THE SPRING KERNEL: A NEW PARADIGM FOR HARD REAL-TIME OPERATING SYSTEMS

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

COINS Technical Report 89-101

September 28, 1989

The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems[*][†]

John A. Stankovic
e-mail: stankovic@cs.umass.edu
phone: 413-5450720
Krithi Ramamritham
e-mail: krithi@nirvan.umass.edu
phone: 413-5450196

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

September 28, 1989

Abstract

Next generation, critical, hard real–time systems will require greater flexibility, dependability, 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. The Spring Kernel is being implemented in stages on a network of (68020 and 68030 based) multiprocessors called SpringNet. A preliminary version of the Kernel is now operational. Our research 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 on a method for on–line dynamic guarantee of deadlines. The purpose of this paper is to provide an overview of the major ideas of this new paradigm and show how the Kernel implements these ideas. Detailed descriptions of both the Kernel and the algorithms referred to in this paper can be found in the referenced material.

KEYWORDS: real--time kernel, multiprocessor kernel, real--time scheduling, integrated scheduling, new paradigm, next generation systems, dynamic time guarantees, deadlines, operating systems.

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

[†]This paper is a significantly updated version of an unrefereed paper that appeared in *ACM SIGOPS Review*, July 1989.

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 of their complexity varies widely from the very simple to the very complex. Current real–time computing systems are used in applications such as 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, teams of robots operating in hazardous environments such as chemical plants and undersea exploration, systems found in intelligent manufacturing, and the space station. These next generation real–time systems will be large, complex, distributed, adaptive, contain many types of timing constraints, operate in non-deterministic environments, and evolve over a long system lifetime. Many advances are required to address these next generation systems in a scientific manner. For example, one of the most difficult aspects will be in demonstrating that these systems meet their performance requirements including satisfying specific deadline and periodicity constraints. Timing constraints of today's systems 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 [11]. New paradigms, algorithms, architectures, design and implementation techniques, languages, operating systems, tools, etc. are required to support predictability, dependability, and flexibility so that next generation real–time systems can be carefully analyzed, can react to non-deterministic environments in a flexible manner, and can be constructed and maintained in a cost-effective manner.

We will not address all of these issues here. Rather we focus only on describing a new real-time operating system, called the Spring Kernel, and show how it provides basic support for next generation real-time systems. In developing this new operating system our research 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 on a method for on-line dynamic guarantee of deadlines. The purpose of this paper is to provide an overview of the major ideas of this new paradigm and to show how the Kernel implements these ideas.

In Section 2 we briefly identify current real-time operating system paradigms and say why we feel they are wrong for next generation real-time systems. In order to place the discussion of our new ideas in perspective, in Section 3, we then give a high level overview of the Spring Kernel. Section 4 presents our real-time operating system paradigm discussing the details of how the Spring Kernel supports this paradigm. Concluding remarks are made in Section 5.

2 Current Real-Time Operating Systems

Most current real-time operating systems (e.g., [7,1,3,8]) contain the same basic paradigms found in timesharing operating systems. These kernels are simply stripped down and optimized versions of timesharing operating systems. For example, while they stress fast mechanisms such as a fast context switch and the ability to respond to external interrupts quickly, they retain the main abstractions of timesharing operating systems including:

- viewing the execution of a task as a random process, in which if resources requested by a task are available, they are granted, otherwise the task is blocked,
- assuming that very little is known about the tasks *a priori* so that little (or no) semantic information about tasks is utilized at run time, and

- attempting to maximize throughput or minimize average response time[1].

In addition, very often today's real-time kernels use priority scheduling. Priority scheduling is a mechanism which provides no direct support for meeting timing constraints. For example, the current technology burdens the designer with the unenviable task of mapping a set of specified constraints on task executions into task priorities in such a manner that all tasks will meet their deadlines. Thus, when using the current paradigms together with priority scheduling it is difficult to *predict* how tasks, dynamically invoked, 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. Basically, currently used scheduling policies are inadequate for three main reasons: (1) they do not address the need for an integrated cpu scheduling and resource allocation scheme, (2) they don't handle the end-to-end scheduling problem, and (3) they are not used in a planning mode, thereby containing a myopic view of the system capabilities. We will define and further discuss these three important issues in the context of the Spring Kernel. Other research efforts are also challenging the current paradigms (e.g., see [4,12]).

3 The Spring Kernel - A High Level Overview

Rather than simply providing a factual description of the primitives in our Kernel, we will present the major abstractions (paradigms) supported by the Kernel. This approach lets us concentrate on the new ideas found in the Kernel. Before we do this, however, we first set the stage for the presentation of these new ideas by stating the general requirements (Section 3.1), describing the environments of applicability (Section 3.2), and outlining the structure of the hardware and operating system (Section 3.3). In Section 4, we then concentrate on the major new ideas found in the Kernel, showing how the Kernel supports these ideas and how they, in turn, provide basic support for next generation real-time systems.

[1]For a more detailed discussion of the problems with today's real-time kernels see [10].

3.1 Requirements

We believe that next generation, critical, real–time systems should be based on the following considerations:

- Tasks are part of a single application with a system-wide objective. The types of tasks that occur in a real–time application are known *a priori* and hence can be analyzed to determine their characteristics. There is no need to treat a task as a random process. Many characteristics of tasks (such as their importance, as well as their timing and resource requirements) can be determined *a priori* and utilized at run time. Further, designers must follow strict rules and guidelines in programming tasks, e.g., tasks must not have a large variance in their execution time. These facts can be exploited in developing a solution to real–time sytems and also in facilitating subsequent analysis of timing requirements.

- 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 (e.g., a very negative value or zero) if it does not make its deadline. Fairness and minimizing average response times are not important metrics for tasks with hard timing constraints.

- 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 properties such as timing and fault tolerance).

- Flexibility should be ensured so that system modifications and on-line dynamics are more easily accommodated.

3.2 The Environment and Definitions

Real–time systems interact heavily with the environment. We assume that the environment is dynamic, large, complex, and evolving. In a system interacting with such an environment there exist many types of tasks. Our approach categorizes the types of tasks found in real–time applications depending on their interaction with and impact on the environment. This gives rise to two main criteria on the basis of which to classify tasks: importance and timing requirements. Our Kernel then treats the different classes of tasks differently thereby reducing the overall complexity.

Based on importance and timing requirements we define three types of tasks: critical tasks, essential tasks, and non-essential tasks. Tasks' timing requirements may range over a wide spectrum including hard deadlines, soft deadlines, periodic execution requirements, while other tasks may have no explicit timing requirements. *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). It must be shown *a priori* that these tasks will always 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 are necessary to the operation of the system, have specific timing constraints, and will degrade the performance of the system if their timing constraints are not met. However, essential tasks will not cause a catastrophe if they are not finished on time. There are a large number of such tasks. 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.

Another timing issue relates to the closeness of the deadline. Some tasks may have extremely tight deadlines. These tasks cannot be dynamically guaranteed since it would take more time to ascertain the schedule for them than exists before the task's deadline. Such tasks must be treated differently, e.g., they might run in a front end using a cyclic scheduler, or a rate monotonic algorithm, or 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, communication constraints, and fault tolerance constraints. While we will not specifically address each of these issues in this paper, it would be unrealistic to design a real–time operating system for a large system that could not support these types of tasks.

3.3 A SpringNet Node

SpringNet (Figure 1) is a physically distributed system composed of a network of multiprocessors each running the Spring Kernel. Each multiprocessor contains one (or more) application processors, one (or more) system processors, and an I/O subsystem. Application processors execute previously guaranteed and relatively high level application tasks. System processors[2] offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and so **that external interrupts and OS overhead do not cause uncertainty in executing guaranteed tasks.** The I/O subsystem is partitioned away from the Spring Kernel and it handles non-critical I/O, slow I/O devices, and fast sensors.

Not surprisingly, the main components of the Kernel can be grouped into task management and scheduling, memory management, and intertask communication (ITC). While this sounds

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

similar to many other kernels, the abstractions supported are quite different and represent a new paradigm for real-time operating systems, as we shall see. Before we discuss the new ideas in detail (the subject of Section 4), we provide a brief overview of the main components of the Kernel in order to provide a better perspective for understanding the ideas in Section 4.

Task Management and Scheduling: The task management primitives support executable and guaranteeable entities called tasks and task groups. A task consists of reentrant code, local data, global data, 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. A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single group deadline. Each task acquires resources before it begins and releases 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). More flexible types of task groups are currently being investigated.

Tasks are characterized by:

- ID
- C (a worse case execution time - may be a formula that depends on various input data and/or state information)
- D (Deadline) or period or other real-time constraint
- importance (this is an indication of the value imparted to the system by the execution of the task)
- preemptive or non-preemptive property
- maximum number and type of resources needed (this includes memory segments, ports, etc.)
- type: critical, essential, or non-essential

- incremental task or not (incremental tasks compute an initial answer quickly and then continue to refine the answer for the rest of its requested computation time)
- location of task copies,
- Group ID, if any (tasks may be part of a task group)
- 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).

All the above information concerning a task is maintained in the task descriptor (TD). We have plans for adding semantic information concerning a task's fault tolerance requirements to the TD. Much of the above 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 importance are handled in a similar way.

Scheduling is an integral part of the kernel and the abstraction provided is one of a guaranteed task set. It is the single most distinguishing feature of the kernel. Since much of Section 4 is devoted to dicussing the merits of our scheduling approach, here we simply identify the scheduling components.

Our scheduling approach separates policy from mechanism and is composed of 4 levels. At the lowest level multiple dispatchers exist; one type of dispatcher running on each of the application processors, and another type executing on the system processor. The *application dispatchers* simply remove the next (ready) task from a system task table (STT) that contains previously guaranteed tasks arranged in the proper order for each application processor. The *system dispatcher* provides for the periodic execution of systems tasks, and asynchronous invocation when it can determine that allowing these extra invocations will not adversely affect guaranteed tasks, nor the minimum guaranteed periodic rate of other system tasks. Asynchronous invocation of system tasks are ordered by importance, e.g., the local scheduler is of higher importance than the meta level controller (see below).

The three higher level scheduling modules are executed on the system processor. The second level is a *local scheduler*. The local scheduler is responsible for locally *guaranteeing* that a new task or task group 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 and details can be found in [9]. The local scheduler, also called the guarantee routine, is invoked periodically or asynchronously as discussed above, and when invoked attempts to guarantee any new tasks or task groups that arrived since the last activation of the local scheduler. It guarantees the new task if the task can be scheduled to complete before its deadline and if the previously guaranteed tasks are not jeopardized by the execution of the new task.

The third scheduling level is the *distributed scheduler* which attempts to find a site for execution for any task or task group (or even partial task group) that cannot be locally guaranteed [5]. The fourth level 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, are not part of the Spring kernel itself, and are still being refined.

When a task is activated, any dynamic information about its resource requirements or timing constraints is computed and set into the TCB; the guarantee routine then determines if it will be able to make its deadline. 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, at activation time tasks always identify their maximum resource requirements; this is feasible in a hard real–time system).

Memory Management: Memory management primitives create various well defined resource segments such as code, stacks, task control blocks (TCB), task descriptors (TD), local data, global data, ports, virtual disks, and non segmented memory. 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 for real–time applications with a need to guarantee timing constraints. Instead, the Spring Kernel memory management adheres to a memory segmentation rule with a fixed memory management scheme. Tasks, as defined, require a maximum number of memory segments of each type, but at activation time a task may request fewer segments. All of the currently required segments are allocated when the task starts execution as part of the integrated scheduling and allocation scheme we use. If a task is programmed to dynamically ask for segments, then the worst case time for this task must include time to invoke the bounded Kernel primitives to get these resources which have already been allocated by the local scheduling algorithm.

Inter-Task Communication (ITC): Tasks can communicate using shared memory or ports. The ITC primitives support communication via ports between tasks, local or remote. The scheduling algorithm automatically handles synchronization over shared memory. We will not discuss ITC in this paper.

The final point we would like to make in this brief overview is that to enhance predictability, system primitives have capped execution times, and some primitives execute as iterative algorithms where the number of iterations it will make for a particular call depends on its capped execution time and on other state information including available time.

4 The New Paradigm

In light of the complexities of real–time systems, 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 assess the performance of the system with respect to requirements, especially timing requirements. In particular, the Spring Kernel stresses the real–time and flexibility requirements, and also contains several features to support fault tolerance. Our approach to supporting this new paradigm combines the following ideas resulting, we believe, in a flexible yet predictable system. The first three ideas are not new, but are quite useful and, consequently, we make use of them. They are:

- 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,
- use of the scheduler in a planning mode,
- the separation of importance and timing constraints, e.g., a deadline,
- end-to-end scheduling, and
- the utilization of significant information about tasks at *run time* including timing, task importance, fault tolerance requirements, etc. and the ability to dynamically alter this information.

We now indicate how the Spring Kernel incorporates the above ideas, thereby supporting predictability and flexibility.

Resource Segmentation: All resources in the system are partitioned into well defined entities. As mentioned, the Kernel supports the resource abstractions of tasks and task groups, and various resource segments such as code, stacks, TCBs, TDs, local data, global data, ports, virtual disks, and non segmented memory. It is important to note that tasks and task groups (which includes the operating system primitives) are *time and resource segmented and bounded* meaning that they are composed of well defined segments and that both the worst case execution times and the worst case resource requirements for these tasks are known. Kernel primitives are also time and resource segmented and bounded. There exists a prologue (as part of an Invoke primitive) that uses formulas for worst case needs to compute the timing and resource requirements for the current invocation. Resource segmentation thereby provides the scheduling algorithm with a clear picture of all the individual resources that must be allocated and scheduled. This contributes to the *microscopic* predictability, i.e., each task upon being activated is bounded in time and resource requirements. Microscopic predictability is necessary, but not sufficient condition for overall system predictability.

Functional Partitioning: As stated earlier, each node in SpringNet is a multiprocessor. There is a system processor, a communications processor, one or more application processors, and one or more front end I/O processors. Upon failure of the system processor, one of the application processors can become the systems processor. Functional partitioning provides many benefits including dividing a large problem into more manageable pieces, allowing us to treat critical, essential and non-essential tasks differently, allowing different solutions for different levels of granularity of timing constraints, and enabling the isolation of tasks that run on the application processors from unpredictable interrupts generated by the non-deterministic environment. This latter point is extremely important and together with our *guarantee algorithm* allows us to construct a more macroscopic view of predictable performance since the collection of tasks currently guaranteed to execute by their deadline are not subject to unknown, environment-driven interrupts. The unexpected interrupts can occur, but they affect the current tasks in a very predictable manner due to our on-line guarantee approach.

Many real–time constraints arise due to I/O devices including sensors. The set of I/O devices that exist for a given application will be relatively static in most systems. Even if the I/O devices change, since they can be partitioned from the application processors and changes to them are isolated, these changes have minimal impact on 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 predetermined and not part of the dynamic on-line guarantee. For example, the I/O processor might be running a cyclic scheduler or a rate monotonic scheduler, etc. However, the slow I/O devices might invoke a task which does have a deadline and which 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 processors might be front-end I/O processors or one or more of the application processors (See Figure 1). 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 higher level real–time tasks. However, it is precisely because of the tight timing constraints and the relatively static nature of the collection of sensors that we preallocate resources for the fast I/O sensors. In summary, our strategy suggests that some of the tasks which have real–time constraints can be dealt with statically, and others by a dynamic scheduling algorithm in the front-end. This leaves a smaller number of tasks which typically have higher levels of functionality and can tolerate a greater latency, for the dynamic, on-line guarantee routine.

Selective Preallocation: Critical tasks and tasks with very fast I/O requirements are preallocated. Further, 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, and sets up stacks, TCBs, TDs, local data, global data, ports, virtual disks and non segmented memory using the Kernel primitives. Multiple instances of a task or task group 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 (the first item on a list) these segments rather than creating them. While facilities also exist for dynamically creating new segments of any type, such facilities should not be used under hard real-time constraints. Using this approach, the system can be fast and predictable, yet still be flexible enough to accomodate major changes in non hard real-time mode.

***A Priori* Guarantee for Critical Tasks:** The notion of guaranteeing timing constraints is central to our approach. However, because we are dealing with large, complex systems in non-deterministic environments, the guarantee is separated into two main parts: an *a priori* guarantee for critical tasks and an on-line guarantee for essential tasks. All critical tasks are guaranteed *a priori* and resources are reserved for them either in dedicated processors, or as a dedicated collection of resource slices on the application processors (this is part of the selective preallocation policy used in Spring). Hence, critical tasks are guaranteed for the entire lifetime of the system. While *a priori* dedicating resources to critical tasks is, of course, not flexible, due to the importance of these tasks, we have no other choice!

On-line Guarantee for Essential Tasks: Due to the large numbers of essential tasks and to the extremely large number of their possible invocation orders, preallocation of resources to essential tasks is not possible due to cost, nor desirable due to its inflexibility. Hence, this class of tasks is guaranteed on-line. This allows for many task invocation scenarios to be handled dynamically (partially supporting the flexibility requirement). However, the notion of on-line guarantee has a very specific meaning as described in the first itemized point below. The basic notion and properties of guarantee for essential tasks have been developed elsewhere [6] and have the following characteristics:

- it allows the unique abstraction that at any point in time the operating system knows exactly which tasks have been 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, However, because of the non-deterministic environment the capabilities of the system may change over time, so the on-line guarantee for essential tasks is an *instantaneous* guarantee that refers to the current state. Consequently, at any point in time we have the *macroscopic* view that *all* critical tasks will make their deadlines and we know *exactly* which essential tasks will make their deadlines given the current load[4],

- 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). Basically, resource conflicts are solved by scheduling tasks at different times if they contend for a given resource,

- 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 guaranteed to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,

- provides early notification; by performing the guarantee calculation when a task arrives there may be time to reallocate the task to another host of the system via the distributed scheduling module of the scheduling approach; early notification also has *fault tolerance* implications in that it is now possible to run alternative error handling tasks early, before a

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

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

deadline is missed,

- within this approach there is the notion of still "possibly" meeting the deadline even if the task is not guaranteed, that is, if a task is not guaranteed it could receive idle cycles at this node, and, in parallel, there can be an attempt to get the task guaranteed on another host of the system subject to location dependent constraints, or based on the fault tolerance semantics of the task, various alternatives could be invoked,
- the guarantee routine supports the co-existence of real–time and non real–time tasks, and note that this is non-trivial when non real–time tasks might use some of the same resources as real–time tasks,
- the guarantee can be subject to computation time requirements, deadline or periodic time constraints, resource requirements where resources are segmented, importance levels for tasks, precedence constraints, I/O requirements, etc. depending on the specific guarantee algorithm being used in a given system.

Integrated CPU Scheduling and Resource Allocation: Current real–time scheduling algorithms schedule the CPU independently of other resources. For example, consider a typical real–time scheduling algorithm, earliest deadline first. Scheduling a task which has the earliest deadline does no good if it subsequently blocks because a resource it requires is unavailable. Our approach integrates CPU scheduling and resource allocation so that this blocking never occurs. Scheduling is an integral part of the Kernel and the abstraction provided is one of a guaranteed task set.

Because hard real–time scheduling in a multiprocessor with resource constraints is NP-hard, we use a heuristic approach. Scheduling a set of tasks to find a feasible schedule is actually a search problem. The structure of the search space is a search tree. An intermediate vertex of the search tree is a partial schedule, and a leaf, a terminal vertex, is a complete schedule. It should be obvious that not all leaves, each a complete schedule, correspond to feasible schedules. The heuristic scheduling algorithms we use try to determine a full feasible schedule for a set of tasks in the following way. It starts at the root of the search tree which is an empty schedule and tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, we use a heuristic function, H, which synthesizes various characteristics of tasks affecting real-time scheduling decisions to actively direct the scheduling to a plausible path. The heuristic function, H, (in a straightforward approach) is applied to each of the tasks that remain to be scheduled at each level of search. The task with the smallest value of function H is selected to extend the current schedule. A more efficient scheme we use allows application of the H function to only K tasks at each level. See [6].

The heuristic that we employ combines a task's worst case computation time, its deadline (or other timing constraint), and its resource requirements into a relatively simple formula. An innovation in our work is the way we quantify the resource requirements. Briefly stated, we quantify resource requirements by computing an Earliest Start Time, i.e., the earliest time by which all the resources required by a task will be available given the current partial schedule. The earliest start time incorporates both resource requirements and worst case computation time considerations. We then simply combine the earliest start time and deadline in a weighted formula to quantify the needs of each task[5]. Other considerations such as precedence constraints are handled by additional logic in the algorithm and not directly in the H function.

One very important aspect of this work, different from previous work, is that we not only specifically consider resource requirements, but we also 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.

By integrating cpu scheduling and resource allocation at run time, we are able to understand (at each point in time), the current resource contention and completely control it so that task

[5]See the Appendix for more details. If the paper is accepted we expect that the Appendix would become an accompanying overleaf box typical of IEEE Computer articles.

performance with respect to deadlines is predictable, rather than letting resource contention occur in a random pattern resulting in an unpredictable system.

Use of Scheduler in Planning Mode: Another important feature of our scheduling approach is how and when we use the scheduler, i.e., we use it in a *planning* mode when a new task is invoked. When a new task is invoked, the scheduler attempts to plan a schedule for it and some number of other tasks so that all tasks can make their deadlines. This enables our system to understand the total load of the system and to make intelligent decisions when a guarantee cannot be made, e.g. see the next point below. This is at odds with other real-time scheduling algorithms which, as mentioned earlier, have a myopic view of the set of tasks. That is, these algorithms only know *which task to run next* and have no understanding of the total load or current capabilities of the system. This planning is done on the system processor in parallel with the previously guaranteed tasks so it must account for those tasks which may be completed before it itself completes. A number of interesting race conditions had to be solved to make this work.

Separation of Importance and Deadline: A major advantage of our approach is that we can separate deadlines from importance. Again, all critical tasks are of the utmost importance and are *a priori* scheduled. Essential tasks are not critical, but each is assigned a level of importance which may vary as system conditions change. To maximize the value of executed tasks, *all* critical tasks should make their deadlines and as many essential tasks as possible should also make their deadlines. Ideally, if any essential tasks cannot make their deadlines, then those tasks which do not execute should be the least important ones. In the first phase of the guarantee algorithm, scheduling is done ignoring importance. If all tasks are guaranteed then the importance value plays no part. On the other hand, when a newly invoked essential task is not guaranteed, then the guarantee routine will remove the least important tasks from the system task table if those preemptions contribute to the subsequent guarantee of the new task. The low importance eliminated tasks, or the original task, if none, are then subject to distributed scheduling. Various algorithms for this combination of deadlines and importance have been developed and analyzed [2].

It is important to point out that our approach is much more flexible at handling the combination of timing and importance than a static priority scheduling mechanism typically found in real–time systems. For example, using static priority scheduling a designer may have a task with a short deadline and low importance, and another task with a long deadline and high importance. For average loads it is usually acceptable to assign the short deadline task the higher priority, and under these loads all tasks probably make their deadlines. However, if there is overload, it will be the high importance task which ends up missing its deadline. This condition would not occur with our scheme.

End-to-End Scheduling: Most *application* level functions (such as stop the robot before it hits the wall) which must be accomplished under a timing constraint are actually composed of a set of smaller dispatchable tasks. Previous real–time kernels do not provide support for a collection of tasks with a single deadline. The Spring Kernel supports tasks and task groups and is currently developing support for 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 the worst case computation time and resource requirements of each task can be determined. A dependent task group is the same as a task group except that computation time and resource requirements of only those tasks with no precedence constraints are known at invocation time. Needs of the remaining tasks of the dependent group can only be known when all preceding tasks are completed. The dependent task group requires some special handling with respect to guarantees which we have not done at this time. Precedence constraints are used to model end-to-end timing constraints both for a single node and across nodes and the scheduling heuristic we use can account for precedence constraints.

Dynamic Utilization of Task Information: Information about tasks and task groups is retained at run time and includes formulas describing worst case execution time, deadlines or other timing requirements, importance level, precedence constraints, resource requirements, fault tolerance requirements, task group information, etc. The Kernel then dynamically utilizes this information to guarantee timing and other requirements of the system. In other words, our approach retains significant amounts of semantic information about a task or task group which can be utilized at run time. Kernel primitives exist to inquire about this information and to dynamically alter the information. This enhances the flexibility of the system.

5 Summary

Most critical, real–time computing systems require that many competing requirements be met including hard and soft real–time constraints, fault tolerance, protection, and security requirements [11]. In this list of requirements, the real–time requirements have received the least formal attention. We believe that it is necessary to raise the real–time requirements to a central, focusing issue. This includes the need to formally state the metrics and timing requirements (which are usually dynamic and depend on many factors including the state of the system), and to subsequently be able to show that the system indeed meets the timing requirements. Achieving this goal is non-trivial and will require research breakthroughs in many aspects of system design and implementation. For example, good design rules and constraints must be used to guide real-time system developers so that subsequent implementation and *analysis* can be facilitated. Programming language features must be tailored to these rules and constraints, must limit its features to enhance predictability, and must provide the ability to specify timing, fault tolerance and other information for subsequent use at run time. Execution time of each primitive of the Kernel must be bounded and predictable, and the operating system should provide explicit support for all the requirements including the real–time requirements. The hardware must also adhere to the rules and constraints and be simple enough so that predictable timing information can be obtained, e.g., caching, memory refresh and wait states, pipelining, and some complex instructions all contribute to timing analysis difficulties. An insidious aspect of critical real–time systems, especially with respect to the real–time requirements, is that the weakest link in the entire system can undermine careful design and analysis at other levels. Our research is attempting to address all of these issues in an integrated fashion. However, in this paper we restricted our comments to the Spring Kernel. We claimed that current real–time operating systems are using the wrong paradigm. We proposed a new paradigm and discussed how the Spring kernel supports this paradigm.

The salient features of the Spring approach are:

- Given that a majority of tasks in a real–time application are known *a priori* and hence can be analyzed to determine their characteristics, our schemes use this information in preallocation and for on-line guarantee.
- *Predictability* is achieved by a combination of schemes, including resource segmentation/partitioning, functional partitioning of application tasks, executing system support tasks on a separate processor, and the use of integrated scheduling algorithms.
- *Flexibility/adaptability* is improved by dynamic (decentralized) task scheduling, and the use of meta-level control.
- The *value* of tasks executed is maximized through resource preallocation for critical tasks and the use of dynamic scheduling algorithms (that take task importance values into account) for essential and non-essential tasks.

The value of our approach has been repeatedly demonstrated by simulation [2,5,13]. We are now in the process of implementing the Kernel, in stages, on a network of multiprocessors. A preliminary version of the Kernel is operational. For more details on the kernel design see [9].

6 Appendix - Details of the Spring Scheduling Algorithm

The goal of our scheduling algorithm is to dynamically guarantee new task arrivals in the context of the current load. Specifically, if a set S of tasks has been previously guaranteed and a new task T arrives, T is guaranteed if and only if a feasible schedule can be found for tasks in the set $S \bigcup T$. Hence, determining whether a feasible schedule exists for a set of tasks, i.e., whether all the tasks in the set can be scheduled to meet their timing constraints, is the crux of the problem.

In practice, the actual algorithm that determines a feasible schedule must consider many issues including whether tasks are preemptive or not, precedence constraints (which is used to handle task groups), multiple importance levels for tasks, and race conditions. The race conditions arise because part of the set S may have completed execution or be in execution by the time time scheduling algorithm finishes. To simplify the discussion we will not consider these complications. Rather, we assume that tasks are characterized by the following:

- Task arrival time T_A;
- Task deadline T_D or period T_P
- Task worst case computation time T_C;
- Task resource requirements $\{T_R\}$;
- Tasks are non-preemptive.
- A Task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes.
- Task earliest start time, T_{est}, at which the task can begin execution; (T_{est} is calculated when the task is scheduled and T_{est} accounts for resource contention among tasks. It is a key ingredient in our scheduling strategy.)

As mentioned in the body of the paper, scheduling a set of tasks to find a feasible schedule is actually a search problem. The structure of the search space is a search tree. An intermediate vertex of the search tree is a partial schedule, and a leaf, a terminal vertex, is a complete schedule. In the worst case finding a feasible schedule requires an exhaustive search. Consequently, we take a heuristic approach.

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

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

However, it is possible to backtrack to continue the search even after a non-strongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it by a different task. The task chosen is the one with the *second* smallest H value. Even though we allow backtracking, the overheads of backtracking can be restricted either by restricting the maximum number of possible backtracks or by restricting the total number of evaluations of the H function. We use the latter scheme because we found it to be more effective.

The algorithm works as follows:

The algorithm starts with an empty partial schedule. Each step of the algorithm involves (1) determining that the current partial schedule is strongly-feasible, and if so (2) extending the current partial schedule by one task. In addition to the data structure maintaining the partial schedule, tasks in the task set S are maintained in the order of increasing deadlines. This is realized in the following way: When a task arrives at a node, it is *inserted*, according to its deadline, into a (sorted) list of tasks that remain to be executed. This insertion takes at most $O(n)$ time. Then when attempting to extend the schedule by one task, three steps must be taken: (1) strong-feasibility is determined with respect to the first (still remaining to be scheduled) N_k tasks in the task set, (2) if the partial schedule is found to be strongly-feasible, then the H function is applied to the first N_k tasks in the task set (i.e., the k remaining tasks with the earliest deadlines), and (3) that task which has the smallest H value is chosen to extend the current schedule. Given that only N_k tasks are considered at each step, the complexity incurred is $O(nk)$ since only the first N_k tasks (where $N_k \leq k$) are considered each time. If the value of k is constant (and in practice, k will be small when compared to the task set size n), the complexity is linearly proportional to n, the size of the task set. While the complexity is proportional to n, the algorithm is programmed so that it occurs a fixed worst case cost by limiting the number of H function evaluations permitted in any one invocation of the algorithm. Also, see [6] for a discussion on how to choose k.

Before we list possible H functions, we should clarify some terms. Whereas typically nonperiodic tasks are invoked with a deadline for completion and can be started anytime after they are invoked, the deadline and start times of periodic tasks can be computed from the period of the tasks. (There are more efficient ways to deal with periodic tasks, for example, by generating a separate scheduling template applicable to them, but we will not go into that here.)

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

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

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

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

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

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

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

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., J. Stankovic, and K. Ramamritham, "The Integration of Criticalness and Deadline In Scheduling Hard Real-Time Tasks," *Real-Time Systems Symposium*, Dec. 1988

[3] 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.

[4] Levi, S., S. Tripathi, S. Carson, and A. Agrawala, "The MARUTI Hard Real-Time Operating System," *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989.

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

[6] Ramamritham, K., J. Stankovic, and P. Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems," *Proc. Int. Conf. on Parallel Processing*, August 1989.

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

[8] 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.

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

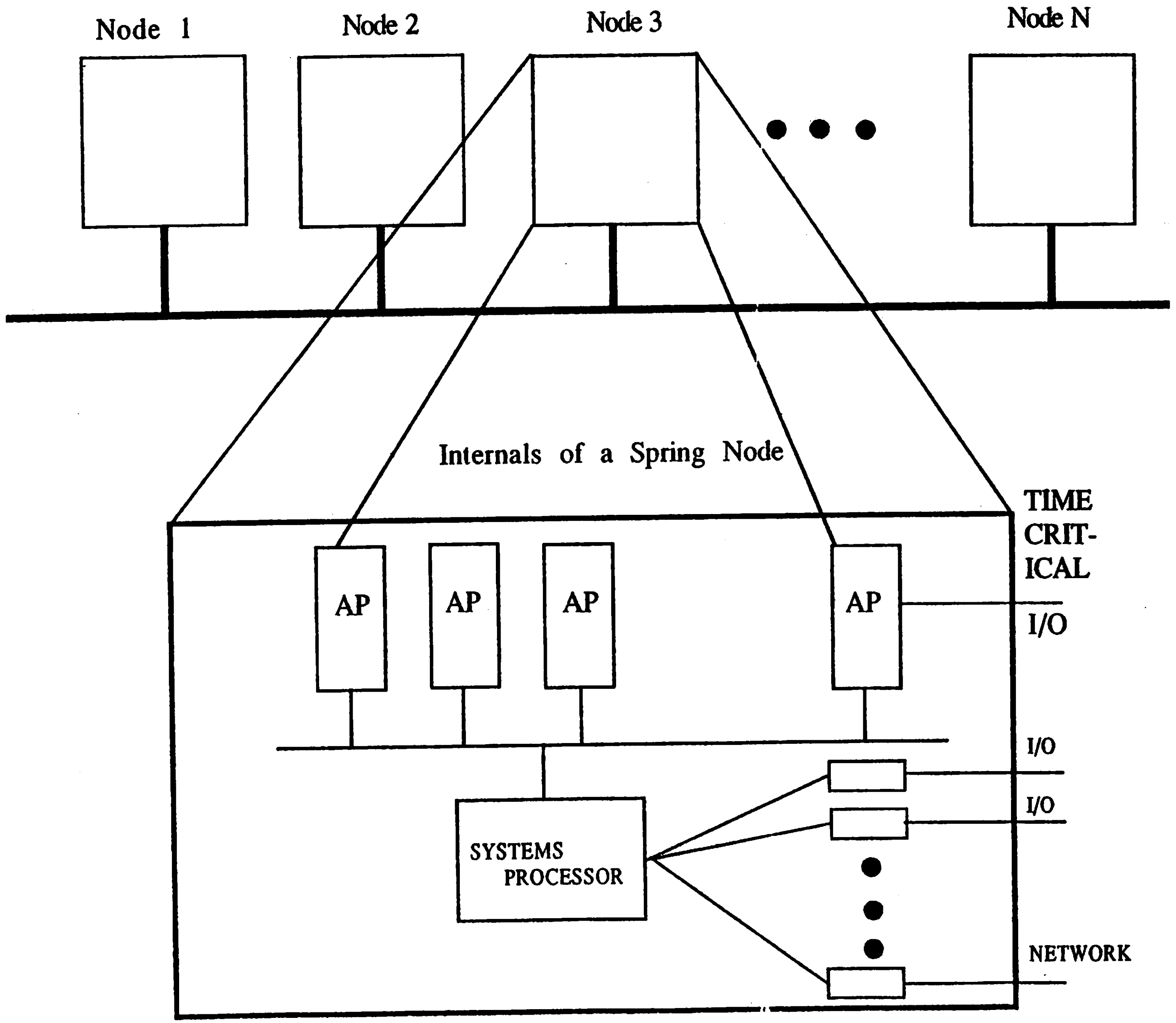
[10] Stankovic, J. and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989, pp. 54-71.

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

[12] Tokuda, H., and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989.

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

FIGURE 1: SpringNet


Node 1
Node 2
Node 3
Node N
Internals of a Spring Node
TIME CRIT-ICAL
I/O
AP
AP
AP
AP
I/O
I/O
SYSTEMS PROCESSOR
NETWORK