OVERVIEW OF THE SPRING PROJECT

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

COINS Technical Report 89-03
Replaces COINS TR 88-51

January 11, 1989

OVERVIEW OF THE SPRING PROJECT [1]

Krithi Ramamritham
(413) 545-0196

John A. Stankovic
(413) 545-0720

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

January 11, 1989

ABSTRACT

The Spring Project at the University of Massachusetts is a research and development effort aimed at studying *next generation* time-critical systems. In addition to being fast and predictable, these systems will have to be flexible, adaptive, and reliable. These requirements arise from the fact that future systems will be large and complex and will operate in environments that are dynamic, distributed, and fault-inducing. In order to achieve its goals, the Spring Project takes a synergistic approach involving the development of scheduling algorithms, operating system support, distributed system architecture, and application development tools. In addition, the following topics are also being investigated as part of the project: Transaction management in real-time databases, support for real-time applications involving artificial intelligence, and formal approaches to the specification and verification of real-time systems. This report summarizes the current status of the project.

[1]This work is part of the Spring Project at the University of Massachusetts which is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332. We also acknowledge the support of Texas Instruments.

1 Introduction to the Spring Project

A number of new and sophisticated applications are currently being contemplated by government and industry. Space stations, automated factories of the future, and future command and control systems are examples of such systems. These applications exhibit a set of common features:

- They will be large and complex.
- They will function in physically distributed environments.
- They will have to be maintainable and extensible due to their evolving nature and projected long lifetimes.
- They will consist of many interacting time-critical components.
- They will result in severe consequences if logical and timing correctness are not met.

Such systems in which tasks have deadlines that must be met are termed ***distributed time-critical*** systems or ***distributed hard real-time*** systems.

Available tools for the development of such systems are woefully inadequate since they are primarily aimed at applications that are static, operate in centralized environments, and either ignore explicit timing constraints or treat them in a very ad hoc manner. Also, many real-time systems are founded on the premise that time-critical systems need to be ***fast***, as opposed to being ***fast and predictable.*** Clearly, there is a need for a ***fresh and flexible*** approach to building distributed time-critical systems. Hence the Spring project.

In order to achieve the major goals of high performance (i.e., the need to be fast) and predictability, we are currently exploring the following areas in a synergistic fashion:

- Scheduling algorithms for distributed time-critical systems.
- Operating system support for time-critical systems.
- Architectural support for time-critical systems.
- Tool support for building time-critical systems.
- Protocols for time constrained communication.

In addition, transaction management for real–time databases, support for real-time applications involving artificial intelligence, and formal approaches needed for specifying and verifying real-time systems are also being investigated as part of the Spring project.

To motivate the approach taken by the Spring Project, we first discuss the limitations of current real-time systems and then present the new paradigm adopted in Spring. This is followed by a summary of our work in each of the areas mentioned above.

2 Limitations of the Current Approach to Real-Time Systems

Today's real-time systems attempt to support time critical applications by implementing a set of primitives which are very fast. This is a laudable goal. However, fast is a relative term and not sufficient when dealing with real-time constraints. The main problems with simply making timesharing operating systems fast is that it is the wrong paradigm, there is no *explicit* support for meeting real-time constraints, and the use (without extensive simulations) of these systems does not provide system designers with a high degree of confidence that the system will indeed meet its real-time constraints. For example, the current technology burdens the designers with the unenviable task of mapping a set of real-time constraints into a priority order such that all tasks will meet their deadlines. Thus, when using the current paradigms it is difficult to *predict* how tasks may interact with each other, 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. As real-time systems become more dynamic and more sophisticated, it will be necessary to develop more effective ways to guarantee real-time constraints and to meet the flexibility and predictability requirements.

Hence, our claim is that the basic paradigms for current real-time operating systems are wrong. In real-time systems the behavior of each task is well understood as a function of the state of the system and inputs. There is no need to treat it as a random process. Fairness and minimizing average response time are not important in a real-time system. What is important is that *all* critical tasks complete by their deadlines and that as many other tasks as possible also complete by their deadlines (usually weighted by their importance). In other words, more appropriate metrics than fairness and average response time are required. Finally, a real-time system supports a single application with all the tasks acting as members of a team to accomplish the best system-wide behavior.

In addition to the above problems with the basic paradigms of current operating systems, many other more specific problems exist with today's real-time systems. For example, many real-time systems are highly static and consequently contain static scheduling policies. However, for next generation real-time systems the dynamics, the need for adaptability and reliability, and the inherent complexity will make it impossible to precalculate all possible combinations of tasks that might occur. This precludes use of static scheduling policies. Further, even in those current kernels where more dynamic scheduling algorithms occur, they are inadequate for two main reasons: (1) they do not address the need for an integrated cpu scheduling and resource allocation scheme, and (2) they don't handle the end-to-end scheduling problem. We will further discuss these two important issues in subsequent sections.

We believe that the new real-time OS paradigm should be based on the following considerations:

- tasks in a real-time application are known a priori and hence can be analyzed to determine their characteristics. Tasks are part of a single application with a system-wide objective,
- the value of tasks executed should be maximized, where the value of a task that completes before its deadline is its full value (depends on what the task does) and some diminished value (including a very negative value or zero) if it does not make its deadline,
- predictability should be ensured so that the timing properties of both individual tasks and the system can be assessed (in other words we have to be able to categorize the performance of tasks and the system with respect to their timing properties), and
- flexibility should be ensured so that system modifications and on-line dynamics are more easily accommodated.

The next section shows how the Spring paradigm *is* based on these considerations.

3 The Paradigm Underlying Spring

This section presents the basic environment we are assuming and the general structure for the real-time operating system paradigm we are proposing.

We assume that the environment is dynamic, large and complex. In this environment there exists many types of tasks. There are critical tasks, essential tasks, non-essential tasks, and tasks' deadlines may range over a wide spectrum. *Critical* tasks are those tasks which must make their deadline, otherwise a catastrophic result might occur (missing their deadlines will contribute a minus infinity value to the system). These tasks must be verified to always be able to meet their deadlines subject to some specified number of failures. Resources will be reserved for such tasks. That is, a worst case anaysis must be done for these tasks to guarantee that their deadlines are met. Using current OS paradigms such a worst case analysis, even for a small number of tasks is complex. Our more predictable kernel facilitates this worst case analysis. Note that the number of truly critical tasks (even in very large systems) will be small in comparison to the total number of tasks in the system. *Essential* tasks are tasks that have deadlines and are important to the operation of the system, but will not cause a catastrophy if they are not finished on time. There are a large number of such tasks. 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. Our approach applies an on-line, dynamic guarantee to this collection of 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, maintenance functions, etc. fall into this category. Some non-critical tasks may have extremely tight deadlines. These tasks cannot be dynamically scheduled since it would take more time to ascertain the schedule than there exists before the task's deadline. Such tasks must also have preallocated resources. These tasks usually occur in the data acquisition front ends of the real-time system.

Task characteristics are complicated in many other ways as well. For example, tasks may be preemptable or not, periodic or aperiodic, have a variety of timing constraints, precedence constraints and communication constraints. All these task characteristics must be addressed together with the dynamic, distributed and evolving environment characteristics.

In light of these complexities, the key to next generation real-time operating systems will be finding the correct approach to make the systems predictable yet flexible in such a way as to be able to guarantee and predict the performance of the system. Our approach to supporting this new paradigm combines the following ideas resulting, we believe, in a flexible yet predictable system:

- resource segmentation/partitioning,
- functional partitioning,
- selective preallocation,
- a priori guarantee for critical tasks,
- an on-line guarantee for essential tasks,

- integrated cpu scheduling and resource allocation, and
- end-to-end scheduling.

4 System and Task Characteristics

Given that future time-critical systems are expected to function in physically distributed environments, it is appropriate for the computing environment also to be distributed. Since we are interested in high performance and reliability, each node in this distributed system should contain multiple processors and the nodes themselves connected by a high-speed subnet. However, several questions still remain, especially those relating to the architecture of each node and the nature of the subnet. Hence we are proceeding in the following two directions.

First, since we do need a working hardware configuration to implement, test, and evaluate the kernel, we have put together, using off-the-shelf components, a distributed architecture, called *SpringNet* that while being reasonable, is also easily modifiable. In the current implementation, each node consists of a number of processors (Motorola 68020's) and a number of memory modules all on a common bus (VME). We plan to configure the processors and the memory such that each processor has quick access to one specific memory module while having relatively slower access to the remaining memory modules. Thus all the memory is shareable by all the processors.

Concurrently, we are building a software testbed for experimenting with alternative distributed architectures. This testbed serves as a tool for designers and implementers of time-critical systems. We discuss the software testbed tool in section 7.

In the rest of this section, we discuss the contents of a node in the Spring network and the characteristics of tasks being considered by Spring.

4.1 A Spring Node

We assume that the Spring system is physically distributed and composed of a network of multiprocessors. Each multiprocessor contains one (or more) application processors, one (or more) system processors, and an I/O subsystem. System processors[2]

[2]Ultimately, system processors could be specifically designed to offer harware support to our system activities such as guaranteeing tasks.

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 guaranteed tasks. All system tasks are resident in the memory module 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 $VRTX^{TM}$, or by completely dedicating processors or cycles on processors to these devices.

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. To facilitate this, the code for tasks is replicated at various nodes, so that only signals, partial state information, or input to the tasks need be transmitted when distributed scheduling occurs, rather than transmitting the task code itself.

To be more specific, the system processors run most of the operating system, as well as application specific tasks that do not have deadlines. The scheduling algorithm separates policy from mechanism and is composed of 4 modules. At the lowest level multiple dispatchers exist, one running on each of the application processors. The dispatcher simply removes the next (ready) task from a system task table (STT) that contains all guaranteed tasks arranged in the proper order for each application processor. The rest of the scheduling modules are executed on the system processor. The second module is a local scheduler. The local scheduler can be used in two ways. First, 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 logic involved in this algorithm is a major innovation of our work. Second, the local scheduler can also be invoked as a *time* planner – valuable for real-time AI applications. This important idea means that it is possible to consider the impact of system level allocations and resource conflicts on the execution time properties of application tasks and that this information can then be used by the application to more accurately accomplish goals on time. The third scheduling module is the global (distributed) scheduler which attempts to find an execution site for any task that cannot be locally guaranteed. The final module is a Meta Level Controller (MLC) which has the responsibility of adapting various parameters or switching scheduling algorithms by noticing significant changes in the environment. These capabilities of the MLC support some of the dynamics required by next generation real-time systems. All system tasks that run on the system processor have a minimum periodic rate which is guaranteed. However, they can also be invoked asynchronously due to events such as the arrival of a new task, if that asynchronous invocation would not violate the periodic execution constraint of other system tasks. Asynchronous events are ordered by importance, e.g., a local scheduling routine is of higher importance than the meta level controller.

4.2 Characteristics of Tasks

Tasks are characterized by:

- ID
- group ID, if any (tasks may be part of a task group or a dependent task group - these are more fully explained below)
- C (a worse case execution time) (may be a formula that depends on various input data and/or state information)
- deadline (D) or period or other real–time constraints
- criticalness (this is an indication of the importance of this task)
- preemptive or non-preemptive property
- maximum number and type of resources (this includes memory segments, ports, etc.) needed
- type: non real-time or real-time
- incremental task or not (incremental tasks compute an answer immediately and then continue to refine the answer for the rest of its requested computation time)
- precedence graph (describes the required precedence among tasks in a task group or a dependent task group)
- communication graph (list of tasks with which a task communicates), and type of communication (asynchronous or synchronous)
- location of task copies
- conditional precedence (not discussed in this report)

5 Scheduling Algorithms for Distributed Time-Critical Systems

Our scheduling algorithms are characterized by the fact that they are ***decentralized*** - scheduling components at individual nodes cooperate to schedule tasks, ***dynamic*** - tasks are scheduled as they arrive in the system, and ***adaptive*** - the algorithm adapts to changes in the state of the system.

In traditional real-time systems, tasks are scheduled according to some policy, say based on their priorities, and if a task does not complete before its deadline, an exception condition is raised. We believe that because of the time-critical nature of the system, the check for whether or not an arriving task will meet its deadline should be done soon after task arrival and an exception condition raised if necessary. In this case, the initiator of the task will have more time to handle the exception condition than in traditional approaches. The nature of exception handling is dependent on the application; the task initiator may resubmit the task with a later deadline or greater importance, or initiate an exception-handling task with greater importance. This important feature of our algorithm has implications for improved fault tolerance.

One of the key notions of our scheme is the notion of *guarantee*. Our scheduling algorithm is designed so that as soon as a task arrives, the algorithm attempts to *guarantee* the task. The guarantee means that barring failures [3] and the arrival of higher importance tasks, this task will execute by its deadline, and that all previously guaranteed tasks with equal or higher importance will also still meet their deadlines. This notion of *guarantee* underlies our approach to scheduling and distinguishes our work from other scheduling schemes. It is also one of the major ingredients for developing flexible, maintainable, predictable, and reliable real-time systems - our major goal.

5.1 Overview of the Scheduling Scheme

Soon after a task arrives at a node, the scheduling component on that node invokes the guarantee routine to determine if that task can be executed locally and completed before its deadline. If the task is not guaranteed locally, then the scheduling component on that node communicates with its counterparts on other nodes to determine if any other node is in a position to guarantee the task. (A copy of the task will already exist on multiple nodes in the distributed system.) If one such node

[3]Failures are handled by various techniques such as guaranteeing multiple instances of a task with proper timing considerations among the instances, other types of task replication, and reallocation of tasks after a host fails [16].

exists, then a signal is sent to that node and a guarantee is attempted on that node. The guarantee algorithm as well as the scheme used for cooperation *explicitly* take the timing and resource constraints into account. Thus, ***ours is a scheduler that is driven by the timing and resource constraints rather than by priorities which encode the timing constraints.***

5.2 The Notion of Guarantee

The basic notion and properties of guarantee have been developed elsewhere [6] and have the following characteristics:

- the approach of providing for on-line dynamic guarantee of deadlines for essential tasks allows the unique abstraction that at any point in time the operating system knows exactly what set of tasks are guaranteed to make their deadlines, what, where and when spare resources exist or will exist, and which tasks are running under non-guaranteed assumptions, (in effect, the algorithm is an on-line time planner),
- it integrates cpu scheduling with resource allocation,
- 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 previously validated to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,
- early notification: 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; 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,
- using precedence constraints it is possible to guarantee end-to-end timing constraints,
- within this approach there is a 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,

- 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 reclaimed when resource conflicts don't prohibit this reclamation,
- worst case execution time is computed for a specific invocation of a task and hence will be less pessimistic than the absolute worst case execution time,
- the guarantee routine supports the co-existence of real-time and non real-time tasks, and
- the guarantee can be subject to computation time requirements, deadline or periodic time constraints, resource requirements where resources are segmented, criticalness levels for tasks, 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.

5.3 Details of the Scheduling Algorithms

Due to the real-time constraints on tasks, the scheduling algorithm itself should be very efficient. That is, we must minimize the scheduling and communication delays. This implies that the decisions, such as whether a task can be guaranteed on a node as well as where to send the task when it cannot be guaranteed locally, must be made efficiently. The problem of determining an optimal schedule even in a multiprocessor system is known to be NP-hard. A distributed system introduces further problems due to communication delays. All of these factors necessitate a heuristic approach to scheduling.

We began our explorations into specific scheduling algorithms by developing algorithms for scheduling simple tasks and then progressively extended our algorithms to deal with tasks having more complex structures and requirements. Following this approach we have developed a number of variants of our scheduling algorithms, the differences between the variants arising from the factors they take into account.

In the basic version of our scheduling algorithm, only timing constraints, i.e., tasks' computation times and deadlines were taken into account [6]. We considered both periodic tasks and nonperiodic tasks. After evaluating this algorithm [13] [14], we extended it to handle, among other things, resource requirements of tasks. This is a significant accomplishment because handling resources is a complicated problem ignored by most researchers. Our work as described in [29] presents a non-preemptive algorithm for guaranteeing tasks that have deadlines and need resources in exclusive mode. In [32], we consider the situation where resources can be used in both shared as well as exclusive modes. Preemptive scheduling on a node is the subject of [30].

Scheduling on multiprocessors is the subject of [10] and [23]. In particular, in [10] we were able to optimize a multiprocessor real-time scheduling algorithm so that it runs in linear time and still provides excellent performance.

We have also developed a suite of algorithms for dealing with a task that is not guaranteed locally, i.e., for distributed scheduling:

- 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. The simulation studies examine how communication delay, task laxity, load differences on the nodes, and task computation times affect the performance of the algorithms. The results show that distributed scheduling is effective even in a hard real-time environment and that the relative performance of these algorithms is a function of the system state [6], [14], [26], [9], and [27].

In parallel with the extensions involving resource constraints, we considered extensions to the basic algorithm to include precedence constraints among tasks. In our approach [3,4], a task consisting of subtasks related by precedence constraints is scheduled in an atomic fashion. We assume that the computation costs of subtasks as well as the communication costs between subtasks are known when a task arrives at a node. Nodes attempt, in parallel, to schedule subtasks within the constraints imposed by precedence relationships; thus, once guaranteed, subtasks can be executed in parallel at different nodes. [4] reports on evaluation of this scheduling strategy as well as its efficacy in different situations.

We have also developed and evaluated two algorithms which integrate both deadline constraints and criticalness factors in making scheduling decisions [1,2]. Any realistic scheduling algorithm must consider the importance of the tasks along with their timing constraints.

During the evaluation of the various algorithms, we made the following observation: In a dynamic system where the system state and task characteristics change dynamically, no single scheduling algorithm performs well in all situations. We need to select the algorithm(s) needed for a particular situation depending on the state of the nodes and the communication network as well as the task characteristics. In one sense, what this amounts to is the *control of scheduling* [7]. Since scheduling is the control of task executions, we term this higher-level control as ***meta-level control*** [7]. Such control will enhance the *adaptability* of resource allocation schemes and since adaptability is one of our goals, we are currently studying the problem of meta-level control in time-critical systems. Meta-level control can be used for the following: Selecting the algorithm(s) used for scheduling tasks on a node and for cooperation among nodes and selecting the values of scheduling parameters used in the chosen algorithm(s). Given the potential uses of meta-level control, a question that deserves special attention is the price. vs. performance of meta-level control techniques. We are currently seeking an answer to this question by investigating various ways of implementing meta-level control, and the cost and complexity of meta-level control, in particular, its communication and processing costs. Meta-level control can also serve as an interface to application semantics and we are also investigating this possibility. In particular we are looking at an air traffic control application and an avionics application both of which may be running some form of expert system.

Even though we believe that we have made a number of substantial contributions in the area of scheduling in time-critical systems, a number of problems still remain. These include:

- Integrated scheduling schemes for nonperiodic tasks which have deadlines, resource requirements, criticalness, precedence constraints, and placement constraints.
- Scheduling such complex nonperiodic tasks in the presence of complex periodic tasks.
- Coping with resources other than those on individual nodes, in particular, the communication subnet.
- Scheduling tasks with precedence constraints on multiple nodes; in a complete scheduling scheme, the scheduling of these tasks will have to be done in conjunction with the scheduling of messages along the communication subnet.
- Strategies for scheduling tasks with a wide spectrum of timing constraints, i.e., where tasks have a large range of deadlines.

- Scheduling schemes for soft real-time tasks coexisting with hard real-time tasks.

We are in the process of extending our current scheduling schemes to deal with the first two issues. Also, we have made a beginning with regard to the issue of dealing with the communication subnet. Specifically, we have developed an algorithm that can be used for preallocation of processor and network resources for periodic *safety-critical* tasks that have substasks with communication, fault-tolerance, and precedence constraints [11]. Evaluation of this algorithm is currently in progress.

In summary, our work on scheduling continues, as we seek integrated solutions that take into account the complex characteristics of tasks in time-critical systems and the nature of resources that these tasks require. In addition we are exploring the efficacy of meta-level techniques that will contribute to the flexibility and adaptability of the solutions.

Recall that our scheduling algorithm is based on the notion of guaranteeing that a task will complete execution before its deadline. For this guarantee to be useful, it should be done with respect to a task's worst-case requirements. For this guarantee to hold, in a sense, the scheduling algorithm has to reserve the resources needed by a task with respect to its worst-case behavior. Since a task has resource requirements, may have complex precedence constraints, and may invoke operating system primitives, a task's worst-case behavior should be determined with respect to its worst-case resource requirements, the worst-case communication time between its subtasks, and the worst-case execution time for the operating system primitives. Needless to say, if a task's worst-case parameters are much larger than their average-case parameters, an underutilized system results. This suggests careful design of the architecture and operating system underlying time-critical systems and the provision of design rules and constraints for the tasks that constitute an application. The rules and constraints can then form the basis for the tools that aid in the development of time-critical applications. It should also be pointed out that our scheduling approach is actually doing on-line *planning* for resource allocation under time, resource, and criticalness constraints. Such a planning capability can be used by higher level application tasks, possibly integrating additional semantics into the scheduler. Such extensions are being considered at this time (see section 10).

6 The Spring Kernel

As was already pointed out, the crucial ingredient of our kernel, called *the Spring kernel* [17,22] is the ability to guarantee a task with respect to its real-time constraints. The major innovations exhibited in the Spring kernel lie in the scheduling algorithm itself and in the way in which the rest of the kernel supports the scheduling algorithm. Of course, the kernel contains features which closely resemble functions found in other real-time kernels. The difference is that extreme care has been taken to ensure predictability of system tasks which when coupled with our scheduling algorithm provides predictability for the application. This predictability of the former implies that we know how long system tasks take to execute and what their resource requirements are. Predictability of the application assures us about the timely execution of tasks that form the application as well as their resource requirements.

The kernel supports the abstractions of tasks, various resource segments such as code, Task Control Blocks (TCBs), Task Descriptors (TDs), local data, data, ports, non segmented memory, and IPC among tasks. It is possible to share memory (one or more data segments) between tasks. Scheduling is an integral part of the kernel and the abstraction provided is one of a guaranteed task set. 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 I/O interrupts are primarily handled by the front end I/O subsystem. It is important to note that the Spring kernel could be considered a back-end hard real-time kernel that deals with deadlines of high level tasks. Because of this, interrupts handled by the Spring kernel itself are well controlled and accounted for in timing constraints.

To enhance predictability, system primitives have capped execution times, and some primitives execute as iterative algorithms where the number of iterations it will currently make depends on state information including available time.

A brief overview of several of these aspects of the Spring kernel is now given.

6.1 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. For example, all tasks with hard real-time requirements are core resident, or are made core resident before they can be invoked with hard deadlines. In addition, a system initialization program loads code, set up TCBs, TDs, local data, data, ports, and non segmented memory using the kernel primitives. Multiple instances of a task may be created at initialization time and multiple free TCBs, TDs, and ports may also be created at initialization time. Subsequently, dynamic operation of the system only needs to free and allocate these segments rather than creating them. Facilities also exist for dynamically creating new segments of any type, but with proper design such facilities should be used sparingly and not under hard real-time constraints. Using this approach, the system can be fast and predictable, yet still be flexible enough to accomodate change.

More specifically, a task consists of reentrant code, local data, dynamic data segments, a stack, a task descriptor and a task control block. Multiple instances of a task may be invoked. In this case the reentrant code and task descriptor are shared. All the above information concerning a task is maintained in the task descriptor. Much of the information is also maintained in the task control block with the difference being that the information in the task control block is specific to a particular instance of the task. For example, a task descriptor might indicate that the worst case execution time for TASK A is 5z milliseconds where z is the number of input data items at the time the task is invoked. The actual worst case time for this module is computed at invocation time and the computed value is then inserted into the TCB. The guarantee is then performed for this specific task instance. All the other fields dealing with time, computation, resources or criticalness are handled in a similar way.

6.2 Memory Management

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, these memory management techniques (as currently implemented) are not suitable to real-time applications with a need to guarantee timing constraints. Instead, the Spring Operating System memory management adheres to a memory segmentation rule with a fixed memory management scheme.

Memory segments include code, local data, data (including shared data), ports, stacks, TCBs, TDs and non-segmented memory. Tasks require a maximum number of memory segments of each type, but at invocation time a task might dynamically require different amounts of segments. The maximum is known a priori. Tasks can communicate using shared memory or ports. However, recall that the scheduling algorithm will automatically handle synchronization over this shared memory or ports. Tasks may be replicated at one or more sites. A program named the Configurator, calling the kernel primitives, initially loads the primary memory of each site with the entire collection of predetermined memory segments. Changes occur dynamically to this core resident set, but it is done under strict timing requirements or in background mode.

When a task is activated, any dynamic information about its resource requirements or timing constraints are computed and set into the TCB; the guarantee routine then 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 segments such as the ports, data segments, etc. at the right time. Again, tasks always identify their maximum resource requirements; this is feasible in a real-time system. If a task is guaranteed it is placed in the system task table (part of the memory associated with the system processor) for use by the application processor dispatcher. A separate dispatcher exists for system tasks which are executing on the system processor. 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 assumes. Also, pre-allocating as much as possible increases the speed of the OS with a loss in generality. One of the main engineering issues of hard real–time systems is where to make this tradeoff between pre-allocating resources and flexibility. Our approach makes this tradeoff by dedicating front–end processors to both I/O and tasks with short time constraints. As functionality and laxity of tasks increase, we employ on–line, dynamic techniques to acquire flexibility.

6.3 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. Even if the set of I/O devices changes, since they can be partitioned from the main system and changes to them are isolated, these changes have minimal impact on the rest of the kernel. 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. 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 interact more closely with the real-time application and have tight time constraints. They might invoke subsequent real-time higher level tasks. However, it is precisely because of the tight timing constraints and the relatively 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.

6.4 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 process, 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 three schedulers: one that schedules interrupts (usually immediately) on the front end processors in the I/O subsystem (what was discussed above), another that is part of the Spring kernel that guarantees and schedules high level application tasks that have hard deadlines, and a third which schedules the OS tasks that execute on the system processor. Interrupts from the front end I/O subsystem to the Spring kernel are handled by the system processors so this doesn't affect application tasks. In other words, I/O interrupts are treated as instantiating a new task which is subject to the guarantee routine just like any other task. The system processor fields interrupts (when turned on) from the I/O front end subsystem and shields the application tasks, running on the application processors from the interrupts.

It should be obvious from the discussion in this section that the Spring kernel is under development [17]. We expect to complete an implementation by the Summer of 1989. To reduce implementation time and costs, we are building the kernel by modifying the commercially available $VRTX^{TM}$ real-time kernel. Modifications are particularly needed to substitute our scheduling algorithms for the priority-based algorithm embedded in VRTX and to implement resources in a way that allows for segmentation and preallocation. In addition, enhancements will be needed to allow decentralized scheduling and execution in an environment where each node is a multiprocessor. The slow front end processing can be supported by the VRTX kernel itself and its IOX package. This is acceptable to us because the front end is relatively slow, not critical, and need not be very flexible. Since fast I/O typically requires periodic processing, nodes executing a simple version of the Spring kernel (in particular, containing the kernel modules that deal with periodic tasks) can serve the needs of fast I/O.

Clearly, in the design of the Spring Kernel, we have made a number of assumptions, in particular, those that relate to resource preallocation, handling slow and fast I/O, and the handling of interrupts. Our design decisions are also based on the assumption that the possibility of appropriately segmenting resources will give us a means by which to tame worst-case behavior. Experience with the kernel will help us understand the implications of these assumptions, as well as indicate whether they are reasonable.

7 Tool Support for Building Time-Critical Systems

Based on the discussions so far, we can classify the tools required as follows:

- A hardware testbed to implement, test, and evaluate the kernel (including the scheduling algorithm).
- A software testbed for evaluating alternative kernel algorithms, specifically, scheduling algorithms.
- A software testbed for evaluating competing architectures for time-critical systems.
- A repertoire of tools for designing time-critical applications.

The software testbeds will help us study the following:

- The effect of different segmentation strategies.
- The effect of using different types and numbers of processors to construct a multiprocessor node.
- The effect of different types of node interconnection structures as well as the protocols appropriate for them.
- The effect of the use of different scheduling algorithms appropriate for the different system configurations.

The effectiveness of a particular choice will be measured with respect to its impact on system performance, predictability, reliability, and flexibility. We see the testbed as serving a number of purposes. We discuss some of these now.

As opposed to a general timesharing system, architecture and operating system support for time-critical systems will be closely tied to the applications. Hence the testbed can be used to choose the appropriate mix of architecture and operating system components that is suitable for a particular (set of) application(s). Results of experimentation with the testbed will also be used to develop rules for segmenting resources, in particular, memory, secondary storage devices, as well as buses and channels connecting processors and nodes respectively. Experience with structuring applications using the testbed will assist us in building a set of rules and constraints that can then be used by the application development tools to be discussed next. Finally, observations made in the course of the experimentation will give us a wealth of information that will be used to produce a *knowledge base* to drive the meta-level control component of the Spring kernel.

This testbed is operational, but it is continually being enhanced. This testbed allows us to experiment with different scheduling strategies, both local and global, and study the impact of different parameter settings on the performance of the scheduling algorithm.

Let us now discuss the tools needed for designing tasks with predictable behavior. These tools will assist designers in the building of applications based on a set of constraints and rules that are geared to produce predictable systems with enhanced performance. The rules and constraints are related to the structuring, i.e., units of segmentation, of resources. They are designed to minimize the variations in task execution time and minimize the resource requirements of task components. Thus, the rules and constraints will help a designer to take a single task that requires different resources at different times during its execution and has wide variations in its execution times and divide it into a set of subtasks related by precedence constraints; a subtask will request resources that it needs during most of its execution; also, each subtask will have minimal variations in its execution time. Clearly, the rules and constraints are related to the types of resources in a distributed system, their segmentation properties, and the manner in which the kernel allocates these resources, i.e., schedules the tasks.

We have already formulated a set of preliminary rules and constraints which we plan to refine as we implement and experiment with the Spring kernel. We also envisage further rules as we apply existing rules to structure simple applications and study their predictability and performance properties both by implementing them using the Spring kernel and by experimenting with the testbed discussed earlier.

It is also possible to package our various real-time scheduling algorithms and baseline scheduling algorithms into a design tool. This tool would then provide system designers with timing information about a particular system configuration and workload. Continued modifications of the configuration and re-use of the tool would help converge on the proper system configuration needed to meet the timing requirements of the system. Due to lack of manpower we have not actually implemented such a tool.

8 Time Constrained Communication

Distributed real-time systems will have multiple, distributed tasks cooperating to achieve the application's goals. An important issue in having such tasks satisfy their timing constraints is the ability to deliver messages on-time. Such message transmission must be integrated with scheduling. This is one of the major open problem areas for distributed, real-time systems referred to as the *end-to-end* problem. For example, process A might want to communicate with process B which is physically remote. The entire step by step process of running process A, executing the send message primitive, invoking the OS, physically transferring the message, receiving the message, invoking the receiving task, processing the message, and replying all must be accomplished within a deadline. This requires an integrated scheduling and resource allocation policy. Our approach enables such a policy to be employed.

There are two broad ways of dealing with scheduling in the presence of message delays. The first is based on utilizing information about the maximum delay that a message will encounter [8]. Thus, if the nodes in a distributed time-critical system are connected by a local area network and the channel access protocol is designed to guarantee message delivéry within bounded time then communicating tasks can be scheduled assuming bounded message delivery delays. The second method to deal with scheduling tasks in the presence of message delays is to compute a deadline for each message delivery from the deadline requirements of the tasks; use a communication protocol that transmits messages so that they are delivered before their deadlines.

We have already developed two protocols appropriate for time constrained communication. One called the *Virtual-Time CSMA protocol* [31] belongs to the class of Inference avoidance protocols; the other is a *window protocol* [34] and belongs to the class of Inference seeking protocols. The protocols attempt to minimize the number of messages lost, i.e., minimize the number of messages that do not reach their destinations before their deadlines. Both protocols have been thoroughly evaluated via simulation to determine their appropriateness for real-time systems with different system and application characteristics. While these protocols are definitely better than those that do not take message deadlines into account, they are not sufficient to handle the types of messages that occur in real-time systems. For example, these protocols have to be extended to handle critical messages, i.e., those messages which if lost, may lead to a catastrophy. We are currently investigating *integrated protocols* which guarantee that critical messages will not be lost while minimizing loss for non-critical messages. This will maximize the value, to the system, of messages transmitted and at the same time increase channel utilization. Also, it still remains to better integrate these new protocols with the cpu scheduling algorithm.

9 Real–Time Transactions

It is accepted that the synchronization, failure atomicity, and permanence properties of transactions aid in the development of distributed systems. Many computations that occur in real-time systems possess these properties. However, little work has been done in utilizing transactions in a real–time context. Hence we have been looking at the use of ideas from Spring in the database context. Specifically, our current work in this area involves the development of an integrated approach for supporting real-time transactions. The following topics are being investigated:

- Concurrency control protocols,
- Recovery techniques,
- Deadlock resolution strategies, and
- CPU scheduling algorithms for transactions.

All the algorithms being developed are to be implemented, evaluated and compared on an existing distributed database testbed, called CARAT. CARAT contains all the major functional components of a distributed transaction processing system (transaction management, data management, log management, communication management, and catalog management) in enough detail so that the performance results will be realistic. CARAT runs on a 5-node distributed system. To support real-time transaction research we have expanded the CARAT testbed into a distributed real-time database testbed called RT-CARAT.

We have already developed four access control protocols based on locking and one based on forward optimistic concurrency control. The first locking protocol is based on the notion of a *virtual clock*, the second on a pairwise comparison of value functions, the third on separating deadlines and criticalness, and the fourth based on assuming that the system has an estimate of a transaction's execution time. For the locking based protocols we have already developed versions which can prevent deadlock rather than relying on deadlock detection [18], [5]. Evaluations of these protocols is currently in progress with encouraging preliminary results. Through further studies, we plan to understand the *combined* effect of concurrency control and recovery protocols, CPU and I/O scheduling algorithms, deadlock resolution strategies, and time constrained communication protocols on the performance of real-time databases.

10 Support for Real-time AI Applications

Many next generation real-time systems will include one or more expert system components or other AI programs. For example, applications such as avionics, the space station, process control, command and control, and robotics are all investigating this approach. Consequently, a significant amount of research is now being conducted in the area of real-time AI (RTAI). However, to date this work has not examined scheduling and resource allocation issues. Ignoring these issues can seriously affect the timing properties of the system.

To meet the requirements of AI applications, the operating system should have the following abilities:

- The ability to dynamically change criticalness, timing requirements, resource needs, precedence constraints, and even the structure of a computation, possibly as a function of various conditions.
- The ability to plan future execution times of functions that may subsequently change in various complicated ways.
- The ability to perform tradeoff analyses.
- The ability to use semantic information supplied by the application program and to return appropriate system information to the application.

To support AI applications, the Spring OS must therefore contain primitives to handle the requirements listed above as well as to support sophisticated task sets [24]. It must do this in such a manner as to make the system predictable, especially with respect to its timing properties. For AI applications, we plan to extend the Spring operating system with a Time Planner (this could consist of multiple versions of the local scheduler being invoked for different purposes, i.e., to suggest tradeoffs, to perform planning of future schedules for tasks or to suggest the causes of the problems in the current plan), and a Condition Monitor (this implements a general (situation,action) pairs capability), various Demons (including possibly a garbage collector that runs in parallel). The Meta Level Controller discussed earlier will be especially useful for complex applications such as AI.

The scheduling algorithm approach we are proposing for AI applications is an extension of the basic algorithm. The extension can be viewed from different perspectives. The original algorithm collects all application tasks that have been guaranteed to make their deadlines into a system task table (STT) which is ordered for dispatching. Executing the tasks in this order, guarantees that deadlines will be met. While we still maintain this dispatch list, we add other dimensions to the table, each additional dimension representing a different way of looking at the system (i.e., different scheduling plans and/or alternatives). These other dimensions are required due to the increased complexity of tasks, task interactions, and task dependencies as well as due to the ever changing environment and system conditions. The scheduler can be invoked on one or more dimensions to aid in planning future actions by obtaining tradeoff analyses among alternatives. The invocation of the scheduler as a *time planner* can occur from the application level, or from the OS level via simple conditional events or more complicated situation-action pairs.

Clearly, our examination of necessary system support for AI applications is in its initial stages. We propose to pursue these ideas further by focussing on one or more typical AI applications.

11 Formal Specification and Verification of Real-time Systems

We have recently embarked upon the development of a formal method specifically to aid in the specification and verification of real-time systems. Formal methods developed to reason about sequential and even concurrent processes often permit generality and computational power that obscure the fundamentally limited resources of any digital computer — limits that designers of real-time systems cannot ignore. *Modal Transition Arithmetic* (MTA) is being designed to address this problem; we are especially interested in developing *modelling and proof techniques* that are applicable to systems with large state spaces.

MTA combines elements from the algebraic theory of finite state automata, modal logics and primitive recursive arithmetic [25]. The system to be verified is modelled as a product of structured finite state automata. The key concepts here, in addition to the well-understood finite-state automata, are structured automata and product of automata. Structured automata allow us to capture the encapsulation and layering typical in computer systems. Also, this allows us to tame the state-space explosion problem that haunts typical automata-based approaches. Product(s) of automata are used to capture the concurrency inherent in most non-trivial computer systems. We specify and analyze these machines via a modal extension of primitive recursive arithmetic. There are two modalities: One corresponds to paths in a finite state machine, and the other corresponds to factors of algebraic products of state machines. The first modality permits us to reason about the dynamics of state change, and the second permits us to reason about systems composed of layers of state machines.

Modal Grammars are used to *indirectly* specify finite state systems. Specifications via grammars is especially useful for systems that are too large and complicated to be handled with traditional methods like state diagrams. This is because grammars allow us to precisely specify automata while avoiding the tedious enumeration of state sets or manipulation of regular expressions that have traditionally plagued finite state methods.

The difficulties of specification and verification are compounded when higher and lower levels of specification are given in different languages. We define a relatively simple syntax in which one can make assertions about the current state, states of factor automata, enabled transitions and future states. Within this formal syntax we can define quite high level operators including analogs of the operators of branching time, interval and real-time temporal logics.

Time is modeled through the use of *tick* transitions. Timing constraints can be concisely imposed using *tick* transitions, e.g., permitting the traversal of a transition labelled, say, α only if at least (or exactly or at most) n *ticks* have been traversed since the previous α transition was traversed. This construction and our use of products of automata also make it possible to describe systems composed of objects that change state at different rates or at different granularities of time.

We are currently working on MTA to better understand the algebraic properties of the model structures, to further develop the proof mechanism, and to build automated tools needed to prove properties of complex systems. Specifically, the following topics are being investigated:

- Implications of, and techniques for hierarchical proofs.
- Axiomatization of MTA and relative advantages of semantic and syntactic proof methods.
- Integration of model checking and proof theoretic paradigms.
- Complexity of answering queries about state machines specified by MTA grammars.
- Complexity of proof procedures and the effect of restricting the power of the arithmetic base on the complexity of proofs.

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, "The Integration of Deadline and Criticalness in Hard Real–Time Scheduling," *Proc. Real-Time Systems Symposium*, Dec. 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*, Dec 1986.

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

[5] J. Huang, "Protocols for Real-time Databases", *Ph.D. Thesis*, University of Massachusetts, (forthcoming), 1989.

[6] K. Ramamritham and J.A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, pp. 65-75, July 1984.

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

[8] K. Ramamritham, "Channel Characteristics in Local Area Hard Real-Time Systems," *ISDN and Computer Networks*, 1987.

[9] K. Ramamritham, J.A. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadline and Resource Requirements," submitted to *IEEE Trans. on Computers*, Oct. 1988.

[10] K. Ramamritham, J. Stankovic, P. Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems," submitted for publication.

[11] K.Ramamritham, "Scheduling Complex Periodic Tasks", (forthcoming) 1989.

[12] Shiah, Perng-Fei, "Real-Time Multiprocessor Scheduling Algorithms," MS Thesis, Univ. of Mass., January 1989.

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

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

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

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

[17] J.A. Stankovic and K. Ramamritham, "The Design of the Spring Kernel," *Proc Real-Time Systems Symposium*, Dec. 1987.

[18] J. A. Stankovic and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, 1988.

[19] J. A. Stankovic, "Real-Time Computing Systems: The Next Generation," UMASS Technical Report, TR 88-06, 1988.

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

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

[22] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," (to appear) *ACM SIGOPS Review*, 1989.

[23] J. A. Stankovic, K. Ramamrithm, P. Shiah, and W. Zhao, "Real-Time Scheduling Algorithms for Multiprocessors," submitted for publication.

[24] J. A. Stankovic, "The Spring Kerenl: Support for Real-Time AI Applications," extended abstract, submitted to EuroMico Workshop, Como, Italy, Dec. 1988.

[25] V. Yodaiken and K. Ramamritham, "A Modal Finite State Arithmetic for Specification," Technical Report, University of Massachusetts, Jan 1989.

[26] W. Zhao and K. Ramamritham, "Distributed Scheduling Using Bidding and Focussed Addressing," *Symp. on Real-Time Systems*, pp. 103-111, Dec 1985.

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

[28] W. Zhao and K. Ramamritham, "A Virtual-Time CSMA Protocol for Hard Real-Time Communication," *Real-Time Systems Symposium*, Dec 1986.

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

[30] W. Zhao, K. Ramamritham, and J.A. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *Special Issue on Real-Time Systems, IEEE Transactions on Computers*, Aug 1987.

[31] W. Zhao and K. Ramamritham, "Virtual Time CSMA Protocols for Hard Real-Time Communication," *IEEE Transactions on Software Engineering*, 1987.

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

[33] W. Zhao, J. Stankovic, K. Ramamritham, "A Multi-Access Window Protocol for Transmission of Time Constrained Messages," *8th Int. Conf on Distributed Computing Systems*, June 1988.

[34] W. Zhao, J. Stankovic, K. Ramamritham, "A Window Protocol for Transmission of Time Constrained Messages," *IEEE Transactions on Computers*, to appear.

[35] Zlokapa, G., "A Multiprocessor Architecture for Real-Time Systems," unpublished memo, University of Massachusetts, May 1985.