to the optimal algorithm that uses exhaustive search.

The pseudo code for our scheduling algorithm is given in Figure 1. The algorithm maintains two vectors EAT^s and EAT^e, each element of the vector corresponding to a resource. EAT^s and EAT^e, respectively, indicate the earliest available times of resources in shared and exclusive mode, given that the tasks in *schedule* have been scheduled and tasks in the *task_set* remain to be scheduled. At each level of search, according to the earliest available times of resources EAT^s and EAT^e, the algorithm calculates the earliest start time T_{est} for each task which remains to be scheduled. The detailed computation methods for EAT^s, EAT^e, and T_{est} are discussed later.

The algorithm invokes a boolean function called *strongly-feasible*. A feasible partial schedule is said to be strongly-feasible if all schedules obtained by extending this schedule

```
Procedure Scheduler(task_set: task_set_type; var schedule: schedule_type; var schedulable: boolean);

(*parameter task_set is the given set of tasks to be scheduled*)

VAR EAT^e, EAT^c: vector_type; (* Earliest Available Times of Resources *)

BEGIN

 schedule := empty;

 schedulable := true;

 EAT^c := 0;  (* a zero vector*)

 EAT^e := 0;

 WHILE (NOT empty(task_set)) AND (schedulable) DO

 BEGIN

   calculate T_est for each task T ∈ task_set;

   IF NOT strongly-feasible(task_set, schedule) THEN

     schedulable := false;

     ELSE

     BEGIN

       apply function H to each task in task_set;

       let T be the task with the minimum value of function H;

       T_sst := T_est;

       task_set := task_set - T ;

       schedule := append(schedule, T); (* append T to schedule *)

       calculate new values of EAT^e and EAT^c;

     END;

 END;

END;
```

Figure 1: Heuristic Scheduling Algorithm

Tasks	Processing Time	Deadline	Resource Requirements		
			R_1	R_2	R_3
T_1	20	30	2	2	1
T_2	10	90	2	0	2
T_3	15	40	2	1	0
T_4	20	55	0	2	2
T_5	20	65	0	0	1

Table 1: Parameters of 5 Tasks

one more level with any one of the remaining tasks are also feasible. If extending a feasible partial schedule by any one of the remaining task makes the extended schedule infeasible, then in none of the possible future extensions will this task meet its deadline. Hence it is appropriate to stop the search when a partial schedule is not strongly-feasible.

From the pseudo-code, we see that beginning with the empty schedule, the algorithm searches the next level by expanding the current vertex (a partial schedule) to *only* one of its immediate descendants. If the new partial schedule is strongly-feasible, the search continues until a full feasible schedule is met. At this point, the searching process (i.e., the scheduling process) succeeds and the task set is known to be schedulable.

If at any level, a schedule that is not strongly-feasible is met, the algorithm stops the searching (scheduling) process and announces that this set of tasks is not schedulable and typically either an error message is sent, an error handler is executed, or distributed scheduling is invoked. On the other hand it is also possible to extend the algorithm to continue the search even after a failure, for example, by limited backtracking. While we do not discuss backtracking in detail, we will later present some performance results where we allow some limited amount of backtracking.

Let us now consider some scheduling examples to clarify both the algorithm and the above terminology. Assume that we are attempting to schedule a set of 5 tasks having parameters as shown in Table 1.

If the scheduling algorithm uses an H function defined as $H(T) = T_P$, where T_P is the processing time, we have

$$H(T_2) < H(T_i)$$

for all i = 1, 3, 4, and 5. Hence, the algorithm will select T_2 to to be scheduled first. This new schedule is strongly-feasible and hence the scheduling process will continue with the selection of T_3. This partial schedule is not strongly-feasible since T_1 will miss its deadline. Hence the scheduling process stops immediately.

If the scheduling algorithm uses $H(T) = T_D$, then

$$H(T_1) < H(T_i)$$

for all i = 2, 3, 4, and 5. Hence, T_1 will be selected, followed by T_3 and T_4 at subsequent levels. However, at this point the partial schedule is not strongly-feasible since T_5 will miss the deadline. A smarter scheduler, after selecting T_1, would schedule T_5 next, because T_1 and T_5 can run in parallel. Then by scheduling T_3, T_4 and T_2 in this order, every task will finish before its deadline.

Clearly, at each level of search, effectively and correctly selecting the immediate descendant is difficult, but very important for the success of the algorithm. Function H becomes the core of the algorithm.

Before we describe the function H we need to discuss the method used to compute EAT^s, EAT^e, and T_{est}.

EAT^s and EAT^e, indicate the *Earliest Available Times* of resources for *shared* and *exclusive* modes respectively:

$$EAT^s = (EAT_1^s, EAT_2^s, \ldots, EAT_r^s), \text{ and} \quad (1)$$
$$EAT^e = (EAT_1^e, EAT_2^e, \ldots, EAT_r^e) \quad (2)$$

where EAT_i^s (EAT_i^e) is the earliest time when resource R_i will become available for shared (exclusive) usage. Each time the partial schedule is extended, EAT^s and EAT^e will be updated taking into account the newly added task's start time, processing time, and resource requirements. From the definitions of EAT_i^s and EAT_i^e, it is clear that $EAT_i^s \leq EAT_i^e$; between EAT_i^s and EAT_i^e some task(s) is using R_i in shared mode (and hence the resource will not be available for exclusive use until EAT_i^e).

At each level of the search tree, using the following scheme, the scheduler computes T_{est}, the earliest start time for each task T which remains to be scheduled. Since a task can run only when all the resources it needs are available [4]

$$T_{est} = \text{MAX}(EAT_i^u) \quad (3)$$

where $u = s$ if T needs shared use of R_i, or $u = e$ if T needs exclusive use of R_i.

T_{est}, the earliest start time of the task selected to extend the schedule at the current level becomes the task's scheduled start time. The earliest start time of the remaining tasks will be re-computed at the next level. This is done after the scheduler computes the new values of EAT^s and EAT^e according to the parameters of the task selected at the

[4]Here we are assuming that a task can begin execution anytime after it arrives. At the end of this section, we examine the case where this is not true.

current level. We now give the formula for computing the new values for EAT^s and EAT^e, indicated by $EAT^{s'}$ and $EAT^{e'}$.

$$EAT^{s'} = (EAT_1^{s'}, EAT_2^{s'}, \ldots, EAT_r^{s'}), \text{ and} \quad (4)$$
$$EAT^{e'} = (EAT_1^{e'}, EAT_2^{e'}, \ldots, EAT_r^{e'}) \quad (5)$$

where

$$EAT_i^{s'} = \begin{cases} T_{est} + T_P, & \text{if T needs } R_i \text{ in exclusive mode,} \\ EAT_i^s & \text{otherwise.} \end{cases} \quad (6)$$

$$EAT_i^{e'} = \begin{cases} T_{est} + T_P, & \text{if T needs } R_i, \\ EAT_i^e & \text{otherwise.} \end{cases} \quad (7)$$

We now illustrate this via an example. Assume a system having 6 resources $R_1, R_2, \ldots, R_6$. Let current EAT^s and EAT^e be:

$$\begin{aligned} EAT^s &= (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s, EAT_6^s) \\ &= (5, 25, 10, 5, 5, 10), \text{ and} \end{aligned} \quad (8)$$

$$\begin{aligned} EAT^e &= (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e, EAT_6^e) \\ &= (5, 25, 10, 10, 5, 15). \end{aligned} \quad (9)$$

Let task T, which is being selected by the scheduler, have processing time $T_P = 10$, and need exclusive use of R_1 and R_4, and shared use of R_6.

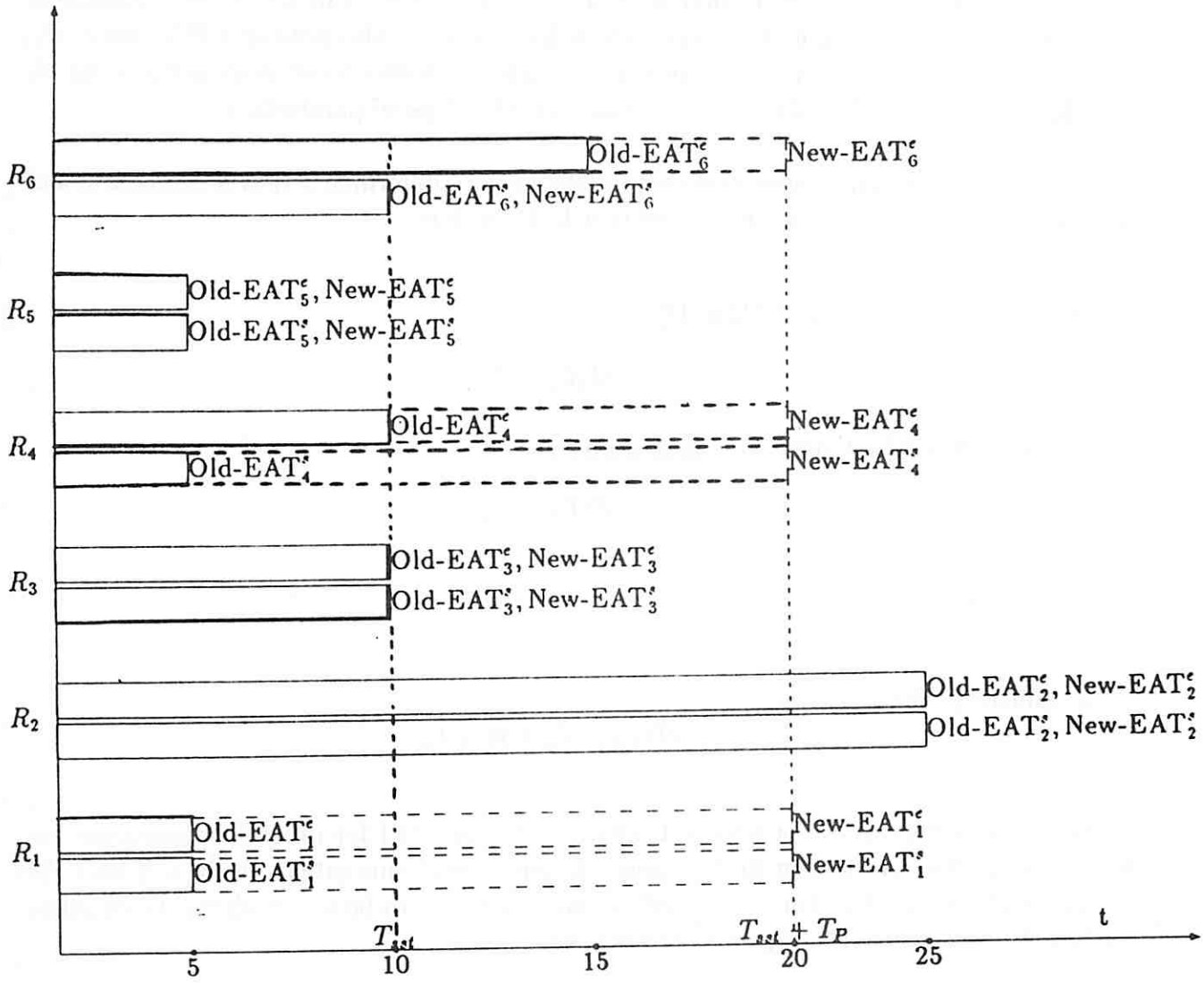
T_{est}, the earliest time T can start, is the earliest time resources needed by T are available in the required modes. So

$$\begin{aligned} T_{est} &= MAX (EAT_1^e, EAT_4^e, EAT_6^s) \\ &= MAX (5, 10, 10) \\ &= 10. \end{aligned} \quad (10)$$

Suppose T is selected to extend the schedule at the current level. Then its scheduled start time is 10. New values for EAT^s and EAT^e, are

$$\begin{aligned} EAT^s &= (EAT_1^s, EAT_2^s, EAT_3^s, EAT_4^s, EAT_5^s, EAT_6^s) \\ &= (20, 25, 10, 20, 5, 10), \text{ and} \end{aligned} \quad (11)$$

$$\begin{aligned} EAT^e &= (EAT_1^e, EAT_2^e, EAT_3^e, EAT_4^e, EAT_5^e, EAT_6^e) \\ &= (20, 25, 10, 20, 5, 20). \end{aligned} \quad (12)$$



Task T requires use of R_1 and R_4 in the exclusive mode, and use of R_6 in the shared mode.

T_P, the processing time of task T, = 10.

$T_{sst} = T_{est} = 10$ is the time T is scheduled to start its execution, and at time 20, it will finish.

Old-EAT: The EAT before T is scheduled.

New-EAT: The EAT after T is scheduled.

Figure 2: Example of Updating EAT Values

This is because T is assumed to start at time 10 and runs for 10 units of time while holding R_1 and R_4 in exclusive mode, and R_6 in shared mode.

Figure 2 presents this example in a graphic form. In Figure 2, the old values of EAT_i^ss and EAT_i^es are presented as solid lines while the new values of EAT_i^ss and EAT_i^es are presented as dashed lines. Note that R_1 will be idle from time 5 to 10. A good scheduling algorithm should be designed to minimize such idle times. Also note that if T, using R_1, R_4, and R_6, is scheduled next, it can run concurrently with one or more tasks using R_2 and R_6. A scheduling algorithm should maximize this type of parallelism.

From extensive simulations reported in [28] we have determined that a combination of two factors is an excellent heuristic function H. Consider:

- minimum deadline first (Min_D):

$$H(T) = T_D;$$

- minimum earliest start time first (Min_S):

$$H(T) = T_{est};$$

and let $H =$

- Min_D + Min_S:

$$H(T) = T_D + W * T_{est};$$

In the above formulas, W is a weight, and may be adjusted for different application environments. We have shown that no single heuristic performs satisfactorily and that the above combination of factors does perform well. These two factors address the deadline and resource contention – the two critical issues.

One important aspect of this study, different from previous work, is that we specifically consider resource requirements and model resource use in two modes: exclusive mode and shared mode. We have shown that by modeling in such way, more task sets are schedulable. Further, this algorithm takes realistic resource requirements into account, and it has the appealing property that it avoids conflicts (thereby avoiding waits) over resources. It is important to note that resource conflicts are solved by scheduling tasks which contend for a given resource at different times. This avoids locking and its consequent unknown delays. If task A is composed of multiple task segments, and task A needs to hold a serially shareable resource across multiple task segments, then that resource is dedicated to task A during that entire period, call it X, and other tasks which need that resource cannot

be scheduled to overlap with task A during time X[5]. Other tasks can overlap with task A. This strategy also minimizes context switches, since tasks are not subject to arbitrary interrupts fueled by tasks arbitrarily waiting for resources. As an aside, if a particular hard real-time system has no conflict over resources except the CPU then it it is possible to assume that resources are always available to ready tasks one may use our preemptive algorithm [27], instead of the non-preemptive algorithm presented in this paper.

Run time cost of the non-preemptive scheduling algorithm is an important factor to consider. We measured the execution time cost for our algorithm on a VAX 11/780 for task sets of different sizes and for 7 critical resources[6] at each site. The cost of the algorithm depends on the number of backtracks allowed.(In applying the H function, we have capped the number of backtracks.) For example, for 10 tasks and no backtracks the algorithm runs in 22.5 ms. For 10 tasks with with a maximum of n^2 backtracks, the algorithm runs in 51.6 ms. For 30 tasks with backtrack we measured the algorithm as needing 160 ms. These figures are presented to give the reader a feeling for the cost of the algorithm. Hence, without further optimizations to the scheduling algorithm, we expect that the Spring kernel is usually dealing with tasks with laxities of 200 ms or greater. However, we believe that significant optimizations on the algorithm are possible, and have preliminary results in this area. Also, there exists a number of possible techniques to integrate tasks with short deadlines into our scheme. We do not present these here.

Many extensions to the algorithm described above are possible: First of all, it is easy to immediately extend the algorithm to handle the case where each resource may have multiple instances and a task may request one or more instances of a resource. For this case, the vectors EAT^s and EAT^e will be matrices, each row corresponding to a resource, and each matrix element corresponding to an instance of a resource. Hence, handling multiple application processors is simple, and is accomplished by making the exclusive resource entry for the processor, a vector.

Second, the algorithm can be extended to handle the case where tasks can be started only after some instance of time in the future. For example, this occurs for periodic tasks, and for non-periodic tasks with future start times. Conceptually, the only modification that needs to be made to our scheme is in the definition of tasks' scheduled start time:

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

where u = s (e) if T needs R_i in shared (exclusive) mode. However, more efficient techniques to handle periodic tasks are being investigated. These techniques are based on a guaranteed template so that periodic tasks need not be guaranteed one instance at a time.

[5]General real-time system design rules encourage a programming style in which no task holds a resource for a long period of time (over many segments).

[6]Many other resources may exist, but if they are not shared in some way then they need not be addressed by the guarantee routine.

Third, in order to handle precedence constraints we simply add another factor to the heuristic function that biases those eligible tasks with long critical paths to be chosen next. Tasks become eligible to execute only when all of its ancestors are scheduled. Again, various optimizations are being investigated here.

5 Other Kernel Features

The Spring kernel design has some similar features to current real-time operating systems such as the VRTX kernel[7], but extends these features in many ways including support for multiprocessors, support for a distributed system, a new scheduling algorithm, and careful rework of some features to make them integrate with our new scheduling algorithm.

The kernel supports the abstractions of tasks, resource objects, and IPC among tasks. Memory is considered as multiple resource objects. Scheduling is an integral part of the kernel and the abstraction provided is one of a guaranteed task. The scheduling algorithm handles resource allocation, *avoids* blocking, and guarantees tasks; the scheduling algorithm is the single most distinguishing feature of the kernel. I/O and interrupts are primarily handled by the front end I/O subsystem. Interrupts handled by the Spring kernel itself are well controlled and accounted for in timing constraints. A brief overview of these aspects of the Spring kernel is now given.

5.1 Task Management

The Spring kernel contains task management primitives that utilize the notion of preallocation where possible to improve speed and to eliminate unpredictable delays. The primary task management primitives include:

- CREATE - a new task structure is set up including a task control block, some minimum amount of memory, etc. This command is only invoked at system initialization time, or possibly after failures to preallocate task structures to support the maximum level of multiprogramming.
- ALLOCATE - assigns a new task a preallocated task slot and fills it with task data where appropriate; however, if task is in memory but passive then the ALLOCATE primitive simply reactivates it.
- DEALLOCATE - this statement removes the assignment of a particular task to a particular task slot; however, the slot is marked as passive but not deleted.

[7] VRTX is a popular current real-time executive.

- DELETE - task is cleared from memory.
- SUSPEND - a task is placed on a wait queue; only a task without a hard real-time deadline can SUSPEND. Such non real-time tasks use either idle cycles or cycles dedicated to non real-time tasks, if any.
- RESUME - this is the mechanism for pulling a SUSPENDed non hard real-time task off the suspend wait list.
- CRITICAL - sets the criticalness of a task. This is normally a static value, but it is possible that the MLC or a human operator might be permitted to modify this parameter.
- ASK - inquires as to the criticalness, the real-time constraint, and/or resources required by this task.
- DELAY - a hard real-time task can DELAY itself only if it is willing to be reevaluated for guarantee again and with a new deadline, i.e., a task DELAYING itself is treated as a future arrival of a non-periodic task (it will be reevaluated at that future arrival time or earlier if spare cycles exist).
- SET - sets real-time constraint; can be issued by application task and other system primitives such as the DELAY primitive.

The task primitives are largely responsible for getting tasks in and out of the ready state.

5.2 Memory Management

While it is desirable to allow as many tasks to share physical memory as possible, it must not be done at the expense of missing timing constraints. That is, memory management techniques must not introduce erratic delays into the execution time of a task. Since page faults and page replacements in demand paging schemes create large and unpredictable delays, the current virtual memory management techniques are not suitable to real-time applications with a need to guarantee timing constraints. Instead, the Operating System memory management adheres to a memory segmentation rule. Let us now provide an example. We require that there be a reasonable amount of memory at each host[8], and that a fixed partition memory management scheme be used. Therefore, we have two primary primitives:

[8]Many real-time systems are composed of disjoint phases, e.g., in the Space Shuttle [6] there are pre-flight processing, liftoff, space cruising, and descent phases. In this type of non-distributed system, the amount of memory needed is enough to contain the largest phase, not the entire system.

- GETMEM
- RELMEM

Tasks require a maximum number of memory objects. This maximum is known a priori.

Let us now tie in memory management with the task primitives, and the scheduling algorithm. In the Spring kernel, the OS is core resident (possible because the OS is smaller than a general purpose OS). A set of N task control blocks (TCBs) are CREATED at system initialization time and thereafter ALLOCATed dynamically to guaranteed tasks. A control region for each task that includes a user and system stack is maintained by the OS within the OS region. Stack sizes are capped; if a task attempts to push onto a full stack, an error occurs (treated similarly as an attempt to divide by zero). Each TCB points to at most M I/O buffers for this process, and Q dynamic data segments including message buffers. I/O buffers, data segments and message buffers are all memory objects. There exists two sizes of code segments. When a task is activated, the guarantee routine determines if it will be able to make its deadline using the algorithm described in section 4. Note that the execution of the guarantee algorithm ensures that the task will obtain the necessary resources such as the TCB, memory blocks, etc. and at the right time. (Tasks always identify their maximum resource requirements; this is feasible in a real-time system). If a task is guaranteed (which includes the time to accomplish the following things), its code segment is filled, the TCB is set up, and it is placed in the system task table (part of OS memory), and marked ready for dispatching. Note that a fixed partition memory management scheme (of multiple sizes) is very effective when the sizes of tasks tend to cluster around certain common values, and this is precisely what our system experiences. Also, pre-allocating as much as possible increases the speed of the OS at a slight loss in generality. One of the engineering issues is where to make this tradeoff between pre-allocating resources and flexibility. A possible problem is what happens when memory is full. Is predictability and the guarantee violated in this case? The answer is no. The reason is that our scheduling algorithm handles this directly because it takes resource requirements into account when guaranteeing. That is, the task is guaranteed if there is a slot of time in which the task can obtain all its required resources (including memory) and finish executing before its deadline.

5.3 IPC

The kernel also supports synchronization and communication with 5 interprocess communication (IPC) primitives. The IPC primitives are:

- SEND - send a message to a mailbox and don't wait

- RECV - receive a message from a mailbox and don't wait
- SENDW - send a message to a mailbox and wait a time B
- RECVW - receive a message from a mailbox and wait a time B
- CREATMB - creates a mailbox; maximum number of CREATMB a task might execute must be known and is a component of the resources requested of a task; mailboxes are memory objects.

Real-time tasks should be programmed with the SEND and RECV primitives whenever possible. In many cases a little thought will enable the programmer to implement real-time tasks with such primitives. In certain cases it may not be possible or desirable to have a task continue without first waiting for a message. A real-time programmer is permitted to wait for a bounded time B in this case. This bounded time is taken into account in guaranteeing a task. When the bound expires the task must still have some logic to decide what to do in this case. This set of primitives along with the guarantee algorithm prevent a process from hanging past its deadline.

In timesharing systems semaphores and monitors are typical solutions for guaranteeing mutual exclusion. While it is possible to implement semaphores and their associated P and V operations using the SENDW and RECVW primitives, our full scheduling approach *specifically avoids the need for semaphores by implementing mutual exclusion directly in the schedule.*

Currently, any task that needs to wait for an event or a combination of events must be programmed to do that with the above bounded SENDW and RECVW primitives, or provide enough information to the scheduler so that it is handled by the scheduler. Because of the features of the the Spring kernel including the scheduling algorithm, the preallocation strategy, the dedication of resources, and the programming strategy of not waiting, we expect the system to be largely free of blocking. Synchronization is achieved largely by the scheduler, and if not in this manner, then by bounded waits and polling. The time for the bounded waits is already accounted for in the guarantee process.

For example, if a sensor is producing data, that data is processed and filtered by a dedicated processor in the I/O subsystem. When the data is ready it is passed to a higher level task whose job it is to take action on that data. Hence, an implicit precedence relation exists. Passing the data to a higher level task means that the data is left for the higher level task, which has been guaranteed to run periodically. Now it may happen that when the higher level task runs the data is not actually ready (i.e., the precedence constraint is not met). The higher level task might then do a bounded wait if it were already accounted for in the guarantee, or if the application semantics permits it just goes on and performs some action such as letting the next periodic instance of the higher level task operate on

the data if it becomes available by that time, or performs yet some other action such as using an estimation made from the old data. Again this depends on the application.

Note that the above primitives can also be used to implement a TIMED MONITOR. A timed monitor provides mutual exclusion with a bounded delay and is a useful primitive for real-time applications.

5.4 I/O

Many of the real-time constraints in a system arise due to I/O devices including sensors. The set of I/O devices that exist for a given application will be quite static in most systems. Special independent driver processes must be designed to handle the special timing needs of these devices. In Spring we separate slow and fast I/O devices. Slow I/O devices are multiplexed through a front end dedicated I/O processor. System support for this is preallocated and not part of the dynamic on-line guarantee. However, the slow I/O devices might invoke a task which does have a deadline and is subject to the guarantee. The slow front end is supported by a VRTX kernel [23] and its IOX package. This is acceptable to us because the front end is relatively slow, not critical, and reasonably inflexible. Fast I/O devices such as sensors are handled with a dedicated processor, or have dedicated cycles on a given processor or bus. The fast I/O devices are critical since they more closely interact with the real-time application and have tight time constraints. They might generate subsequent real-time higher level tasks for the Spring kernel. However, it is precisely because of the tight timing constraints and the relative static nature of the collection of sensors that we pre-allocate resources for the fast I/O sensors. In summary, our strategy suggests that many tasks which have real-time constraints can be dealt with statically, leaving a smaller number of tasks which typically have higher levels of functionality and higher laxity for the dynamic, on-line guarantee routine.

Note that VLSI techniques have provided the capability for the peripheral subsystem (in all computer systems) to attain a much higher level of intelligence, autonomy and processing power than in the past. For this reason, peripheral functions are becoming more of a factor in overall system design. While peripheral subsystems have always played an important role in real-time systems, the new potential created by VLSI needs to be exploited to fully realize the I/O segmentation advantages.

5.5 Interrupts

Another important issue is interrupts. Interrupts are an environment consideration which causes problems because they can create unpredictable delays if treated as a random pro-

cess as is done in most timesharing operating systems. Further, in most timesharing systems, the operating system often gives higher priority to interrupt handling routines than that given to application tasks, because interrupt handling routines usually deal with I/O devices that have real-time constraints, whereas most application programs in timesharing systems don't. In the context of a real-time system, this assumption is certainly invalid because the application task delayed by interrupt handling routines could in fact be more urgent. Therefore, interrupts are a form of event driven scheduling, and, in fact, the Spring system can be viewed as having two schedulers: one that schedules interrupts (usually immediately) on the front end processors in the I/O subsystem (what was discussed above), and the other that is part of the Spring kernel proper that schedules all other tasks. Interrupts are then treated as instantiating a new task which is subject to the guarantee routine just like any other task. If deadlines are very short for the tasks invoked via interrupts and to be handled by the Spring kernel scheduler, then time for these tasks might have to be preallocated.

Intraprocessor interrupts like zero divide, overflow, accessing restricted memory, execution of a priviledged instruction, machine failure, and parity errors are considered errors. The current task is in error (maybe through no fault of its own) and may miss its deadline. If an error is recoverable and the executing task can still execute within its worst case time, then our system would still permit it to execute by its deadline and this would not affect other tasks because all of the tasks have been guaranteed with respect to worst case times[9]. If a task needs to execute longer than its anticipated worst case time, say to handle error recovery, then it is possible that it may still execute by using idle cycles up to its deadline, but it is treated as a task without hard real-time constraints. If it does not finish in time, this is considered an error, just like a divide by zero.

Overall, the dilemma caused by interrupts is that interrupts are unpredictable, but we require predictability. Since we are interested in real-time systems which are largely interrupt driven, we must also keep interrupts turned on as much as possible. However, our design isolates guaranteed application tasks from the interrupts. I/O interrupts are handled in the front ends, interrupts from the front ends into the Spring kernel are handled by the system processors and doesn't affect the application tasks, and application tasks generally run to completion with any acceptable (and known) interrupt times accounted for in their specification of their worst case computation time.

If the system degrades to one processor for the Spring kernel and the application tasks then there is more of a problem. The system (and guarantee routine) must allow for enough free cycles to handle a bounded (including zero) number of interrupts at any given time. Processing time for interrupts must be accounted for as well as the impact of this time on the guarantees of tasks, i.e., a guarantee is done with respect to a certain number and type of interrupts being permitted. The bound is prevented from being exceeded by masking interrupts at the appropriate times, allowing interrupts to occur only at the beginning

[9]In general, real-time system designers and programmers need to prevent long worst case times for tasks.

of cpu time segments, and not allowing interrupts to affect the application processors by pre-allocation of resources or by dedicated devices with more autonomy.

6 Current Status

The Spring project has 4 major thrusts. The first one is the development of dynamic on-line real-time scheduling algorithms. This work is well underway with many such algorithms already developed and evaluated, each based on a different set of assumptions. The plan is that any of these algorithms can slip into the kernel depending on the requirements of the application. The second major thrust is the development and implementation of the Spring kernel. The design of the kernel is now complete. and a plan for rapid prototyping the kernel has been established. As described in this paper, we believe that the Spring kernel contains a number of innovative features for hard real-time systems. The third major thrust of the Spring Project is to build the multiprocessor nodes so as to directly support the kernel and the scheduling algorithm. A preliminary design has been completed, and we are beginning to construct the first of these nodes. The last major research area within the Spring Project is the development of real-time tools; a sophisticated simulator is being developed in this area.

References

[1] Alger, L. and J. Lala, "Real-Time Operating System For A Nuclear Power Plant Computer," *Proc. 1986 Real-Time Systems Symposium*, Dec. 1986.

[2] Blazewicz, J., and Weglarz, J., "Scheduling under Resource Constraints — Achievements and Prospects", *Performance of Computer Systems*, edited by M. Arato, A. Butrimento, and E. Gelenbe, North-Holland Publishing Company, 1979.

[3] Blazewicz, J., "Deadline Scheduling of Tasks with Ready Times and Resource Constraints", *Information Processing Letters*, Vol. 8, No. 2, February 1979.

[4] Blazewicz, J., Lenstra, J. K. and Kan, A. H. G. R., "Scheduling Subject to Resource Constraints: Classification and Complexity", *Discrete Applied Mathematics*, 5, 1983.

[5] Bonuccelli, M. and Bovet, D. P., "Scheduling Unit Time Independent Tasks on Dedicated Resource Systems", *Report S-86-21, Univ. degli studi di Pisa, Instituo di Scienze dell Informazione*, 1976.

[6] Carlow, G., "Architecture of the Space Shuttle Primary Avionics Software System," CACM, Vol. 27, No. 9, Sept. 1984.

[7] Daniels, D. and H. Wedde, "Real-Time Performance of a Completely Distributed Operating System," *Proc. 1986 Real-Time Systems Symposuim*, Dec. 1986.

[8] Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transaction on Software Engineering*, Vol. Se-11, No. 1, January 1985.

[9] Garey, M.R., and Johnson D.S., "Complexity Results for Multiprocessor Scheduling under Resource Constraints", *SIAM J. Comput.*, 4, 1975.

[10] Graham, R.L., Lawler, E.L., Lenstra, J.K., and A.H.G.R. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey", *Annals of Discrete Mathematics*, 5, 1979.

[11] Leinbaugh, D., "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Trans on Soft Eng*, Vol. SE-6, January 1980.

[12] Lenstra, J. K. Rinnooy, A. H. G., and Brucker, P., "Complexity of Machine Scheduling Problems", *Annals of Discrete Mathematics*, 1. 1977.

[13] Liu, C, and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, January 1973.

[14] Mok, A., and L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conf. Comp Sys*, Nov. 1978.

[15] Ramamritham, K. and J. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, Vol. 1, No. 3, July 1984.

[16] Ramamritham, K., J. Stankovic, and W. Zhao, "Distributed Scheduling of Hard Real Time Tasks Under Resource Constraints in the Spring System," submitted to *IEEE Transactions on Computers*, Feb. 1986.

[17] Ramamritham, K., J. Stankovic, W. Zhao, "Meta-Level Control in Distributed Real-Time Systems," submitted to 7th Distributed Computing Conference, December 1986.

[18] Schrage, L., "Solving Resource-Constrained Network Problems by Implicit Enumeration — Non-preemptive Case", *Operations Research*, 10, 1970.

[19] Schwan, K., W. Bo and P. Gopinath, "A High Performance, Object-Based Operating System for Real-Time, Robotics Application," *Proc. 1986 Real-Time Systems Symposium*, Dec. 1986.

[20] Sha, Lui, J. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc 1986 Real-Time Systems Symposium*, Dec. 1986.

[21] Stankovic, J., K. Ramamritham, and S. Cheng, "Evaluation of a Bidding Algorithm for Hard Real-Time Distributed Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, Dec. 1985.

[22] Stankovic, J., and L. Sha, "The Principle of Segmentation," Technical Report, 1987.

[23] VRTX/68020 User's Guide, Hunter and Ready, Doc No. 591333001, August 1986.

[24] Ward, Paul, and S. Mellor, *Structured Development for Real-Time Systems*, Yourdan Press, Vol. 1 and Vol. 2, N.Y., N.Y., 1985.

[25] Wirth, N., "Toward a Discipline of Real-Time Programming," *Communications of the ACM*, Vol. 20, No. 8, August 1977.

[26] Zhao, W., Ramamritham, K., and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," to appear *IEEE Transactions on Software Engineering*, May 1987.

[27] Zhao, W., Ramamritham, K. and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," to appear *IEEE Transactions on Computers*, August 1987.

[28] Zhao, W. and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," to appear in *Journal of Systems and Software*, 1987.