The Spring Kernel: A New Paradigm for Real-Time Operating Systems*

John A. Stankovic
Krithi Ramamritham

Dept. of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

February 24, 1989

Abstract

Next generation real-time systems will require greater flexibility and predictability than is commonly found in today's systems. These future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications. The Spring kernel is a research oriented kernel designed to form the basis of a flexible, hard real-time operating system for such applications. Our approach challenges several basic assumptions upon which most current real-time operating systems are built and subsequently advocates a new paradigm based on the notion of predictability and a method for on-line dynamic guarantees of deadlines. The Spring kernel is being implemented on a network of (68020 based) multiprocessors called SpringNet.

1 Introduction

Real-time computing is that type of computing where the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced. Real-time computing systems play a vital role in our society and the spectrum

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

of complexity of such systems varies widely from the very simple to the very complex. Examples of current real-time computing systems are the control of laboratory experiments, the control of engines in automobiles, command and control systems, nuclear power plants, process control plants, flight control systems, space shuttle and aircraft avionics, and robotics. Next generation systems will include the autonomous land rover, controllers of robots with elastic joints, systems found in intelligent manufacturing, the space station, and undersea exploration. The more complicated real-time systems are expensive to build and their timing constraints are verified with ad hoc techniques, or with expensive and extensive simulations. Minor changes in the system result in another extensive round of testing. Different components of such systems are extremely difficult to integrate with each other, and consequently add to the cost of such systems. Millions (even billions) of dollars are currently being spent (wasted) by industry and government to build today's real-time systems. The current brute force techniques will not scale to meet the requirements of guaranteeing real-time constraints of the next generation systems [13].

One of the serious problems in scaling to larger systems and in enhancing the performance predictability of these systems, we believe, is that the OS paradigms currently being used in these systems are wrong. In this paper we discuss why they are wrong (Section 2), and then present a new paradigm discussing the details of how the Spring kernel supports this new paradigm. These details are given in Sections 3 (the general system structure), Section 4 (the guarantee algorithm and its benefits) and Section 5 (other specifics regarding the Spring kernel). Concluding remarks are made in Section 6.

2 Limitations of the Current RTOS Paradigm

Many of the more complicated real-time systems include a real-time kernel [9][1] [6] [10]. These kernels are simply stripped down and optimized versions of timesharing operating systems. Consequently, they contain the same basic paradigms found in timesharing operating systems which, in many cases, are not suitable for real-time systems. In this section we discuss the limitations of the *current* real-time operating system paradigm.

The basic characteristics of current real-time kernels include:

- a fast context switch,
- a small size (with its associated minimal functionality),
- the ability to respond to external interrupts quickly,
- multi-tasking with task coordination being supported by features such as ports, events, signals, and semaphores,
- fixed or variable sized partitions for memory management (no virtual memory),
- the presence of special sequential files that can accumulate data at a fast rate,

- priority scheduling,
- the minimization of intervals during which interrupts are disabled,
- support of a real-time clock,
- primitives to delay tasks for a fixed amount of time and to pause/resume tasks, and
- special alarms and timeouts.

These kernels:

- assume that application tasks request resources as if they were random processes; if resources requested by a task are available, they are granted, otherwise the task is blocked, or blocking the tasks when the resources are not available
- assume that very little is known about the tasks a priori,
- attempt to insure fairness among the tasks,
- attempt to minimize average response time, and
- assume that there exist many independent tasks with independent address spaces.

Most real-time operating 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 (i.e., the system is too unpredictable).

Note that even though current kernels are successfully used in many of today's real-time embedded systems, it is only at greater cost and inflexibility than is desired. For example, the current technology burdens the designer with the unenviable task of mapping a set of specified real-time constraints into a priority order in such a manner that all tasks will meet their deadlines. Thus, when using the current paradigms it is difficult to *predict* how tasks invoked dynamically interact with other active tasks, where blocking over resources will occur, and what the subsequent effect of this interaction and blocking is on the timing constraints of all the tasks. As the next generation hard real-time systems become more dynamic and more sophisticated, it will be necessary to develop cheaper ways to guarantee real-time constraints and to meet the flexibility and predictability requirements.

Once again, 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. The

specific characteristics of tasks that can be assumed known are described later in Section 4.2. 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 deadline and that as many as possible other tasks also complete by their deadline (usually weighted by their importance). In other words, more appropriate metrics than fairness and average response time are required to be supported. 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. Tasks are not all independent and it may not be necessary to support independent address spaces.

In addition to the problems listed above concerning the basic paradigms of current operating systems, many other more specific problems exist with today's real-time kernels. 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 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 Section 4.3.

In summary, 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. Further, designers usually follow strict rules and guidelines in programming tasks, e.g., tasks will usually not have too long an execution time nor have a large variance in execution time. These facts can be exploited in developing a solution to real-time sytems. 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.

3 The General System Structure

This section presents the basic environment we are assuming and the general structure for the new 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 new, 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 it 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, a task may be preemptable or not, periodic or aperiodic, have a variety of timing constraints, precedence constraints and communication constraints. All these task complications must be addressed together with the environment characteristcs which will be dynamic, distributed and evolving.

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.

Of course, such systems will also have to be fault tolerant, but we do not discuss this aspect in this paper. In the remainder of this paper we hope to show how the Spring kernel incorporates the above ideas.

4 The Guarantee Algorithm and Its Benefits

The notion of guaranteeing timing constraints is central to our approach. Again, all critical tasks are guaranteed a priori and resources are reserved for them (this is part of the selective preallocation policy used in Spring). The essential tasks are guaranteed on-line. This allows for many task invocation scenarios to be handled dynamically (supporting the flexibility requirement). In this section we will discuss the dynamic guarantee of essential tasks in more depth, pointing out many of its advantages.

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[1] offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and **so that this overhead does not cause uncertainty in executing guaranteed tasks.** All system tasks are resident in the memory of the system processors. The I/O subsystem is a separate entity from the Spring kernel and it handles non-critical I/O, slow I/O devices, and fast sensors. The I/O subsystem can be controlled by some current real-time kernel such as VRTX [9], 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 portion[2] of our algorithm. Some modifications to our previously reported work have been made for implementing distributed scheduling on SpringNet. Most notably, the code for tasks is now 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, one of which can be used in two different ways. At the lowest level multiple dispatchers exist, one running on each of the application

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

[2] See [7] [8] for details on distributed scheduling.

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 inovation 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. Using the local scheduler as a planner is considered a high level OS activity and therefore will not be discussed any further in this paper. The third scheduling 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 (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. The distributed scheduling component and the MLC are not discussed any further in this paper since they can be considered upper levels of the OS and are not part of the Spring kernel itself. All OS tasks that run on the system processor have a minimum periodic rate which is guaranteed, but 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 Tasks

At the kernel level there exists an executable and guaranteeable entity called a task. A task consists of reentrant code, local data, dynamic data segments, a stack, a task descriptor (TD) and a task control block (TCB). Multiple instances of a task may be invoked. In this case the reentrant code and task descriptor are shared.

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 paper).

All the above information concerning a task is maintained in the task descriptor (TD). Much of the information is also maintained in the task control block (TCB) 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. At invocation time a short procedure is executed to compute the actual worst case time for this module and this value is then inserted into the TCB. The guarantee is then performed against this specific task instance. All the other fields dealing with time, computation, resources or criticalness are handled in a similar way.

While the kernel supports tasks, the local scheduler not only guarantees tasks, but also supports the abstractions of task groups and dependent task groups. A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single deadline. Each task acquires resources before it begins and can release the resources upon its completion. For task groups, it is assumed that when the task group is invoked, all tasks in the group can be sized (this means that the worst case computation time and resource requirements of each task can be determined at invocation time). A dependent task group is the same as a task group except that only those tasks with no precedence constraints can be sized at invocation time. The remaining tasks of the dependent group can only be sized when all preceding tasks are completed. The dependent task group requires some special handling with respect to guarantees. Our work in this area is tentative and hence is not discussed further in this paper.

4.3 The Guarantee Algorithm

The basic notion and properties of guarantee have been developed elsewhere [7] 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[3], what, where and when spare resources exist or will exist, a complete schedule for the guaranteed tasks, and which tasks are running under non-guaranteed assumptions,
- 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 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.

We now describe the guarantee algorithm and give an example. We begin the discussion by concentrating on the most difficult aspect of scheduling, handling the resource requirements of tasks. It is this aspect of scheduling which integrates cpu scheduling and resource

[3]In contrast, current real-time scheduling algorithms such as earliest deadline have no global knowledge of the task set nor of the systems ability to meet deadlines; they only know which task to run next.

allocation and thereby provides resource conflict avoidance and predictability. To simplify the discussion, we first describe the algorithm for independent tasks on nodes with a single application processor and a single system processor. At the end of this section we describe the extensions needed to handle precedence constraints (which, in turn, can be used to deal with end-to-end scheduling), periodic tasks, multiple application processors and criticalness. We will also discuss the run time costs of the algorithm.

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

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

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

   and

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

A *partial schedule* is a subset of the tasks in $\mathcal{T}$ whose scheduled start times have been assigned. A partial schedule $\mathcal{S}$ is *feasible* if the scheduled start times are such that all the tasks in $\mathcal{S}$ will meet their deadlines, i.e., $\forall T \in \mathcal{S}, (T_{sst} + T_P \leq T_D)$. For the tasks in a feasible schedule, the resources required by each task are available in the mode required by the task at its scheduled start time. A set of tasks is *schedulable* if there exists a feasible schedule for it. Thus, the scheduler must determine if a feasible schedule for a set of tasks exists. Also, it should be obvious from the above description that we are interested in non-preemptive scheduling. Thus, once a task begins execution, it will release its resources only after it has executed for T_P units of time.

Suppose tasks in set Γ have been previously scheduled and a new task arrives. We attempt to schedule the set of tasks $\Pi = \Gamma \bigcup \{\text{new task}\}$. If this set of tasks is found schedulable, the new task is scheduled, otherwise not. In either case, tasks in Γ remain scheduled.

For a given set of tasks, the problem of finding a feasible schedule is, in fact, a search problem. The structure of the search space is a *search tree*. The *root* of the search tree is the empty schedule. An *intermediate vertex* of the search tree is a partial schedule. A *leaf*,

a terminal vertex, is a complete schedule. Note that not all leaves correspond to feasible schedules. The goal of the scheduling algorithm is to search for a leaf that corresponds to a feasible schedule.

An optimal algorithm, in the worst case, may make an exhaustive search which is computationally intractable. In order to make the algorithm computationally tractable even in the worst case, we take a heuristic approach for this search. We develop a heuristic function, H. That is, on each level of the search, function H is applied to each of the tasks that remain to be scheduled. The task with the minimum value of function H is selected to extend the current (partial) schedule. As a result of the above directed search, even in the worst case, our scheduling algorithm is not exponential. Fortunately, our simulation studies, described in [14], [15], and [16] show that algorithms using linear combinations of simple heuristics perform very well — very close to the optimal algorithm that uses exhaustive search.

The pseudo code for our scheduling algorithm is given in Figure 1. The algorithm maintains two vectors EAT^s and EAT^e, each element of the vector corresponding to a resource. EAT^s and EAT^e, respectively, indicate the earliest available times of resources in shared and exclusive mode, given that the tasks in *schedule* have been scheduled and tasks in the *task_set* remain to be scheduled. At each level of search, according to the earliest available times of resources EAT^s and EAT^e, the algorithm calculates the earliest start time T_{est} for each task which remains to be scheduled. The detailed computation methods for EAT^s, EAT^e, and T_{est} are not discussed here (see [16]).

The algorithm invokes a boolean function called *strongly-feasible*. A feasible partial schedule is said to be strongly-feasible if all schedules obtained by extending this schedule one more level with any one of the remaining tasks are also feasible. If extending a feasible partial schedule by any one of the remaining tasks makes the extended schedule infeasible, then in none of the possible future extensions will this task meet its deadline. Hence it is appropriate to stop the search when a partial schedule is not strongly-feasible.

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

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

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

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

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

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

BEGIN

  schedule := empty;

  schedulable := true;

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

  EAT^s := 0;

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

  BEGIN

    calculate T_est for each task T ∈ task_set;

    IF NOT strongly-feasible(task_set, schedule) THEN

      schedulable := false;

      ELSE

      BEGIN

        apply function H to each task in task_set;

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

        T_sst := T_est;

        task_set := task_set - T ;

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

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

      END;

  END;

END;
```

Figure 1: Heuristic Scheduling Algorithm

From extensive simulations reported in [16] we have determined that a combination of two factors is an excellent heuristic function H. Consider:

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

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

One important aspect of this study, different from previous work, is that we specifically consider resource requirements and model resource use in two modes: exclusive mode and shared mode. We have shown that by modeling two access modes, more task sets are schedulable than if only exclusive mode were used. Further, this algorithm takes realistic resource requirements into account, and it has the appealing property that it avoids conflicts (thereby avoiding waits) over resources. It is important to note that resource conflicts are solved by scheduling at different times tasks which contend for a given resource. This avoids locking and its consequent unknown delays. If task A is composed of multiple task segments, and task A needs to hold a serially shareable resource across multiple task segments, then that resource is dedicated to task A during that entire period, call it X, and other tasks which need that resource cannot be scheduled to overlap with task A during period X[4]. Other tasks can overlap with task A. This strategy also minimizes context switches, since tasks are not subject to arbitrary interrupts generated by tasks arbitrarily waiting for resources. As an aside, if a particular hard real-time system has no conflict over resources except the CPU then it is possible to assume that resources are always available to ready tasks and one may use our preemptive algorithm [15], instead of the non-preemptive algorithm presented in this paper.

It is also important to note that the execution time cost of the algorithm is a function of the average number of resources that *each* task requires, and not the total number of resources. Consequently, even if there are 100's of critical resources defined per site, as long as each task requires some small number of them (e.g., less than 10), then the impact of the number of resources on the execution time of the algorithm is negligible.

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

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

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

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

where u = s or e if T needs R_i in shared or exclusive mode, respectively. However, more efficient techniques to handle periodic tasks are being investigated. These techniques are based on a guaranteed template so that each instance of a periodic task need not be guaranteed separately.

Third, in order to handle precedence constraints we simply add another factor to the heuristic function that biases those eligible tasks with long critical paths to be chosen next. A task becomes eligible to execute only when all of its ancestors are scheduled. Precedence constraints are used to model end-to-end timing constraints both for a single node and across nodes [4]. Again, various optimizations are being investigated here.

Fourth, a major advantage of our approach is that we can separate deadlines from criticalness. To date, we have equated the criticalness of a task with its value. Hence, to maximize value in the system, as many tasks as possible should make their deadline and, if any tasks cannot make their deadlines, then those tasks should be the least critical (least valuable) ones. Now, in describing the algorithm, for ease of explanation and to emphasize the *avoidance* and resource requirements aspects of our scheduling approach, it was described using deadlines only. In actuality, in the first phase of the algorithm the guarantee is performed as described above using deadlines and resource constraints. If the task is guaranteed then the criticalness value plays no part. On the other hand, when a task is not guaranteed, then the guarantee routine will remove the least critical tasks from the system task table if those preemptions contribute to the subsequent guarantee of the new task. The lowest criticalness tasks which were preempted, or the original task, if none, are then subject to distributed scheduling. Various algorithms for this combination of deadlines and criticalness, and local and distributed scheduling have been developed and analyzed [2].

5 Other Kernel Features

The kernel supports the abstractions of tasks, task groups, dependent task groups, various resource segments such as code, TCBs, TDs, local data, data, ports, virtual disks, 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 additional aspects of the Spring kernel is now given.

5.1 Task Management

The Spring kernel contains task management primitives that utilize the notion of preallocation where possible to improve speed and to eliminate unpredictable delays. 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, virtual disks and non segmented memory using the kernel primitives. Multiple instances of a task may be created at initialization time and multiple free TCBs, TDs, ports and virtual disks 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.

5.2 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. Let us now provide an example. We require that there be a reasonable amount of memory at each host[5], and that memory be considered a single address space.

Memory segments include code, local data, data (including shared data), ports, stacks, virtual disks, 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

[5]Many real-time systems are composed of disjoint phases, e.g., in the Space Shuttle [3] there are pre-flight processing, liftoff, space cruising, and descent phases. In this type of non-distributed system, the amount of memory needed is enough to contain the largest phase, not the entire system.

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. and 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 memory in 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.

5.3 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.

5.4 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 proper 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.

6 Summary

In this paper we claim that current real-time operating systems are using the wrong paradigms. We propose a new set of paradigms and discuss the Spring kernel which supports these new paradigms. The value of our approach has been repeatedly demonstrated by simulation [2,7,8,11,14,15,16]. We are now in the process of implementing the kernel on a network of multiprocessors. For more details on the kernel design see [12].

References

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

[2] Biyabani, S., "The Integration of Criticalness and Deadline Considerations in Hard Real-Time Systems," Masters thesis, Univ. of Mass, May 1988.

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

[4] Cheng, S., "Dynamic Scheduling in Hard Real–Time Systems," PhD thesis, Dept of Computer Science, UMASS, 1987.

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

[6] Holmes, V. P., D. Harris, K. Piorkowski, and G. Davidson, "Hawk: An Operating System Kernel for a Real–Time Embedded Multiprocessor," Sandia National Labs Report, 1987.

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

[8] Ramamritham, K., J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," submitted to *IEEE Transactions on Computers*, Oct. 1988.

[9] Ready, J., "VRTX: A Real-Time Operating System for Embedded Microprocessor Applications," *IEEE Micro*, pp. 8-17, Aug. 1986.

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

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

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

[13] Stankovic, J., "Misconceptions About Real-Time Computing," *IEEE Computer*, Vol. 21, No. 10, Oct. 1988.

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

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

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