OVERVIEW OF THE SPRING PROJECT [1]

COINS Technical Report 87-54

Krithi Ramamritham
(413) 545-0196

John A. Stankovic
(413) 545-0720

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

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.

[1]The Spring Project is funded in part by the Office of Naval Research under contract 048-716/3-22-85 and by the National Science Foundation under grant DCR-8500332.

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 a physically distributed environment.
- 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.

Predictability allows us to determine if the required timing constraints can be met for a given system configuration. 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, they 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 approach to building distributed time-critical systems. Hence the Spring Project.

The Spring project began with the aim of developing scheduling algorithms for distributed time-critical systems and our major thrust so far has been on the development and evaluation of scheduling algorithms. However, in order to achieve the major goals of high performance (i.e., the need to be fast) and predictability, the rest of the system, namely, the remaining functions of the operating system, the hardware architecture, and the system development tools, must also be designed with these goals in mind. Hence, we are currently exploring the following four 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.

In the following sections we present an overview of the ongoing research in each of these areas. It is important to remember their interrelationships while purusing these sections.

2 Scheduling Algorithms for Distributed Time-Critical Systems

Good scheduling algorithms form the most essential component of operating systems underlying time-critical applications. Most tasks in time-critical applications have timing constraints such as deadlines or have a need for periodic execution. In addition, tasks typically have criticalness, resources requirements, precedence constraints, and placement or affinity constraints.

Current real-time operating systems use priority-based schedulers. This requires that actual timing constraints be mapped into priorities to meet these constraints - a time consuming, expensive, and error-prone task. Typically, one assigns priorities to tasks and via extensive tests verifies whether this assignment results in tasks' meeting their timing constraints. In dynamic systems task priorities have to be changed according to the nature of tasks in the system at any given time and hence the problem of priority assignment is further exacerbated in dynamic 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 criticalness, or initiate an exception-handling task with greater criticalness.

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 [2] and the arrival of higher criticalness

[2] Failures are handled by various techniques such as guaranteeing multiple instances of a task with proper

tasks, this task will execute by its deadline, and that all previously guaranteed tasks with equal or higher criticalness 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.

Here is an overview of our 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. An approach combining bidding and focussed addressing is used to determine such a node. If one such node exists, then the task 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.*

The scheduling algorithm separates policy from mechanism and is composed of 4 modules. At the lowest level there exists a dispatcher. The dispatcher simply removes the next task from a dynamically changing system task table (STT) that contains all guaranteed tasks already arranged in the proper order. 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.

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

timing considerations among the instances, other types of task replication, and reallocation of tasks after a host fails [10].

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 [3]. We considered both periodic tasks and nonperiodic tasks. After evaluating this algorithm [7] [8], 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 [15] presents a non-preemptive algorithm for guaranteeing tasks that have deadlines and need resources in exclusive mode. In [18], 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 [16]. Extensions to the scheme for cooperation among nodes to explicitly handle the presence of resources are discussed in [12], [4], and [13].

In parallel with the extensions involving resource constraints, we considered extensions to the basic algorithm to include precedence constraints among tasks. In our approach [1], 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. [2] reports on evaluation of this scheduling strategy as well as its efficacy in different situations.

Our evaluations showed that dynamic and distributed hard real-time scheduling is feasible and that the system can substantially benefit from distributed scheduling. 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*. Since scheduling is the control of task executions, we term this higher-level control as *meta-level control* [5]. 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 concerns 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.

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.
- Stragegies 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.
- Design of kernel components, in particular, interrupt handlers, resource management modules, and I/O subsystems, to support our scheduling approach based on the notion of guarantee.

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.

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 [6]. 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 delivery 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 developed and evaluated a new class of protocols that we have termed *virtual time CSMA protocols* [14], [17] specifically tailored for communication in time-critical systems.

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.

In the following sections we examine how our approach to scheduling leads to a specific set of requirements with respect to the rest of the operating system, architecture, and support tools. 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.

3 Operating System Support for Time-Critical Systems

Just like time-sharing operating systems, operating systems supporting time-critical applications have to provide facilities for process management, resource management, memory management, and device management. However, as we observed at the end of the last section, the requirements are much more stringent than in time-sharing operating systems.

As was already pointed out, the crucial ingredient of our kernel, called *the Spring kernel* 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 design of the kernel is based on the principle of segmentation as applied to hard real-time systems [9]. 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. The goals of using segmentation in hard real-time systems are to develop well defined units of each resource, to design

resource units such that fragmentation is minimized and utilization is maximized, and to allow us to allocate these units via an on line algorithm in such a manner as to provide predictability with respect to timing constraints.

The Spring kernel contains task management primitives that utilize the notion of pre-allocation where possible to improve speed and to eliminate unpredictable delays. The kernel's memory management scheme adheres to a memory segmentation rule: Divide physical memory into fixed size partitions that relate to space requirements of executing tasks and block size requirements of secondary storage. This is so that memory management techniques do 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 for real-time applications with tight timing constraints.

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. When a task is activated, the guarantee routine determines if it will be able to make its deadline which includes obtaining the necessary resources such as the TCB, memory blocks, etc. (Tasks always identify their maximum resource requirements; this is feasible in a real-time system). If a task is guaranteed, its code segment is filled, the TCB is set up, and it is placed in the system task table (part of OS memory), and ready for dispatching. Pre-allocating in a careful way increases the predictability and 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.

Because of the features of the the Spring kernel including the scheduling algorithm, the preallocation strategy, and the dedication of resources, we expect the system to be largely free of blocking. Synchronization is achieved primarily by the scheduler, and if not in this manner, then by bounded waits. The time for the bounded waits is accounted for in the guarantee process.

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. 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 timing 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 needed for processing information obtained from 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.

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 time-sharing 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 (as discussed above), and the other that is part of the Spring kernel proper that schedules all other tasks. Interrupts in the latter case are treated as initiating a new task which is subject to the guarantee routine just like any other task. If deadlines are very short on the tasks invoked via interrupts and to be handled by the Spring kernel scheduler, then time for these tasks might have to be preallocated.

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. To achieve this we plan to execute system tasks, such as those needed for scheduling, on separate *system processors*; this 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. Time for handling interrupts that are task-specific as well as waiting times, say for communication, are accounted for in the specification of the worst case computation time of application tasks.

It should be obvious from the discussion in this section that the Spring kernel is currently in the design stage [11]. We expect to begin implementation by the Summer of

1987. To reduce implementation time and costs, we plan to build the kernel by modifying the commercially available VTRX (TM) real-time kernel. Modifications will be 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.

In addition to the above assumptions, with a view to improving predictability, we have placed some burden on both the underlying architecture (for example, to provide the I O front end and processors for executing system tasks) as well as on the design of the application tasks (for example, to design tasks which are managable by the scheduling algorithm and the rest of the kernel). Hence, in the next two sections, we deal with these two aspects of the Spring project.

4 Architectural Support for Time-Critical Systems

We have already alluded to the various architectural features of time-critical systems. In this section, we consolidate our ideas on architecture and discuss the outstanding issues.

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 two directions. Since we do need a working hardware configuration to implement, test, and evaluate the kernel, we plan to put together, using off-the-shelf components, a distributed architecture, called *SpringNet* that while being rea-

sonable, is also easily modifiable. We discuss such an architecture later in this section. Concurrently, we will be building a software testbed for experimenting with alternative distributed architectures. This testbed can thus serve as a tool for designers and implementers of time-critical systems. We discuss such a tool in the next section.

SpringNet consists of a number of nodes for executing application tasks, one or more nodes for processing slow I/O, and one or more nodes for processing fast I/O. Nodes for application tasks and for fast I/O will typically be multiprocessor nodes. As was pointed out earlier, our design calls for the isolation of application tasks from external interrupts and from the overheads caused by the execution of kernel modules. Such an isolation contributes to predictable behavior. To achieve this isolation, we plan to execute all kernel services on a dedicated processor. External interrupts will be directed to and handled by this processor.

There are two ways to approach the design of this dedicated processor. The first is to choose an architecture for the dedicated processor that is tailored to the needs of the kernel modules. This will contribute to efficiency of the kernel. The remaining processors on a node can be general-purpose processors for executing application tasks. The second alternative is to populate a node with processors of one type. One of the processors can be designated as the dedicated processor. The advantage of this approach is that idle cycles on this processor may be used for executing application tasks.

Given the simplicity of the latter approach, we will pursue this for our initial implementation. In this implementation, each node will consist 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 module of the memory (associated with that processor) while having relatively slower access to the remaining memory. Thus all the memory is sharable by all the processors.

In the proposed implementation, two types of inter-node connections are possible. One is a point-to-point interconnection. Since initially our system will consist of a small number of nodes, this is plausible. An alternative is to connect the nodes via an Ethernet. Of course, this is not conducive to communication in time-critical systems since the protocols used are probabilistic in nature.

Given the inadequacies of both these possibilities, in parallel, using the software testbed mentioned earlier, we will be studying more appropriate alternatives. In this regard, we should recognize the presence of two types of messages in distributed time-critical systems. Typically, messages exchanged between scheduling components on nodes are not time-constrained. However, messages between subtasks of a guaranteed task, where the subtasks execute on different nodes, will be time-constrained. Also, as was pointed out in Section 2, in this case, the scheduling of the subtasks will have to be considered along with the scheduling of transmission of these messages. Thus, having a single subnet or a single

protocol for both types of messages will, in all probability, result in a complex solution. Hence we will be investigating the possibility of having subnets and protocols targeted for each type of message.

To summarize, with regard to the choice of architecture for time-critical systems, we have more open questions than we have answers. Hence our two-pronged attack on this problem: While experimenting with a prototype implementation consisting of off-the-shelf hardware components, we will be evaluating alternative architectures using a testbed.

5 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.
- 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.

We discussed the hardware testbed in the previous section. Given the interactions between the underlying architecture and the kernel algorithms, the two software testbeds could be built as one. This testbed will help us study the following:

- The effect of different segmentation strategies.
- The effect of using different types and number 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 time-sharing 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.

We have already built a part of this testbed. It 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.

References

[1] 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.

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

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

[4] K. Ramamritham, J.A. Stankovic, and W. Zhao, "Distributed Scheduling of Hard Real Time Tasks Under Resource Constraints in the Spring System", submitted to *IEEE Trans. on Computers*, Dec 1985.

[5] Ramamritham, K., J. Stankovic, W. Zhao, "Meta-Level Control in Distributed Real-Time Systems," (to appear in) *Conference on Distributed Computing Systems*, Sep 1987.

[6] K. Ramamritham, "Channel Characteristics in Local Area Hard Real-time Systems", to appear in *Computer Networks*, 1987.

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

[8] 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.

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

[10] J.A. Stankovic, "Decentralized Decision Making for Task Allocation in a Hard Real-Time System", submitted for publication, August 1986.

[11] J.A. Stankovic and K. Ramamritham, "The Design of the Spring Kernel", submitted for publication, April, 1987.

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

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

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

[15] 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.

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

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

[18] W. Zhao 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.

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